UNIVERSITY OF Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

Bounded Model Checking of Multi-threaded Programs via Sequentialization

by

Omar Inverso

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

November 2015

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Omar Inverso

In order to achieve greater computational power, processors now contain several cores that work in parallel and, consequently, multi-threaded software is rapidly becoming widespread.

The inherently nondeterministic nature of concurrent computations can cause errors that show up rarely and are difficult to reproduce and repair. Traditional testing techniques perform an explicit exploration of the possible program executions, and are thus not adequate to spot such bugs. They need to be complemented by symbolic verification techniques that analyse multiple thread interactions simultaneously.

Sequentialization consists in translating a given concurrent program into a corresponding non-deterministic sequential program that simulates executions of the original program. We investigate on whether combining sequentialization (to symbolically represent thread interleavings) with bounded model-checking (BMC) can be effective for finding errors in concurrent software.

Specifically, we target multi-threaded C programs with POSIX threads. We make the following contributions: (1) evaluate the Lal-Reps sequentialization schema in combination with BMC; (2) propose and evaluate a new sequentialization schema specifically tailored to BMC and aimed at fast bug finding; (3) present a framework for building tools based on sequentialization.

Contents

List of Figures

List of Tables

Acknowledgements

The work presented in this thesis would not have been possible without the continuous guidance, support, and encouragement of my supervisor, Gennaro Parlato, who never hesitated to advise me throughout the course of my studies. Gennaro and Bernd Fischer introduced me to my current research area of program analysis and verification. Special thanks to Bernd Fischer and Salvatore La Torre for having kindly agreed to review my drafts, providing meticulous feedback that has considerably improved this dissertation. Thanks to Denis Nicole and Michael Tautschnig for having agreed to examine my work. Thanks to Ermenegildo Tomasco and Truc Nguyen Lam for their friendship, collaboration, and enthusiasm in tuning our tools up for the software verification competitions. I would like to say thank you to 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 ECS for funding my doctoral studies. Thanks to my family.

Chapter 1

Introduction

The steady exponential increase in processor performance has reached the inevitable turning point, at which it is no longer feasible to increase the density or clock frequencies of individual processors. In order to achieve greater computational power, processors now contain several cores that work in parallel and, consequently, multi-threaded software is rapidly becoming widespread.

Multi-threaded programs consist of several *threads* of computation active at the same time and communicating either by sharing variables or sending messages. An inherent source of complexity is in the number of possible thread interleavings, which grows exponentially with the number of threads and statements in the program. Multi-threaded software is hard to implement: software developers not only have to guarantee the correctness of each individual thread, but need to take into account the possibly complex interactions between threads; to avoid unwanted behaviour, expedients such as synchronisation need to be used, which add complexity to the code, and can introduce concurrency-specific errors.

The nondeterministic thread interactions can cause errors that show up rarely and are difficult to reproduce and repair [BBdH+09]. Due to their explicit exploration of the possible executions of a program, traditional testing-based techniques are not adequate to spot such bugs, and thus need to be complemented by automated verification techniques for detecting errors in a systematic and symbolic way.

However, the symbolic verification of concurrent programs poses additional challenges. Concurrency exacerbates the theoretical limitations of automated software analysis due to the exponential blow up of the state space, and in practice by introducing additional complexity at different levels: (1) at the level of multi-threaded applications using high-level synchronisation primitives, (2) at the level of the software layer that implements the concurrency libraries and synchronisation mechanisms, and (3) at the level of the memory models adopted by compilers and modern multicore architectures in order to optimise

performance. Consequently, the state of the art for concurrent program verification lags behind that for sequential programs.

Researchers have successfully explored a wide range of techniques and tools to address real-world sequential programs, and software used in practice can already be successfully analysed; organisations such as IBM, Intel, Microsoft, NASA, and NEC are building dedicated divisions that regularly use program verification in industrial projects. As a result, there are several mature techniques and tools for the analysis of sequential programs based on model-checking, data-flow analysis, abstract interpretation, deductive verification, etc. This dichotomy (i.e., the availability of strong verification tools for sequential programs, and the lack of similarly strong tools for concurrent programs) is the starting point of our research.

Verification of concurrent software has been the subject of extensive research over the last few decades, and a variety of different approaches have been proposed. The canonical approach, which is implemented for example by SPIN [Hol97], VeriSoft [God05], CHESS [MQB+08], and ESBMC [CF11], is to explicitly explore the individual thread interleavings; however, their large number makes scaling-up difficult. Therefore, symbolic approaches that analyse simultaneously different thread interactions are highly desirable. One symbolic approach is to model executions of concurrent programs using partial orders [SW11, CKL04]. This has led to effective bug hunting tools based on bounded model checking [BCCZ99] that leverage modern satisifiability (SAT/SMT) solvers for the analysis.

A different symbolic approach, called *sequentialization* and originally proposed by Qadeer and Wu [QW04], is to translate the concurrent programs so that verification techniques or tools that were originally designed for sequential programs can be reused without any changes. This is the main theme of our research. Sequentialization can be implemented as a code-to-code translation from the concurrent program into a corresponding non-deterministic sequential program that simulates all executions of the original program. The sequential program contains both the mapping of the threads in the form of functions, and an encoding of the scheduler, where the non-determinism allows to handle different concurrent schedules collectively. This approach has three main 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; (3) sequentializations can be designed to target multiple backends for sequential program analysis.

Most sequentializations proposed in the literature focus on capturing under-approximations of concurrent programs communicating through shared memory. Lal and Reps [LR09] showed a sequentialization (LR) that works for any given program with a finite number of threads and captures all program's executions up to a given number of context switches. A *lazy* sequentialization for bounded context rounds that ensures that sequential program

explores only reachable states of the concurrent program was defined by La Torre et al. [LMP09a], and it was empirically shown to be more efficient than LR on multi-threaded Boolean programs. A sequentialization for an unbounded number of threads and bounded round-robin rounds of context-switches is also known [LMP10]. Other sequentializations cope with the problem of handling dynamic thread creation [BEP11, EQR11]. A sequentialization targeting real-time systems [CGS11] and distributed applications where threads communicate through FIFO channels [BE14] has also been proposed.

1.1 Contributions

All the sequentialization schemas proposed to date are fundamentally theoretical in their nature. The intellectual effort is in fact uniquely spent to capture the semantics of the original programs, ignoring the details of the underlying technology for sequential analysis. The evaluation does not go beyond proof-of-concepts or prototypes, and in the most interesting cases is limited to show that on abstractions of the original program it is possible to achieve good results in combination with BDD-based analysis [LR09, LMP09a]. In general, little evidence is provided for the effectiveness of sequentialization-based approaches on general-purpose real-world software, and there is no trace of a systematic and comprehensive evaluation on combining existing sequentializations with existing mature techniques for sequential analysis. In particular, we have argued that the symbolic representation of thread interleavings is essential for successful analysis of concurrent software. It is not clear whether combining sequentialization (to symbolically represent thread interleavings) with bounded model-checking can be effective for finding errors in real-world concurrent software.

In order to address this question, in this thesis we fix our target to multi-threaded C programs [ISO11] with POSIX threads [ISO09], because it is a fairly standard program category for most operating systems, largely used in device drivers and embedded systems.

Our research then makes the following contributions:

1. we evaluate the Lal-Reps sequentialization schema in combination with BMC (Chapter 3);
2. we develop, implement, and evaluate a novel lazy sequentialization schema specifically tailored to BMC and aimed at fast bug finding (Chapter 4);
3. we present our CSeq framework for building sequentialization tools (Chapter 5).

CSeq, including the implementations of the two schemas above, can be found at the project's homepage at `http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html`.

The following subsections give an overview of those contributions.

1.1.1 Evaluation of the Lal-Reps Schema

We choose the Lal-Reps schema (LR) [LR09] as the starting point of our research, following up on a preliminary empirical study [GHR10] where it is shown to be well suited to BMC, in contrast to the schema proposed in [LMP09a] that suffers from exponentially sized verification conditions due to the inlining performed by the back-end on the sequentialized file. Moreover, LR is a context-bounded analysis method and so should fit well into the general BMC framework as a bug finding approach; this hypothesis is motivated by previous empirical work that shows that errors typically occur within few context switches [QW04, MQ07, TDB14].

LR is a simple and elegant sequentialization that simulates up to a given number of round-robin executions of the original program. The sequentialized program simulates each thread using separate, non-deterministically initialised copies of the shared memory. These copies are left totally unconstrained during the simulation, and the values corresponding to unfeasible executions are pruned away only at the end (whence the term *eager* exploration).

The schema was originally proposed for Boolean programs and for a fixed number of threads. We work out the details to adapt it to multi-threaded C programs with POSIX threads and dynamic thread creation.

The evaluation of our prototype in combination with three different bounded model-checkers shows some initial encouraging results on a few non-trivial test cases, where the combined system formed by LR-CSeq and the bounded model-checker is competitive with native concurrency handling. However, we identify the theoretical limitations intrinsic to the LR schema as major barriers to extending our prototype on more complex real-world software.

In the sequentialized program, the large number of extra variables (that grows with the number of threads, the number of rounds, and the size of the global memory that is shared by the threads) and the resulting high degree of nondeterminism overwhelm the backend, causing performance problems.

Eager guessing severely limits backend integration. Implicit safety properties, such as array bounds violations or invalid pointer dereferences that are handled by the backend, must be translated into explicit assertions, and their check by the backend must be suppressed, in order to prevent spurious triggering due to eager guessing. This adds overhead to the translation and therefore negatively affects the performance of the backend.

A bigger problem is caused by heap-allocated memory that is accessible to all threads, and so needs to be treated similarly to global variables. Intuitively, the checker would have to guess and carry around memory blocks of possibly variable size. Such an explicit modelling

of the global memory would add even further overhead to the translated program. Moreover, implementing a memory model forces low-level details into consideration, which clashes with the very idea of sequentialization intended as a separation of concern that allows to focus on concurrency aspects.

1.1.2 A New Lazy Sequentialization Schema

We address the question as to whether it is possible to derive a novel schema that does not suffer from the drawbacks evidenced by the LR schema. In particular, our goal is to keep the focus on BMC, to simulate the same schedules as in LR (for a fair comparison), to avoid eager guessing in order to improve on backend performance and integration, and to reduce the overall translation overhead by limiting the non-determinism and keeping the control-flow structure simple in the translated program.

We design a new, surprisingly simple lazy sequentialization schema that works well in combination with BMC and is very effective for finding bugs. In contrast to the LR schema that performs the sequentialization upfront, the new schema works after the program unfolding stage and thus aggressively exploits the structure of bounded programs. The idea is to convert each thread into a *thread simulation function* that is invoked as many times as the given number of rounds. We inject at each visible statement non-invasive control code that simulates thread preemption and resuming at nondeterministically guessed context switch points. The statements are identified by unique numerical labels, so that for each thread we maintain the point at which the context switch was simulated in the previous round and where the computation must thus resume in the current round. The thread-local variables are made persistent by forcing their storage class to be static, so we do not need to re-compute them when resuming suspended executions, therefore avoiding the exponentially growing formula sizes observed earlier [GHR10].

This translation introduces very small memory overheads and very few sources of nondeterminism, and thus results in simple formulae. The new schema only explores reachable states of the input program (*lazy* approach) and thus requires no built-in error checks nor any special dynamic memory allocation handling, but can rely on the backend for these. The resulting sequentialized program simulates all bounded executions of the original program for a bounded number of rounds.

We implement this sequentialization in the Lazy-CSeq tool, which is highly competitive and ranked first in the Concurrency category of the last two editions of the SV-COMP Software Verification Competition (SV-COMP). The tool successfully handles all the proposed test cases without false results. A detailed evaluation shows its effectiveness in a bug-hunting setting, and in particular the superiority over our own implementation of

the LR schema as well as its competitiveness with the state-of-the-art tools with built-in concurrency handling.

1.1.3 A Sequentialization Framework

We present our framework for fast prototyping and development of sequentialization-based tools, CSeq, that subsumes our experience in working on sequentialization.

Reasoning at the level of the source code can offer a very abstract and expressive representation, and thus can indeed support more intricate reasoning than feasible at lower levels (e.g., in terms of SSA form or of directly on the verification condition) where potentially relevant information on the program is inevitably lost. In the particular case of multi-threaded programs, sequentialization as source translation provides a clear-cut separation of concern that allows the tool designers to focus on concurrency, disregarding any other detail of the program. Our framework emphasises these aspects by supporting a modular approach to source-to-source translation. We use string-based transformation, which is more intuitive than rewrite rules, and thus closer to a developer's standpoint.

CSeq reduces the overall engineering effort required in order to build a new tool by providing concurrency-aware parsing and data structures. In addition, it comes with a few built-in standard transformations (such as loop unrolling, function inlining, etc.) commonly used in program analysis, and a set of built-in functionalities (such as the backend integration, and the support for counterexample translation).

Our framework has already been used to build two other tools [TIF+15, NFLP15] (besides the ones developed for this thesis) within very compressed development frames and with modest engineering effort. We release it as open-source software, free for the community to use.

1.2 Structure of the Thesis

This thesis is organised as follows. In Chapter 2 we provide a short overview of bounded model-checking and sequentialization, including a brief review of work related to ours. In Chapter 3 we present our evaluation of the Lal-Reps schema. In Chapter 4 we present and evaluate our second contribution, our novel lazy sequentialization schema. In Chapter 5 we present our third contribution, the CSeq sequentialization framework. We conclude in Chapter 6 with our final considerations and possible directions for future work.

Chapter 2

Background

2.1 Preliminary Notions

In this section we introduce the main concepts used throughout the thesis.

2.1.1 Alphabets, Words and Languages

An *alphabet* Σ is a non-empty finite set of symbols.

A *string* or *word* over Σ is a sequence of elements of Σ. A *finite word* w of length n over Σ is a sequence $w = (a_0, a_1, \ldots, a_{n-1})$ where $a_i \in \Sigma$ for $0 \leq i \leq n-1$. The *empty word* ε has length 0. A finite word of length n can also be seen as a function $w : \{0, 1, \ldots, n-1\} \to \Sigma$, where $w(i)$ gives the symbol at position i. Similarly, an *infinite word* w over Σ is a function $w : \mathbb{N} \to \Sigma$.

The set of all words of length n over Σ is denoted as Σ^n. The set of all finite words, or *closure* of Σ, is $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$. The set of all finite non-empty words is Σ^+.

A *language* $\mathcal{L}$ over an alphabet Σ is a (possibly infinite) set of finite-length words over that alphabet, i.e., $\mathcal{L} \subseteq \Sigma^*$.

2.1.2 Propositional Logic

The alphabet of propositional logic consists of:

- a countably-infinite set $V = (p, q, r, \ldots)$ of *propositional variables*
- the *logical connectives* or *Boolean operators* NOT ($\neg$), AND ($\wedge$), OR ($\vee$)
- parentheses (and).

The set $\mathcal{P}$ of *well-formed formulae of propositional logic* is inductively defined as follows:

- all propositional variables are well-formed formulae (called *atoms*)
- if $\varphi \in \mathcal{P}$, then $\neg\varphi \in \mathcal{P}$ and $(\varphi) \in \mathcal{P}$
- if $\varphi_1 \in \mathcal{P}$ and $\varphi_2 \in \mathcal{P}$, then $(\varphi_1 \wedge \varphi_2) \in \mathcal{P}$ and $(\varphi_1 \vee \varphi_2) \in \mathcal{P}$.

Further logical connectives, such as IMPLIES ($\rightarrow$) and EQUIVALENT TO ($\leftrightarrow$), are derived from the standard connectives ($\neg$,$\wedge$,$\vee$) in the usual way. A *literal* is a variable p, also called *positive literal* or its negated form $\neg p$, then called *negative literal*. The formulae $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$ are referred to as *conjunction* and *disjunction* of φ_1 and φ_2, respectively. A *clause* is disjunction of literals, and it is also known as *unary clause* if it contains just a single literal. A *Horn clause* is a clause that contains at most one positive literal. The formula $\neg\varphi$ is referred to as *negation* of φ. In the rest of this section and throughout the thesis we use the terms formula, Boolean formula, and well-formed formula interchangeably.

An *interpretation*, *valuation*, or *assignment* is a function $I : V \rightarrow \{\bot, \top\}$ that assigns either TRUE ($\top$) or FALSE ($\bot$) to every propositional symbol in V. If $I(p) = \top$, then p is said to be *true under the interpretation* I. If $I(p) = \bot$, then p is *false under the interpretation* I.

Given a well-formed formula φ and an assignment I, either I satisfies φ or it does not. This is indicated with $I \models \varphi$ or $I \not\models \varphi$, respectively, and inductively defined by the following rules:

- if φ is an atom p, then $I \models \varphi$ if and only if $I(p) = \top$
- if φ is a negation $\neg\psi$, then $I \models \varphi$ if and only if $I \not\models \psi$
- if φ is a conjunction $\psi_1 \wedge \psi_2$, then $I \models \varphi$ if and only if $I \models \psi_1$ and $I \models \psi_2$
- if φ is a disjunction $\psi_1 \vee \psi_2$, then $I \models \varphi$ if and only if $I \models \psi_1$ or $I \models \psi_2$.

A formula φ is *satisfiable* if there exists an interpretation I under which the formula is true; if no such interpretation exists, φ is *unsatisfiable*, or a *contradiction*. Two formulae φ_1 and φ_2 are *equisatisfiable* if they are both satisfiable or unsatisfiable; they are *equivalent*, denoted with $\varphi_1 \equiv \varphi_2$, if for any assignment I, $I \models \varphi_1$ if and only if $I \models \varphi_2$. A formula that evaluates to true under all possible assignments is *valid*, or a *tautology*.

A *canonical form* is a way of representing formulae such that two equivalent formulae will have the same representation. This is useful for software verification and in general any automated method that handles logic formulae.

A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, where a clause is a disjunction of literals (note that CNF is not a canonical form). Every propositional formula can be transformed into an equivalent CNF formula using the well-known rules of logical equivalence, such as De Morgan's laws, but the size of the formula can increase exponentially. However, Tseitin's transformation [Tse68] can convert an arbitrary propositional formula into an equisatisfiable (not equivalent) CNF formula

with only a linear increase in size. In this case the resulting CNF formula will contain more variables than the original one, but a satisfying assignment of variables on the new formula can be converted into a satisfying assignment for the original formula by discarding the assignments for the new variables.

The *decision problem* for a given Boolean formula φ consists in determining whether φ is a tautology. A procedure to solve the decision problem is *sound* if, when it determines that a given input formula φ is a tautology, then φ is a tautology (i.e., if the procedure terminates with an answer, that answer is correct); the procedure is *complete* if, for any input formula φ, (a) it terminates, and (b) if φ is a tautology, then the procedure determines that φ is a tautology (i.e., the procedure terminates on any possible input). Note that there may be sound but incomplete procedures (i.e., procedures that do not terminate or fail to produce an answer on some input), and complete but unsound procedures (i.e., procedures that produce wrong answers). The *boolean satisfiability problem*, also known as *propositional satisfiability problem* and often abbreviated as SAT, consists in determining whether a given propositional formula is satisfiable.

The question of the validity of a formula can be rewritten as to one involving satisfiability. In particular φ is valid if and only if $\neg\varphi$ is unsatisfiable. On the other hand, φ is satisfiable if and only if $\neg\varphi$ is invalid. Satisfiability is one of the most intensively studied problems in computer science, and many well-known problems reduce to checking the satisfiability of a propositional formula.

2.1.3 Propositional Satisfiability

SAT is NP-complete [Coo71], and all known deterministic algorithms to solve it have exponential worst-case complexity. Despite this, the outstanding advancements over the last fifteen years have brought us methods capable of handling formulae of considerable size and complexity. In practice, there are heuristics that allow rather fast solution for many large instances of SAT, both satisfiable and unsatisfiable, from a broad range of real-world applications.

A *SAT solver* is a decision procedure that takes as input a given formula φ (typically in CNF) and returns as output an assignment I of the variables of φ that satisfies φ, or $\bot$ if the φ is unsatisfiable. In this last case, some SAT solvers may additionally provide an unsatisfiable core of clauses or a resolution proof of unsatisfiability.

SAT solvers are mostly based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62]. DPLL solvers follow three main steps: decision, propagation, and backtracking. The decision step chooses an unassigned variable and assigns it a value. In the propagation step, Boolean constraint propagation (also known as BCP) is performed. The implications of the decision on the variable and its value are propagated by applying the unit clause rule. If a clause is a *unit clause*, i.e. contains only a single unassigned

literal p, there is only one possible value of p that can make the clause true. This prunes an otherwise exhaustive search space by avoiding unnecessary decisions. Sometimes it can be done repeatedly, excluding large parts of the search space. In the propagation step, pure literal elimination is also performed. A literal in a CNF formula is *pure* when it occurs only with one polarity, i.e., either negated or not. Pure literals can always be set to a value such that all clauses containing them are true. These clauses can then be removed from the formula, as they no longer constrain the search. Since further literals become pure, this simplification may need to be applied repeatedly in order to obtain a formula without pure literals. Backtracking happens when the propagation generates conflicts. The idea is to flip one of the decisions variables that has been assigned but not yet flipped, marking it as flipped, and then re-applying propagation. The solver terminates if there are no unassigned variables, in which case the formula is satisfiable, or if there are no decision variables to invert, i.e., the formula is unsatisfiable. Plain backtracking tends to be inefficient, and modern solvers implement Conflict-Driven Clause Learning (CDCL), based on different refinements of the basic backtracking algorithm, such as *backjumping* or *clause learning*. These methods identify the reason (i.e., the variable assignments) for the conflict and prevent reaching the same conflict again by restricting the next possible decisions of the solver accordingly. Clause learning uses binary resolution to identify the reason for the conflict, generates the *learnt* conflict cause and appends it to the input formula. Note that since these clauses are implied by the input formula itself, this does not affects satisfiabilty. After learning a new conflict clause the solver progressively invalidates decisions up to a certain point and restarts the whole process.

In order to achieve good efficiency, different SAT solvers rely on many different heuristics to guide their choices, such as the selection of variables and their values, the backtracking depth, BCP clause selection, and so on. In addition, modern solvers use further optimisations that focus on more practical aspects, such as preprocessing (useful for instance to simplify the input formula upfront), fast restarts, and highly efficient data structures. The reader is referred to [Bie09, KS08, GPFW96, ZM02] for further details.

2.1.4 Bounded Model Checking

Bounded model-checking (BMC) is a symbolic technique for program analysis where only subsets of feasible program behaviours are explored. It checks, given a program, a property, and a bound k, if the property (that typically represents the negated form of some error condition) can be violated within k execution steps [BCCZ99, BCC+03, CES09].

BMC efficiently reduces program analysis to propositional satisfiability (described in Section 2.1.3). An initial program unfolding procedure enforces the bound by transforming the input program into a *bounded program* and then simplifies it using an *intermediate representation*; this is in turn compiled into a propositional formula, or *verification condition*, that is satisfiable if and only if there exists an execution of the program

(a) input program

```
x:=x+y;

if(x!=1) then
   x:=2;
else
   x:=x+1;

assert(x<=3);
```

(b) SSA form

```
x1:=x0+y0;

x2:=2;
x3:=x1+1;
x4:=(x1!=1)?x2:x3;

assert(x4<=3);
```

(c) verification condition

$$\begin{aligned} \varphi_C := \quad & x_1 = x_0 + y_0 \wedge \\ & x_2 = 2 \wedge \\ & x_3 = x_1 + 1 \wedge \\ & x_4 = (x_1 \neq 1) ? x_2 : x_3 \\ \\ \varphi_P := \quad & x_4 \leq 3 \end{aligned}$$

FIGURE 2.1: Bounded model-checking: static single assignment (SSA) form and verification condition (VC) (example slightly adapted from [CKL04])

that in at most k steps violates the property. The satisfiability of the formula implies the existence of an error in the initial program, while the absence of detected errors is indecisive, because an error might still occur beyond the given execution bound. Completeness is in fact relinquished for decidability, making the technique mostly suited for finding errors rather than for verifying their absence. This process is outlined in Figure 2.1 and described below in more detail.

