Predicate abstraction is a key enabling technology for applying model checkers to programs written in mainstream languages. It has been used very successfully for debugging sequential system-level C code. Although model checking was originally designed for analysing concurrent systems, there is little evidence of fruitful applications of predicate abstraction to shared-variable concurrent software. The goal of the present thesis is to close this gap.

Monotonicity in Shared-Memory Program Verification

Alexander Kaiser

Alexander Kaiser

Monotonicity in Shared-Memory Program Verification

A thesis submitted for the degree of Doctor of Philosophy

Kellogg College, University of Oxford, Trinity term 2013

Für Julia und Emelie*

* . . . und ihr noch namenloses Geschwisterchen

Abstract

Predicate abstraction is a key enabling technology for applying model checkers to programs written in mainstream languages. It has been used very successfully for debugging sequential system-level C code. Although model checking was originally designed for analysing concurrent systems, there is little evidence of fruitful applications of predicate abstraction to shared-variable concurrent software. The goal of the present thesis is to close this gap. We propose an algorithmic solution implementing predicate abstraction that targets safety properties in non-recursive programs executed by an *unbounded* number of threads, which communicate via shared memory or higher-level mechanisms, such as mutexes and broadcasts. As system-level code makes frequent use of such primitives, their correct usage is critical to ensure reliability.

Monotonicity is a natural and common feature of human-written concurrent software. It is also useful: if every thread's memory is finite, monotonicity often guarantees the decidability of safety properties even when the number of running threads is unspecified. In this thesis, we show that the process of obtaining finite-data thread abstractions for model checking is not always compatible with monotonicity. Predicate-abstracting certain mainstream asynchronous software such as the ticket busy-wait lock algorithm results in non-monotone multi-threaded Boolean programs, despite the monotonicity of the input program: the monotonicity is *lost in the abstraction*. As a result, the unbounded-thread Boolean programs do not give rise to well quasi-ordered systems [1], for which sound and complete safety checking algorithms are available. In fact, safety checking turns out to be *undecidable* for the obtained class of abstract programs, despite the finiteness of the individual threads' state spaces. Our solution is to restore the monotonicity in the abstraction, using an inexpensive closure operator that precisely preserves all safety properties from the (non-monotone) abstract program without the closure.

As a second contribution, we present a novel, sound and complete, yet empirically much improved algorithm for verifying abstractions, applicable to general well quasi-ordered systems. Our approach is to gradually *widen* the set of safety queries during the search by program states that involve fewer threads and are thus easier to decide, and are likely to finalise the decision on earlier queries. To counter the negative impact of "bad guesses" the search is supported by a engine that generates such states; these are never selected for widening.

We present an implementation of our techniques and extensive experiments on multi-threaded C programs, including device driver code from FreeBSD and Solaris. The experiments demonstrate that by exploiting monotonicity, model checking techniques—enabled by predicate abstraction—scale to realistic programs even of a few thousands of multi-threaded C code lines.

Acknowledgements

I would like to thank my supervisor Daniel Kroening without whom this work would have not been possible. It has been a great opportunity to be in his group. I owe sincere and earnest thankfulness to my co-supervisor Thomas Wahl, who supported me throughout my thesis, even after he left Oxford for a position at the Northeastern University in Boston. By setting up a 3 1/2 year studentship, he offered me the chance to pursue my studies without financial concerns. In particular, this thesis would not have been possible without the financial support of the Engineering and Physical Sciences Research Council.*

My thanks also go to various members of the Oxford Computer Science Department with whom I have interacted during the course of my studies, in particular to Gérard Basler, Vijay D'Silva, Alastair Donaldson, Ramon Granell Leopold Haller, Stefan Kiefer, Vincent Nimal, Hristina Palikareva, Philipp Rümmer, Michael Tautschnig and Georg Weissenbacher. I also want to express my thanks to Luke Ong and Manuel Oriol, for the discussion and feedback that made my viva such a pleasurable experience.

Special thanks to Adele Skowronski, Jochen Vollmer, Richard Kaiser and Robert Marschinski for their careful proofreading, suggestions and corrections, and to Helga Mayer-Kaiser and Ulrike Reichle for taking care of my daughter when time was running short.

Completing this dissertation has been a challenging task for which I have relied greatly on the wonderful Julia Vollmer and my daughter Emelie (for her enduring energy to remind me to take breaks), my family and friends.

Alexander Kaiser
University of Oxford, July 2013

*EPSRC project EP/G026254/1.

Contents

List of Figures

List of Tables

1 Introduction

Multi-threading is becoming the premier technology for accelerating computations in a post frequency-scaling era. The widespread availability of thread libraries for mainstream languages including C and Java, as well as for all major operating systems, makes this technology easily accessible, which can in turn entrap the inexperienced programmer to create code with puzzling and irreproducible behaviour. The programming language community is called upon to help prevent a renewed software crisis, by providing formal technology that reliably analyses concurrent programs, and exposes potential bugs before deployment.

The object of study in this thesis are non-recursive procedures executed by multiple threads (e.g. dynamically generated, and possibly unbounded in number), which communicate via shared variables or higher-level mechanisms such as mutexes, and broadcasts on condition variables. OS-level code, including Windows, UNIX and Mac OS device drivers, makes frequent use of such concurrency APIs, whose correct use is therefore critical to ensure a reliable programming environment. An example of thread communication via mutexes and POSIX-style broadcasts in the FreeBSD operating system is shown in Figure 1.

```
static u_int
akbd_read_char(keyboard_t *kbd, ...) {
  //...
  sc = (struct adb_kbd_softc *)(kbd);
  mtx_lock(&sc->sc_mutex);
  if (!sc->buffers && wait)
    cv_wait(&sc->sc_cv,&sc->sc_mutex);
  if (!sc->buffers) {
    mtx_unlock(&sc->sc_mutex);
    return (0); }
  adb_code = sc->buffer[0];
  for (i = 1; i < sc->buffers; i++)
    sc->buffer[i-1] = sc->buffer[i];
  sc->buffers--;
  key = adb_code;
  mtx_unlock(&sc->sc_mutex);
  return (key); }
```

```
static u_int
adb_kbd_receive_packet(device_t dev, ...) {
  struct adb_kbd_softc *sc;
  sc = device_get_softc(dev);
  if (command != ADB_COMMAND_TALK)
    return 0;
  if (reg != 0 || len != 2)
  {
    return (0);
  }
  mtx_lock(&sc->sc_mutex);
  kbd = kbd_get_keyboard(/*...*/);
  //...
  mtx_unlock(&sc->sc_mutex);
  cv_broadcast(&sc->sc_cv);
  //...
  return (0); }
```

Figure 1: **Apple Bus protocol** — Code fragment of a recent implementation of an Apple Bus protocol in the FreeBSD OS, involving kernel thread synchronisation via mutexes and POSIX-style broadcasts; functions `mtx_(un)lock`, `cv_wait` and `cv_broadcast`

Based on the celebrated success achieved with sequential programs, we propose in this thesis an extension of *predicate abstraction* to multi-threaded programs that enables thread-specific reasoning about intricate data relationships. The utility of predicate abstraction is known to depend critically on the choice of predicates: the consequences of a poor choice range from inferior performance to flat-out unprovability of certain properties. Our abstraction method permits predicates expressing constraints between shared (global-scope) variables, the active (executing) thread's local variables, and local variables of a passive (non-executing) thread. Our predicate language hence supports

(i) *shared-variable* relationships, e.g. "shared variables `s` and `t` are equal",
(ii) *single-thread* relationships, e.g. "local variable `m` of thread i is less than shared variable `s`", and
(iii) *inter-thread* relationships, e.g. "local variable `m` of thread i is less than variable `m` *in all other threads*".

Why such a rich predicate language? For many mainstream concurrent algorithms such as the widely used *ticket* busy-wait lock algorithm (the default locking mechanism in the Linux kernel since 2008; see Figure 5), the verification of elementary safety properties **requires** such single- and inter-thread relationships. Another example is the pattern exhibited by the CUDA program in Figure 2. Many CUDA programs operate correctly only if every thread identifier idx is unique among the threads (otherwise some entries may accidentally be operated on multiple times). The uniqueness property is an inter-thread relationship. As we demonstrate, the ticket lock algorithm can in fact be verified by predicate abstraction with respect to inter-thread predicates, at this point for a finite number of threads. Similarly, tracking the success of compare-and-swap primitives requires single-thread relationships, namely through predicate $\mathtt{s} = \mathtt{m}$ in the implementation in Figure 3; the same predicate is needed to prove that the assertion in the atomic counter in Figure 4 cannot fail.

In the first main part of the thesis, we address the problem of full parameterised (unbounded-thread) program verification with respect to our rich predicate language. Such reasoning requires first that the n-thread abstract program $\hat{\mathcal{P}}^n$, obtained by existential predicate abstraction, is rewritten into a single template program $\tilde{\mathcal{P}}$ to be executed by (any number of) multiple threads. A consequence of the use of single- or inter-thread predicates in the abstraction is that the obtained programs $\hat{\mathcal{P}}^n$ are no longer asynchronous. In order to capture the semantics of these

```
#include <cuda.h>

// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx]; }

//main routine that executes on the host
int main(void) { //...
  int num_threads = get_num_threads(); //how many thread shall execute
  int n_blocks = N/num_threads + (N%num_threads == 0 ? 0:1);
  //do calculation on device
  square_array <<< n_blocks, num_threads >>> (a_d, N);
  //...
```

Figure 2: **CUDA program** — The program squares every element of a matrix using num_threads concurrent threads

programs in the template $\tilde{\mathcal{P}}$, the template programming language must itself permit variables that refer to the currently executing or a generic passive thread; we call such programs *dual-reference (DR)*. We describe how to obtain $\tilde{\mathcal{P}}$, namely essentially as an overapproximation of $\hat{\mathcal{P}}^{\mathsf{b}}$, for a constant b that scales linearly with the number of inter-thread predicates used in the predicate abstraction.

Given the Boolean dual-reference program $\tilde{\mathcal{P}}$, and with classical results on *well quasi-ordered transition systems* in mind [2], we then expect the absence of recursion in the unbounded-thread replicated program $\tilde{\mathcal{P}}^{\infty}$ to guarantee decidability of safety properties. Somewhat surprisingly, however, this is not the case: the expressiveness of dual-reference programs renders parameterised program location reachability undecidable, despite the finite-domain variables. The root cause is the lack of *monotonicity* of the transition relation with respect to the standard partial order over the space of unbounded thread counters. That is, adding passive threads to the source state of a valid transition can invalidate this transition. Note that, in contrast,

```
// executes atomically
__atomic bool CAS(volatile natural& s, natural m, natural l) {
  if (s = m) { s := l; return true; }
  else { return false; } }
```

Figure 3: **Compare-and-swap function (CAS)** — The function compares the value of the *shared* variable `s` to a given *local* value `m` and, only if they are equal, modifies `s` to `l`

```
volatile natural s := 0; //shared counter
natural inc() { //executed by an unbounded number of threads
    natural m, l; //local
ℓ1: do { m := s;
ℓ2:     if(m = 0u−1) return 0; //overflow, increment failed
ℓ3:     l := m + 1;
ℓ4: } while (!CAS(s, m, l));
ℓ5: assert(s > m); //property to be checked
ℓ6: return l; } //increment succeeded
```

Figure 4: **Atomic counter using CAS [154]** — The assertion checks whether the shared counter is increased by each call to function `inc` that returns a non-zero value (`inc` is executable by an unbounded number of concurrent threads); the program is safe

the input C programs are asynchronous and thus perfectly monotone; we say the monotonicity is *lost in the abstraction*. As a result, our abstract programs are in fact not well quasi-ordered.

We address this problem by restoring the monotonicity using a simple *closure operator*, which enriches the transition relation of the abstract program $\tilde{\mathcal{P}}$ such that the obtained program $\tilde{\mathcal{P}}_m$ engenders a monotone (and thus well quasi-ordered) system *and* has the same safety properties as $\tilde{\mathcal{P}}$: the closure introduces no spurious error traces. The closure operator essentially terminates passive threads that block transitions allowed by other passive threads. This technique resembles earlier approaches that *enforce* (rather than restore) monotonicity in genuinely non-monotone systems [27, 3, 4, 136, 91], at the cost of overapproximating the transition relation. In contrast, we exploit the fact that the asynchronous input programs do feature monotonicity, with the result that the monotonicity closure $\tilde{\mathcal{P}}_m$ is safety-equivalent to the intermediate program $\tilde{\mathcal{P}}$.

Safety checking the enriched transition relation is decidable by reduction to the *coverability problem* for well quasi-ordered systems. In program terms, coverability of a given n-thread program state e consisting of shared variable values plus n valuations of the local variables, asks for the existence of a number $n' \geq n$ and the reachability of a state v in the n'-thread instantiation that differs from e only in that it contains $n' - n$ additional threads. Coverability thus allows us to express precisely the types of assertion or synchronisation failures we are after:

(i) "can a thread violate an assertion?", or
(ii) "can a critical section be concurrently accessed by two or more threads?" etc.

Coverability for the class of well quasi-ordered systems, which subsumes popular concurrency models such as various forms of Petri nets, is known to be Ackermann-hard. For example, for plain Petri nets and vector-addition systems, the problem was shown to be complete in exponential space [35]. Extensions such as *transfer transitions* or our (equivalent) monotone Boolean DR program model increase the complexity of the problem further [164]. These daunting computational costs, and the significance of the coverability problem in practical concurrent program verification, have led to a flurry of activity in crafting solutions viable in practice [75, 88, 86].

In the second main part of the thesis, we introduce a new solution to the coverability problem in well quasi-ordered systems that shares soundness and completeness with existing methods, but follows a fundamentally different search strategy: before a state is selected for expansion, the set of target elements is *widened* by its downward closure, thus adding to the target set many smaller (in the partial state order) elements whose coverability has not yet been decided. The motivation behind this approach is two-fold: (i) the coverability of smaller elements, which involve fewer threads, is often less costly to decide, and (ii) *un*coverability of smaller elements immediately implies uncoverability of any larger elements, including the original query target. The strategy to first settle the coverability of smallest-possible elements, side-stepping the original query, not only accelerates the search, but also makes the search structure more compact, as we will demonstrate.

"Bad guesses", i.e. elements in the widening that end up coverable, permit no conclusion about the coverability of the original query elements. In this case we backtrack out of the widening: all elements found to be coverable are purged from the search structure, and the search continues with another smallest-possible and currently undecided element. Such backtracking can impair the algorithm's efficiency. As a countermeasure, the backward search can be assisted by an engine that generates coverable elements; none of these are ever selected for widening. All we require of such an engine is that it soundly report coverability of elements. It does not need to be complete, as its job is only to accelerate the backward search by flagging coverable states early. An ideal candidate is a Karp-Miller like forward-directed search for Petri nets with transfer arcs, which generates (possibly incomplete) coverability information quickly.