Program unfolding (also known as *program bounding* or *program flattening*) includes *loop unrolling* and *function inlining*. Loop unrolling [DAC71, Sar01] replaces each loop statement with k copies of the loop body, each guarded by the loop condition, followed by a *loop unwinding assertion* on the negated loop condition that fails if the given loop bound is not sufficient to fully unfold the loop; backwards jumps also generate loops and therefore are handled likewise. Function inlining [AJ88] replaces each function call with the body of the invoked function, transforming the return statements into an assignment to a newly introduced variable that stores the return value (if any) followed by a jump to the end of the function. Recursive calls are handled similarly to loop unwinding, by asserting in the end that the recursion does not exceed the bound.

In the bounded program resulting from the procedure above there are no loops or function calls, all jumps are forward, and each statement is executed at most once in any feasible execution. Additionally, the transformation from bounded program to an intermediate representation, or static single assignment (SSA) form [CFR+89, LA04], guarantees that each variable is assigned at most once. This is essentially a matter of introducing a new variable to replace the targeted variable within each assignment statement, updating all other occurrences of that variable in the rest of the program accordingly (see Figure 2.1(b)). The intermediate representation of the bounded program is then compiled into a propositional formula (see Figure 2.1(c)) that is satisfiable if and only if there exists an execution of the program that in at most k steps violates the property. The verification condition is eventually analysed by a SAT solver.

An advantage of this approach is in that it exploits the considerable performance gains achieved by modern SAT solvers; another advantage is in the ability to provide a *counterexample*, or *error trace*: 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. The above considerations, and the fact that BMC is fully automatic, make it particularly attractive for industrial applications.

Completeness

Extending BMC to a complete analysis method has been considered of fundamental importance since the inception of the technique [BCCZ99, CES09].

A possible approach is to pre-compute a *completeness threshold* [Bie09] before the actual analysis, so to choose a bound k that is sufficient to cover the entire program's state space. Finding out the exact completeness threshold is computationally very expensive and can be as hard as the model checking problem itself [CKOS05]. Research thus focuses on over-approximation. There are many methods available that depend on the program and the kind of property to prove; these methods 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) [BKA02, KS03, MS03, CKOS04, BK04, Kro06, KOS+11, BOW12]. Nevertheless, the problem of efficiently determining reasonably accurate completeness thresholds in the general case remains still open.

An alternative technique is k-induction, which is essentially a mechanised inductive reasoning [SSS00, ES03, Bra11]. The idea is to use invariants to construct a k-step inductive proof. Different variations have been proposed on this approach [DKR10, DHKR11] and in general require auxiliary invariants to be provided externally, usually through manual source-code annotations. Techniques to automatically generate invariants [AS06, BHMR07, BM08] require additional effort and are not guaranteed to provide invariants powerful enough to imply the correctness of the property.

2.2 Shared-memory Multi-threaded Programs

We describe multi-threaded programs using a simple imperative language. It features dynamic thread creation, thread join, and mutex locking and unlocking operations for thread synchronization. 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.

During the execution of a multi-threaded C program, we can assume that only one thread is *enabled* at any given time. Initially, only the *main thread* is enabled; new threads can

be spawned from any thread by invoking `create`. Once created, a thread is added to the pool of active threads. At a *context switch* the currently enabled thread is suspended and becomes active, and one of the active threads is resumed and becomes the new enabled thread. When a thread is resumed its execution continues either from the point where it was suspended or, if it becomes enabled for the first time, from the beginning.

All threads share the same address space: they can write to or read from global (*shared*) variables of the program to communicate with each other. We assume the *sequential consistency* memory model: when a shared variable is updated its new valuation is immediately visible to all the other threads [Lam79]. We further assume that each statement is atomic. This is not a severe restriction, as it is always possible to decompose a statement in a sequence of statements, each involving at most one shared variable [Mül06]. [1]

2.2.1 Syntax

The syntax of multi-threaded programs is defined by the grammar shown in Figure 2.2. Terminal symbols are set in typewriter font. Notation $\langle n\ \mathtt{t}\rangle^*$ represents a possibly empty list of non-terminals n that are separated by terminals `t`; x denotes a local variable, y a shared variable, 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, integer 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.

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 sequential, or a concurrent statement (also known as *visible statement*), or a sequence of statements enclosed in braces known as *compound 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, a labelled sequential statement, or a jump to a label. Local variables are considered uninitialised

[1] The restricted grammar considered and the assumptions on sequential consistency and atomicity of statements simplify our presentation, but exclude many subtle circumstances that actually do occur in practice, due to the increasing complexity of language specifications, compiler technologies, hardware designs, and to the intricate interplay between them [Boe05]. For instance, for performance reasons sequential consistency is often violated both at compile time and at the hardware layer (most modern processors do so) due to the reordering of memory operations. There are other sources of trouble especially related to multi-threaded C programs using POSIX threads, such as the complex notion of sequence points in C, and the assumptions of the POSIX thread model that are indeed stronger than the actual C standard. All these aspects should be properly taken into account in order to design industrial-strength verification tools.

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

FIGURE 2.2: Syntax of multi-threaded programs.

right after their declaration, which means that they can take any value from their 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 non-deterministic choice of any possible value of the corresponding type (e.g., `x:=*;`).

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), 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. For the sake of simplicity, we assume that the default value is 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.

2.2.2 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, the local variables and the program counter are restored. Any other statement follows the standard *C-like semantics*.

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 sh is a valuation of the shared variables, $en \in \{i_1, \ldots, i_n\}$ is the identifier of the only thread that is enabled to make a transition, and 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`.

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 $j = en$. 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 and no other transitions can continue from that configuration; in this case c' is said to be an *assertion-failure* configuration. In contrast, an `assume` statement will not allow any further transitions from that thread if its condition does not hold. 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} to make any further transitions. The execution of a thread `unlock` statement on a mutex m, held by t_{en}, allows to free it. 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 $\xrightarrow[P]{}$ to be the union of all relations $\xrightarrow[P]{j}$.

Let P be a multi-threaded program with configurations c and c'. A *run* or *execution* of P from c to c', denoted $c \overset{}{\underset{P}{\rightsquigarrow}} c'$, is any sequence of zero or more transitions $c_0 \xrightarrow[P]{} c_1 \xrightarrow[P]{} \cdots \xrightarrow[P]{} c_n$ where $c = c_0$ and $c' = c_n$. A configuration c' is *reachable* in P, if $c \overset{}{\underset{P}{\rightsquigarrow}} c'$ 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 $c_0 \xrightarrow[P]{t} c_1 \xrightarrow[P]{t} \cdots \xrightarrow[P]{t} c_n$ for some n, where $c = c_0$, $c' = c_n$. A run $c \overset{}{\underset{P}{\rightsquigarrow}} c'$ is *k-context bounded* if it can be obtained

```
int m; int c;

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

void C() {
   assume(c>0);
   c:=c-1;
   assert(c>=0);
}

void main() {
   c:=0;
   init m;
   int p0,p1,c0,c1;
   p0:=create P(5);
   p1:=create P(1);
   c0:=create C();
   c1:=create C();
}
```

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

by concatenating at most k contexts of P, i.e. there exist $c_0, c_1, \ldots, c_{k'}$ 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.

2.2.3 Reachability

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

Example. The program shown in Figure 2.3 contains a reachable assertion failure. It models a producer-consumer system, with two shared variables, a mutex `m` and an integer `c` that stores the number of items that have been produced but not yet consumed.

The `main` function initializes the mutex and spawns two threads executing `P` (*producer*) and two threads executing `C` (*consumer*). Each producer acquires `m`, increments `c`, and terminates by releasing `m`. Each consumer first checks whether there are still elements not yet consumed; if so (i.e., the `assume`-statement on `c` > 0 holds), it decrements `c`, checks the assertion `c` ≥ 0 and terminates. Otherwise it terminates immediately.

The mutex ensures that at any point of the computation at most one producer is operating. However, the assertion can still be violated since there are two consumer threads, whose behaviors can be freely interleaved: with $\mathtt{c} = 1$, both consumers can pass the assumption, so that both decrement `c` and one of them will write the value -1 back to `c`, and thus violate the assertion.

2.3 Bounded Model Checking of Multi-threaded Programs

Multi-threaded programs are formed from sequential programs, or threads, communicating through a shared memory. A computation of such programs is an interleaving of the computations of each thread, and thus can be seen as a sequence of contexts where only one thread is enabled (see Section 2.2). Attempts to extend BMC to the analysis of multi-threaded programs face the problem of state space explosion, as the number of possible interleavings grows exponentially with the number of threads and statements in the bounded program.

There are two main approaches to address this problem: *context bounding* (see Sections 2.3.1 and 2.3.2) limits the analysis to a given number of context switches; *partial-order reduction* (see Section 2.3.3) prunes the search space by avoiding the exploration of multiple executions leading to the same state.

2.3.1 Context-bounded Analysis

Context-bounded analysis (CBA) methods limit the number of context switches they explore and so fit well into the general BMC framework. Their use for under-approximate analysis is empirically justified by work that has shown that errors typically occur within few context switches [QW04, MQ07, TDB14]

CBA can be performed by unfolding the set of running threads up to a given context bound to build a reachability tree and then performing an explicit, depth-first exploration of the different interleavings on the tree. The solver is invoked whenever the last statement in an interleaving is reached. The process stops when a bug if found, or all possible interleavings have been explored [CF11].

2.3.2 Sequentialization

CBA can be also implemented by translating the multi-threaded program into a non-deterministic sequential program that simulates all possible schedules up to the context switch bound. This translation or *sequentialization* idea was proposed by Qadeer and Wu [QW04], with the goal to reuse verification tools originally developed for sequential

programs also to analyse multi-threaded programs. This first schema simply scheduled the threads such that they all use a unique call stack, i.e., at each step the stack can be split into contiguous parts each corresponding to the *whole* stack of an executed thread, and each thread at the top of the stack can be either executed, or suspended, or terminated. However, this limits the maximum number of contexts switches that can be considered (e.g., for two threads only two context switches can be simulated in this schema).

Eager Sequentialization

Lal and Reps subsequently proposed a generalised schema for Boolean programs with a fixed number of threads and a parameterised number of round-robin schedulings [LR09]. The basic idea behind the Lal-Reps schema (LR) is that the sequentialized program simulates all round-robin schedules of the threads in the concurrent program in a fixed order, in such a way that (*i*) each thread is run to completion, and (*ii*) each simulated round works on its own copy of the shared global memory. The first thread eagerly guesses the initial values of all memory copies and the context switch points. At each context switch point it switches over to the memory copy for the next round. Context switches are thus simulated by "memory switches". The subsequent threads follow the same schema, but work with the values of the shared memory copies left by their respective predecessors. After the simulation of the last thread has finished, a *checker* prunes away all initial guesses that do not correspond to feasible computations, i.e., where the values guessed for one round do not match the values computed at the end of the previous round). This requires a second set of memory copies. However, since each thread is simulated to completion, its local variables can be discarded at termination, and the global memory copies serve only as interfaces between the threads. This is known as *eager* sequentialization, as the data non-determinism induces the exploration of unreachable states that are pruned away only at the end of simulation, by the checker.

LR was originally designed for programs where threads are only created at the beginning of the execution. Delay bounded sequentialization overcomes this issue to handle thread-creation [EQR11]. The idea of delay bounded is similar to LR with the difference that threads are transformed into function calls and are simulated at the point they are spawn. Similarly, further extensions allowed modelling of unbounded, dynamic thread creation [BEP11, LMP12], and dynamically linked data structures allocated on the heap [ABQ11]. LR is relatively easy to implement, and has been applied in several tools [CGS11, LQR09, Qad11, LQL12, FIP13a, FIP13b].

Lazy Sequentialization

Since the set of states reachable by a concurrent program can be much smaller than the whole state space, techniques that explore only the reachable states (so-called *lazy* exploration) are often desirable [LR09]. A fixed-point algorithm for the verification of concurrent Boolean programs is given in [LMP09b] and a sequentialization algorithm in [LMP09a]. These have been extended to parametric Boolean programs (i.e., concurrent Boolean programs with unboundedly many threads) in [LMP10, LMP12].

Similarly to LR, the LMP lazy sequentialization schema [LMP09a] also simulates k round-robin schedules of the original concurrent program P, using k copies of the shared memory. In contrast, however, these copies are not guessed, but are computed also for the first thread, and the simulation proceeds round-by-round. Note, that since the call-stack and the program counter of a thread are not stored on context-switches, when a thread is resumed it is necessary to recompute the values of its thread-local variables from the first context. This recomputation of threads is not an actual problem for tools that compute the function summaries since when simulating a thread for a new round, the recomputation is avoided by using the summaries computed in the previous iterations. However, the recomputation seems to be a serious drawback for applying LMP in connection with BMC [GHR10].

2.3.3 Partial-order Reduction

Partial-order reduction (POR) exploits the traditional representation of concurrent systems executions as partial orders [Lam78, Pra86, BF94]. The key observation is that different executions of a multi-threaded program can lead to the same state, hence the explicit enumeration of the thread interleavings potentially explores unnecessary executions. The idea is then to partition the program's executions into equivalence classes, such that (ideally) only one representative execution for each equivalence class is considered during the analysis. This can prune the state space and speed-up the analysis.

Lightweight POR can be achieved via static analysis by conservatively detecting potential collisions between threads [DHRR04, God97, VHB+03]. Specific techniques concentrate either on reasoning about potential thread interferences due to future transitions [Val89], or using information recorded about past computations [God96]. Sophisticated mechanisms have been proposed recently that dynamically detect thread interference, with the goal to improve the precision and thus effectively prune the state space [FG05, GFYS07, KWG09].

Partial-order reduction can be tailored to SAT-based BMC and implemented at the level of the propositional formula. In [AKT13] the approach is to build a propositional formula for each thread following the standard approach for sequential BMC but leaving

global variables unconstrained; the individual formulae are then put in conjunction with an additional formula that encodes the computations as a partial order. This approach has been integrated within the CBMC tool and has been shown to yield a reduction of the formula size over the use of total orders, and, in practice, an improved scalability due to a reduced memory footprint.

Chapter 3

Eager Sequentialization

In this chapter, mostly based on our published paper [FIP13a], we present and evaluate a slightly extended version of the Lal-Reps sequentialization (see Section 2.3.2).

The schema was originally proposed for Boolean programs; here we work out all the details to adapt it to multi-threaded programs with dynamic thread creation.

We evaluate our implementation, LR-CSeq, on multi-threaded C programs using three different bounded model-checkers for the sequential analysis.

3.1 Overview

The Lal-Reps sequentialization schema (LR) [LR09] translates a multi-threaded program into a non-deterministic sequential program that simulates all round-robin schedules of the original program in a fixed order and up to a given round bound.

The idea of the schema is the following:

- each thread is transformed into a thread simulation function;
- all calls to thread routines (e.g. `create`, `join`, `lock`, `unlock`, etc.) are replaced by calls to functions that simulate them;
- the global memory is copied as many times as the number of rounds to simulate, each copy is initialised with non-deterministic values, and accessed to in such a way that each simulated round uses a distinct copy of the memory;
- each thread simulation function is executed exactly once (regardless of the number of rounds to simulate); at non-deterministically chosen context-switch points, the thread simulation function switches over to the memory copy for the next round; and

- memory consistency across the different memory copies is enforced at the end of the simulation, when all executions based on global memory guesses that do not correspond to feasible computations are pruned away.

Essentially, LR replaces the control non-determinism by data non-determinism and simulates context-switches by "memory switches".

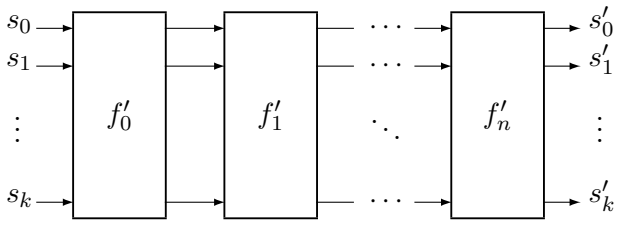

FIGURE 3.1: Propagation of global memory snapshots in the Lal-Reps sequentialization schema: $f'_0, \ldots, f'_n$ represent the thread simulation functions, $s_0, \ldots, s_k$ the global memory snapshot for the first simulated thread, and $s'_0, \ldots, s'_k$ the snapshot at the end of the simulation of the last thread.

More precisely, the schema is defined on a fixed number of threads and a parameterised number of round-robin schedulings. Consider a program P with $n+1$ threads $f_0, \ldots, f_n$, where f_0 is the program's main thread, or `main` function. Let k be the bound on the number of round-robin schedulings to simulate.

The schema translates P into a sequential but non-deterministic program P' that simulates all k round-robin schedules of P's threads in the order $f_0, \ldots, f_n$. P' has $n+2$ functions, $f'_0, \ldots, f'_n, checker$, where each f'_i is obtained by transforming f_i, and $checker$ is the main function of P' that drives the simulation. In addition, the global memory s of the program is duplicated $k+1$ times, $s_0, \ldots, s_k$.

The checker sets $s_0 = s$ (so that the global memory is initially the same as in the original program) and initialises all other memory copies $s_1, \ldots, s_k$ with non-deterministic data. Then it launches $f'_0, \ldots, f'_n$ following the fixed round-robin order.

Let us now consider Figure 3.1. The first thread simulation function, f'_0, starts executing its statements following the same order as in the original program, until non-deterministically it decides to simulate a context-switch. At a context-switch, the thread simulation function switches over to a different copy of the global memory (namely, at context-switch j it switches to s_j), and keeps following the normal order of execution of the statements. The simulation then continues until the next context-switch. This process is repeated up to k times.

The subsequent threads $f'_1, \ldots, f'_n$ follow the same schema, but work with the values of the shared memory copies left by their respective predecessors. Each copy of the memory thus represents the snapshot seen by the thread simulations executing during the corresponding round of a round-robin schedule.

In order to ensure that the thread simulations work on consistent snapshots, before starting the simulation the checker stores the initial global memory in a second copy $s''_1, \ldots, s''_k$, and checks at the end (i.e., after all simulation functions have terminated) that the last thread in each round has ended its simulation with consistent initial guesses for the next round, i.e., $s'_i = s''_{i+1}$ for $i = 0, \ldots, k-1$. This *thread mapping* guarantees that the execution of P' corresponds to an actual execution of P.

LR is possibly the most well-known sequentialization schema and has been implemented in several tools. Rek [CGS11] implements sequentialization for C via code-to-code transformation; it is targeted at real-time systems and hard-codes a specific scheduling policy. STORM [LQR09] extends the LR schema to C programs, by first translating from concurrent C programs to concurrent Boogie programs, and then applying an LR sequentialization integrated into a CEGAR approach. Similarly, Poirot [Qad11] also verifies concurrent C programs via sequentialization; it first translates them into Boogie and then implements the sequentialization transformation at the Boogie level.

In this chapter, we present and evaluate our implementation of LR, slightly modified to handle dynamic thread creation, that works directly on multi-threaded C programs with POSIX threads and is not necessarily tied to a specific back-end for the actual analysis.

3.2 Lal-Reps Sequentialization of multi-threaded Programs

In this section we work out all the details to instantiate the LR schema on multi-threaded programs.

Our implementation assumes that the input program can be divided into three blocks of code: declarations of global variables, function definitions, and `main` function definition. Fig. 3.2 sketches this structure. We assume that the above blocks do not mix and that their order is as shown.

We show how to model the basic thread functionalities: thread creation and join, mutex lock and unlock. In particular, we extend the LR schema so that thread creation statements can be at any point in the input code, therefore dynamic thread creation is now supported as well.

In the rest of the section, we use the definitions introduced in Section 2.2, in a few cases abusing the notation for the sake of simplicity. In particular, we use pointer to functions without explicit notation (following the simplified call-by-reference semantics already used for all other ordinary variables).

```
type_g1 g1;  type_g2 g2;  ...

void f_n() {
   type_x1 x1;  type_x2 x2;  ...
   stmt_1;
   stmt_2;
   ...
}

...

void f_0() {
   ...
}
```

(a) multi-threaded program

```
type_g1 g1[K];  type_g2 g2[K];  ...                          (i)

void f'_n() {
   type_x1 x1;  type_x2 x2;  ...
   cs(); if(ret) then return; stmt'_1;                       (ii)
   cs(); if(ret) then return; stmt'_2;
   ...
}

...

void f'_0() {
   ...
}

void main() {
   type_g1 _g1[K];  type_g2 _g2[K];  ...                     (iii)

   for(i:=1;i<K;i:=i+1) do {                                 (iv)
      _g1[i]:=g1[i];
      _g2[i]:=g2[i];
   }

   t[0]:=f'_0;                                               (v)
   born[0]:=0;

   for(i:=0;i<N;i:=i+1) do {
      if(born[i]>-1) then {
         ret:=0;
         k:=born[i];
         t[i]();
      }
   }

   for(i:=0;i<K-1;i:=i+1) do {                               (vi)
      assume(_g1[i+1]=g1[i]);
      assume(_g2[i+1]=g2[i]);
      ...
   }

   assert(err=0);                                            (vii)
}
```

(b) sequentialized program

FIGURE 3.2: Structure of original (a) and translated program (b). Function f'_n is the sequentialized version of function f_n. Functions f_0 and f'_0 denote, respectively, the main function of the concurrent program and the corresponding sequentialized function. Function `main` is the *checker*. The auxiliary data structures, the context-switch simulation, and the simulation functions for thread routines are omitted (see Section 3.2.1, Figure 3.3, and Figure 3.4, respectively).

3.2.1 Auxiliary Data Structures

Let `N` = $n+1$ be the maximum number of threads in P and `K` = k the number of round-robin schedules to simulate. We use the following data structures to simulate concurrency:

- `int k` and `int ct` keep track of the index of the current round being simulated and of the currently running thread, respectively.
- `int ret` is set to a value different than `0` to force the termination of the current thread.

- `bool error` is set whenever an error is found and checked after thread-wrapping.
- `int born[N]` stores the round number where a thread is created (or `-1` if the thread has not been created yet).
- `int status[K][N]` maintains the thread statuses (either `RUNNING` or `TERMINATED`) for all rounds.
- `thread[N]()` is an array of (pointers to) thread functions used as argument to `create` to spawn new threads.

3.2.2 Non-deterministic State Replication

In our implementation, we replace each global variable `g` by a `k`-indexed entry `g[k]` in an array of size `K`, where `k` is is the current round counter and `K = k` is the round bound (see Fig. 3.2 (*i*)); we use the notation $stmt'$ to denote the statement resulting from this replacement (e.g., Fig. 3.2 (*ii*)).

For each global variable `g` we also also keep a second copy `_g[]` (see Fig. 3.2 (*iii*)) that contains the guesses that the first thread uses in each round; note that only the guesses for the second and subsequent rounds are copied into the first copy, to prevent overwriting the initializations done by the original program (see Fig. 3.2 (*iv*)).

3.2.3 Thread Simulation

All thread-specific statements are mapped into function calls to the corresponding simulation functions (see Figure 3.4).