Contributions*. To summarise, the central contribution of this thesis is a fully automated predicate abstraction strategy for unbounded-thread asynchronous C programs, with respect to the rich language of inter-thread predicates. This language allows the abstraction to track properties that are essentially universally quantified over all passive threads. To this end, we first develop such a strategy for a fixed number of threads. Second, in preparation for extending it to the unbounded case, we describe how the abstract model, obtained by existential predicate abstraction for a given thread count n, can be expressed as a template program that can be multiply instantiated. Third, we show a sound and complete algorithm for reachability analysis for the obtained parameterised Boolean dual-reference programs. We overcome the undecidability of the problem by building a monotone closure that enjoys the same safety properties as the original abstract dual-reference program. Fourth, we present a novel, sound and complete, yet empirically much improved algorithm for verifying monotone Boolean DR programs, applicable to general well quasi-ordered systems.

We have implemented our techniques in the publicly available infinite-state model checker monabs for concurrent C programs and the breach (back-end) model checker, and show extensive experimental results on system-level concurrent software and synchronisation algorithms. We make two observations: first, our tool is able to verify certain parameterised programs, such as the ticket lock algorithm, that are principally beyond *all* existing tools we are aware of [100, 37, 47, 176, 40, 95, 58, 69, 71]. The reasons vary from their inability to deal with unbounded threads, to the lack of support for inter-thread predicates. Second, our tool is, still *for unbounded threads*, often more efficient than other tools *for fixed and small thread counts*.

The tools and all benchmarks used in this work are publicly available:

- monabs (integrated with satabs*) at www.cprover.org/satabs and
- breach (aka. bfc) and all benchmarks at www.cprover.org/bfc, or alternatively at www.akaiser.net/bfc.

*Major contents of this thesis were previously presented in [59, 58, 117, 118, 119].

*Although monabs has been fully integrated with version 3.2 of the (originally sequential program) verifier satabs [43], and is not further maintained, we retain the initial name to avoid confusion.

```
struct Spinlock {
      natural s := 1; // ticket being served
      natural t := 1; }; // next free ticket

struct Spinlock lock; // shared

void spin_lock() {
      natural m := 0; // local
  ℓ1: m := fetch_and_add(lock.t);
  ℓ2: while (m ≠ lock.s)
            /* spin */; }

void spin_unlock() {
  ℓ3: lock.s++; }
```

Figure 5: **The ticket algorithm** — Shared variable lock has two integer components: `s` holds the ticket currently served (or, if none, the ticket served next), while `t` holds the ticket to be served after all waiting threads have had access. To request access to the locked region, a thread atomically retrieves the value of `t` and then increments `t`. The thread then busy-waits until local variable `m` agrees with shared `s`. To unlock, a thread increments `s`. Our goal is to verify "unbounded-thread mutual exclusion": no matter how many threads try to acquire and release the lock concurrently, no two of them should simultaneously be between the calls to functions spin_lock and spin_unlock

2 Background and Notation

In this section, we provide background on well-quasi-orderings, transition systems, the coverability problem, strictly asynchronous programs and predicate abstraction. A reader familiar with these concepts could well skip this section.

Well-quasi-orderings. The theory of well-quasi-orderings was pioneered by Graham Higman [101] and later by Erdos and Rado. The theory has become a fundamental (and frequently discovered [125]) concept in computer science and, if defined over the infinite set of states of system, provides termination arguments for many algorithmic methods.

A binary relation $\leq$ over a set Σ is a *quasi-ordering* if for any elements v_1, v_2 and v_3 from Σ, $v_1 \leq v_1$ is true (reflexivity), and $v_1 \leq v_2 \leq v_3$ implies $v_1 \leq v_3$ (transitivity). Any quasi-ordering induces an equivalence relation (a quasi-ordering that is symmetric), namely $v_1 \equiv v_2$ if and only if $v_1 \leq v_2 \leq v_1$, and induces a partial ordering between the equivalence classes. As usual, we write $v_2 \geq v_1$ for $v_1 \leq v_2$, and $v_1 < v_2$ for $v_1 \leq v_2 \not\leq v_1$. An example of a quasi-ordering is $(\mathbb{Z}^k, \preceq)$, the set of vectors of k integer numbers, $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$, with component-wise ordering $\preceq$, i.e.

$$(x_1, \ldots, x_k) \preceq (y_1, \ldots, y_k) \quad \text{if and only if} \quad x_i \leq y_i \quad \text{for all} \quad i = 1, \ldots, k\,. \tag{1}$$

A *well-quasi-ordering* is a quasi-ordering such that every infinite sequence of elements contains an increasing pair:

Definition 1. *A **well-quasi-ordering** is a quasi-ordering $\leq$ such that every infinite sequence of elements $v_1, v_2, v_3, \ldots$ contains an increasing pair, i.e. $v_i \leq v_j$ for some $i < j$.*

This is tantamount to saying that every infinite set of elements has at least one but no more than a *finite* number of (non-equivalent) minimal elements (see [101, 125] for more alternative definitions). The set of vectors of k integer numbers with component-wise ordering, $(\mathbb{Z}^k, \preceq)$, is *not* well (notice the existence of infinite strictly descending sequences). Yet, $(\mathbb{N}^k, \preceq)$, the set of vectors of k *natural* numbers, $\mathbb{N} = \{0, 1, \ldots\}$, with the component-wise ordering from eq. (1) is *well* by Dickson's lemma [54]; Figure 6 illustrates the ordering for $k = 4$.

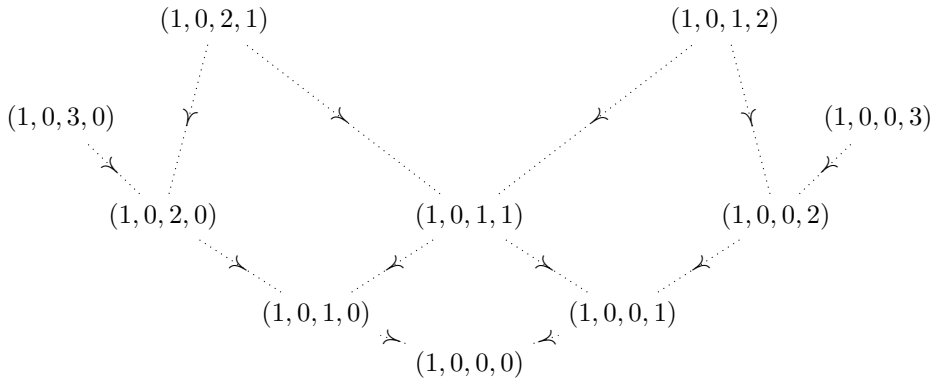

Figure 6: **Component-wise ordering $\preceq$ over $\mathbb{N}^4$** — An upward line from v to t indicates that $v \preceq t$, and no state r satisfies $v \prec r \prec t$; only vectors (a, b, c, d) with $a + b = 1$ and $c + d \leq 3$ are shown

Upward and downward closed sets. We write $\uparrow P$ for the $\geq$*-upward closure* of a set P,

$$\uparrow P \;=\; \{r \;:\; \exists v \in P \,.\, r \geq v\}\,.$$

An element v of P is *minimal* if there is no $r \in P$ such that $r < v$. Two minimal elements of a set are either incomparable or equivalent; the well-quasi-orderedness ensures that the equivalence relation $\equiv$ has a finite index.

We denote by $\min P$ a set of canonical representatives for each subset of equivalent minimal elements of P; note that with this definition, $\min P$ is always finite. Set P is *upward closed* if $\uparrow P = P$; in that case $\min P$ is a minimal subset M of P such that $\uparrow M = P$. Every upward-closed set P is representable as $\uparrow \min P$, for the finite set $\min P$. We abbreviate $\uparrow\{v\}$ by $\uparrow v$.

The concept *downward closed* and the symbol $\downarrow P$ are defined analogously (although, in contrast, not every downward-closed set can be represented by a finite set of *maximal* elements). For the set of vectors of 4 natural numbers, $\mathbb{N}^4$, with component-wise ordering, the set $\{(1,0,0,0), (1,0,0,1), (1,0,0,2), \ldots\}$ is upward-closed, and $\{(1,0,0,0), (1,0,0,1)\} = \downarrow(1,0,0,1)$ is downward-closed.