We model the status of each thread and each lock as an integer variable. `lr_create` (which simulates the thread creation function, `create`) simply inserts a pointer to the thread function into the array `threads`, and records the round in which the thread was created in the array `born`; the pointer is then used later on to start the simulation of the threads in the round stored in `born`.

The program's main function, f_0, is handled as a thread created in round 0 (f_0' in Fig. 3.2) and its pointer is inserted as the first item in the array of threads, to start the simulation (see Fig. 3.2 (*v*)). `lr_join` uses an `assume` statement on the thread status to prune away simulations in which the thread has yet not terminated. The mutex lock and unlock operations similarly set and check the lock variable.

```
void cs() {
        int j;
        assume(j>=0);
        assume(k+j<K);
        k:=k+j;
        if (k=K-1 ∧ *) then ret:=1;
}
```

FIGURE 3.3: Context-switch simulation (Lal-Reps schema).

```
void lr_create(int id1, t()) {
        assume(ct<T);
        ct:=ct+1;
        born[ct]:=cr;
        thread[ct]:=t;
        status[cr][ct]:=RUNNING;
        id1:=ct;
}

void lr_join(int tid) {
        assume(status[cr][tid]=FINISHED);
}
```

```
void lr_init(int m) {
        m:=FREE;
}

void lr_destroy(int m)
        m:=DESTROY;
}

void lr_lock(int m) {
        if (lock=0) then {
                lock:=ct+1;
        } else {
                ret:=1;
        }
}

void lr_unlock(int lock) {
        assume(lock=ct+1);
        lock:=0;
}
```

FIGURE 3.4: Thread simulation stubs (Lal-Reps schema).

3.2.4 Context Switch

The sequentialized program simulates the threads in the order in which they are created via `lr_create`. It simulates a context switch by non-deterministically increasing `k` up to the round bound. If `k` reaches the bound, non-deterministically an early exit can be enforced (i.e., the thread is forced to exit and never gets to run again). We insert this simulation code (as shown in Fig. 3.3) at all sequence points of the original program threads (see Fig. 3.2 (*ii*)). Early thread exit is enforced by setting the control variable `ret`.

3.2.5 Consistency Check

The first simulated thread, in each round, accesses a fresh copy of the memory with non-deterministically chosen values, while the subsequent threads continue with the state left by their predecessor. The initial guesses are stored in `_g[]`; at the end of the simulation (see Fig. 3.2 (*vi*)) we check that each round has ended with the guesses

```
void lr_assert(int expr) {
       if (!expr) then { ret:=1; error:=1; }
}

void lr_assume(int expr) {
       if (!expr) then { ret:=1; }
}
```

FIGURE 3.5: Modelling `assert` and `assume` statements (Lal-Reps schema).

that are used in the next round; simulations that do not satisfy this condition do not correspond to feasible runs, and are discarded.

3.2.6 Error Detection

Since infeasible runs are only discarded at the end, in order to prevent false results, errors can only be reported after the checker has run. In particular, assertion checking must be integrated with the sequentialization (see Figure 3.5). We thus replace all `assert` statements by conditionals that set an error variable `error` and exit from the thread. The error variable is then checked at the end of the simulation (see Fig. 3.2 (*vii*)). `assume` statements require a similar handling in that they force the current thread to exit in case the assumption does not hold, but the `error` variable is not set.

3.3 Evaluation

We have implemented the LR schema for multi-threaded C programs using POSIX threads in our prototype tool LR-CSeq. We have evaluated LR-CSeq over 24 test cases taken from the `pthread_atomic` and `pthread` sections of the `Concurrency` category of the SV-COMP 2013 Software Verification Competition [Bey13], with a total of approximately 2.2k lines of code. The 10 benchmarks that end on `_unsafe` contain an error condition that we encoded as `assert(0)`. We used LR-CSeq to translate the benchmarks into all supported formats and then used CBMC (v4.5), ESBMC (v1.22), and LLBMC (v2012.2a) to verify the translated programs.

We did a few preliminary tests using abstraction-based backends (such as SATABS [CKSY05] or CPAchecker [BK11]) where we could have achieved context-bounded analysis without bounding of the program. However due to either restrictions on the input programs or poor performance, none of the considered backends worked on the sequentialized files.

We also evaluated the native concurrency handling of CBMC [AKT13] and of Threader (c0.92) [PR13], the fastest verifier in the Concurrency category at the SV-COMP 2013 Software Verification competition [Bey13]. Here we ran CBMC with the same setting for

TABLE 3.1: Comparison of sequentialization and native concurrency handling. * - program rejected. † - internal error.

	Sequentialized version					Concurrent version	
	n	k	u	CBMC	ESBMC	CBMC	Threader
dekker_safe	2	3	5	4.7	*4.2*	**0.5**	**0.5**
lamport_safe	2	3	5	52.0	*23.0*	**7.1**	63.8
peterson_safe	2	3	5	*0.6*	0.8	**0.3**	7.4
rw_lock_safe	4	2	5	*1.6*	2.8	**0.6**	1.8
rw_lock_unsafe	4	2	5	-†	*4.5*	**0.4**	2.6
scull_safe	-	-	5	-†	-†	**1.5**	171.2
szymanski_safe	2	3	5	*0.9*	1.1	**0.6**	21.3
time_var_mutex_safe	2	3	5	*1.0*	2.1	**0.7**	7.2
fib_longer_safe	2	7	7	*23.4*	TO	18.1	**11.2**
fib_longer_unsafe	2	7	7	*7.2*	TO	**3.0**	10.1
fib_safe	2	6	6	**6.6**	65.2	6.8	7.7
fib_unsafe	2	6	6	*6.3*	45.8	**0.5**	6.8
indexer_safe	-	-	130	-†	-†	TO	**0.7**
lazy_unsafe	3	2	7	*0.9*	1.8	**0.5**	0.7
queue_safe	2	2	5	144.5	**10.1**	71.6	-†
queue_unsafe	2	2	5	*249.2*	TO	**86.0**	-†
reorder_2_unsafe	-	-	8	-†	-†	6.2	**2.4**
reorder_5_unsafe	-	-	8	-†	-†	6.5	**3.5**
stack_safe	2	2	5	**8.7**	TO	64.7	TO
stack_unsafe	2	2	5	*9.2*	TO	**3.7**	144.4
stateful_safe	2	2	5	0.8	**0.7**	0.9	3.9
stateful_unsafe	2	2	5	**0.7**	1.1	**0.7**	0.8
sync_safe	2	2	5	4.5	**1.4**	4.9	2.5
twostage_3_unsafe	-	-	5	-*	-*	28.6	**24.1**

the context switch bound and Threader for complete analysis. All the tests were made on an otherwise idle Gentoo Linux standard PC with 12GB of memory and an Intel Xeon CPU with 2.67GHz. The timeout was set to 400s.

Table 3.1 summarizes the results. Here n and k denote the number of threads and rounds, respectively, used for the sequentialization translation, and u denotes the unwinding bound for bounded model-checking (not used by Threader). Times are given in seconds; for the sequentialized versions they also include LR-CSeq's runtime, which is generally neglible (approx. 0.1secs). *TO* denotes timeout. The time of the fastest tool for each benchmark is shown in bold; the time of the fastest LR-CSeq backend is shown in cursive.

Note that LLBMC uses a more precise memory model and a more accurate syntactic checks than the other tools. It resulted sensibly slower than CBMC and ESBMC, timing out on many instances, and reporting a memory error on the sequentialized version of `stack_safe` (we do not know whether this error is spurious or not). Therefore LLBMC running times on the sequentialized files are not shown in the table.

LR-CSeq failed to translate five benchmarks due to the restrictions already mentioned: on the first file due to non-standard include files, on the last file because of dynamic

memory allocation (the tool finds a call to `malloc` and rejects the file) and on the other files because the passing of parameters to the main function is not supported at the moment. Threader fails on both the versions for the `queue` test cases.

Overall, the native concurrency handling is faster than sequentialization, but the time difference is generally reasonably small; moreover, for some benchmarks LR-CSeq even outperforms the native concurrency handling. Within LR-CSeq, CBMC slightly outperforms ESBMC as backend, but again the differences are small, and may be caused by the fact that we have mainly used CBMC for testing during development, and as a result the code generated from LR-CSeq is now somewhat optimized for that specific backend.

For an evaluation of this schema on a larger benchmark suite (and in particular for a comparison against our novel schema proposed in Chapter 4), see Section 4.6.2.

3.4 Conclusions

Our prototype has several limitations, mostly due to the strong assumptions on the input that we adopted to simplify the implementation. For instance, we assume that the declarations for the global variables precede those for all functions, that there are no static variables and no global multi-dimensional arrays, and that local variables cannot shadow global variables. These limitations did not significantly affect the evaluation of our prototype tool, due to the simplicity of the test cases.

Another limitation of our prototype is that the counterexample provided by the backend to the wrapper script refers to the sequentialized code and is not translated back to the original input code. However, this can be done by mapping back line numbers, reverting the state replication, and rearranging the order of the statuses in the counterexample, shuffled by non-determinism. This would take a negligible computational effort and would hardly have any impact on the overall analysis performance.

The above implementation-specific limitations could be removed with some engineering effort, and we observed encouraging results on a few non-trivial test cases. However, the inherent limitations to the LR schema remain. In the rest of this section we briefly discuss the main problems that motivated us to develop a newer schema.

A limitation of LR is in the fact that it only works for round-robin thread schedules assuming a fixed thread ordering. Note that, since empty execution contexts are allowed (i.e., threads can be pre-empted without executing any computation) other (shorter) schedules that can fit in a round-robin schedule are captured. For example, with three threads T_1, T_2, T_3 and a fixed ordering $\rho = 1, 2, 3$, a round bound $k = 2$ is sufficient to capture the scheduling $\langle T_2, T_1, T_3 \rangle$, but not enough for the scheduling $\langle T_3, T_2, T_1 \rangle$, which requires one more round. Naively, context-bounded simulation up to a given bound can be achieved by setting the round bound to that bound. However, in this way the

simulation will also unduly include other schedules that actually exceed the context bound. It is still not clear how to achieve a more efficient context-bounded analysis, that does not consider schedules beyond a given bound.

Eager guessing inevitably limits the integration with the applied backend verification tool. For instance, implicit safety properties, such as array bounds violations or invalid pointer dereferences that are handled by the backend, must be translated into explicit assertions, and their detection by the backend must be explicitly suppressed, in order to prevent false results due to eager guessing. While adding further checks to our prototype is certainly possible, it would add overhead to the translation and therefore negatively affect the performance of the backend.

Another problem is heap-allocated memory. Since this memory is accessible to all threads, it needs to be treated similarly to global variables. Intuitively, the checker would have to guess and carry around memory blocks of possibly variable size. This would require an explicit modelling of the global memory and therefore would add even more non-determinism and overhead to the translated program. It is not clear how to do this efficiently without overwhelming the backend. This leaves an open research question that is beyond the scope of this thesis.

Chapter 4

Lazy Sequentialization

In this chapter we present our novel sequentialization schema for efficient bug finding, which is based on the idea of lazy analysis and is specifically targeted to bounded model-checking.

We introduce our sequentialization schema in Section 4.1. We define bounded multi-threaded programs in Section 4.2 and outline the schema in Section 4.3. We give a detailed description with an informal correctness argument in Section 4.4, a formal description based on rewrite rules in Section 4.4.5, and a formal proof of correctness in Section 4.4.6. We discuss two variations on the original schema in Section 4.5. We summarise the experimental evaluation in Section 4.6, and finally give our conclusions in Section 4.7.

The content of this chapter is largely based on our published work [ITF+14a, INF+15].

4.1 Introduction

Sequentialization is a technique to re-use on concurrent software existing tools for the analysis of sequential software [QW04]. It can be implemented as a code-to-code translation of the input program into a corresponding nondeterministic sequential program, and the tool for analysis of sequential software is used as a backend (see Section 2.3.2). Such translations alter the original program structure by injecting control code that is an overhead for the backend. Therefore, the design of well-performing tools under this approach requires careful attention to the details of the translation.

The LR sequentialization schema [LR09] evaluated in Chapter 3 uses a large number of extra variables; the number of assignments involved in handling these variables, the high degree of nondeterminism, and the late pruning of infeasible runs can all negatively impact the performance of the backend tool. Moreover, due to the eager exploration, LR

cannot rely on built-in error checks of the backend and also requires specific techniques to handle programs with heap-allocated memory [LQR09].

Since the set of reachable states of a concurrent program can be much smaller than the whole state space explored by LR, *lazy* techniques that explore only the reachable states can be much more efficient. For instance, an alternative schema uses, like LR, several copies of the shared memory but rather than guessing values, it computes them precisely [LMP09a]. However, in the schema from [LMP09a] since the local state of a thread is not stored on context switches, the values of the thread-local variables must be recomputed from scratch when a thread is resumed. This re-computation poses no problem for tools that use function summarisation because they can re-use the summaries from previous rounds [LMP09a, LMP10], but it is a serious drawback when applying the schema on the top of BMC: each recomputation causes a duplication of the formula corresponding to the thread, and this causes an exponential blow-up in the size of the verification condition [GHR10]. It was thus an open question whether it is possible to design an effective lazy sequentialization for BMC-based backends.

In this chapter, we answer this question and design a new, surprisingly simple but effective lazy sequentialization schema that works well in combination with bounded model-checkers.

Typically, a sequentialization schema is not conceived for any particular underlying technology and performs the source transformation upfront in the whole analysis process. This is the case of the LR schema evaluated in Chapter 3 as well. In contrast, our schema specifically targets bounded model-checking, and in particular the source transformation takes place between the program bounding procedure and the generation of the verification condition (see Section 2.1.4). Working directly on bounded programs allows us to aggressively exploit their structure. The translation is in fact carefully designed to introduce very small memory overheads and very few sources of nondeterminism, so that it produces simple formulae, and is thus very effective in practice. In contrast to LR, only reachable states of the input program are explored, and thus the translation requires no built-in error checks nor any special dynamic memory allocation handling, but can rely on the backend for these. The resulting sequentialized program simulates all bounded executions of the original program for a bounded number of rounds, but avoids their re-computation and thus the exponentially growing formula sizes observed above [GHR10]. The formula size is instead proportional to the product of the size of the original program, the number of threads and the number of rounds.

We have used our CSeq sequentialization framework (see Chapter 5) to implement our lazy schema on sequentially-consistent multi-threaded C programs with POSIX threads [ISO11, ISO09]. The prototype tool, Lazy-CSeq (see Section 5.5), implements both bounding and sequentialization as source-to-source translations and supports the full C language and the main parts of the POSIX thread API, such as dynamic thread

creation and deletion, and synchronisation via thread join and locks. The resulting sequential C program can be analysed with *any* existing verification tool for sequential C programs.

We have tested Lazy-CSeq using BLITZ [CDS13], CBMC [CKL04], ESBMC [CFM12], and LLBMC [MFS12] as backends. We have evaluated our approach and tool over the SV-COMP benchmark suite [Bey14, Bey15]. Lazy-CSeq [ITF+14a] won the concurrency category at SV-COMP 2014 and SV-COMP 2015. The positive results thus justify the general sequentialization approach, and in contrast to the findings by Ghafari et al. [GHR10], also demonstrate that a lazy translation can be more suitable for use in BMC than the more commonly applied LR translation [LR09, EQR11], as Lazy-CSeq also significantly outperforms our own LR-CSeq tool described in Chapter 3.

4.2 Bounded multi-threaded Programs

Bounded multi-threaded programs represent the starting point for our sequentialization schema. Intuitively, bounded multi-threaded programs are multi-threaded programs (see Section 2.2) in which there are no loops, and all the functions, except for the main function, are never called explicitly but passed as arguments to the thread creation routine to spawn new threads. The bounded version of any multi-threaded program can be obtained by applying the program unfolding procedure described in Section 2.1.4 and, if needed, by duplicating the functions definitions, such that multiple threads spawned from the same function (if any) use distinct copies of that function (see Figure 4.2(a)).

$$
\begin{array}{lcl}
P & ::= & (dec\texttt{;})^* \ (\texttt{void}\ f_i\ (\langle dec\texttt{,}\rangle^*)\ \texttt{\{}(\texttt{static}\ dec\texttt{;})^* stm\texttt{\}})_{i=0,\ldots,n} \\
dec & ::= & type\ z \\
type & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{void} \\
stm & ::= & seq \mid conc \mid \texttt{\{}\langle stm\texttt{;}\rangle^*\texttt{\}} \\
seq & ::= & \texttt{assume}(b) \mid \texttt{assert}(b) \mid x := e \mid \texttt{return}\ e \\
 & & \mid \texttt{if}(b)\ \texttt{then}\ stm_1\ \texttt{else}\ stm_2 \mid l\texttt{:}\ seq \mid \texttt{goto}\ l \\
conc & ::= & x := y \mid y := x \mid t := \texttt{create}\ f_i(e) \mid \texttt{join}\ t \\
 & & \mid \texttt{init}\ m \mid \ \texttt{lock}\ m \mid \texttt{unlock}\ m \mid \texttt{destroy}\ m \mid l\texttt{:}\ conc
\end{array}
$$

FIGURE 4.1: Syntax of bounded multi-threaded programs.

More precisely, *bounded multi-threaded programs* are defined by the syntax given in Figure 4.1. Note that for effect of program bounding there are no loops in these programs, so the additional property not captured by the given syntax is that all `goto` statements

are forward-only. In our definition thread-local variables use the `static` storage class. We used this convention to make our presentation simpler. Static variables are in practice equivalent to local variables, except that uninitialised local variables contain undefined values, while static variables are initialised to 0 by default. Thus, after the declaration of these variables we assign them with a nondeterministic value. For instance, `int tmp` is turned into `static int tmp:=*`. This directly applies to all primitive types and can be done at the level of the components for programming languages that have arrays and structured types.

In addition, we restrict our attention to bounded multi-threaded programs that satisfy the following assumptions:

- *(functions)* every function, which we refer to as *thread function*, is used exactly once to spawn a new thread,
- *(arguments)* every function has at most one argument, the type of which is `int`,
- *(main)* there exists a thread function f_0 corresponding to the `main` thread,
- *(exits)* the `return` statement occurs exactly once in each function, as the last statement of that function,
- *(labels)* in all functions there are numerical labels in increasing order starting from `0`, immediately before the first statement, every visible statement, and the last statement; any other label of the program is non-numerical.

In addition, for the sake of conciseness, we adopt the usual square-bracket notation commonly used for arrays to indicate elements of fixed-sized sets of scalar variables. Note that all of the above assumptions can be enforced in any bounded multi-threaded program by simple source transformations.

4.3 Overview

Our translation transforms a bounded multi-threaded program into a bounded sequential program that simulates round-robin schedules of the initial program up to a fixed number of rounds. The basic idea is the following:

- each thread is transformed into a thread simulation function;
- all calls to thread routines (e.g. `create`, `join`, `lock`, `unlock`, etc.) are replaced by calls to functions that simulate them;

```
int m; int c;

void f1(int p) {
   0: static int l; l:=p;
   1: lock m;
   2: if(c>0) then
   3:    c:=c+1;
      else {
   4:    c:=0;
         if(!(l>0)) then
            goto _l1;
   5:    c:=c+1;
         l:=l-1;
         if(!(l>0)) then
            goto _l1;
   6:    c:=c+1;
         l:=l-1;
         assume(!(l>0));
         _l1:
      }
   7: unlock m;
   8: return;
}

void f2(int p) {...}

void f3() {
   0: assume(c>0);
   1: c:=c-1;
      assert(c>=0);
   2: return;
}

void f4() {...}

void f0() {
   0: c:=0;
   1: init m;
      int p0,p1,c0,c1;
   2: p0:=create f1(5);
   3: p1:=create f2(1);
   4: c0:=create f3(0);
   5: c1:=create f4(0);
   6: return;
}
```

(a) bounded multi-threaded program

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

void f1^seq(int p) {
   0:J(0,1) static int l; l:=p;
   1:J(1,2) seq_lock(m);
   2:J(2,3) if(c>0) then
   3:J(3,4)    c:=c+1;
            else { G(4);
   4:J(4,5)    c:=0;
               if(!(l>0)) then
                  goto _l1;
   5:J(5,6)    c:=c+1;
               l:=l-1;
               if(!(l>0)) then
                   goto _l1;
   6:J(6,7)    c:=c+1;
               l:=l-1;
               assume(!(l>0));
              _l1:  G(7);
            } G(7);
   7:J(7,8) seq_unlock(m);
   8:       return;
}

void f2^seq(int b) {...}

void f3^seq() {
   0:J(0,1) assume(c>0);
   1:J(1,2) c:=c-1;
            assert(c>=0);
   2:       return;
}

void f4^seq() {...}

void f0^seq() {
   0:J(0,1) c:=0;
   1:J(1,2) seq_init(m);
            static int p0,p1,c0,c1;
   2:J(2,3) p0:=1; seq_create(5,1);
   3:J(3,4) p1:=2; seq_create(1,2);
   4:J(4,5) c0:=3; seq_create(0,3);
   5:J(5,6) c1:=4; seq_create(0,4);
   6:       return;
}