Well quasi-ordered systems. A *transition system* is a pair $(\Sigma, \rightarrowtail)$, where Σ is a set of elements called *states*, and $\rightarrowtail \subseteq \Sigma \times \Sigma$ is the set of *transitions*. Equipped with an ordering $\leq$, we call the system *monotone* if larger states have larger successors:

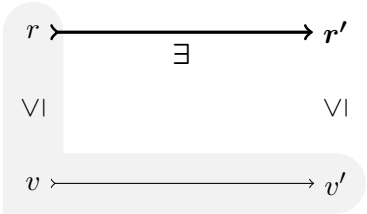

Figure 7: **Monotonicity** — Larger states have larger successors

Definition 2 ([2])**.** *A transition system* $(\Sigma, \rightarrowtail)$ *is* ***monotone*** *with respect to an ordering* $\leq$ *if for all states* v, v', r*, if* $v \leq r$ *and* v' *is a successor of* v*, i.e.* $v \rightarrowtail v'$*, then there exists a successor* r' *of* r *such that* $v' \leq r'$ *(Figure 7).*

We write $\mathsf{Pre}(X)$ for the set of *direct predecessors* of states in X, i.e. the set $\{r \in \Sigma \ : \ r \rightarrowtail v \in X\}$, or simply $\mathsf{Pre}(v)$ if $X = \{v\}$. An immediate consequence of Definition 2 is that for monotone transition systems operator Pre preserves upward-closedness, and hence the set of predecessors of an upward-closed set of states is *again* upward-closed and can precisely be characterised by its (non-equivalent) minimal elements.

Definition 3 ([2])**.** *A transition system* $(\Sigma, \rightarrowtail)$ *with a decidable ordering* $(\Sigma, \leq)$ *is said to be* ***well-quasi ordered*** *if*

(i) $\leq$ *is a well-quasi-ordering (Definition 1),*
(ii) the transition system is monotone with respect to $\leq$ *(Definition 2), and*
(iii) for each state $v \in \Sigma$*, the set* $\min \mathsf{Pre}(v)$ *is computable.*

Coverability problem. Let $(\Sigma, \rightarrowtail)$ be a transition system with ordering $(\Sigma, \leq)$, and $I \subseteq \Sigma$ be a set of initial states (downward-closed). We write $\mathsf{Cover} \subseteq \Sigma$ for the set of *coverable states*, i.e. those "covered" by some reachable state: $\mathsf{Cover} = \{v \ : \ i \rightarrowtail^* r \geq v \text{ for some } i \in I \text{ and } r \in \Sigma\}$. The reachability problem for upward-closed sets is frequently referred to as the *coverability problem*:

Definition 4. *Given a transition system ordered by* $\leq$*, initial states* $I \subseteq \Sigma$ *(downward-closed), and error states* $E \subseteq \Sigma$ *(upward-closed). The* ***coverability problem*** *asks for the existence of a reachable state* $v \in E$*.*

Reachability of a state in E (and thus the coverability problem) can be checked by computing the set of states that are *backward-reachable* from E, i.e. have an emanating execution leading to a state in E. Hence, by computing $\mathsf{Pre}^*(E)$, the limit of sequence

$$E = E_0 \subseteq E_1 \subseteq \cdots \text{ with } E_{i+1} = E_i \cup \mathsf{Pre}(E_i) \text{ for } i \geq 1\,. \tag{2}$$

The crux of combining both the theory of well-quasi-ordering (Definition 1) with monotonicity (Definition 2) into well quasi-ordered systems (Definition 3) is that together they are sufficient to ensure convergence and computability of the limit of eq. (2), and thus decidability of the coverability problem (Definition 4) [2]. We will see next, why this is useful for verifying (strictly asynchronous) programs.

Strictly asynchronous programs. We model a strictly asynchronous program $\mathcal{P}$, designed for execution by $n \geq 1$ concurrent threads, as follows. The variable set V of a program $\mathcal{P}$ is partitioned into sets S and L. The variables in S, called *shared*, are accessible jointly by all threads, and those in L, called *local*, are accessible by the individual thread that owns the variable. We assume the statements of $\mathcal{P}$ are given by a transition formula $\mathcal{R}$ over unprimed (current-state) and primed (next-state) variables, V and $V' = \{\mathtt{v}' \ : \ \mathtt{v} \in V\}$. Further, the initial and error states are characterised by the initial and error formulas, $\mathcal{I}$ and $\mathcal{E}$, respectively, over V. We assume these formulas are expressible in reasonable logics for which existential quantification is computable.

As usual, the computation may be controlled by a local program counter $\mathtt{pc}$, and involve non-recursive function calls. If we represent programs as "code fragments" like in Figures 4 and 5, threads are assumed to interleave with line-level granularity; see the discussion in Section 8 on this subject. Moreover, we write "$=$" for the equality operator on program variables, "::" to define relations, "$:=$" for the parallel assignment operator [21] (e.g. $\mathtt{m}, \mathtt{l} := \mathtt{l}, \mathtt{m}$ swaps the value of variables $\mathtt{m}$ and $\mathtt{l}$), and "$*$" to denote the nondeterministic choice expression, i.e. $*$ nondeterministically evaluates to true (`T`) or false (`F`).

Example 5. *The transition relation for the compare-and-swap instruction in Figure 3 is*

$$\mathcal{R} :: (\mathtt{m}' = \mathtt{m}) \wedge (\mathtt{l}' = \mathtt{l}) \wedge \big((\mathtt{s} = \mathtt{m}) \Rightarrow (\mathtt{s}' = \mathtt{l})\big) \wedge \big((\mathtt{s} \neq \mathtt{m}) \Rightarrow (\mathtt{s}' = \mathtt{s})\big)\,.$$

The initial formula of the atomic counter in Figure 4 is $\mathcal{I} :: \mathtt{pc} = \ell_1 \wedge \mathtt{s} = 0$*, and the error formula induced by the assertion is* $\mathcal{E} :: (\mathtt{pc} = \ell_5) \not\Rightarrow (\mathtt{s} \geq \mathtt{m})$.

When executed by n threads, $\mathcal{P}$ gives rise to *n-thread program states* consisting of the valuations of the variables in $V_n = S \cup L_1 \cup \ldots L_n$, where $L_i = \{\mathtt{m}_i \,:\, \mathtt{m} \in L\}$. Call a variable set *uniquely-indexed* if its variables either all have no index, or all have the same index. For a formula f and two uniquely-indexed variable sets X_1 and X_2, let $f\{X_1 \triangleright X_2\}$ denote f after replacing every occurrence of a variable in X_1 by the variable in X_2 with the same base name, if any; unreplaced if none. We write $f\{X_1 \Subset X_2\}$ short for $f\{X_1 \triangleright X_2\}\{X_1' \triangleright X_2'\}$. As an example, given $S = \{\mathtt{s}\}$ and $L = \{\mathtt{m}\}$, we have

$$(\mathtt{m}' = \mathtt{m} + \mathtt{s})\{L \Subset L_a\} = (\mathtt{m}'_a = \mathtt{m}_a + \mathtt{s})\,.$$

Finally, let $\overline{\exists} X$ abbreviate $\exists X, X'$, and $X \overset{\circ}{=} X'$ stand for $\forall x \in X \,.\, x = x'$.

The *n-thread instantiation* $\mathcal{P}^n$ is defined for $n \geq 1$ as

$$\mathcal{P}^n = (\mathcal{R}^n, \mathcal{I}^n, \mathcal{E}^n) \tag{3}$$

$$= \left(\textstyle\bigvee_{a=1}^{n} \mathcal{R}(a)^n,\ \bigvee_{a=1}^{n} \mathcal{I}(a)^n,\ \bigvee_{a=1}^{n} \mathcal{E}(a)^n\right) \tag{4}$$

with

$$\mathcal{R}(a)^n :: \mathcal{R}\{L \Subset L_a\} \wedge \textstyle\bigwedge_{p:p\neq a} L_p \overset{\circ}{=} L'_p \tag{5}$$

$$\mathcal{I}(a)^n :: \mathcal{I}\{L \triangleright L_a\} \wedge \textstyle\bigwedge_{p:p\neq a} \mathcal{I}\{L \triangleright L_p\} \tag{6}$$

$$\mathcal{E}(a)^n :: \mathcal{E}\{L \triangleright L_a\}\,. \tag{7}$$

Applied to n-thread program states, formula $\mathcal{R}(a)^n$ encodes that the shared variables, and the variables of the executing (active) thread a are updated according to $\mathcal{R}$, while the local variables of other (passive) threads $p \neq a$ are not modified. Analogously, a state is *initial* (*erroneous*) if all threads are (some thread is) in a state satisfying $\mathcal{I}$ ($\mathcal{E}$). An *n-thread execution* is a sequence of n-thread program states, starting in $\mathcal{I}^n$ and pairwise related by $\mathcal{R}^n$. An erroneous execution is one ending in a state satisfying $\mathcal{E}^n$. We call $\mathcal{P}$ safe if no erroneous execution exists, and unsafe otherwise. For brevity, we sometimes omit the error component.

We note that "parameterised concurrency", via an n-thread instantiation $\mathcal{P}^n$ of a single template program $\mathcal{P}$, is elegant and conveniently formalisable, but does not directly reflect the more common case of dynamic thread creation at runtime. The latter scenario is, however, easily supported in the parameterised model using standard techniques. All our benchmark programs spawn threads during program

execution. Also note that the atomicity of *local computation* steps, i.e. of instructions that involve only local variables, is unimportant for safety reasoning [14, 13]. Different threads could just as well perform local computation simultaneously as in a multi-core system rather than being interleaved with one another in a single instruction stream.

A strictly asynchronous program $\mathcal{P}$ induces an infinite-state transition system $(\Sigma, \rightarrowtail)$ and sets I and E as follows:

(i) Σ is the set containing arbitrary-thread program states,
(ii) $v \rightarrowtail v'$ if and only if v and v' are pairwise related by formula $\mathcal{R}^n$ for some n, and
(iii) I and E contain states satisfying formulas $\mathcal{I}^n$ and $\mathcal{E}^n$ for some n, respectively.

Notice that the induced sets I and E are downward- and upward-closed, and hence coverability problem suffices to *exactly* decide classical safety properties like program location reachability and program assertions in strictly asynchronous programs.

Well-quasi-orderedness. We write an n-thread program state $v \in \Sigma$ in the form

$$(s \mid c_1, \ldots, c_n)\,,$$

emphasising v's valuation of the shared variables S of $\mathcal{P}$ (here s), and listing the valuations of the local variables in v for each of the n threads (components $c_1, \ldots, c_n$); if the variable ordering is unambiguous we sometimes omit their names. As an example, for Boolean-type variables $S = \{\mathtt{s}\}$ and $L = \{\mathtt{b}\}$, $(\neg\mathtt{s} \wedge \neg\mathtt{b}_1 \wedge \mathtt{b}_2 \wedge \neg\mathtt{b}_3)$, $(\neg\mathtt{s} \mid \neg\mathtt{b}, \mathtt{b}, \neg\mathtt{b})$ and $(\mathtt{F} \mid \mathtt{F}, \mathtt{T}, \mathtt{F})$ characterise the same 3-thread program state. Specifically, for our case of local state tuples we choose for $\min P$ the *lexicographically least* among all equivalent minimal elements in the set as representative, such that $\min\{(\mathtt{F} \mid \mathtt{F}, \mathtt{T}), (\mathtt{F} \mid \mathtt{T}, \mathtt{F})\} = \{(\mathtt{F} \mid \mathtt{F}, \mathtt{T})\}$, and $\min\{(\mathtt{F} \mid \mathtt{T}, \mathtt{F})\} = \{(\mathtt{F} \mid \mathtt{T}, \mathtt{F})\}$.

Let all variables of $\mathcal{P}$ be Boolean-type. A well-quasi-ordering on program states is the *covered relation*, which is defined as follows (illustrated in Figure 8):

Definition 6. *Given two program states v and r from Σ. State $v = (s \mid c_1, \ldots, c_n)$ is **covered** by $r = (t \mid m_1, \ldots, m_{n'})$, written $v \leq r$, if and only if (i) $n \leq n'$, and (ii) there exists a permutation π of $1, \ldots, n'$ such that $v = (t \mid m_{\pi(1)}, \ldots, m_{\pi(n)})$.*

In order to see that $(\Sigma, \leq)$ is a well-quasi-ordering, observe that each program state can be characterised as a vector of $2^{|S|} + 2^{|L|}$ natural numbers (all variables are

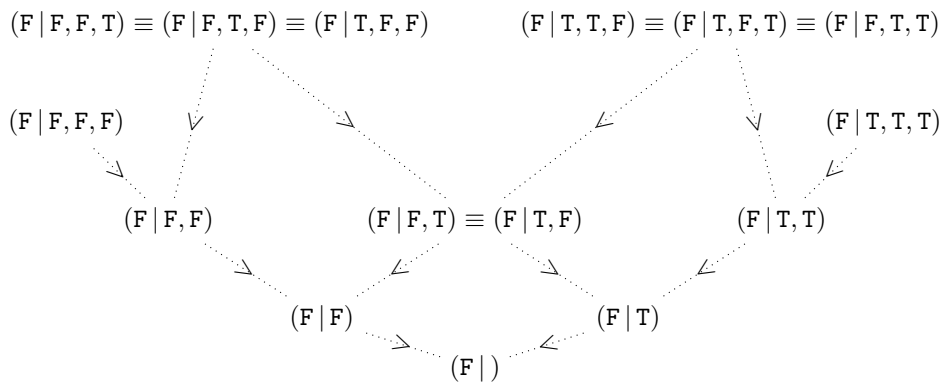

Figure 8: **Covered relation for variables L=$\{\mathtt{b} \in \mathbb{B}\}$ and S=$\{\mathtt{s} \in \mathbb{B}\}$** — States that are identical up to permutations ("≡") are grouped; only states $(\mathtt{s} \mid c_1, \ldots, c_n)$ for $n \leq 3$ and $\mathtt{s} = \mathtt{F}$ are shown

Boolean-type): of the first $2^{|S|}$ components, a unique non-zero component encodes the shared state s, and the remaining components store the Parikh vector [153] of the sequence $c_1, \ldots, c_n$, i.e. the vector that counts the occurrences of the respective local states in some order.

Example 7. *Consider the Boolean-type variables $S = \{\mathtt{s}\}$ and $L = \{\mathtt{b}\}$, and let the "component-mapping" be $(\mathtt{s} \stackrel{?}{=} \mathtt{F}, \mathtt{s} \stackrel{?}{=} \mathtt{T}, \#_{\mathtt{b}=\mathtt{F}}, \#_{\mathtt{b}=\mathtt{T}})$. The 1 and 3-thread program states $(\mathtt{T} \mid \mathtt{F})$ and $(\mathtt{F} \mid \mathtt{F}, \mathtt{T}, \mathtt{F})$ then yield vectors $(0, 1, 1, 0)$ and $(1, 0, 2, 1)$ from $\mathbb{N}^4$, respectively.*

The crux is that for two states v and r with associated vectors x and y, $v \leq r$ if and only if $x \preceq y$. Observe how Figure 8 morphs into Figure 6, and vice versa. Hence, $(\Sigma, \leq)$ is a well-quasi-ordering by Dickson's lemma. Since, on the other hand, $(\Sigma, \rightarrowtail, \leq)$ is clearly a monotone transition system according to Definition 2 we obtain that strictly asynchronous Boolean programs can be encoded as well quasi-ordered systems.