void main() {...see Fig. 4.3...}
```

(b) sequentialized program

FIGURE 4.2: Lazy sequentialization example. The program on the left is the bounded program resulting from applying the method described in Sec. 2.1.4 to the multi-threaded program from Figure 2.3. Note that some statements are preceded by a numerical label, following the assumptions from Sec. 4.2. The program on the right is the sequentialized program. The code injected by the source transformation is gray. For practical reasons we abuse the notation and use C-style syntax to define the macros `G()` and `J()`.

- for each round, the thread simulation functions are called in a fixed order and non-deterministically they can exit at any visible statement to simulate a context switch; we maintain the program location of the (simulated) context switch where the computation must resume from in the next round;
- on thread resuming, the control jumps back to the locations stored as above, then executes, again, a non-deterministically selected number of steps, and jumps out of the function;
- the local variables of all threads are persistent, so that the simulation does not need to recompute them.

Note that the above mechanism only works because the original program is bounded, so that (*i*) there is a bounded number of activations for each function, and (*ii*) we can associate unique identifiers as jump targets with each statement of the sequential program.

Figure 4.2(b) shows the resulting sequentialized program for the Producer-Consumer example (cf. Figure 2.3), with an unwinding bound of 2. The parts in black correspond to the unwound original program, those in light gray are injected to achieve the wished sequentialization, as described in Section 4.4. Note that in the bounded program we get two separate copies of each of the functions P and C, since the original program spawns two producer and two consumer threads.

4.4 Lazy Sequentialization for Bounded Programs

We now describe our code-to-code translation from a bounded multi-threaded program P to a sequential program P_k^{seq} that simulates all round-robin executions with $k > 0$ rounds of P.

Assume that P consists of $n + 1$ functions $f_0, \ldots, f_n$, where f_0 denotes the unwound `main` function. By definition, P contains n calls to `create`, which spawn (at most) n threads using as start functions $f_1, \ldots, f_n$, respectively. Each start function is associated with at most one thread, so that we can identify threads and functions.

For round-robin executions, we fix an arbitrary schedule ρ by permuting $f_0, \ldots, f_n$. For any fixed ρ, our translation guarantees that P fails an assertion in a k round-robin execution if and only if P_k^{seq} fails the *same* assertion. Moreover, the translation preserves not only bounded reachability, but allows us to perform on the bounded multi-threaded program all the analyses that are supported by the sequential backend tool.

P_k^{seq} is composed of a new function `main` and a thread simulation function f_i^{seq} for each thread f_i in P. The new `main` of P_k^{seq} calls, in the order given by ρ, the functions f_i^{seq}

for k complete rounds. For each thread it maintains the numerical label at which the context switch was simulated in the previous round and where the computation must thus resume in the current round.

Each f_i^{seq} is essentially f_i with few lines of additional control code (note that we have assumed the existence of numerical labels in f_i to denote the relevant context switch points in the original code). When executed, each f_i^{seq} jumps (in multiple hops) to the saved position in the code and then restarts its execution until the label of the next context switch is reached. Since the local variables are made persistent (i.e., of storage class `static`) we do not need to re-compute them when resuming suspended executions.

We now describe our translation in a top-down fashion. We also convey an informal correctness argument as we go along. A formal proof of correctness of the sequentialization schema is provided in Section 4.4.6. We start by describing the (global) auxiliary variables used in the translation in Section 4.4.1. Then, we give the details of function `main` of P_k^{seq} in Section 4.4.2, and illustrate how to construct each f_i^{seq} from f_i in Section 4.4.3. Finally, we discuss how the thread routines are simulated in Section 4.4.4.

4.4.1 Auxiliary Data Structures

During the simulation of P, the sequentialized program P_k^{seq} maintains the following data structures.

Let `N` be a symbolic constant denoting the maximal number of threads in the program, i.e., $n + 1$.

- `bool active[N]` tracks whether a thread is active, i.e., has been created but not yet terminated. Initially, only `active[0]` is `true` since f_0^{seq} simulates the `main` function of P;
- `int arg[N]` stores the argument used for thread creation (recall that for simplicity we have assumed an implicit call-by-reference semantics in Section 2.2);
- `int size[N]` stores the largest numerical label for each thread simulation function;
- `int pc[N]` stores the label of the last context switch point for each thread simulation function;
- `int ct` tracks the index of the thread currently under simulation;
- `int cs` contains the (pre-guessed) numerical label at which the next context switch for thread `ct` will happen.

Note that the thread simulation functions f_i^{seq} read but do not write any of the above data structures. `N` and `size[]` are constants computed from the bounded program and

remain unchanged during the simulation. `arg[]` is set by `seq_create` (that simulates `create`) and remains unchanged once set. `active[]` is set by `seq_create` and unset by the main driver as described in the next section. `pc[]`, `ct`, and `cs` are updated by the main driver following the mechanism shown in the next section.

4.4.2 Main Driver

Figure 4.3 shows the new function `main` in P_k^{seq} that drives the simulation. For simplicity, we assumed that the fixed schedule corresponds to the ordering $0, \ldots, n$.

`K` is a symbolic constant that gives the bound on the round-robin schedules to simulate, i.e., `K` evaluates to k.

Each iteration of the loop simulates an entire round of a computation of P. The simulation of each thread $f_{\texttt{ct}}$ invokes the corresponding simulation function $f_{\texttt{ct}}^{seq}$ with the argument `arg[ct]` that was originally used to create the thread. The order in which the functions are called corresponds to the fixed round-robin schedule ρ, here $0, \ldots, n$.

For each active thread the driver thus executes the following steps:

1. nondeterministically guess the label for next context switch and store it in `cs`,
2. check that the value is appropriate,
3. simulate the thread from `pc[ct]` through to `cs`, and
4. store `cs` in `pc[ct]`, since in the next round the computation must restart from this label.

The choice of an appropriate value for `cs` is simplified by the structure of P, more precisely, by the fact that the control flow always moves forward because all jumps are forward. We can thus pick any value for `cs` that is between the value stored in `pc[ct]` (corresponding to the case that the thread will not make any progress, hence skips the round) and the largest label in $f_{\texttt{ct}}^{seq}$ that is added in the translation (which corresponds to the last possible context switch point in the code of the corresponding thread $f_{\texttt{ct}}$). We stress that this guess is the only source of nondeterminism introduced by our translation.

4.4.3 Thread Translation

In our schema, each function f_i representing a thread in P is converted into a thread simulation function f_i^{seq} in P_k^{seq} that is obtained as follows.

```
void main() {
       round:=0;

       while(round<K) do {
              ct:=0;
              if (active[ct]) then {
                     cs:=pc[ct] + *;
                     assume(pc[ct]<=cs<=size[ct]);
                     f_0^seq(arg[ct]);
                     pc[ct]:=cs;
              }

              ...

              ct:=n;
              if (active[ct]) then {
                     cs:=pc[ct] + *;
                     assume(pc[ct]<=cs<=size[ct]);
                     f_n^seq(arg[ct]);
                     pc[ct]:=cs;
              }

              round:=round+1;
       }
}
```

FIGURE 4.3: P_k^{seq}: main driver.

Persistence of Thread Local Storage

Each thread f_i in P is simulated in P_k^{seq} by repeated calls to f_i^{seq}; each invocation executes a fragment of the code according to the context switch points that are guessed nondeterministically in the `main` function. Since each thread simulation function is called once in each round, and the thread-local variables are persistent (`static`) between consecutive invocations (because their storage class is `static`), the inefficient re-computation of their values is thus avoided.

Thread Pre-emption and Resuming

When a function f_i^{seq} is called for the first time (i.e., in the first round), it starts its execution from the beginning. In the subsequent calls, it must skip over the statements already executed in previous calls, in order to resume the simulation from its last context switch point. When the control reaches the label guessed for the context switch, it must return without executing any further statements. Different solutions exist to implement this using `goto` statements and distinct labels associated with every meaningful context switch point in the code. We tried to use a multiplexer at the top of the thread's body, implemented with a `switch` and a series of `goto` statements, to jump over the statements already executed, directly to the starting label. We also injected additional code at the

context-switch label to return immediately when the thread is pre-empted. However, this schema has performed poorly in our experiments, possibly because it introduces complex control flow branching.

In contrast, the schema we present here, although at first it may look counterintuitive, actually scales well when used together with BMC backends. We use `goto` statements in a way that avoids complex branching in the control flow. We remark that we use consecutive natural numbers as labels, starting with 0 for the first statement in each function, and label the other statements with numbers increasing in program order (see Figure 4.2). To reduce the nondeterminism, we insert the labels (which are only used to simulate the context switches) only at the first statement, the last statement, and every visible statement (see Section 2.2). Note that this suffices, as we are only interested in assertion violations and in general properties involving only the shared memory and the local state of one thread [Mül06].

Right after each numerical label i (except for the last one) we inject a conditional jump of the form

`if (pc[ct]>`i $\vee$ i `>=cs) then goto` *i+1*`;` (`J` macro)

in front of the statement. Note that the fragment *i+1* is evaluated at translation time, and thus simplifies to an integer literal that also occurs as label. When the thread simulation function tries to execute statements before the context switch of the previous round, or after the guessed context switch, the condition becomes true, and the control jumps to the next label without executing actual statements of the thread. This achieves the positioning of the control at the program counter corresponding to `pc[ct]` with potentially multiple hops, and similarly when the guessed context switch label is reached, the fall-through to the last statement of the thread (which is by assumption always a `return`). Note that, whenever the control is between these two labels, the injected code is immaterial, and the statements of $f_{\mathtt{ct}}^{seq}$ in this part of the code are executed as in the original thread. We use a macro `J` to package up the injected control code (see Figure 4.2(b)).

As an example, consider the sequentialized program in Figure 4.2(b), and assume that f_1^{seq} is called (i.e., `ct`=1) with `pc[1]`=2 and `cs`=6. At label 0, the condition of the injected `if` statement holds true, thus the `goto` statement is executed and the control jumps to label 1. Again, the condition is true, and then the control jumps to label 2. Now, the condition check fails, thus the underlying code is executed, up to label 5. At label 6, the condition of the injected `if`-statement holds again, thus the control jumps to label 7, and then to label 8, thus reaching the `return` statement without executing any other code of the producer thread.

Handling Control-flow Branching

Eager sequentializations such as the Lal-Reps schema (see Chapter 3) need to prune away guesses for the shared variables that lead to infeasible computations. A similar issue arises in our schema for the guesses of context switches. We remark that this is the only source of nondeterminism introduced by our translation.

Consider for example the `if-then-else` in `f1`, as shown in Figure 4.2(b), and assume that `pc[1]`=2 and `cs`=3, i.e., in this round the sequentialized program is assumed to simulate (feasible) control flows between labels 2 and 3. However, if `c`$\leq$0, then the program jumps from label 2 to the `else`-branch right before label 4; if we ignore the `G(4)` macro, the condition in the `if` statement inserted by `J(4,5)` would be tested, and since it would hold, the control flow would slide through to label 8, and return to the `main` driver, which would then set `pc[1]` to 3. In the next round, the computation would then duly resume from this label—which in this execution should be unreachable! Similar problems may occur when the context switch label is in the body of the `else`-branch, and with `goto` statements.

Note that assigning `pc` in the called function rather than in the `main` driver would fix this problem. However, this would require to inject at each possible context-switch point an assignment to `pc` guarded by a nondeterministic choice. This has performed poorly in our experiments. The main reason for this is that the control code is spread "all over" and thus even small increments of its complexity may significantly increase the complexity of the formulae computed by the backend tools. We therefore simply prune away simulations that would store unreachable labels in `pc`. For this, we use a simple guard of the form

```
assume(cs>=j);                                    (G macro)
```

where `j` is the next inserted label in the code. We insert such guards at all control flow locations that are target of an explicit or implicit jump, i.e., right at the beginning of each `else` block, right after the `if` statement, and right after any label in the actual code of the simulated thread (which can be the target of a `goto`-statement of the starting program). Again, we package this up in a macro called `G` (see Figure 4.2(b)).

This solution prunes away all spurious control flows. Consider first the case of `goto` statements. We assume without loss of generality that the statement's execution is feasible in the multi-threaded program and that the target's label `l` is in the code after the planned context switch point. But then the inserted `G` assumption fails, and the simulation is correctly aborted. The argument for `if` statements is more involved but follows the same lines. First consider that the planned context switch is the `then` branch. If the simulation takes the control flow into the `else` branch, then the guard fails because the first label in this branch is guaranteed to be greater than any label in the `then` branch, and the simulation is aborted. In the symmetric scenario, the guard after the

```
void seq_create(int arg, int id) {
       active[id]:=true;
       arg[id]:=arg;
}

int seq_join(int tid) {
       assume(pc[tid]=size[tid]);
}
```

```
void seq_init(int m) {
       m:=FREE;
}

void seq_destroy(int m) {
       m:=DESTROY;
}

void seq_lock(int m) {
       assert(m!=DESTROY);
       assume(m=FREE);
       m:=t;
}