Predicate abstraction for sequential programs. Let $\mathcal{P}$ be a strictly asynchronous program, yet assume a single thread of execution ($n = 1$). Further, let $\{Q[c] : 1 \leq c \leq m\}$ be a *set of predicates* defined over the program variables V. In the presence of multiple instructions, we assume that there exists a predicate $\mathtt{pc} = \ell$ for each instruction label ℓ.

Concrete transition ($n = 1$)				Abstract relation $\hat{\mathcal{R}}$			
$\mathtt{v}$	$\mathtt{w}$	$\mathtt{v}'$	$\mathtt{w}'$	$\mathtt{b}[1]$	$\mathtt{b}[2]$	$\mathtt{b}[1]'$	$\mathtt{b}[2]'$
2	0	0	0	F	T	F	T
2	1	0	1	F	T	T	T
2	3	0	3	T	T	T	T
3	0	1	0	F	F	F	F
3	2	1	2	F	F	T	F
3	4	1	4	T	F	T	F

Table 1: **Abstraction of a decrement instruction on integer variables** $\mathtt{v}$ **and** $\mathtt{w}$ — Each row lists a concrete transition for instruction $\mathtt{v} := \mathtt{v} - 2$, and the abstract transition it induces with respect to predicates $\mathtt{v} < \mathtt{w}$ and $(\mathtt{v} \bmod 2) = 0$, tracked by Boolean variables $\mathtt{b}[1]$ and $\mathtt{b}[2]$, respectively

For sequential programs, the goal of *existential predicate abstraction* [42, 93] is to derive an abstract program $\hat{\mathcal{P}}$ by treating the equivalence classes induced by the predicates as abstract states. The m Boolean variables of $\hat{\mathcal{P}}$ are

$$\hat{V} = \{\mathtt{b}[1], \ldots, \mathtt{b}[m]\}\,.$$

Variable $\mathtt{b}[c]$ tracks the truth of predicate c.

Formula $\mathcal{D}$ below specifies the correspondence between concrete thread states (valuations of V) and abstract thread states (valuations of $\hat{V}$):

$$\mathcal{D} :: \bigwedge_{c=1}^{m} (\mathtt{b}[c] \Leftrightarrow Q[c])\,. \tag{8}$$

For a formula f, let f' denote f after replacing each variable by its primed version; the components of $\hat{\mathcal{P}}$ are

$$\hat{\mathcal{R}} :: \exists V.\ \mathcal{R} \wedge \mathcal{D} \wedge \mathcal{D}' \tag{9}$$

$$\hat{\mathcal{I}} :: \exists V.\ \mathcal{I} \wedge \mathcal{D} \tag{10}$$

$$\hat{\mathcal{E}} :: \exists V.\ \mathcal{E} \wedge \mathcal{D}\,. \tag{11}$$

Example 8. *Consider the assignment* $\mathtt{v} := \mathtt{v} - 2$ *in the presence of integer variables* $\mathtt{v}$ *and* $\mathtt{w}$ *(*$\mathtt{w}$ *is not modified), and the two predicates* $Q[1] :: \mathtt{v} < \mathtt{w}$ *and* $Q[2] :: (\mathtt{v} \bmod 2) = 0$*. Plugging both into eq.* (9)*, we get 6 abstract transitions, which are*

```
typedef struct cell {
  int val;
  struct cell* nex;
} *list;

list c, p, newl, nexc;
list partition(list *m, int l) {
  c := *m;
  p := NULL;
  newl := NULL;
  while (c ≠ NULL) {
    nexc := (*c).nex;
    if ((*c).val > l) {
      if (p ≠ NULL) {(*p).nex := nexc;}
      if (c = *m) {*m := nexc;}
      (*c).nex := newl, newl := c; }
    else {p := c;}
    c := nexc; }
  return newl; }
```

(a)

```
//Q[1] :: c = NULL
//Q[2] :: p = NULL
//Q[3] :: (*c).val > l
//Q[4] :: (*p).val > l

bool b[1], b[2], b[3], b[4];
void partition() {
  b[1],b[3] := *,*;
  b[2],b[4] := true,*;

  while(¬b[1]) {

    if (b[3]) {
      if (¬b[2]) {skip;}
      if (*) {skip;}
    }
    else {b[2],b[4] := b[1],b[3];}
    b[1],b[3] := *,*;}
  return; }
```

(b)

Figure 9: **Abstraction of a sequential list algorithm (from [17])** — (a) function that separates the list `m` points to into one with cells strictly greater than `l`, and one for others; (b) Boolean program-characterisation of the existential abstraction obtained for the predicates on top

listed in Table 1. Equivalent characterisations as parallel assignment and formula are $\mathtt{b}[1], \mathtt{b}[2] := \mathtt{b}[1] \vee *, \mathtt{b}[2]$ *and* $\hat{\mathcal{R}} :: (\mathtt{b}[1] \Rightarrow \mathtt{b}[1]') \wedge (\mathtt{b}[2] \Leftrightarrow \mathtt{b}[2]')$*, respectively. The first conjunct in* $\hat{\mathcal{R}}$ *reflects that* $\mathtt{v} < \mathtt{w}$ *is true after executing assignment* $\mathtt{v} := \mathtt{v}-2$*, if it was true previously (Table 1, rows 3 and 6), and otherwise it is either true or false. The second conjunct reflects that* `v`*'s parity is preserved. Analogously, if* $\mathcal{I} :: \mathtt{v} = 1 \wedge \mathtt{w} = 5$ *and* $\mathcal{E} :: \mathtt{v} = 0$ *are the initial and error formulas, respectively, then* $\hat{\mathcal{I}} :: \mathtt{b}[1] \wedge \neg \mathtt{b}[2]$ *and* $\hat{\mathcal{E}} :: \mathtt{b}[2]$.

Figure 9 illustrates the process for a sequential list algorithm. The key is that existential abstraction is conservative for reachability properties [42]: For every execution in $\mathcal{P}^1$ there exists a safety-equivalent execution in $\hat{\mathcal{P}}^1$. Safety of $\hat{\mathcal{P}}^1$ hence implies safety of $\mathcal{P}^1$.

3 Lost in Abstraction: Monotonicity in Multi-Threaded Programs*

In this section, we present our abstraction strategy for unbounded-thread strictly asynchronous programs with respect to predicates tracking certain relationships that are quantified over local variables of all threads. To this end, we first present such a strategy for fixed thread numbers, then show how to compute an abstract program template that can be instantiated for any thread number, much like strictly asynchronous programs, and finally illustrate a natural extension to programs with condition variables, which are more expressive than strictly asynchronous programs when every thread's memory is finite.

3.1 Inter-Thread Predicate Abstraction

We introduce single- and inter-thread predicates, with respect to which we then formalise existential predicate abstraction. Except for the predicate language, these concepts are mostly standard, but lay the technical foundations for the contributions of this thesis.

3.1.1 Predicate Language

We extend the predicate language from [58] to allow the use of the passive-thread variables $L_P = \{\mathtt{m}_P \,:\, \mathtt{m} \in L\}$, each of which represents a local variable owned by a generic passive thread. We classify our predicates as follows.