void seq_unlock(int m) {
       assert(m=t);
       m:=FREE;
}
```

FIGURE 4.4: Thread simulation stubs (lazy schema).

`if` statement will do the job because `cs` is guaranteed to be smaller than the next label used as argument in the `G`. Note that the `J` macro at the last context switch point in the `else` branch (in the example `J(6,7)`) jumps over this guard so that it never prunes feasible control flows.

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

4.4.4 Simulation of Thread Routines

For each thread routine we provide a verification stub, i.e., a simple standard function that replaces the original implementation for verification purposes. Figure 4.4 shows the stubs for the routines used in this paper. Spawned threads are simply mapped to integers, which serve as unique thread identifiers; all other relevant information is stored in the auxiliary data structures, as described in Section 4.4.1.

In `seq_create` we simply set the thread's `active` flag and store the argument to be passed (later, from within the main driver) to the thread simulation function. Note that we do not need to store the thread start function, as the `main` driver calls all thread simulation functions explicitly, and that the `seq_create` stub uses an additional integer argument `id` that serves as thread identifier. The `id` values correspond to the order in which the calls occur in the unwound program and are statically added to the `seq_create` calls.

From the semantics of multi-threaded programs, a thread invoking `join(t)` blocks until `t` is terminated. In the simulation a thread is terminated if it has reached the thread's last numerical label, but there is no notion of blocking and unblocking. Instead, the stub `seq_join` uses an `assume` statement with the condition `pc[t]=size[t]` (which checks that the argument thread `t` has reached its last numerical label) to prune away any simulation that corresponds to a blocking join. We can then see that this pruning does not change the reachability of error states. Assume that the joining thread `t` terminates after the invocation of `join(t)`. The invoking thread should be unblocked then but the simulation has already been pruned. However, this execution can be captured by another simulation in which a context switch is simulated right before the execution of the `join`, and the invoking thread is scheduled to run only after thread `t` is terminated, hence avoiding the pruning as above.

For mutexes we need to know whether they are free or already destroyed, or which thread holds them otherwise. We thus model mutexes as integers, and define two constants `FREE` and `DESTROY` that have values different from any possible thread index. When we initialise or destroy a mutex we assign it with the appropriate constant. If we want to lock a variable we assert that it is not destroyed and then check whether it is free before we assign to it the index of the thread that has invoked `mutex_lock`. Similarly to the case of `join`, we block the simulation if the lock is held by another thread. If a thread executes `unlock`, we first assert that the lock is held by the invoking thread and then set it to `FREE`.

4.4.5 Code-to-code Translation

We now formalise the general translation, described in the previous sections, using rewriting rules on the syntax grammar of bounded multi-threaded programs. Let P be a bounded multi-threaded program and $[\![P]\!]_k$ be the sequentialized program for P, where k is the bound on the number of round-robin schedules. The rewrite rules are given in Figure 4.5.

The resulting sequentialized program is formed as follows. We start with the declaration of the auxiliary data structures introduced by the sequentialization (see Section 4.4.1), followed by the declaration of the original global variables, that remains unchanged. All thread functions are first sequentialized (as discussed in Section 4.4.3) and then appended to the program. Then, all simulation procedures for thread routines from Figure 4.4 are inserted. Finally, the main driver shown in Figure 4.3 is appended.

Every call to a thread routine (`create`, `join`, `init`, `lock`, `unlock`, `destroy`) is transformed into a call to the corresponding simulation function.

Branching statements are treated as described in Section 4.4.3. In particular, within every `if` statement we introduce two guards, one appended at the end of the original code

$$
\begin{array}{rcl}
 & & \texttt{bool active[T] = \{1,0,...,0\};} \\
 & & \texttt{int cs,ct;} \\
 & & \texttt{int arg[T],pc[T],size[T];} \\
\left[\!\!\left[\begin{array}{l} (dec;)^* \ (\texttt{void}\ f_i\ (\langle dec,\rangle^*) \\ \{(dec;)^* stm\})_{i=0,\ldots,n} \end{array} \right]\!\!\right] & \stackrel{\text{def}}{=} & \begin{array}{l} (dec;)^*\ (\texttt{void seq\_}f_i\ (\langle dec,\rangle^*) \\ \{(\texttt{static}\ dec;)^* [\![stm]\!]\})_{i=0,\ldots,n} \\ \texttt{seq\_create(int i, int arg)\{...\}} \\ \texttt{seq\_join(int m)\{...\}} \\ \texttt{seq\_init(int m)\{...\}}\ \ \texttt{seq\_destroy(int m)\{...\}} \\ \texttt{seq\_lock(int m)\{...\}}\ \ \texttt{seq\_unlock(int m)\{...\}} \\ \texttt{main()\{...\}} \end{array} \\
[\![\texttt{assume}(b)]\!] & \stackrel{\text{def}}{=} & \texttt{assume}(b) \\
[\![\texttt{assert}(b)]\!] & \stackrel{\text{def}}{=} & \texttt{assert}(b) \\
[\![x := e]\!] & \stackrel{\text{def}}{=} & x := e \\
[\![\texttt{return}\ e]\!] & \stackrel{\text{def}}{=} & \texttt{return}\ e \\
\left[\!\!\left[\begin{array}{l} \texttt{if}(b)\ \texttt{then}\ stm_1 \\ \texttt{else}\ stm_2 \end{array} \right]\!\!\right] & \stackrel{\text{def}}{=} & \begin{array}{l} \texttt{if}(b)\ \texttt{then}\ [\![stm_1]\!] \\ \texttt{else}\ \{\texttt{G}(\ell'(stm_1));[\![stm_2]\!]\}\,\texttt{G}(\ell'(stm_2));\end{array} \\
[\![l\colon seq]\!] & \stackrel{\text{def}}{=} & \begin{cases} l\colon \texttt{J}(l,l+1);\ [\![seq]\!], & \text{if } l=0, \\ l\colon \texttt{G}(\ell''(seq));\ [\![seq]\!], & \text{otherwise.} \end{cases} \\
[\![\texttt{goto}\ l]\!] & \stackrel{\text{def}}{=} & \texttt{goto}\ l \\
[\![x := y]\!] & \stackrel{\text{def}}{=} & x := y \\
[\![y := x]\!] & \stackrel{\text{def}}{=} & y := x \\
[\![t := \texttt{create}\ f_i(e)]\!] & \stackrel{\text{def}}{=} & \{\, t := i;\ \texttt{seq\_create}(e,i)\,\} \\
[\![\texttt{join}\ t]\!] & \stackrel{\text{def}}{=} & \texttt{seq\_join}(t) \\
[\![\texttt{init}\ m]\!] & \stackrel{\text{def}}{=} & \texttt{seq\_init}(m) \\
[\![\texttt{lock}\ m]\!] & \stackrel{\text{def}}{=} & \texttt{seq\_lock}(m) \\
[\![\texttt{unlock}\ m]\!] & \stackrel{\text{def}}{=} & \texttt{seq\_unlock}(m) \\
[\![\texttt{destroy}\ m]\!] & \stackrel{\text{def}}{=} & \texttt{seq\_destroy}(m) \\
[\![l\colon conc]\!] & \stackrel{\text{def}}{=} & \begin{cases} l\colon \texttt{J}(l,l+1);\ [\![conc]\!], & \text{if } l \text{ is numerical,} \\ l\colon \texttt{G}(\ell''(conc));\ [\![conc]\!], & \text{otherwise.} \end{cases}
\end{array}
$$

FIGURE 4.5: Rewriting rules for the lazy sequentialization.

and one inserted right at the beginning of the `else` block. The guards are implemented by the `G` macro and their arguments are calculated by a function ℓ', which returns the numerical label (plus 1) for a single statement, or the last numerical label (plus 1) in a compound statement. In the absence of numerical labels, ℓ' returns a special value that causes the guards to have no effects. Sequential and concurrent statements preceded by

a non-numerical label are both translated in a similar way, with a `G` guard being inserted between the label and the actual statement in the simulated code. In that case, the argument for the macro is calculated by a function ℓ'' that returns the value of the last numerical label seen before that statement, plus 1. See Figure 4.2(b) for an example of this transformation.

Finally, we guard every visible statement through the macro `J` using as argument the value of the numerical label l that (by definition) occurs immediately before that statement. Any other statement is left as in the original program.

Example 4.1. *Figure 4.2(b) is the result of the translation map $[\![\cdot]\!]$ of Figure 4.5 applied to the bounded multi-threaded program shown Figure 4.2(a).*

4.4.6 Correctness

In this section, we provide a correctness proof for the lazy-sequentialization schema. We show that any k round-robin execution of a bounded multi-threaded program P can be simulated by the sequentialized program $[\![P]\!]_k$, and that every execution of $[\![P]\!]_k$ *embeds* a k round-robin execution of P. Thus, P fails an assertion (that is, reaches an assertion-failure configuration) within the given round-robin bound if and only if $[\![P]\!]_k$ fails the *same* assertion by reaching an *equivalent* configuration.

Henceforth, P denotes a bounded multi-threaded program with at most $n+1$ threads, and k is a bound on the number of round-robin schedules of P. We assume that any potentially blocking operation (i.e., `join` or `lock`) does not block during the execution of P. Note that this assumption does not affect the reachability of error states. In fact, any execution of P containing a `join` from a thread t_1 on a thread t_2 can always be captured by another execution where t_1 is pre-empted immediately before the `join` and is re-scheduled only after t_2 terminates (if at all). For the sake of simplicity, we also assume that all local variables of P and $[\![P]\!]_k$ are transformed into global variables initialised with non-deterministic values at the beginning of the main function; furthermore, we inline all the functions in $[\![P]\!]_k$. We refer to a program obtained by modifying $[\![P]\!]_k$ as above as the *simplified inlining* of $[\![P]\!]_k$ and denote it with S in the rest of this section.

Let us now define how to map program counters from P to S. By definition, each thread function in P is translated into a unique thread simulation function in $[\![P]\!]_k$, and each statement of P is translated to either a single statement, or a block of statements in $[\![P]\!]_k$. Each program counter of P is therefore unequivocally mapped into a program counter of $[\![P]\!]_k$, i.e., the program counter of the statement, or of the first statement of the block, to which the original statement is translated. Note that in the main driver of S each call to a thread simulation function is inlined k times, one for each simulated round. Hence we can define a map, denoted $linemap_{[P,k]}$, from pairs (pc, r), where pc is the program counter of a statement in a thread function of P and r is a round number,

into the program counter of the corresponding statement in the r-th inlined copy of the sequentialized version of that function in the main driver. We also introduce a map $label_P$ that associates to the program counter of a visible statement in P the value of the numerical label preceding it.

We now define a notion of *equivalence* between configurations of P and S. Let V be the common variables of P and S and recall that S has the following auxiliary control variables introduced by the sequentialization: `active[]`, `cs`, `ct`, `arg[]`, `pc[]`, and `size[]`. Intuitively, a configuration c of P and a configuration $\widehat{c}$ of S are equivalent if the valuations of their common variables coincide, and the valuation of the auxiliary control variables in $\widehat{c}$ is consistent with the configuration c, that is: the valuation of `active` is consistent with the threads present in c, the valuation of `ct` identifies the currently enabled thread, the valuation of `pc` matches the numerical labels at the program counters of all threads in c except possibly for the enabled one, and the program counter corresponds to that of the running thread in c.

Definition 4.1 (Equivalent Configurations)**.** Let k be a positive integer, P be a bounded multi-threaded program, $c = \langle sh_P, en_P, th_{i_1}, \ldots, th_{i_\ell} \rangle$ be a configuration of P where $th_i = \langle pc_i \rangle$[1] for any identifier of active threads $i \in \{i_1, \ldots, i_\ell\} \subseteq [0, n]$, and $\widehat{c} = \langle sh_S, pc_S \rangle$ be a configuration of the sequentialized program S.

Let C and $\widehat{C}$ be the sets of configurations of P and S, respectively. For a round number $r \in [1, k]$, we define the binary relation $\equiv_r \subseteq (C \times \widehat{C})$ as follows: $c \equiv_r \widehat{c}$ (i.e., c is equivalent to $\widehat{c}$ w.r.t. r), if the following holds:

1. $sh_P(v) = sh_S(v)$, for every variable $v \in V$;
2. for any $i \in [1, n]$, $sh_S(\texttt{active}[i]) = true$ iff $i \in \{i_1, \ldots, i_\ell\}$;
3. $sh_S(\texttt{ct}) = en_P$;
4. for any $i \in (\{i_1, \ldots, i_\ell\} \setminus \{en_P\})$, pc_i points to a visible statement of P, and $sh_S(\texttt{pc}[i]) = label_P(pc_i)$;
5. $pc_S = linemap_{[P,k]}(pc_{en_P}, r)$. □

Let $\pi = c_0 \underset{P}{\rightarrow} c_1 \underset{P}{\rightarrow} \cdots \underset{P}{\rightarrow} c_m$ be an execution of P. We fix the schedule as the sequence of threads ordered by increasing identifier[2] (*increasing-id schedule*). Let en_P^j be the identifier of the enabled thread in configuration c_j. We define a map that from each prefix of π according to this schedule returns the minimal round-robin bound. Formally,

[1] Since by assumption thread functions do not have function calls or local variables, the configuration of each thread in P only consists of its program counter.

[2] Note that this is exactly the order in which thread functions are called in the main driver of $[\![P]\!]_k$.

$round_\pi : [0, m] \to [1, k]$ is inductively defined as follows:

$$round_\pi(j) = \begin{cases} 1 & \text{if } j \in \{0, 1\}; \\ round_\pi(j-1) + 1 & \text{if } j \in [2, m] \wedge en_P^j < en_P^{j-1}; \\ round_\pi(j-1) & \text{otherwise (i.e., } j \in [2, m] \wedge en_P^j \geq en_P^{j-1}). \end{cases}$$

We are now ready to prove that every k round-robin execution of P (according to the increasing-id schedule) can be simulated by S, hence by $[\![P]\!]_k$. Intuitively, the proof shows that an execution π of P can be simulated by an execution $\widehat{\pi}$ of S. The key observation is that any statement executed in P is simulated by a sequence of one or more statements of S. We split such sequence of statements in two sequences and consider these in two separate parts of the proof. The first sequence corresponds to executing in S exactly the same statement as in P, which leads to an intermediate configuration $\widehat{c}$. For this part we consider separately the case when the statement is not a call to a thread routine, and one individual case for each of the thread routines modelled in our schema. For the second sequence (that starts from $\widehat{c}$) we show that zero or more transitions are performed to correctly position the program counter of S to the next statement to simulate (that corresponds to the next statement executed in P according to π). The proof at this point proceeds by case inspection on the kind of statement. If the statement is non-visible, there are no further steps and the lemma trivially holds. If the statement is visible, the proof proceeds by considering separately the case when a context switch happened in P (and thus needs to be simulated in S) or when it does not.

Lemma 4.2. *Let P be a bounded multi-threaded program, k be a positive integer, and S be the simplified inlining of $[\![P]\!]_k$. For every k round-robin execution*

$$\pi = c_0 \underset{P}{\to} c_1 \underset{P}{\to} \cdots \underset{P}{\to} c_m$$

of P with respect to the increasing-id schedule, there is an execution

$$\widehat{\pi} = \widehat{I} \underset{S}{\rightsquigarrow} \widehat{c_0} \underset{S}{\rightsquigarrow} \widehat{c_1} \underset{S}{\rightsquigarrow} \cdots \underset{S}{\rightsquigarrow} \widehat{c_m}$$

of S such that $c_j \equiv_{r_j} \widehat{c_j}$ with $r_j = round_\pi(j)$, for every $j \in [0, m]$.

Proof. For $j \in [0, m]$, let $c_j = \langle sh_P^j, en_P^j, pc_{t_1}^j, \ldots, pc_{t_\ell}^j \rangle$, and $\widehat{c_j} = \langle sh_S^j, pc_S^j \rangle$.

Furthermore, we define $cs_\pi : [0, m] \to \mathbb{N}$ as follows:

$$cs_\pi(j) = \begin{cases} size[en_m] & \text{if } j = m; \\ label_P(pc_{en_j}^{j+1}) & \text{if } j < m \wedge en_P^j \neq en_P^{j+1}; \\ cs_\pi(j+1) & \text{otherwise.} \end{cases}$$

In other words, $cs_\pi(j)$ is the value of the numerical label at which the thread enabled in configuration c_j will context-switch out. We denote $cs_j = cs_\pi(j)$, for any $j \in [0, m]$.

The proof now proceeds by showing by induction on $j \in [0, m]$ that the following property $\mathcal{P}(j)$ holds:

> There exists an execution of S, $\widehat{\pi}_j = \widehat{I} \underset{S}{\rightsquigarrow} \widehat{c_0} \underset{S}{\rightsquigarrow} \widehat{c_1} \underset{S}{\rightsquigarrow} \cdots \underset{S}{\rightsquigarrow} \widehat{c_j}$ such that $c_i \equiv_{r_i} \widehat{c_i}$ and $sh^i_S(\texttt{cs}) = cs_i$, for every $i \in [0, j]$.

Base case: $j = 0$. We choose the initial configuration $\widehat{I}$ such that the values of global and local variables coincide with those in c_0. Furthermore, the auxiliary variables of the initial configuration of S are by construction initialised as follows: (1) the valuation of `active[0]` is 1 and the valuation of `active[`i`]` is 0 for any $i \in [1, n]$, as the only active thread is the one corresponding to the main procedure of P that has identifier 0; (2) variable `ct` (that keeps track of the identifier of the thread under simulation) is also set to 0; and (3) all the elements of the array `pc` are set to 0, which is the numerical label of the first statement in any thread. Thus, the equivalence of c_0 and $\widehat{I}$ holds for properties 1-4 of Definition 4.1, but property 5 still does not hold, because the program counter of S is positioned at the beginning of the main driver rather than pointing to the beginning of the sequentialized `main` procedure (i.e., the first thread simulation function).

In the main driver of S (see Figure 4.3), we can execute the first `if` statement in the first loop iteration, where `ct` is 0, and pick the transition that sets `cs` to cs_0 (which is always possible as we can choose any nondeterministic value in the range of the thread labels of the `main` procedure of P). Then, we invoke the sequentialized `main` function, where the condition check of the macro `J(0,1)` guarding the first statement fails, thus the control moves to the first statement, s. This configuration is $\widehat{c_0}$. Since the original variables are not affected by these last transitions, $c_0 \equiv_{r_0} \widehat{c_0}$ and $sh^0_S(\texttt{cs}) = cs_0$, that shows the base case.

Inductive step. Now, assuming that $\mathcal{P}(j-1)$ holds, we prove that $\mathcal{P}(j)$ holds. For this, it suffices to prove that $\widehat{c_{j-1}} \underset{S}{\rightsquigarrow} \widehat{c_j}$, $c_j \equiv_{r_j} \widehat{c_j}$, and $sh^j_S(\texttt{cs}) = cs_j$.

Since $\mathcal{P}(j-1)$ holds, from the definition of equivalent configurations we obtain that $pc^{j-1}_S = linemap_{[P,k]}(pc^{j-1}_{en_{j-1}}, r_{j-1})$, which essentially means that both P and S point to the same statement, say $stmt$, in c_{j-1} and $\widehat{c_{j-1}}$, respectively.

We now split the proof into two parts. We first show that by simulating $stmt$ in S (which may require one or more transitions) there exists an intermediate configuration $\widehat{c}$ where all parts of Definition 4.1 hold except possibly for part 5 that concerns the consistence of the program counters. Then, we show that from $\widehat{c}$ we can take zero or more transitions that position the program counter s.t. part 5 of Definition 4.1 holds while retaining the remaining properties. The reached configuration is $\widehat{c_j}$. The proof of both parts is by case inspection.

We start by considering the first part, i.e., from $\widehat{c_{j-1}}$ to $\widehat{c}$. We distinguish between the different kinds of the simulated statement $stmt$:

- *Thread creation and joining.* A thread `create` statement in P adds to c_{j-1} the initial configuration of the newly created thread with a new identifier, say t, and a program counter pointing to the first statement of that thread. By construction, the `create` statement is transformed into a call to `seq_create` in S that sets `active[`t`]` to *true* and `pc[`t`]` to 0.

  A `join` statement in P invoked with thread identifier t normally blocks the invoking thread (i.e., the thread does not make any further transition) until the thread identified by t terminates its execution. At that point, the configuration for the terminated thread is removed from c_j. By assumption, all calls to a concurrency routine in π are not blocking (see discussion at the beginning of the section). In S, the `assume` statement from `seq_join` (see Figure 4.4), discards all executions where `active[`t`]` is set.

  The above reasoning shows that in both the transitions considered properties 2 and 4 of Definition 4.1 hold for $\widehat{c}$ and c_j. Since the rest of the configuration (except for the program counter) remains unchanged, by inductive hypothesis parts 1–4 hold.

- *Lock acquisition and release.* A thread `lock` operation in P on a mutex m suspends the invoking thread until m becomes available, i.e., it is not held by any other thread, and then sets the variable representing m to en_j. Similarly to thread joining, `seq_lock`, which simulates `lock` in S (see Figure 4.4), uses an `assume` statement to discard any execution where the mutex is not free, and then sets m, now a free mutex, to the same value.

  A thread releasing a mutex m using an `unlock` statement in P sets m to a special value indicating that the lock is now free. This is done in S by the corresponding `seq_unlock` function that performs exactly the same assignment.

  In both cases, the transition of S performs the same memory update as P and thus since everything else, except the program counter, is unchanged, by inductive hypothesis again parts 1–4 of Definition 4.1 hold for $\widehat{c}$ and c_j.

- *Remaining statements.* If *stmt* is not a call to a concurrency routine, it can only involve the common variables of P and S. Being c_{j-1} and $\widehat{c_{j-1}}$ equivalent, after executing *stmt* these variables will hold the same values in P and S[3]

  Since everything else except the program counter is unchanged, by inductive hypothesis parts 1–4 of Definition 4.1 hold for $\widehat{c}$ and c_j.

Now we show that indeed from $\widehat{c}$ there are transitions of S leading to a configuration $\widehat{c_j}$ such that $c_j \equiv_{r_j} \widehat{c_j}$ and $sh^j_S(\texttt{cs}) = cs_j$. Note that these transitions do not modify the common variables of P and S. We set pc to $pc^j_{en_{j-1}}$, which is the program counter of the thread that has executed the statement in the last transition. We proceed again by case inspection:

[3]Nondeterministic transitions of P due to the occurrence of the $*$ operator can be matched by transitions in S where $*$ yields the same evaluation as in P.

- *Non-visible statements.* If pc points to a non-visible statement then $en_j = en_{j-1}$ and $pc_S^j = linemap_{[P,k]}(pc, r_j)$. Thus, $c_j \equiv_{r_j} \widehat{c}$ already holds. Moreover, the valuation of `cs` in $\widehat{c_{j-1}}$ is the same as in $\widehat{c}$ that is $cs_j = cs_{j-1}$, therefore we can take $\widehat{c_j} = \widehat{c}$.

- *Visible-statements with no context-switch.* If pc points to a visible statement, then the corresponding statement in S with program counter $linemap_{[P,k]}(pc, r_j)$ is guarded by the macro `J(`$s, s+1$`)` for some numerical label s. Since no context-switch occurs, s cannot be greater than the guessed context-switch point, and thus this macro is immaterial and the control moves to $linemap_{[P,k]}(pc, r_j)$ in S. The reached configuration, that we denote with $\widehat{c_j}$, is such that $c_j \equiv_{r_j} \widehat{c_j}$.

  We recall that cs_j is the label of the statement at which the enabled thread in configuration c_j context-switches out. By inductive hypothesis, `cs` at $\widehat{c_{j-1}}$ evaluates to cs_{j-1}. Since no context switch occurs in the last transition, $cs_{j-1} = cs_j$. Being `cs` not updated in the last transition, the valuation of `cs` at $\widehat{c_j}$ is exactly cs_j.

- *Visible-statements with context-switch.* The remaining case is when pc points to a visible statement in P and a context-switch occurs. Thus, is S, the value of the current numerical label s must be greater than `cs`, so that the macro `J(`$s, s+1$`)` jumps to label $s+1$. By construction, then the control jumps in multiple hops from one label to the next one and then back to the main driver, right after the (inlined) call to the sequentialized function of the thread identified by en_{j-1} in the r_{j-1}-th iteration of the while loop. Thus, variable `pc[`en_{j-1}`]` is set to `cs`, which corresponds to the value of the numerical label of the statement where the context-switch happened. The other entries of the array `pc` remain as in $\widehat{c_{j-1}}$. Now, going through all subsequents thread simulation blocks within the main driver, we enter the first block that corresponds to the thread identified by en_j. Notice that this block is in the r_j-th iteration of the while loop.[4]

  Variable `ct` is then set to en_j, and we can take the transition of S that sets `cs` to c_j (in S we nondeterministically guess this value). Then, the thread simulation function for the enabled thread en_j is entered. Jumping in multiple hops the control is repositioned to the numerical label `pc[`en_j`]`, which by inductive hypothesis corresponds to the numerical label of the last context switch for this thread. Here the macro `J` has no effect and the control is positioned to $linemap_{[P,k]}(pc, r_j)$. The reached configuration is $\widehat{c_j}$ and is such that $c_j \equiv_{r_j} \widehat{c_j}$.

□

Let us now introduce some additional definitions to prove the other direction of the lemma. A configuration $\widehat{c} = \langle sh_S, pc_S \rangle$ of S is *relevant* if there exists a program counter

[4] Note that other intermediate simulation blocks for other active threads may be entered due to the corresponding `active` flag being set, however appropriate non-deterministic choices of `cs` allow to skip the simulation of those threads.

pc of P and a round number $r \in [1,k]$ such that $pc_S = linemap_{[P,k]}(pc, r)$. A *signature* of an execution $\widehat{\pi}$ of S ending with a relevant configuration is the sequence obtained by removing from $\widehat{\pi}$ all the configurations that are not relevant.

We show now that each k round-robin execution $\widehat{\pi}$ of S ending with a relevant configuration can be simulated by an execution of P that matches all the relevant configurations of $\widehat{\pi}$ with equivalent configurations according to Definition 4.1, thus showing the completeness of our approach w.r.t. k round-robin executions.

Lemma 4.3. *Let P be a bounded multi-threaded program, k a positive integer, and S the simplified inlining of $[\![P]\!]_k$. For every execution*

$$\widehat{\pi} = \widehat{I} \underset{S}{\rightsquigarrow} \widehat{c_0} \underset{S}{\rightsquigarrow} \widehat{c_1} \underset{S}{\rightsquigarrow} \cdots \underset{S}{\rightsquigarrow} \widehat{c_m}$$

of S such that $\widehat{c_0}, \widehat{c_1}, \ldots, \widehat{c_m}$ is the signature of $\widehat{\pi}$, there is a k round-robin execution of P

$$\pi = c_0 \underset{P}{\rightarrow} c_1 \underset{P}{\rightarrow} \cdots \underset{P}{\rightarrow} c_m$$

and a non-decreasing sequence of round numbers $(r_0, r_1, \ldots, r_m) \in [k]^{m+1}$ such that $c_j \equiv_{r_j} \widehat{c_j}$ for every $j \in [0,m]$.

Proof. For $j \in [0,m]$, let $c_j = \langle sh_P^j, en_P^j, pc_{t_1}^j, \ldots, pc_{t_\ell}^j \rangle$, and $\widehat{c_j} = \langle sh_S^j, pc_S^j \rangle$. Given a program counter pc_S of S, we define $prev\_label(pc_S)$ as the last numerical label occurring in S before the statement with program counter pc_S.

The proof now proceeds, showing by induction on $j \in [0,m]$, that the following property $\mathcal{Q}(j)$ holds:

> There is an execution of P, $\pi_j = c_0 \underset{P}{\rightarrow} c_1 \underset{P}{\rightarrow} \cdots \underset{P}{\rightarrow} c_j$ and a non-decreasing sequence of round numbers $(r_0, r_1, \ldots, r_j) \in [k]^{j+1}$ such that for every $i \in [0,j]$, $c_i \equiv_{r_i} \widehat{c_i}$, and $r_i \geq round_{\pi_j}(i)$.

Base case: $j = 0$. We choose the initial configuration c_0 such that the values of global and local variables coincide with those in $\widehat{I}$. Furthermore, the auxiliary control variables of an initial configuration of S are by construction initialised as follows: (1) the valuation of `active[0]` is 1 and the valuation of `active[`i`]` is 0 for any $i \in [1,n]$; (2) variable `ct` is also set to 0; and (3) all the elements of array `pc` are set to 0, which is the numerical label of the first statement in any thread. Thus, c_0 and $\widehat{I}$ are equivalent except that the program counter of S is positioned at the beginning of the main driver, while in P it points to the beginning of the `main` procedure (i.e., the first thread simulation function).

The execution from $\widehat{I}$ to $\widehat{c_0}$ in $\widehat{\pi}$ is deterministic when r_0 is fixed. Observe that r_0 is the number of the loop iteration in the main driver in which the first thread simulation starts by executing at least one statement of the original thread. Now in the main driver

of S (see Figure 4.3), we execute the first `if` statement in r_0-th loop iteration, where `ct` is set to 0, and then the transition that sets `cs` to any nondeterministic value in the range of the numerical labels of the `main` procedure of P. This value must be greater than 0 to make the simulation start. Then, the sequentialized `main` is invoked. The condition check of the macro `J(0,1)` guarding the first statement s of the sequentialized `main` fails, and the control moves to s. This configuration is relevant and corresponds to $\widehat{c_0}$. Since none of the original variables has changed in the above transitions, $c_0 \equiv_1 \widehat{c_0}$, $sh^0_S(\texttt{cs}) > prev\_label(pc^0_S)$, and $r_0 \geq 1 = round_{\pi_0}(0)$ that shows the base case.

Inductive step. Now, assuming that $\mathcal{Q}(j-1)$ holds, we prove that $\mathcal{Q}(j)$ holds. For this, it suffices to prove that $c_{j-1} \underset{P}{\rightarrow} c_j$, $c_j \equiv_{r_j} \widehat{c_j}$, $sh^j_S(\texttt{cs}) > prev\_label(pc^j_S)$, and $r_j \geq round_{\pi_j}(j)$.

Since $\mathcal{Q}(j-1)$ holds, from the definition of equivalent configurations we obtain that $pc^{j-1}_S = linemap_{[P,k]}(pc^{j-1}_{en_{j-1}}, r_{j-1})$, which means that both P and S point to the same statement, say $stmt$, in c_{j-1} and $\widehat{c_{j-1}}$, respectively.

By construction, any execution of S that starts from a relevant configuration and ends at a relevant configuration without visiting other relevant configurations can be split into two executions: a first part that simulates a statement from P, and a second part that positions the program counter to the next statement to execute. Let $\widehat{c_{j-1}} \underset{S}{\rightsquigarrow} \widehat{c} \underset{S}{\rightsquigarrow} \widehat{c_j}$ be such an execution, where $\widehat{c}$ is the configuration resulting from the simulation of $stmt$.

Now, we pick as c_j the configuration of P obtained from c_{j-1} by executing $stmt$, choosing $en^j_P = sh^j_S(\texttt{ct})$ (which is always possible since the enabled thread is nondeterministically selected in P), and matching any other nondeterministic choice in $\widehat{c_{j-1}}$ and $\widehat{c}$.

We observe that for $\widehat{c}$ and c_j Definition 4.1 holds except possibly for parts 3-5. The proof is as for the cases *Thread creation and joining*, *Lock acquisition and release* and *Remaining statements* given in the proof of Lemma 4.2. Therefore, we omit further details on this.

Also, the execution from $\widehat{c}$ to $\widehat{c_j}$ does not modify the common variables of P and S, thus parts 1-2 of Definition 4.1 are preserved up to $\widehat{c_j}$. We now show that this computation from $\widehat{c}$ to $\widehat{c_j}$ indeed satisfies parts 3-5 of Definition 4.1, and so $c_j \equiv_{r_j} \widehat{c_j}$. The proof is by case inspection.

Let pc be $pc^j_{en_{j-1}}$, which is the program counter of the thread that has executed the statement in the last transition in P, and $\widehat{pc}$ be the program counter of S at $\widehat{c}$.

- *Non-visible statements.* If pc points to a non-visible statement, then $en^{j-1}_P = en^j_P$ and (by construction of S) $pc^j_S = \widehat{pc} = linemap_{[P,k]}(pc, r_j)$. Therefore, $\widehat{c}$ is a relevant configuration and $\widehat{c_j} = \widehat{c}$. We now prove that $c_j \equiv_{r_j} \widehat{c_j}$ by showing that parts 3-5 of Definition 4.1 hold. Note that variable `ct` is only updated in the

main driver, thus $sh_S^j(\texttt{ct}) = sh_S^{j-1}(\texttt{ct})$. By inductive hypothesis $sh_S^{j-1}(\texttt{ct}) = en_P^j$, thereby part 3 of the definition holds. With a similar argument we prove part 4. Moreover, we have already shown that $pc_S^j = linemap_{[P,k]}(pc, r_j)$, hence part 5 also holds.

Now we show that $sh_S^j(\texttt{cs}) > prev\_label(pc_S^j)$. Since pc_S^j points to a non-visible statement, it must be the case that $prev\_label(pc_S^{j-1}) = prev\_label(pc_S^j)$. Further, by inductive hypothesis, $sh_S^{j-1}(\texttt{cs}) > prev\_label(pc_S^{j-1})$. Thus, $sh_S^j(\texttt{cs}) = sh_S^{j-1}(\texttt{cs}) > prev\_label(pc_S^{j-1}) = prev\_label(pc_S^j)$.

We conclude the proof of this case by showing that $r_j \geq round_{\pi_j}(j)$. Since $en_P^j = en_P^{j-1}$, $round_{\pi_j}(j) = round_{\pi_j}(j-1)$. Furthermore, the control in S remains in the same sequentialized thread function, hence $r_j = r_{j-1}$. By inductive hypothesis, $r_{j-1} \geq round_{\pi_j}(j-1)$ thereby $r_j \geq round_{\pi_j}(j)$.

- *Visible-statements.* If pc points to a visible statement, then the corresponding statement in S with program counter $linemap_{[P,k]}(pc, r_j)$ is guarded by either the macro `J(`$s, s+1$`)`, or the sequence of macros `G(`s`)` `J(`$s, s+1$`)`, for some numerical label s. We only consider the latter case as it is more general. Observe that the configurations right before and after the execution of `G(`s`)` are not relevant. Since we reach $\widehat{c_j}$, the `assume` statement in `G` does not block the computation. Let us now distinguish the cases based on whether the condition check in `J(`$s, s+1$`)` succeeds. We recall that `J(`$s, s+1$`)` is defined as follows:

  ```
  if( pc[ct]>s || s >=cs ) goto s+1;
  ```

  No context-switch simulation. If the condition *cond* of the if-statement in `J` is evaluated to false, then the control moves to $linemap_{[P,k]}(pc, r_j)$ in S. The reached configuration is relevant and corresponds to $\widehat{c_j}$. This proves parts 1-2, 5 of Definition 4.1. Parts 3 and 4 also hold for the same argument as the *Non-visible statements* case given above. Furthermore, since *cond* does not hold, it must be the case that $sh_S^j(\texttt{cs}) > prev\_label(pc_S^j) = s$. The argument proving that $r_j \geq round_{\pi_j}(j)$ is the same as for the *Non-visible statements* case given above.

  Context-switch simulation. The remaining case is when *cond* is evaluated to true. In this case, we claim that $s = sh_S^{j-1}(\texttt{cs})$ holds.

  Claim's proof. We first prove that first sub-expression in *cond*, i.e., `pc[ct]>`s, is evaluated to false. The proof is by contradiction. If `pc[ct]>`s was evaluated to true, then the same sub-expression in all the other macros `J(`t`)` with $0 \leq t < s$ within the same sequentialized function would be evaluated to true. Thus, in the last call to the current thread function along $\widehat{\pi}$, the control would have moved from `J(0,1)` to `J(`$s, s+1$`)`, jumping in multiple hops through the in-between `J` macros. Consequently, the simulation of *stmt* would not have

taken place, which is indeed a contradiction. Thus, *cond* holds because s `>=cs` is evaluated to true.

We now show that at this point of the execution indeed s corresponds to the evaluation of `cs`. We consider two cases. If the macro `G(`s`)`, which is defined as `assume(cs>=`s`)`, precedes `J(`$s, s+1$`)`, then the evaluation of `cs` after the evaluation of *cond* in `J` must be s. If only the macro `J(`$s, s+1$`)` occurs, the control must come from the previous statement (which is *stmt* by hypothesis) whose program counter is $pc_S^{j-1} = linemap_{[P,k]}(pc_{en_{j-1}}^{j-1}, r_{j-1})$. Note that $prev\_label(pc_S^{j-1}) = s-1$. By inductive hypothesis, $sh_S^{j-1}(\texttt{cs}) > s-1$. Since in the simulation of *stmt* variable `cs` is not modified, the evaluation of `cs` satisfies the same condition right before `J(`$s, s+1$`)`. Thus, similarly as above, when `J(`$s, s+1$`)` is executed the evaluation of `cs` is s. □

The above claim implies that on executions of S leading to relevant configurations, after the simulations of some statements of a thread, the first time that the condition of a `J` macro does not hold always coincides with the case that `cs` evaluates to s. Since the numerical labels follow an increasing order, this will be the case up to the end of the thread simulation function. By construction, the control jumps in multiple hops from one label to the next one and then back to the main driver, right after the unique call to the (inlined) sequentialized function of the thread with identifier en_{j-1} of the r_{j-1}-th iteration of the while loop in the main driver of S. Thus, variable `pc[`en_{j-1}`]` is set to `cs`, and as shown above, corresponding to the numerical label of the statement where the context-switch has happened. The other elements of the array `pc` remain as in $\widehat{c_{j-1}}$. Now, going through all successive thread simulation blocks within the main driver, we enter the first block int the r_j-th iteration that corresponds to the thread identified by $sh_S^j(\texttt{ct}) = en_j$. Variable `ct` is then set to en_j, and rightafter `cs` is set to a non-deterministic value greater than the valuation of `pc[ct]`. Then, the thread simulation function for the enabled thread en_j is entered. Jumping in multiple hops the control is repositioned to the numerical label `pc[`en_j`]`, which by inductive hypothesis corresponds to the label of the last context-switch for this thread. Here the macro `J` has no effect and the control is positioned to $linemap_{[P,k]}(pc, r_j)$. The reached configuration is $\widehat{c_j}$, and $c_j \equiv_{r_j} \widehat{c_j}$.

It is straightforward to show that (1) the valuation of `cs` is greater than the valuation of `pc[ct]` (see above), and (2) using the inductive hypothesis that $r_j \geq round_{\pi_j}(j)$.

□

From Lemma 4.2 and 4.3, we are now ready to claim the main result of this section.

Theorem 4.4 (Correctness)**.** *Let P be a bounded multi-threaded program, and k be a positive integer. P fails and assertion through a k round-robin execution if and only if $[\![P]\!]_k$ fails an assertion.*

4.5 Alternative Scheduling Policies

Our sequentialization schema captures round-robin executions, where in each round the threads are scheduled following a fixed order. It has been carefully fine-tuned with the purpose of optimising the performance of the back-end for bug finding. However, the interdependency, in the resulting schema, between the simulation of thread execution, pre-emption, and resuming (undertaken by self-contained functions, as described in Section 4.4.3) and the scheduler (the main driver presented in Section 4.4.2) grants some degree of control over the considered thread interleavings. This can profitably be exploited for different goals.

In this section, we outline two simple variations of the original schema: the first avoids the analysis of unwanted schedules by filtering them out according to the thread identifiers involved in the simulation; the second one extends the control at the level of the individual execution contexts, by deactivating context-switch points that do not satisfy specific conditions. Both of them can be obtained by slightly modifying the main driver, and can be used, for instance, to (a) guide the analysis on a pruned state space (aiming at leveraging specific facts known on the input program in order to achieve faster analyses), or (b) partition the state space at the level of the translation (desirable for partial or distributed analysis of large programs).

The simplicity of our changes points out, once again, the expressiveness of reasoning at the level of the source code, especially for sequentializations, where concurrency-related aspects can be manipulated at a very high-level. In fact, achieving similar alterations by working at a different level, for instance at the level of the verification condition, or even at the level of the decision procedure, would require considerable efforts.

Coarse-grained Selective Round-robin Scheduling

A first simple variation consists in introducing schedule restrictions on a round-by-round basis. Namely, for each round we can explicitly indicate a pool of threads (i.e., a subset of the threads of the program) to which limit the simulation such that the simulation of threads that do not belong to the pool is bypassed[5].

This is a generalisation of the default round-robin scheduler shown in Figure 4.3 that can be captured by setting the pool of threads to all threads in each round. Also, observe

[5]The pool for the first simulated round must include the main thread, otherwise the simulation cannot start.

that we can narrow the pools down to single elements for each round, thereby forcing a specific thread interleaving. However, we remark that even when the schedule is fixed context switching can still occur at any visible statement.

Assuming that the external loop in the main driver has been unfolded k times, this results in a static simplification of the main driver. More precisely, in each (unfolded) iteration of the main driver, now corresponding to a *restricted* round, the guarded code snippet that simulates a thread is statically removed when that thread is not in the pool of threads for that round. In practice, the translation is tailored to the specific sub-set of possible schedules, and the trimmed-down main driver results in smaller verification conditions. Our current prototype includes this feature (see Section 5.5.2).

Fine-grained Selective Scheduling

We now describe another variation to our schema that, operating on the context switch points, allows to refine the set of analysed program behaviours to a more fine-grained level.

The main driver shown in Figure 4.3 uses the variable `cs` multiple times to guess the next context switch point for the thread under simulation, immediately before invoking the thread simulation function. However, we can guess all the context switch points at once, for each thread and round, at the beginning of the main driver, and access them using double-indexed array elements `cs[round][ct]` instead of one single scalar variable.

This change in the main driver yields an equivalent sequentialization, but, with all the context switch points guessed upfront, restricting conditions may be enforced on them by introducing an `assume`-statement right after their guessing. Note that, due to static single assignment (see Section 2.1.4), the variable `cs` used in the original schema is duplicated anyway along the process that generates the verification condition, hence the proposed variation does not affect the size of the verification conditions[6].

Following a similar reasoning, we can now replace variable `ct` with multiple variables `ct[round]`, and overwrite the constant values used in the original assignments of `ct` according to our preferences. This yields an effect similar to when restricting round-robin schedules using singleton pools (as described above), and in addition it allows fine-grained control over the considered program behaviours, by considering both the scheduled threads and the context-switch point at once. This can for instance be used as a basic mechanism to experiment with partial-order reduction techniques [FG05, KWG09].

[6] At a closer look, since the guesses occur at different points in the program, the resulting verification conditions have different structures and may trigger different heuristic decisions within the underlying SAT procedure, thus potentially leading to performance gaps. However, the purpose of this variation is not to improve the bug-finding performance, and as a matter of fact the standard translation generally yields faster analysis in practice.

4.6 Evaluation

We have implemented our sequentialization approach in the Lazy-CSeq tool for multi-threaded C programs (see Section 5.5) using our CSeq framework (see Chapter 5).

Our evaluation is divided in two main parts:

1. we compare Lazy-CSeq in combination with multiple backends against several tools with built-in concurrency handling, observing the bug finding performance, the state space coverage, and the size of the verification conditions (Section 4.6.1);
2. we compare Lazy-CSeq using the best-performing backend against LR-CSeq using the best-performing backend and against the best-performing bounded-model checker with built-in concurrency handling, observing only the bug finding performance and using a considerably extended benchmark suite for a more in-depth comparison (Section 4.6.2).

4.6.1 Multiple Backends vs. Multiple Concurrency-handling Tools

We have evaluated Lazy-CSeq on the benchmark set from the concurrency category of the SV-COMP 2014 software verification competition [Bey14]. This set consists of 76 concurrent C programs using POSIX threads as a concurrency model, with a total size of about 4,500 lines of code. 20 of the files contain a reachable error location. We chose this benchmark set because it is widely used and all tools (but Corral) we compare against have been trained on this set for the competition.

The experiments are split into two parts. The first part only concerns the unsafe programs, where we investigate the effectiveness of several tools at finding errors. The second part concerns the safe programs, where we estimate whether limiting the round bound to small values allows a more extensive exploration of programs in terms of increased values of loop unwinding bounds.

The tools considered for comparison are BLITZ [CDS13] (4.0), CBMC [AKT13] (4.5 and 4.7), Corral [LQL12], LR-CSeq [FIP13a] (0.5) ESBMC [CFM12] (1.22), LLBMC [MFS12] (2013.1), and Threader [PR13].

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

TABLE 4.1: Bug-hunting performance (unsafe instances); $-^1$: timeout (750s); $-^2$: internal error; $-^3$: manual translation not done; $-^4$: test case rejected; $-^5$: unknown failure.

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

Unsafe instances

The evaluation on unsafe instances is split into two parts. The purpose of the first part is to evaluate the performance of Lazy-CSeq in combination with different sequential backends; the second part compares the performance of Lazy-CSeq against different tools with built-in concurrency support.

The performance of Lazy-CSeq using different backends is shown on the left of Table 4.1. Note that only the backend run-times are given. The additional Lazy-CSeq pre-processing time, which is the same for every backend, is about one second for each file with our current Python prototype implementation. This could easily and substantially be reduced with a more efficient implementation. The results show that the tools were able to process most of the files generated by Lazy-CSeq's generic pre-processing, and found most of the errors. This is in marked contrast to our experience with LR-CSeq, where the integration of a new backend required a substantial development effort, due to the nature of the Lal-Reps schema. They also show that the different backends generally perform relatively uniformly, except for few cases where the performance gap is noticeably wide, probably due to a different handling of subtle corner-cases in the input from the backends. Both observations gives us further confidence that our approach is general and not bound to a specific verification backend tool.

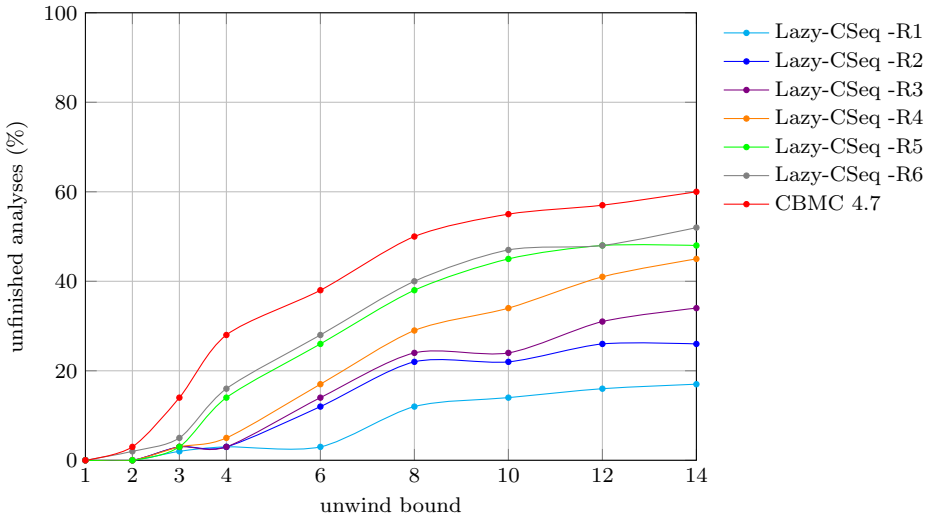

FIGURE 4.6: Evaluation of safe benchmarks for increasing loop unwind bounds.

We then compared the bug-hunting performances of Lazy-CSeq and several tools with different native concurrency handling approaches. CBMC and ESBMC are both bounded model-checkers; CBMC uses partial orders to handle concurrency symbolically while ESBMC explicitly explores the different schedules [CF11]. LR-CSeq is our implementation of the variant of the Lal-Reps schema (see Chapter 3), and uses CBMC as sequential backend. Corral [LQL12] uses a dynamic unwinding of function calls and loops, and implements abstractions on variables with the aim of discovering bugs faster. Threader, the winner in the Concurrency category of the SV-COMP 2013 competition, is based on predicate abstraction. For each tool (except Threader) we adjusted, for each file, all parameters to the minimum needed to spot the error. The results, given on the right of Table 4.1, show that Lazy-CSeq is highly competitive. Of the "native" tools only CBMC is able to find all errors with the most recent version. All other tools time out, crash, or produce wrong results for several files. This shows how difficult it is to integrate concurrency handling into a verification tool—in contrast to the conceptual and practical simplicity of our approach. Moreover, for simple problems (with verification times around one second), Lazy-CSeq performs comparably with the fastest competitor. On the more demanding instances, Lazy-CSeq is almost always the fastest, except for the Fibonacci tests (53, 55 and 57) that are specifically crafted to force particularly twisted interleavings. In most cases (again except for the Fibonacci tests), Lazy-CSeq successfully finds the errors in all test cases using only three rounds, confirming that few context switches are sufficient to find bugs [QW04, MQ07, TDB14].

Safe instances

The evaluation on safe instances consisted in comparing Lazy-CSeq using CBMC v4.7 as backend with the best-performing tool with native concurrency handling (again, CBMC).

We ran nine sets of experiments for CBMC with unwinding bounds to 1, 2, 3, 4, 6, 8, 10, 12, and 14, respectively. Recall that CBMC considers all possible interleavings and does not performs context-bounding. For Lazy-CSeq, we ran six repetitions of the sets, with a bound on the number of rounds from one to six, for each of the above unwinding values, respectively.

As shown in Figure 4.6, we observe that CBMC starts performing worse than Lazy-CSeq, in terms of number of instances on which the analysis is completed, as we increase the loop unwinding bound. Overall, with the settings from the SV-COMP, Lazy-CSeq, is about 30x faster than CBMC for safe instances. This points out how the introduction of an extra parameter for BMC, i.e., the bound on the number of rounds, can offer a different, alternative coverage of the state-space. In fact, it allows larger loop unwindings, and therefore a deeper exploration of loops, than feasible with other methods.

Size of the Verification Condition

We conducted further investigation in order to compare the size of the verification conditions generated from the original files against the size of their sequentialized counterparts. We compared the number of variables and clauses of the resulting formulae reported by CBMC after the propositional reduction on the original and the sequentialized files, with loop unrolling bounds of 1,2,3,4,6, and 8 and round bounds from 1 to 4.

Figure 4.7 shows the curves of the average ratio, over all the safe test cases, between the number of variables (resp. clauses) generated from the original files and the number of variables (resp. clauses) from the sequentialized files, for different values of the loop unwind bound. We observe that this gap grows for increasing values of this parameter.

The major insight here is that introducing a small bound on the number of round-robin schedules keeps the formulae compact when increasing the loop unwinding bound; in contrast, with unbounded context-switches the formula tends to grow very quickly both in the number of clauses and variables.

Note that with unwinding bounds greater than 8 the back-end fails to produce the formula within the given memory and time limits on too many test cases, which makes it unfeasible to extend our comparison beyond that bound.

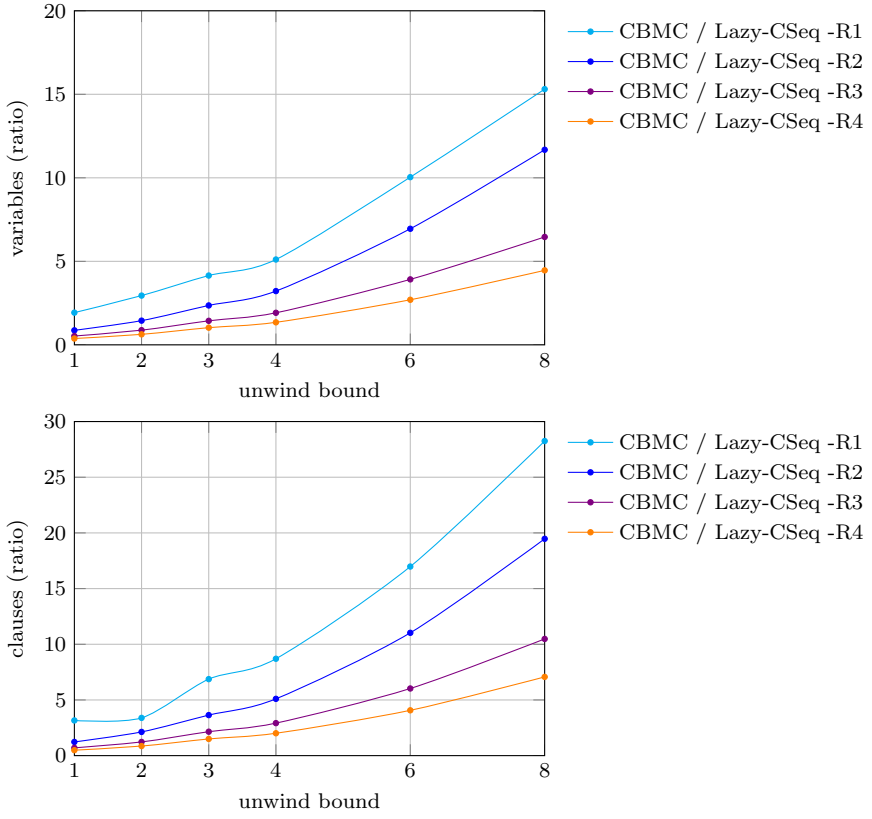

FIGURE 4.7: Verification condition size ratio between CBMC and Lazy-CSeq using different round bounds.

Several approaches [RG05, GG08, SW11, SW10, AKT13] encode program executions as partial orders, in which each thread is an SSA program and operations on the shared memory are constrained by a global conjunct modeling the memory model. In [AKT13] the authors argued that the formula size of their encodings on the considered benchmarks (among which are 36 from SV-COMP 2014) is smaller than those of [RG05, GG08, SW11, SW10]. In our work, we have empirically evaluated the formula size of our encoding against CBMC (see Figure 4.7). The main result is that our approach yields smaller formulae already for small unwind bounds, even for four rounds; with increasing unwind bounds (e.g., $n = 8$), CBMC's formulae contain 5x to 15x more variables and 5x to 25x more clauses, depending on the number of rounds.

4.6.2 Fastest Backend vs. Fastest Concurrency-handling Tool

We recently re-compared the bug-hunting performance of Lazy-CSeq against CBMC (as the best-performing bounded model-checker with native concurrency handling) and LR-CSeq (our implementation of the LR schema described in Chapter 3), on a considerably extended benchmark suite, using the latest available versions of the tools, and a faster machine to reduce the timeouts. We used CBMC as the backend for both our sequentialization tools.

We considered the 783 unsafe files of the 993 files from the Concurrency category of the SV-COMP 2015 benchmark suite [Bey15], with a total of approx. 240K lines of code.

We have performed the experiments on an otherwise idle machine with a Xeon W3520 2.6GHz processor and 12GB of memory, running a Linux operating system with 64-bit kernel 3.0.6. We set a 10GB memory limit and a 750s timeout for the analysis of each subject. For each tool and file, we set the parameters to the minimum value needed to expose the error.

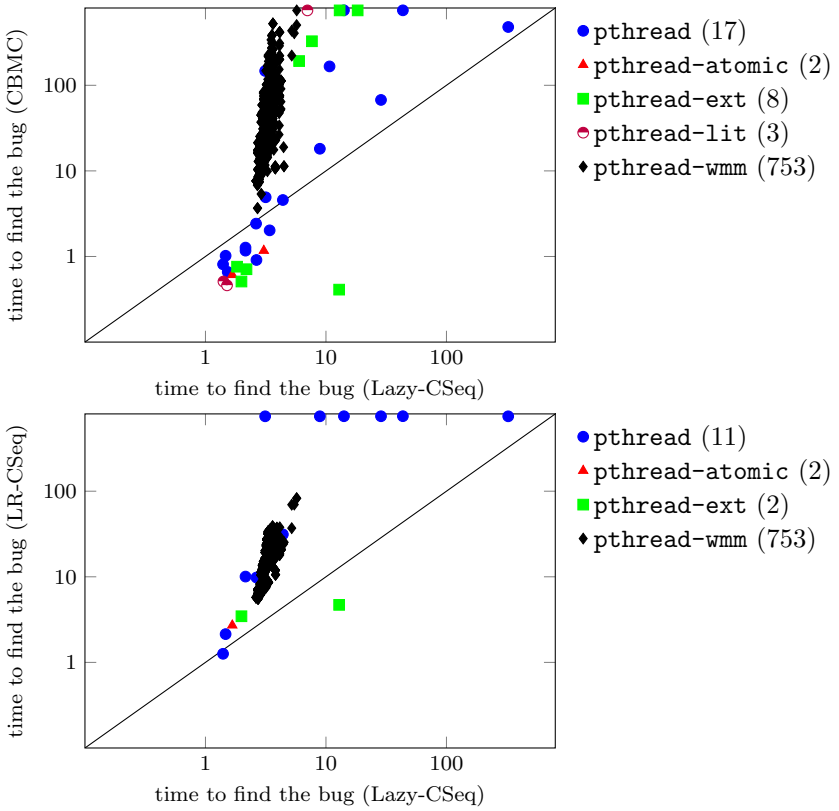

FIGURE 4.8: Lazy-CSeq vs. CBMC and LR-CSeq: bug-hunting performance

The scatter plots (with logarithmic axes) shown in Figure 4.8 summarise the running-time comparison between Lazy-CSeq and CBMC, and Lazy-CSeq and LR-CSeq.

All tools report the correct answers. Both CBMC and LR-CSeq time out on 6 files. Furthermore, LR-CSeq rejects 5 files and returns "unknown" on 10 files (due to the restrictions mentioned in Chapter 3, translation errors or bugs in the tool). The experiments show that Lazy-CSeq outperforms both CBMC and LR-CSeq, except on a handful of small files on which CBMC is faster. Overall, Lazy-CSeq is about 6x and 20x faster than LR-CSeq and CBMC, respectively.

4.7 Conclusions

In this chapter we have presented a novel lazy sequentialization schema for bounded multi-threaded programs that has been carefully designed to take advantage of BMC tools developed for sequential programs. We have implemented our approach for multi-threaded C programs with POSIX threads in the prototype tool Lazy-CSeq as a code-to-code translation (see Section 5.5) using our sequentialization framework, CSeq (see Chapter 5). Lazy-CSeq can be used as a stand-alone model checker that currently supports several BMC tools as backends. The experimental results show that our prototype:

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

Laziness allows us to avoid handling all spurious errors that can occur in an eager exploration (for example when using the Lal-Reps schema evaluated in Chapter 3). Thus, we can inherit from the backend tool all checks for sequential C programs such as array-bounds-check, division-by-zero, pointer-checks, overflow-checks, reachability of error labels and assertion failures, etc.

A core feature of our code-to-code translation that significantly impacts its effectiveness is that it just injects light-weight, non-invasive control code into the input program. The control code is composed of few lines of guarded `goto` statements and, within the added function `main`, also very few assignments. It does not use the program variables and it is clearly separated from the program code. This is in sharp contrast with the existing sequentializations (such as LR, LMP [LR09, LMP09b], which can handle also unbounded programs) where multiple copies of the shared variables are used and assigned in the control code.

As consequence, we get three general benefits that set our work apart from previous approaches, and that simplify the development of full-fledged, robust model-checking tools based on sequentialization. First, the translation only needs to handle concurrency—all other features of the programming language remain opaque, and the backend tool can take care of them. This is in contrast to, for example, LR where dynamic allocation of the memory is handled by using maps [LQR09]. Second, the original motivation for sequentializations was to reuse for concurrent programs the technology built for sequential program verification, and in principle, a sequentialization could work as a generic concurrency preprocessor for such tools. However, previous implementations needed specific tuning and optimizations for the different tools (see [FIP13a]). In contrast,

Lazy-CSeq works well with different backends (currently BLITZ, CBMC, ESBMC, and LLBMC), and the only required tuning was to comply with the actual program syntax supported by them. Finally, the clean separation between control code and program code makes it simple to generate a counter-example starting from the one generated by the backend tool.

Chapter 5

CSeq Framework

In this chapter we present CSeq, our open-source framework for developing sequentialization tools. We describe the architecture of the framework, its main functionalities, and how to develop new source-to-source translations. We present our Lazy-CSeq tool and discuss how it has been developed within this framework [INF+15].

We release CSeq, including Lazy-CSeq, as open-source software. The project's homepage is at: http://users.ecs.soton.ac.uk/gp4/cseq/cseq.html.

5.1 Overview

CSeq is a framework for developing tools for program analysis of C programs [ISO11] that use the POSIX threads shared-memory concurrency model [ISO09]. It follows the sequentialization approach, where the analysis of concurrent programs is reduced to the analysis of sequential (i.e., non-concurrent) programs (as described in Section 2.3.2). There are three main advantages in this approach: (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; (3) sequentializations can be designed to target multiple backends for sequential program analysis.

CSeq subsumes our experience in developing sequentializations, and in particular emphasises the above aspects by: (a) encouraging a modular approach to source translation, where the translation can be designed as a sequence of simple steps; (b) providing a range of built-in modules and functionalities, in order to limit the overall engineering effort and speed-up the prototyping and development of sequentialization-based tools.

In practice, the use of sequentialization can be summarised in three main phases:

- *transformation*: the concurrent program is re-written into a corresponding sequential version (i.e., the sequentialized file), where typically concurrency is removed and replaced by non-determinism
- *analysis*: the sequentialized file is analysed using a backend for sequential analysis that can handle non-determinism
- *feedback*: the output from the backend is processed in order to generate a user-readable report.

The tool design methodology that we propose with our framework consists in splitting the above phases into simple steps, each implemented as a separate component, or *module*. A complete tool is then obtained by conveniently arranging the modules in a sequence, or *configuration*. This modular approach supports the development of complex source transformations.

CSeq borrows concepts related to source-to-source transformation and software analysis frameworks. Source-to-source transformation, sometimes also referred to as code refactoring, was introduced in the 1970s for recursive programs with the goal to improve code maintenance [BD77]. More recently, many source-to-source transformation frameworks have been made available. A possible approach, followed for instance by DMS [BPM04], consists in defining the transformations using rewrite rules. Similarly, TXL [Cor06] is a special-purpose programming language for rapid prototyping of generalised source transformation systems; it targets multiple languages and features language primitives for specifying tree rewriting rules using context-free grammars. Cetus [BML+13] specifically targets C programs and provides a range of built-in transformations for optimised parallelisation and annotation of the source code. ROSE [QSPK01] uses different data structures to capture multiple aspects of the input code, and embeds many source transformations, for instance for loops (loop interchange, loop splitting, loop unrolling, etc). Rather than using rewrite rules, it uses string-based source transformations. Roughly this consists in building the AST from the original code and then visiting the AST to re-generate the code. The transformation is achieved by altering the behaviour of the AST visiting procedures.

CSeq only targets the C programming language [ISO11], and in particular shared-memory multi-threaded programs using POSIX threads [ISO09]; it provides data structures that capture detailed concurrency-related information on the input. In addition, it comes with a few built-in standard transformations (not restricted to multi-threaded programs), such as loop unrolling, function inlining, etc., commonly used in program analysis. It uses string-based transformation, which is possibly more intuitive than using rewrite rules, and thus perhaps closer to a developer's point of view.

There are many well-known tool-development frameworks for software analysis. The CPROVER framework has been used to develop tools for bounded model-checking

[CKL04, MFS12, CFM12, CDS13] and abstraction-based tools such as SATABS [CKSY05]. It uses an intermediate representation that reduces the input program to a control-flow graph. The IKOS [BNSV14] framework is targeted at building tools based on abstract interpretation of avionic software; it uses the intermediate representation of LLVM (LLVM-IR) [LA04].

LLVM's intermediate representation has recently been attracting considerable interest, and many tools both for software analysis and source-to-source translation rely on its well-designed API built on top of a properly layered architecture. However, the frequent changes to the API between releases anticipate additional code maintenance efforts. The C back-end to generate C code from a parsed IR source tree has been removed due stability and other issues, with tentatives to resurrect that functionality being still at an experimental stage and outside the scope of the main branch. As an intermediate representation LLVM-IR is thus not particularly indicated for source transformations, especially on concurrent programs, as the support for multiple threads is not very mature either.

CSeq does not use any low-level intermediate representation of the source code. Intermediate representation, while undoubtedly increasing robustness by making the syntax more regular, inevitably drops along the way potentially relevant information about the input program. On the other hand, working directly on the original source language offers a more abstract, compact, and expressive representation that can indeed support more intricate reasoning, especially when dealing with source-to-source translations. However, this does not prevent tool developers from implementing sequences of multiple transformations to progressively obtain syntactically-restricted programs, whenever this is desirable to simplify the development of complex transformations, as described later in this chapter. In this sense the C language itself is used as an intermediate representation, in a way similarly to CIL [NMRW02].

In this chapter we present the general architecture of CSeq and the Lazy-CSeq tool as a specific instantiation that applies the lazy sequentialization schema discussed in Chapter 4 to realistic multi-threaded C programs. To date, however, CSeq has also been used for developing other tools, namely MU-CSeq, which implements a new schema based on memory unwindings [TIF+14, TIF+15] and a new prototype targeting abstraction-based backends [NFLP15].

5.2 Architecture

The architecture of CSeq and the main component interplay is shown in Figure 5.1. The front-end controls the execution of the system. The user provides through the command-line (1) the input file, or files, to analyse, (2) the name of the definition file

for the configuration to use, (3) a possibly empty extra argument list, depending on the modules used in the configuration.

A configuration definition file is a plain text file that contains a list of modules: the front-end executes them in the order given in the configuration and with the given arguments (if any). The input of the first module is the content of the input file whose name is given as an argument to the front-end, and the input of any of the remaining modules is the output of the corresponding previous module in the configuration. The output of the last module is the output of the front-end shown on the standard output at the end of the execution.

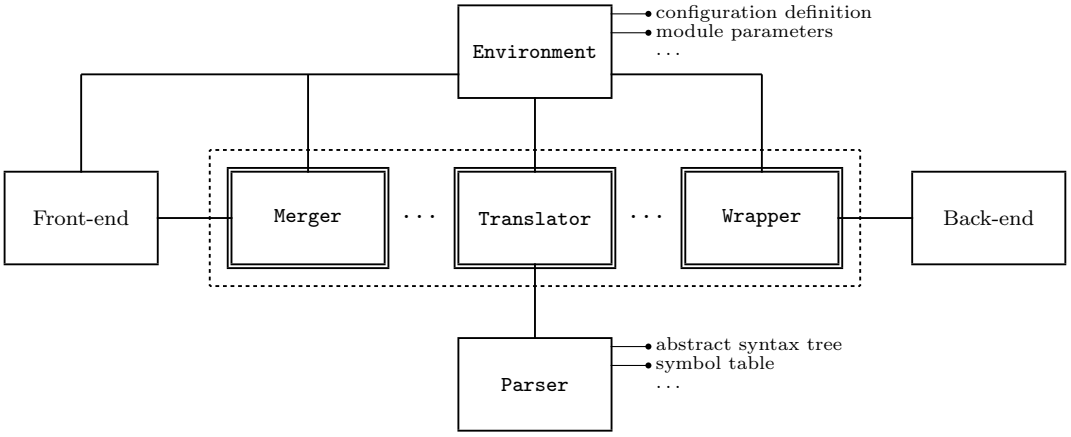

FIGURE 5.1: Architecture of the CSeq framework

The dashed and the double-framed rectangles in the diagram indicate the configuration and the modules, respectively. A configuration starts with the source merging module, `Merger`, then continues with a sequence of `Translator` modules followed by a sequence of `Wrapper` modules.

Each module has an input string, an output string, and zero or more parameters. `Wrapper`s work on generic input and output strings, and are basic units that carry out general-purpose tasks, such as interacting with the operating system to make system calls. More generally, they allow embedding external components within a tool. For example, the built-in module for backend wrapping takes as input a program, invokes an external tool to analyse the program, and returns the output from the tool upon completion. `Translator` modules are specialised modules that extend the basic functionalities of a `Wrapper` in order to support source-to-source program transformation. They take as input a C program and return as output a transformed C program. `Translator`s do not allow preprocessor directives in the input, nor multiple inputs. The `Merger` module, positioned in the beginning of the chain, is an exception. It takes as input one or multiple C programs, performs source merging and preprocessing by invoking the C preprocessor [SW05, EBN02], and returns as output a single C program.

The `Environment` object, shared by the front-end and all modules, keeps tracks of the overall status of the CSeq system and of each module in the configuration. It is a container for shared information that the front-end and the modules can access throughout the execution of a tool. The front-end uses it to store the input and output of each module, the actual parameters, and other data structures described later on in this chapter.

The `Parser` object available to `Translator`s provides a set of data structures useful for reasoning about the input code, such as the abstract syntax tree, the symbol table, and other information extracted using lightweight static analysis. In particular, the `Parser` extracts from the input some concurrency-specific information, such as the (over-approximated) set of visible statement, the list of functions potentially used to spawn new threads, and so on. The `Translator` object is built on top of `pycparser`, an open-source C parser that uses `PLY`, an implementation of Lex-Yacc [LS90, Joh75].

5.3 Modules

A module of CSeq is implemented in module definition file, that is a separate Python file that extends either class `Translator` or `Wrapper`.

Modules work in two steps: initialisation and execution. Module initialisation is performed by method `init`, to initialise all the data structures needed later in the execution step and declare any parameters used by the module. A module is executed by invoking `loadfromstring` on a given input string (see Figure 5.4).

The initialisation and execution of a module are both triggered by the front-end, that at the end of the execution forwards the output of the module to the next module in the configuration, if present, or to the standard output.

The basic mechanisms provided by a `Translator` are: source transformation, argument passing, and line mapping. Argument passing is described in Section 5.3.1, line mapping in Section 5.3.2, source transformation in Section 5.3.3. `Wrapper`s are simpler basic units and only support argument passing.

5.3.1 Argument Passing

Modules can be parameterised for improved flexibility. Input parameters are convenient for parameterised transformations. CSeq's standard `unroller` module for program unfolding, for instance, accepts as an argument an integer number representing the loop unrolling bound; the `instrumenter` module for backend instrumentation accepts a string with the name of the backend for which it needs to instrument the input.

Output parameters can be used to transfer information across modules. For instance the loop unrolling module mentioned above might follow another module that calculates over-approximations for loop bounds and outputs them as an argument for next modules, thereby avoiding the need to provide externally a loop bound to the `unroller`.

Parameters are declared in the module initialisation method, `init`, within the module definition file, by invoking either `addInputParam` or `addOutputParam` (see Figure 5.4). During the initialisation cycle, the front-end invokes the `init` method for each module in the configuration being used, and collects all the parameter definitions. It then attaches the parameter definitions to the `environment` object. The front-end then adjusts the command-line options accordingly. Roughly, if a module declares an input parameter that is not an output parameter for some (previous) module in the configuration, then it should be provided by the user in the command-line. The arguments are later fetched within the module by invoking `getParamValue`, or set by `setParamValue`.

Figure 5.8 shows the argument passing in Lazy-CSeq. The `backend` parameter is shared by multiple modules. Note that `threads`, an actual parameter for the sequentialization schema, is not provided externally but is instead calculated by one of the program bounding modules, and used later.

5.3.2 Line Mapping

`Translator` modules have an automatic *line-mapping* feature. The idea is to keep track of the exact coordinates in the input source where the translated code originates from.

This can help debugging transformation prototypes and can be useful for interpreting the output from the backend. In a traditional bug-finding setting, for instance, the backend generates a counterexample trace with the steps to reproduce a bug. Without tracing the lines back to the original file, the trace may be difficult to understand because it refers to the transformed program rather than to the initial one.

Line mapping is automatically calculated and normally does not require any extra engineering effort; it is available at the end of the module's execution as a map from output line numbers to input line numbers.

The mapping is calculated on a line-by-line basis and regardless from the specific transformation performed, in a similar way to how the C preprocessor (CPP) uses line control information, by inserting explicit `#line` directives in the source code during the generation of the output. However, at the end of the execution of the module, line control information is removed from the output and the information is stored as a map from output line numbers to input line numbers (note that each input line may generate several output lines, for instance when unfolding a loop or inlining a function multiple times).

For example, in the function inlining example of Figure 5.5, line `3` of the input program on the left generates output lines `8` and `9` (note that the jump is used instead of the return statement, and `_f` simulates the passing of the return value). As another example, consider the loop unrolling shown in Figure 5.6. Lines `5,6` and `7` of the input program (a) are translated into lines `5-7`,`9-11` and `13-15`, respectively, in the output program (b). Input line `4` generates output lines `4`, `8` and `12`.

For a more involved example of how code snippets propagate over a longer sequence of transformations, consider Figure 5.8(i)-(v). The source files are initially merged into a single string, and simplified. The line map for this first stage is therefore, and exceptionally, a function from output line numbers to pairs of the kind *(linenumber,filename)*. On subsequent transformations, it will suffice to map only line numbers to line numbers, as all operations after the source merging are on single input and output files.

The simplified program then gets through a program flattening stage that includes loop unrolling and function inlining (see Section 5.4.1). Note that in this example the thread `t1` of the bounded program (*ii*) contains a loop that has been unfolded three times in program (*iii*), meaning that the lines corresponding to the loop body of the input program on the left generate three different sets of output lines. Similarly, a function is inlined twice, in thread `tn` and function `main`. In the next transformation, the mapping from existing lines to output lines is unchanged, but there are extra snippets of code at the top and bottom of the output program (*iv*): no map entries are generated for these lines as these are not translated from the input, but simply added to the output as raw strings. The last transformation does not alter the line map either, as it does not add any new line but in practice only performs a few string substitutions. Note that unmapped lines always refer to code injected at some point during the translation (e.g., the main driver, additional function definitions for the instrumentation, expanded header files, etc.) rather than translated code. They introduce intermediate transitions during the simulation that in the actual counterexample translation (see Section 5.5.1) need to be processed in a way that depends on the specific source transformation (and in some cases can be safely ignored).

Line mapping is available at two different levels of detail: at the level of the individual module, maps are available through the `outputtoinput` and `inputtooutput` instance attributes, that map from output line numbers to input line numbers (and the other way around); at the level of the translation, from output line numbers for a given module to input line numbers for the first module in the configuration (or more precisely, to coordinates of the original input file, or files). This map is available by invoking `generatelinenumbers()` that iteratively composes the line maps by following the configuration in a reversed order. The resulting output can be visualised as a a table of the line maps across all source transformation steps, one row for each output line, one column for each transformation.