Definition 9. *A predicate Q over S, L and L_P is **shared** if it solely contains variables from S, **local** if it solely contains variables from L, **single-thread** if it contains variables from L but not from L_P, and **inter-thread** if it contains variables from L and from L_P.*

In the ticket algorithm (Figure 5), with shared variables $S = \{\mathtt{s}, \mathtt{t}\}$ and local variable $L = \{\mathtt{m}\}$, examples of shared, local, single-thread and inter-thread predicates are: $\mathtt{s} = \mathtt{t}$, $\mathtt{m} = 5$, $\mathtt{s} = \mathtt{m}$ and $\mathtt{m} \neq \mathtt{m}_P$, respectively.

*The main content of this chapter was previously presented in [59, 58].

Semantics. Let $\{Q[c] \ : \ 1 \leq c \leq m\}$ be a list of m predicates (any class). The truth of predicate $Q[c]$ in a given n-thread state v ($n \geq 2$) is defined with respect to a choice of active thread a:

$$\mathcal{C}_a(c) :: \bigwedge\nolimits_{p:p\neq a} Q[c]\{L \triangleright L_a\}\{L_P \triangleright L_p\}\,. \tag{12}$$

The predicate is *true* in v if and only if $v \Rightarrow \mathcal{C}_a(c)$ is valid, and hence captures whether the predicate evaluates to true for all thread pairs (a, p), $a \neq p$. Notice that for single-thread and shared predicates (= without L_P variables), we have

$$\mathcal{C}_a(c) = Q[c]\{L \triangleright L_a\} \quad \text{and} \quad \mathcal{C}_a(c) = Q[c], \text{respectively.} \tag{13}$$

Evaluation of eq. (12) in a state for all predicates and threads yields the abstraction function α, a mapping of n-thread program states to $n \times m$-dimensional bit matrices:

$$\alpha(v)_{a,c} = \begin{cases} \mathtt{T} & \text{if } v \Rightarrow \mathcal{C}_a(c) \text{ is valid} \\ \mathtt{F} & \text{otherwise}\,. \end{cases} \tag{14}$$

Function α partitions the n-thread program state space via m predicates into $2^{n\times m}$ equivalence classes. As an example, consider the inter-thread predicates $\mathtt{m} \leq \mathtt{m}_P$, $\mathtt{m} > \mathtt{m}_P$, and $\mathtt{m} \neq \mathtt{m}_P$ for a local variable $\mathtt{m}$. Then

$$\alpha\left(\begin{pmatrix} \mathtt{m}_1 = 4 \\ \mathtt{m}_2 = 4 \\ \mathtt{m}_3 = 5 \\ \mathtt{m}_4 = 6 \end{pmatrix}\right) = \begin{pmatrix} \mathtt{T} & \mathtt{F} & \mathtt{F} \\ \mathtt{T} & \mathtt{F} & \mathtt{F} \\ \mathtt{F} & \mathtt{F} & \mathtt{T} \\ \mathtt{F} & \mathtt{T} & \mathtt{T} \end{pmatrix}\,. \tag{15}$$

In the matrix, column $c \in \{1, 2, 3\}$ lists the truth of predicate c for each of the four threads in the active role. Predicate $\mathtt{m} \leq \mathtt{m}_P$ captures whether a thread owns the minimal value for local variable $\mathtt{m}$ and thus evaluates to true exactly for $a = 1, 2$; $\mathtt{m} > \mathtt{m}_P$ tracks whether a thread owns the unique maximum value and thus evaluates true only for $a = 4$; finally $\mathtt{m} \neq \mathtt{m}_P$ captures the uniqueness of a thread's copy of $\mathtt{m}$, which applies to both $a = 3$ and $a = 4$.

Due to the universal nature of inter-thread predicates our predicate language is not closed under negation. We can observe this phenomenon in the previous example with predicates $Q[1]$ and $Q[2]$, which syntactically are negations of each other, yet both evaluate to false for thread $a = 3$. For single-thread predicates, however, $Q[c] = \neg Q[c']$ means $\alpha_{a,c} = \neg\alpha_{a,c'}$.

Inter-thread predicates are essential. To emphasise the need for our abstraction strategy to handle inter-thread predicates, we show that proof methods that cannot reason about them (such as [70, 58, 176]) are bound to fail for the ticket algorithm in Figure 5 when threads concurrently and repeatedly (e.g. in an infinite loop) request and release lock ownership.

Lemma 10. *Given the parameterised ticket algorithm where threads call spin_lock and spin_unlock arbitrarily often. Then no Hoare/Floyd-style correctness proof with single-thread predicates exists.*

Proof sketch. We write $\mathtt{pc}_i$ for the pc of thread i, $1 \leq i \leq n$. We first state some easy-to-prove invariants of the ticket algorithm:

$$s \leq t \leq s + n \tag{16}$$
$$\mathtt{pc}_i = \ell_1 \Rightarrow l_i = 0 \tag{17}$$
$$\mathtt{pc}_i = \ell_2 \Rightarrow s < l_i < t \tag{18}$$
$$\mathtt{pc}_i = \ell_3 \Rightarrow l_i = s \tag{19}$$
$$\#(\mathtt{pc} = \ell_2) + \#(\mathtt{pc} = \ell_3) = t - s \tag{20}$$

We can think of ℓ_1, ℓ_2, and ℓ_3 as the non-critical, trying, and locked region of a standard mutex lock. The total number of threads in the trying and locked regions is $t - s$ (Section 3.1.1). If all threads are "non-critical", we have $s = t$, and the l_i are all zero.

Let now

$$E = \bigcup_{i=1}^{n} E^{s,t,l_i} \tag{21}$$

be the **disjoint** union of sets of predicates formulated over the shared variables s and t and any one of the l_i ; in particular, no predicate may refer to several of the l_i. Suppose I is an invariant expressible over E that is strong enough to prove mutual exclusion. Then

$$\forall i, j \;:\; i \neq j \;:\; I \wedge \mathtt{pc}_i = \mathtt{pc}_j = \ell_2 \quad \Rightarrow \quad l_i \neq l_j \,, \tag{22}$$

since otherwise threads i and j can, once s reaches the value l_i $(= l_j)$, escape the busy-wait loop and simultaneously proceed to the critical section.

For any c, there exists a *reachable* global state satisfying $\mathtt{pc}_1 = \mathtt{pc}_2 = \ell_2$ and $(s, l_1, l_2) = (c, c, c+1)$ (that is, thread 1 proceeds to the trying region first, then thread 2), and a *reachable* global state satisfying $\mathtt{pc}_1 = \mathtt{pc}_2 = \ell_2$ and $(s, l_1, l_2) = (c, c+1, c)$ (vice versa). Since c is unbounded, there thus exist infinitely many such assignments that satisfy invariant I.

Let now $\{I_1, \ldots, I_w\}$ be the cubes in the DNF representation of I. Since this set is finite, one can extract a single cube I_k that satisfies both $(s, l_1, l_2) = (c, c, c+1)$ and $(s, l_1, l_2) = (c, c+1, c)$, for some c. We split I_k into the sub-cubes that belong to E^{s,t,l_1}, and those that belong to E^{s,t,l_2}: $I_k = I_k^1 \wedge I_k^2$; note that these sub-cube sets are disjoint (sub-cubes that refer to neither l_1 nor l_2 are apportioned to either side). Then $(s, l_1, l_2) = (c, c, c+1)$ satisfies I_k^1, which does not contain l_2, so in fact $(s, l_1) = (c, c)$ satisfies I_k^1. Symmetrically, one obtains that $(s, l_2) = (c, c)$ satisfies I_k^2. Hence $(s, l_1, l_2) = (c, c, c)$ satisfies $I_k^1 \wedge I_k^2 = I_k$ and hence satisfies I, which contradicts eq. (22).