```
import core.module

class test(core.module.Translator):
   def visit_UnaryOp(self,n):
      if n.op == "p++":
         return "%s = %s + 1" % (n.expr,n.expr)

      return super(test,self).visit_UnaryOp(n)
```

FIGURE 5.2: Source transformation module: from `x++` to `x=x+1`.

5.3.3 Source Transformation

Source-to-source transformation is the main task for a `Translator`. When the front-end executes a `Translator` invoking its `loadfromstring` method, this automatically parses the input code to build the AST, then visits the AST to un-parse it back and generate the transformed input, which is the output of the module.

Source transformation is obtained by modifying the standard AST visit in such a way to produce alterations in the output. This mechanism is implemented by conveniently overriding `pycparser`'s AST-based pretty-printer. Note that by design choice no structural change is made to the AST itself, the transformation is in fact performed on-the-fly by directly modifying the output strings corresponding to fragments of code generated during AST sub-visits.

In the rest of this section we show two small examples of source transformation, and how to wrap up one of them to build a complete tool that can be invoked from the command-line.

A Simple `Translator`

Figure 5.2 shows a small example of source transformation module that replaces every statement using the unary post-increment operator on a variable, say `x`, with a statement that assigns to `x` the result of the addition `x+1`.

The transformation is implemented by overriding the `visit_UnaryOp` method that generates the source code from AST-subtrees representing unary operations. In the AST, nodes for unary operations have two children: `expr` and `op`, representing the variable name and the operator, respectively. The string `p++` represents the operator, where `p` is not the real variable name but a placeholder used in the grammar to distinguish pre- and post-increment operators (a simple `++` would have been ambiguous).

The `if` statement detects whether the unary operation is of the kind being targeted, in which case the amended output is returned. In any other case the method returns the code snippet that would normally be generated during a standard visit of the AST-subtree for unary operations (notice the call to `super`), therefore leaving the input code unchanged.

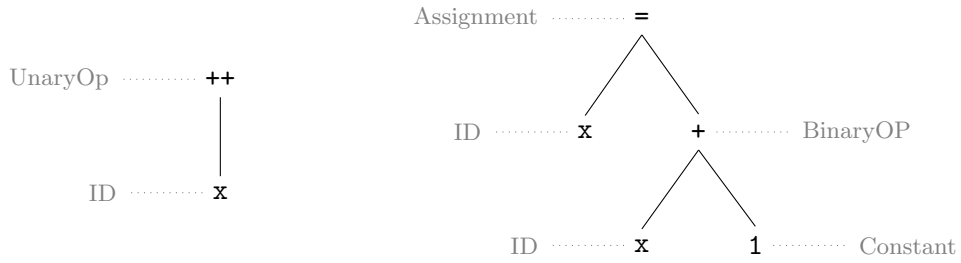

FIGURE 5.3: AST representation for the transformation from Fig. 5.2.

Similarly, the output is untranslated for any other node type: being `visit_UnaryOp` the only method in the module, all other visits are inherited from `Translator`, whence in turn from `pycparser`.

Figure 5.3 shows two AST fragments for a snippet of code affected by the transformation of module `test` from Figure 5.2: the AST on the left is generated from the relevant part of the input of the module, before the transformation; the AST on the right encodes the corresponding transformed code from the output of the module, after the transformation.

Note the simple example described above serves only to provide a first glance at the source translation mechanism implemented by CSeq. The module has indeed several flaws that would prevent it from being used reliably on anything else than very simple programs; these issues are not for us to worry about at the moment and will be discussed later in this section.

A Parameterised `Translator`

Figure 5.4 shows a module for a parameterised transformation. The module replaces every call to a given function into a call to another given function.

The functions identifiers are declared as parameters in the `init` method and can be set by the user externally, as shown later on in section. For now, let us concentrate on the source transformation, assuming that the module execution method, `loadfromstring`, correctly loads the source and destination function identifiers and stores them into two strings, respectively as `oldname` and `newname`, as instance attributes of class `rename`.

The AST node for function calls has two children: `name`, which stores the actual function identifier, and `args`, the root node of a subtree that represents the arguments of the call. Method `visit_FuncCall` is invoked whenever a function call is found during the AST visit. The method is overridden in order to change its behaviour and implement the transformation. The `if` statement in `visit_FuncCall` detects when the name of the called function matches the old name `self.oldname`, in which case the name is changed to the string stored in `self.newname` by overwriting the value of `fref` previously calculated by the first statement of the module.

```
import core.module

class rename(core.module.Translator):
   def init(self):
      self.addInputParam("old","source function","s","a",False)
      self.addInputParam("new","destination function","d","b",False)

   def loadfromstring(self,string,env):
      self.oldname = self.getInputParamValue("old")
      self.newname = self.getInputParamValue("new")
      super(rename,self).loadfromstring(string,env)

   def visit_FuncCall(self,n):
      fref = self._parenthesize_unless_simple(n.name)
      args = self.visit(n.args)

      if fref == self.oldname:
         fref = self.newname

      return fref + "(" + args + ")"
```

FIGURE 5.4: Parameterised source transformation module: function call renaming.

Setting-up a Configuration

Let us now briefly re-consider the argument passing mechanism to show how it can allow the transformation of Figure 5.4 to work for any two given function identifiers. By default the module transforms any call to function `a` into a call to function `b`. During the module initialisation phase, the front-end finds the two parameters declared in the `init` method of the module and automatically adds them as command-line arguments. The function identifiers can thus be set by using the `--old` and `--new` command-line options.

To wrap this up as a standalone tool, we write a configuration definition file, say `rename_test`, containing `rename` as the only module. Then, the following command:

```
cseq.py -i input.c --old f --new g -l rename_test
```

has the effect of applying the transformation to file `input.c` and change all the function calls to `f` into calls to function `g`.

This toy tool can only make one renaming at once. A possible solution to extend it to make multiple renamings in a single pass would be to change the two input parameters to file names rather than variable names, and then store in these file names the function names to replace.

Suggestions

In order to preserve syntactic correctness and semantic equivalence with the input source code, particular attention to detail and corner-cases is essential when writing a new transformation module. The transformation implemented by the module in Figure 5.2, for instance, changes blindly the syntactic category from expression to statement, and can easily break the syntax due to nested expressions.

A problem with the example from Figure 5.4 is that it does not check whether the destination function identifier was already being used in the input source code. This can introduce syntax errors in the transformed program, for instance due to calls to the same function using inconsistent signatures. A basic symbol table lookup to validate the module parameters can easily fix this issue. In general, since coding oversights may lead to non-trivial mistakes, formal reasoning about correctness may be needed on more convoluted transformation steps. This is the case of the lazy sequentialization schema presented in Chapter 4 and implemented in module `lazyseq` of CSeq.

Splitting complex translations into multiple steps instead of just writing a single one-pass task can be convenient. The idea is to devise a sequence of small transformations of increasing complexity and with a progressively restricted input syntax, starting with simple syntactic reductions and then gradually shifting to more elaborate transformations. Assuming a reduced input syntax generally keeps the structure of the module compact and readable, minimising potential corner cases and allowing quick adjustments when needed.

A loop unrolling module could, for instance, assume that there are only `for` loops in the input. This would avoid writing AST transformations for `do..while` and `while` nodes, making the module considerably more compact, shrinking its code down to roughly one third, in practice. Clearly, this would not work on unrestricted inputs. The input would need to be first processed by another module that trivially changes all loops into equivalent `for` statements.

Consider the function inlining module from Figure 5.5. Given the simplicity of the basic case shown in the example, one might be tempted to conclude that a simple repositioning of the function body will do the job. However, a robust implementation requires more attention. For example, if the function call occurs inside the condition block of an `if` statement, the expanded function body would end up in the condition block, possibly breaking the syntax. Nevertheless, to keep the inlining simple one could assume the absence of function calls from within condition blocks anyway; another module, executed before the actual inlining, could move the function calls outside the condition block by introducing additional Boolean variables.

(a) original function call

```
1:  int f(int i)
2:  {
3:     return 123+i;
4:  }
5:
6:  main() {
7:     ...
8:     x = f(x);
9:     ...
10: }
```

(b) function `f` inlined

```
1:  main() {
2:     ...
3:     int _f;
4:     {
5:        int _i;
6:        _i=x;
7:        {
8:           _f = 123+_i;
9:           goto L;
10:       }
11:       L: ;
12:    }
13:    x = _f;
14:    ...
15: }
```

FIGURE 5.5: Function inlining (simplified example).

5.4 Built-in Modules

In this section, we briefly describe CSeq's built-in modules for function inlining, loop unrolling, and backend instrumentation.

5.4.1 Function Inlining and Loop Unrolling

The `inliner` module implements function inlining [AJ88]. A basic example was already shown in Figure 5.5, where the left and right parts show the input and output of the module, respectively. Roughly, the function definition (declaration and body) is removed, and the function call is replaced with the function body. Additional variables are introduced to simulate the passing of arguments, if any. Functions with an undefined body (for example, `extern` functions) or declared as atomic using the prefixes `__VERIFIER_atomic_` or `__CSEQ_atomic_`, are not inlined.

Loop unrolling [DAC71, Sar01] is provided by module `unroller`. Consider the loop in Figure 5.6(a). An example of full unfolding in three iterations is given in Figure 5.6(b). The output code starts by first copying the initialisation statement `k=0`, followed by a copy of the compound statement representing the body of the loop. Similarly, further unfolding iterations will replicate the increment statement `k++` followed again by the loop body. Note that this is potentially incorrect if the loop body updates `k`, thus fully unfolding according to this schema should check that this never happens in the loop body.

In general, when the loop condition is a more complex expression, a *loop unwind bound* on the number of unfolding iteration is set upfront, such that the loop body is duplicated exactly as many times as required and regardless of the loop condition. Figure 5.6(c)

(a) initial loop

```
1:  main() {
2:    int k;
3:
4:    for (k=0;k<3;k++)
5:    {
6:      ...
7:    }
8:  }
```

(b) full unfolding

```
1:  main() {
2:    int k;
3:
4:    k=0;
5:    {
6:      ...
7:    }
8:    k++;
9:    {
10:     ...
11:   }
12:   k++;
13:   {
14:     ...
15:   }
16: }
```

(c) bounded unfolding

```
1:  main() {
2:    int k;
3:
4:    k=0; if(!(k<3)) goto L;
5:    {
6:      ...
7:    }
8:    k++; if(!(k<3)) goto L;
9:    {
10:     ...
11:   }
12:   assert(!(k<3));
13:   L: ;
14: }
```

FIGURE 5.6: Loop unrolling (simplified example).

shows a bounded unfolding (in two iterations) of the same loop. The guarded jumps simulate the loop condition check ($k < 3$) by moving the control at the end of the unfolding as soon as the condition is no longer satisfied, following the same behaviour of the original code.

An assertion with the negated loop condition is added after the last copy of the loop's body (*loop unwind assertion*), so to generate an error when the loop has not been unwound a sufficient number of times. Alternatively, runs longer than the bound can be silently discarded by using an `assume` statement instead.

5.4.2 Backend Instrumentation

Instrumenting the code for a specific backend is in itself a simple standalone transformation undertaken by the `instrumenter` module and consists in replacing the primitives for modelling non-determinism, assumptions and assertions (formally defined in Section 2.2), potentially inserted at any point during the translation, with analogous backend-specific statements. The backends supported by CSeq's standard program instrumentation module are the following:

- bounded model-checkers: `blitz` [CDS13], `cbmc` [CKL04], `esbmc` [CFM12], `llbmc` [MFS12]
- abstraction-based tools: `cpachecker` [BK11], `satabs` [CKSY05]
- symbolic testing tools: `klee` [CDE08].

(a) bounded model-checking: CBMC

```
assert();
__CPROVER_assume();
extern int __nondet_int(void);
```

(b) abstraction-based: CPAchecker

```
void __assert(int x) { if(!(x)) { ERROR: goto ERROR; } }
void __assume(int x) { while(!(x)); }
extern int __nondet_int(void);
```

(c) symbolic testing: KLEE

```
__KLEE_assert();
void __assume(int x) { while(!(x)); }
int __nondet_int() { int x; klee_make_symbolic(&x,sizeof(x),"x"); return x; }
```

FIGURE 5.7: Backend instrumentation. For the two `nondet_int` declared as `extern` the backend already considers every possible return value due to the missing body, so there is no need to define them (`extern` is used to avoid warnings from the backend); all the other functions with a missing body correspond to primitives natively modelled by the backend.

Although non-determinism is typical in modelling languages, and tools for software analysis usually support it, there is no common standard for a precise set of primitives and their semantics. Depending on whether the given verification backend natively models such primitives, the instrumentation requires either a simple function call renaming, or also inserting ad-hoc function definitions.

Figure 5.7 shows the basic differences in the instrumentation for the different families of backend supported. Bounded model-checkers generally model all the needed primitives natively. Explicit implementation of `assume` and `assert` is, however, needed for CPAchecker. We have observed that this is often the case for other tools based on abstraction. The figure also shows the instrumentation for the only testing tool supported, KLEE, which handles assertions, but requires implementations for `assume` and non-deterministic functions. Note that `klee_make_symbolic` can be used for all other data types too.

5.5 Lazy-CSeq

Lazy-CSeq implements the lazy sequentialization schema presented in Chapter 4 for multi-threaded C program.

Figure 5.8 shows the module layout, the argument flow, and a sketch of the translation. The lazy sequentialization schema is implemented by a source translation module, box

LAZY in the diagram. Since the schema only works for bounded programs, modules for function inlining and loop unrolling need to be inserted into the configuration before the sequentialization module.

The complete configuration is defined by a sequence of 18 modules (16 `Translator` and two `Wrapper` modules), which can be conceptually grouped according to the following categories:

1. the source merging module, followed by eight simple transformation modules, the purpose of which is to rewrite the input program into a progressively simplified syntax, so to make it easier to implement the more complex transformations occurring later in the sequence;
2. four `Translator`s for program flattening (that include improved versions of loop unrolling and function inlining modules presented in Section 5.4) to produce a bounded multi-threaded program that is equivalent to the input program up to the given unwind bound;
3. a module implementing the lazy sequentialization schema described in Chapter 4 that yields a backend-independent sequentialized file;
4. standard program instrumentation (discussed in Section 5.4) to instrument the sequentialized file for a specific backend;
5. two `Wrapper`s for backend invocation and user report generation or counterexample translation (presented in Section 5.5.1).

Double-framed boxes in the diagram denote groups of modules. The input and output of the tool are boxes (i) and (vii), respectively. Boxes (ii) to (vi) represent the output of intermediate modules in the configuration. In particular, boxes (ii) to (v) sketch the structure of the output file resulting from the execution of each of the group of translation modules shown above them. Boxes (vi) and (vii) represent the original and translated counterexample traces, respectively. The counterexample from the backend (vi) that refers to the sequentialized file (v) is translated by module `cex` into another counterexample (vii) that refers to the actual input (i).

5.5.1 Counterexample Generation

One of the main usability limitations of sequentialization-based tools is in that when an error is found the error trace is too hard to follow because the counterexample produced by the backend actually refers to the sequentialized file. The built-in `cex` counterexample generation module generates counterexamples that instead refer to the actual input code. Counterexample generation is built on top of CSeq's line-mapping described in

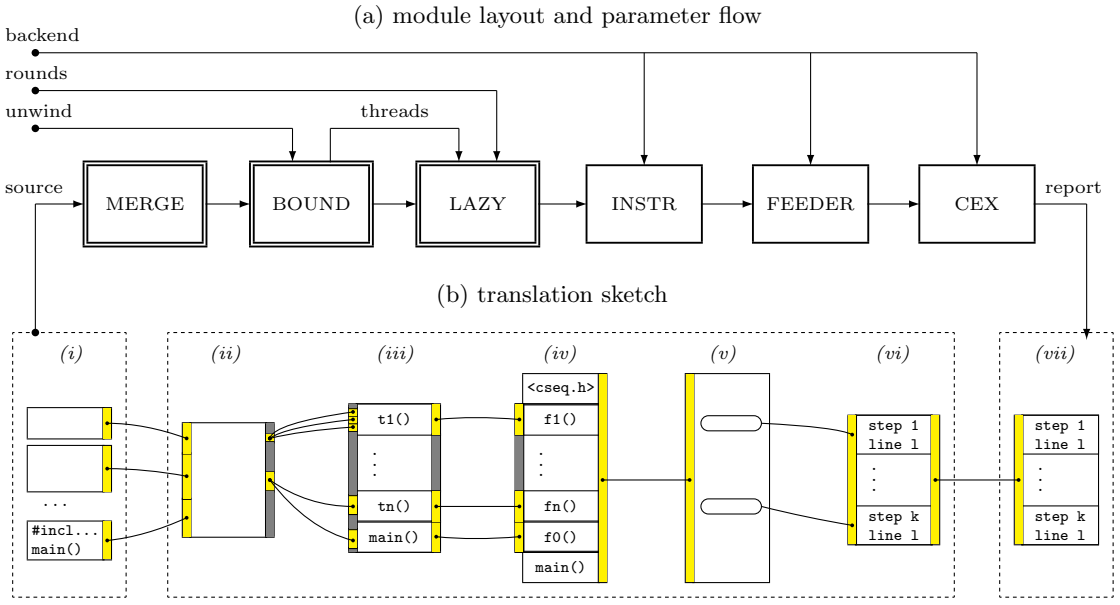

FIGURE 5.8: Lazy-CSeq: module layout and translation sketch.
(a) boxes from MERGE to CEX show the module configuration, with double-border boxes denoting groups of modules. (b) boxes (*i*) to (*vii*) represent the input and output of the modules, or group of modules, above them (for example, (*v*) and (*vi*) represent the input and output, respectively, of module FEEDER).

Section 5.3.2. It is currently only supported for the Lazy-CSeq configuration and the default backend (i.e., CBMC). Note, however, that the line-mapping facility provided by the framework is backend-independent and translation-independent, thus it can be used in other settings. The counterexample module itself may be or may be not easy to adapt to other sequentializations schemas and other backends.

Let us now consider the hypothetical tool's execution shown in Figure 5.8(b). Observe that box (*vi*) is a counterexample for the sequentialized file (*v*). Module `cex` changes that file into another counterexample (*vi*) that actually refer to the original input file (*i*).

To do so, `cex` translates one by one the state transitions listed in the counterexample returned by the backend by tracing back line numbers to their corresponding input coordinates using CSeq's linemapping, and then showing the amended transitions in the same order. Specifically, the translation is done according to the three cases discussed below.

If the output line is traced back to a thread statement (e.g., `lock`, `create`, etc), we append to the new counterexample an intermediate transition to explicitly show concurrency-specific details, such as context-switching, changes to lock conditions, and the like. If the output line is traced back to any other statement, we amend the transition description with the mapped line number and append it to the counterexample. If the output line cannot be mapped back (as explained in Section 5.3.2), it must come from thread simulation code injected during the sequentialization (see Figure 4.2). Therefore the

transition indicated in the counterexample refers to an computation that was performed to simulate concurrency and does not correspond to any transition in the original program (i). Therefore, these transitions can be safely ignored.

5.5.2 Usage

Lazy-CSeq can be executed by invoking CSeq with the `lazy` configuration, through the command:

```
cseq.py -i input.c -l lazy
```

to analyze the file `input.c` and check for reachable error states determined by an `ERROR` label, an assertion failure, or incorrect use of locks, using the default analysis parameters and the default backend. Deadlock checking is off by default and can be enabled with `--deadlock`.

The analysis parameters are the loop unwinding depth and the number of rounds. The default value is `1` for both and can be changed with `--unwind k` and `--rounds k`, respectively. The default backend is `cbmc` and it can be changed using `--backend b` where `b` is any backend supported by the instrumentation module as described in Section 5.4.2.

Our lazy sequentialization schema is tailored to bounded model-checkers as backends. However, since the `instrumenter` module also supports abstraction-based and testing backends, these can still be tried, and occasionally might work well on some test cases. However, achieving accurate analysis would require substantial changes to the sequentialization schema.

The option `--rounds` uses standard round-robin schedules. This can be replaced with restricted schedules using `--schedule r1:...:rn`, which gives schedule restrictions for `n` rounds, as described in Section 4.5.

By default Lazy-CSeq does not generate counterexamples. Counterexample generation can be enabled by using the default backend with the `--cex` option. Alternatively, `--linemap` will show the line mapping table across all source transformation steps.

Chapter 6

Conclusions

6.1 Summary of Work

In this thesis we have investigated on the effectiveness of combining BMC and sequentialization for finding errors in real-world concurrent software, by targeting the largely representative category of multi-threaded C programs with POSIX threads.

We have implemented and evaluated the Lal-Reps sequentialization schema [LR09], in previous empirical work advocated as a suitable technique to complement BMC [GHR10]. In contrast, we have identified several major drawbacks that prevent LR from actually being used on non-idealised software, and in particular we have discussed the need for lazy techniques and the reasons why they can improve backend integration and analysis performance when used on top of BMC.

We have developed a novel lazy sequentialization schema specifically tailored to BMC and provided an extensive empirical evidence of its superiority over our own implementation of the Lal-Reps schema, and its high level of competitiveness with the state-of-the-art bug-hunting tools with built-in concurrency handling. Our tool Lazy-CSeq [ITF+14b, ITF+14a] is aggressively optimised for fast bug finding, and has won the gold medal in the concurrency category in the last two editions of SV-COMP [Bey14, Bey15]. In our tool we have implemented both the program unfolding and the sequentialization using source translations, therefore once again providing evidence that reasoning at the level of the source code can be very beneficial for optimising the program analysis process. With regard to BMC-based techniques, a major insight is that integrating context-bounding within the program unfolding stage can possibly represent one of the most competitive approaches for finding bugs in concurrent software, considering that errors typically occur within a few context switches [MQ07, QW04, TDB14].

We have presented our framework for fast prototyping and development of sequentialization-based tools that subsumes our experience in working on sequentializations. CSeq emphasises the desirable aspects of sequentialization, and in general of source translation, by encouraging a modular approach to designing new schemas, and providing a range of built-in functionalities to reduce the overall engineering effort to build a new tool. During the development of Lazy-CSeq, the framework has been very useful by solidly and comfortably supporting the intricate reasonings at the level of the source code needed to refine our schema. CSeq has also been used to develop other tools not discussed in this thesis [TIF+14, TIF+15, NFLP15] within very compressed development time frames and with a relatively modest effort in comparison to what it would have been required using lower-level program representations. In general, source transformation can be a very expressive, flexible and powerful method for gathering deep insights on the nature of programs, and can thus yield significant improvements to program analysis. Our view is shared by others [Cad15].

6.2 Future Work

The aggressive performance optimisations on our lazy schema were only possible because we concentrated on a specific backend technology. Targeting other families of backends, for instance abstraction-based tools, might be interesting. A few simple, preliminary experiments using our lazy schema in combination with abstraction-based backends produced (not surprisingly) a significant amount of incorrect results due to the variables modelling the program counters of the simulated threads being too coarsely abstracted. As the next step in this direction one could change the translation in such a way to force more or less precise representations of the program counters (while keeping the same level of abstraction on the rest of the program) to see to what extent the overall accuracy of the analysis can be improved, and to understand the possible trade-offs between accuracy and performance.

During our experiments with sequentialization we have observed different program features that can negatively affect the performance of the backend, such as: extensive use of pointers and dynamic memory allocation, number of visible statements, complexity of control-flow structure, number of threads, size of the shared or local memory, use of synchronisation primitives, length of the shortest schedule to find the error, and so on. A systematic study on how these features affect the performance of different backend technologies (not limited to BMC) in combination with different sequentializations could highlight valuable insights.

In particular, we have noticed that on very large instances sometimes bounded model-checkers even struggle to start the analysis (i.e., they fail to generate the verification condition), thus leaving testing as the only possibility to look for bugs. However, testing

is only adequate to detect errors that show up with high probability. Issues that only show rarely are unlikely to be found in large programs. A detailed investigation on well-selected test cases that are currently out of reach for BMC could be the starting point to advance the state of the art.

Partitioning the program's state space is potentially useful on complex instances as it can spread the computational load over multiple machines. It would be interesting to experiment with partitioning at a sequentialization level (for example by partitioning the schedules, as sketched in Section 4.5), and in particular to estimate the overhead due to the redundant computations performed on similar verification conditions representing distinct partitions, and to appreciate the suitability of incremental BMC techniques [SKB+14] in this setting.

Further possible directions for future research consist in extending our existing implementations or designing new schemas to: handle other memory models than sequential consistency to capture the subtle process interactions due to modern hardware designs (previous work has already shown how to reduce weaker memory models to SC using source transformations [ABP11, AKNT13]), support other concurrency models (for instance message-passing, perhaps by just transforming it to multi-threaded systems), and using partial orders [FG05, KWG09] to reduce the number of simulated thread interleavings.

Bibliography

[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.

[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.

[ÁH14] Erika Ábrahám and Klaus Havelund, editors. *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*. Springer, 2014.

[AJ88] Randy Allen and Steve Johnson. Compiling C for vectorization, parallelization, and inline expansion. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 241–249. ACM, 1988.

[AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 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 7792 of *Lecture Notes in Computer Science*, pages 512–532. Springer, 2013.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 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.

[BBdH+09] Thomas Ball, Sebastian Burckhardt, Jonathan de Halleux, Madanlal Musuvathi, and Shaz Qadeer. Deconstructing concurrency heisenbugs. In *ICSE Companion*, pages 403–404. IEEE, 2009.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[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.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.

[BE14] Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. *STTT*, 16(2):127–146, 2014.

[BEP11] Ahmed Bouajjani, Michael Emmi, and Gennaro Parlato. On sequentializing concurrent programs. In Yahav [Yah11], pages 129–145.

[Bey13] Dirk Beyer. Second competition on software verification - (summary of sv-comp 2013). In *TACAS*, pages 594–609, 2013.

[Bey14] Dirk Beyer. Status report on software verification - (competition summary SV-COMP 2014). In Ábrahám and Havelund [ÁH14], pages 373–388.

[Bey15] Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In 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*, pages 401–416. Springer, 2015.

[BF94] Yosi Ben-Asher and Eitan Farchi. Using true concurrency to model execution of parallel programs. *International Journal of Parallel Programming*, 22(4):375–407, 1994.

[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.

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

[BM09] Ahmed Bouajjani and Oded Maler, editors. *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*. Springer, 2009.

[BML+13] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. The cetus source-to-source compiler infrastructure: Overview and evaluation. *International Journal of Parallel Programming*, 41(6):753–767, 2013.

[BNSV14] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In Dimitra Giannakopoulou and Gwen Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, volume 8702 of *Lecture Notes in Computer Science*, pages 271–277. Springer, 2014.

[Boe05] Hans-Juergen Boehm. Threads cannot be implemented as a library. 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 261–268. ACM, 2005.

[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 *ICSE*, pages 625–634. IEEE Computer Society, 2004.

[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.

[BS11] 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*. ACM, 2011.

[Cad15] Cristian Cadar. Targeted program transformations for symbolic execution. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 906–909. ACM, 2015.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Draves and van Renesse [DvR08], pages 209–224.

[CDS13] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. Blitz: Compositional bounded model checking for real-world programs. In Denney et al. [DBZ13], pages 136–146.

[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 C. 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, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 331–340. ACM, 2011.

[CFM12] Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. Smt-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012.

[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.

[CGS11] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Time-bounded analysis of real-time systems. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 72–80. FMCAD Inc., 2011.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 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, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

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

[Cor06] James R. Cordy. The TXL source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[DAC71] International Business Machines Corporation. Research Division, F.E. Allen, and J. Cocke. *A Catalogue of Optimizing Transformations*. 1971.

[DBZ13] Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. IEEE, 2013.

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In Yahav [Yah11], pages 351–368.

[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.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[DvR08] Richard Draves and Robbert van Renesse, editors. *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008.

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Software Eng.*, 28(12):1146–1170, 2002.

[EQR11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In Ball and Sagiv [BS11], pages 411–422.

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

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles*

of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, pages 110–121. ACM, 2005.

[FIP13a] Bernd Fischer, Omar Inverso, and Gennaro Parlato. CSeq: A concurrency pre-processor for sequential C verification tools. In Denney et al. [DBZ13], pages 710–713.

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

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian partial-order reduction. In Dragan Bosnacki and Stefan Edelkamp, editors, *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*, volume 4595 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2007.

[GG08] Malay K. Ganai and Aarti Gupta. Efficient modeling of concurrent systems in BMC. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software, 15th International SPIN Workshop, Los Angeles, CA, USA, August 10-12, 2008, Proceedings*, volume 5156 of *Lecture Notes in Computer Science*, pages 114–133. Springer, 2008.

[GHR10] Naghmeh Ghafari, Alan J. Hu, and Zvonimir Rakamaric. Context-bounded translations for concurrent software: An empirical evaluation. In Jaco van de Pol and Michael Weber, editors, *SPIN*, volume 6349 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2010.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *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. ACM Press, 1997.

[God05] Patrice Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[GPFW96] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1996.

[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.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[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 (Tool Demonstration). In *ASE*, page to appear, 2015.

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

[ISO11] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C.* International Organization for Standardization, Geneva, Switzerland, December 2011.

[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 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 585–602. Springer, 2014.

[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 Ábrahám and Havelund [ÁH14], pages 398–401.

[Joh75] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1975.

[KOS+11] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. Linear completeness thresholds for bounded model checking. In Gopalakrishnan and Qadeer [GQ11], pages 557–572.

[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.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[KWG09] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In Bouajjani and Maler [BM09], pages 398–413.

[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.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam79] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.

[LMP09a] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In Bouajjani and Maler [BM09], pages 477–492.

[LMP09b] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. Analyzing recursive programs using a fixed-point calculus. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 211–222. ACM, 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 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 of *EPTCS*, pages 34–47, 2012.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and Sanjit A. Seshia, editors, *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 427–443. Springer, 2012.

[LQR09] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using smt solvers. In Bouajjani and Maler [BM09], pages 509–524.

[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.

[MFS12] Florian Merz, Stephan Falke, and Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *VSTTE*, volume 7152 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2012.

[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. ACM, 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 Draves and van Renesse [DvR08], pages 267–280.

[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.

[Mül06] Markus Müller-Olm. *Variations on Constants - Flow Analysis of Sequential and Parallel Programs*, volume 3800 of *Lecture Notes in Computer Science*. 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 *TACAS*, volume 9035 of *LNCS*, pages 461–463. Springer, 2015.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble,*

France, April 8-12, 2002, Proceedings, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[PR13] Corneliu Popeea and Andrey Rybalchenko. Threader: A verifier for multi-threaded programs - (competition contribution). In *TACAS*, pages 633–636, 2013.

[Pra86] Vaughan Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[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 *LCPC*, volume 2624 of *LNCS*, pages 383–394. Springer, 2001.

[QW04] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In William Pugh and Craig Chambers, editors, *PLDI*, pages 14–24. ACM, 2004.

[RG05] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005.

[Sar01] Vivek Sarkar. Optimized unrolling of nested loops. *International Journal of Parallel Programming*, 29(5):545–581, 2001.

[SKB+14] Peter Schrammel, Daniel Kroening, Martin Brain, Ruben Martins, Tino Teige, and Tom Bienmüller. Incremental bounded model checking for embedded software (extended version). *CoRR*, abs/1409.5872, 2014.

[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.

[SW05] Richard M. Stallman and Zachary Weinberg. *The C Preprocessor*, 2005.

[SW10] Nishant Sinha and Chao Wang. Staged concurrent program analysis. In Gruia-Catalin Roman and Kevin J. Sullivan, editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, pages 47–56. ACM, 2010.

[SW11] Nishant Sinha and Chao Wang. On interference abstractions. In Ball and Sagiv [BS11], pages 423–434.

[TDB14] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using schedule bounding: an empirical study. In José Moreira and James R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 15–28. ACM, 2014.

[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 Ábrahám and Havelund [ÁH14], pages 402–404.

[TIF+15] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Verifying concurrent programs by memory unwinding. In *TACAS*, volume 9035 of *LNCS*, pages 551–565. Springer, 2015.

[Tse68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

[Val89] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[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.

[Yah11] Eran Yahav, editor. *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*. Springer, 2011.

[ZM02] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002.