The argument also indicates that, for a finite number n of threads, the set of predicates depends on n: there is no c such that, for every n, c predicates suffice to prove the protocol correct for n threads. □

3.1.2 Existential Inter-Thread Predicate Abstraction

Embedded into our formalism, the goal of *existential predicate abstraction* [42, 93] is to derive an abstract program $\hat{\mathcal{P}}^n$ by treating the equivalence classes induced by eq. (14) as abstract states. The $n \times m$ Boolean variables of $\hat{\mathcal{P}}^n$ are

$$\begin{aligned} \hat{V}_n \;=\; \{\mathtt{b}[1]_1, \;\; \cdots \;\; , \mathtt{b}[m]_1, \\ \vdots \qquad\quad \vdots \\ \mathtt{b}[1]_n, \;\; \cdots \;\; , \mathtt{b}[m]_n\}\,. \end{aligned}$$

Variable $\mathtt{b}[c]_a$ tracks the truth of predicate c for active thread a.

Formula $\mathcal{D}^n$ below specifies the correspondence between concrete n-thread states (valuations of V_n) and abstract n-thread states (valuations of $\hat{V}_n$):

$$\mathcal{D}^n :: \bigwedge_{a=1}^{n} \bigwedge_{c=1}^{m} (\mathtt{b}[c]_a \Leftrightarrow \mathcal{C}_a(c))\,. \tag{23}$$

For a formula f, let f' denote f after replacing each variable by its primed version;

Concrete transition ($n = 2$)				Abstract relation $\hat{\mathcal{R}}(1)^2$			
$\mathtt{m}_1$	$\mathtt{m}_2$	$\mathtt{m}'_1$	$\mathtt{m}'_2$	$\mathtt{b}_1$	$\mathtt{b}_2$	$\mathtt{b}'_1$	$\mathtt{b}'_2$
1	0	0	0	F	T	F	F
1	1	0	1	F	F	T	F
1	2	0	2	T	F	T	F
2	0	1	0	F	T	F	T

Table 2: **Abstraction for $n = 2$** — Rows list concrete transitions for instruction $\mathtt{m} := \mathtt{m} - 1$ when executed by thread 1, i.e. valuations of $\mathcal{R}(1)^2$, and the abstract transitions it induces with respect to predicate $\mathtt{m} < \mathtt{m}_P$

the components of $\hat{\mathcal{P}}^n$ are

$$\hat{\mathcal{R}}(a)^n :: \exists\!\exists V_n \,.\, \mathcal{R}(a)^n \wedge \mathcal{D}^n \wedge (\mathcal{D}^n)' \tag{24}$$

$$\hat{\mathcal{I}}(a)^n :: \exists V_n \,.\, \mathcal{I}(a)^n \wedge \mathcal{D}^n \tag{25}$$

$$\hat{\mathcal{E}}(a)^n :: \exists V_n \,.\, \mathcal{E}(a)^n \wedge \mathcal{D}^n \,. \tag{26}$$

As an example, consider the decrement operation $\mathtt{m} := \mathtt{m} - 1$ on a local integer variable $\mathtt{m}$, and the inter-thread predicate $\mathtt{m} < \mathtt{m}_P$. Plugging both into eq. (24) with $n = 2$, we get 4 abstract transitions, which are listed in Table 2. Now observe that strict asynchrony is lost in the abstraction: while the transitions in the 2-thread concrete relation $\mathcal{R}(1)^2$ satisfy $\mathtt{m}_2 = \mathtt{m}'_2$ (the input program is strictly asynchronous), this is no longer the case in $\hat{\mathcal{R}}(1)^2$: in the first row in Table 2 (highlighted), both the pc (not shown) of thread 1 and the value of local variable $\mathtt{b}$ of thread 2 change.

The key is that existential abstraction is conservative for reachability properties.

Corollary 11. *[42] For every thread number n and every execution in $\mathcal{P}^n$ there exists a safety-equivalent execution in $\hat{\mathcal{P}}^n$. Safety of $\hat{\mathcal{P}}^n$ hence implies safety of $\mathcal{P}^n$.*

Proving the ticket lock for fixed threads. To illustrate the approach, we use the ticket algorithm [10], a busy-wait lock that prevents starvation by processing waiting threads in FIFO-order. This has been the default lock in the Linux kernel since 2008 [48]. Figure 5 shows the slightly simplified C code. Consider the predicates

$$Q[1] :: \mathtt{m} \neq \mathtt{m}_P \,, \tag{27}$$

$$Q[2] :: \mathtt{t} > \max(\mathtt{m}, \mathtt{m}_P) \,, \text{ and} \tag{28}$$

$$Q[3] :: \mathtt{s} = \mathtt{m} \,. \tag{29}$$

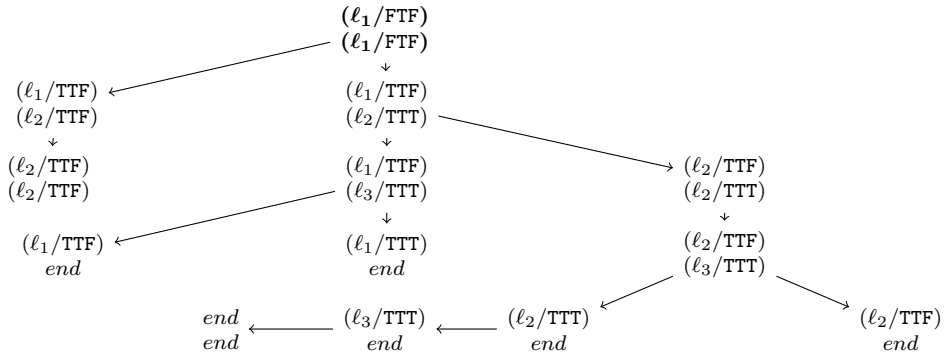

Figure 10: **Abstract reachability tree for the ticket lock for 2 threads** — Each node represents a 2-thread program state (stacked), with edges being labelled by the inducing transition. The initial state is in bold

The first two predicates are inter-thread, while the third is single-thread. They capture the uniqueness of a ticket (eq. (27)), that the next free ticket is strictly larger than all the tickets that are currently owned by threads (eq. (28)), and whether a thread's ticket is currently being served (eq. (29)).

Figure 10 shows the abstract reachability tree (modulo symmetry) for $\hat{\mathcal{P}}^2$ obtained for this set of predicates according to eqs. (24) to (26) and $n = 2$.* It has 13 reachable states for $n = 2$ threads; the tree grows exponentially in n. As an example, initially the program counters both point to the fetch_and_add operation (line ℓ_1), both threads have a non-unique ticket ($\mathtt{m}_1 = \mathtt{m}_2 = 0$), which are furthermore strictly smaller than the next free ticket ($\mathtt{m}_i < \mathtt{t} = 1$ for $i \in \{1, 2\}$), and finally neither of the threads is being served ($\mathtt{m}_i \neq \mathtt{s} = 1$ for $i \in \{1, 2\}$). For example, the root's successor in the middle

$$\begin{matrix}(\ell_1/\mathtt{TTF})\\(\ell_2/\mathtt{TTT})\end{matrix}$$

abstracts the 2-thread concrete states satisfying

$$\bigwedge_{i=1,2} (\mathtt{pc}_i = \ell_i \wedge \mathtt{b}[1]_i \wedge \mathtt{b}[2]_i) \wedge \neg \mathtt{b}[3]_1 \wedge \mathtt{b}[3]_2 \,.$$

Notice that the abstraction is no longer strictly asynchronous with respect to the argument a of a predicate, as the changes of passive-thread variables reveal. As an

*As usual, we assume the existence of a predicate $\mathtt{pc} = \ell_i$ for $i \in \{1, 2, 3\}$. The abstraction therefore retains the control-flow.

example, if the thread with $a = 1$ executes from the initial state, this also implies a change of the other thread's predicates, as is evident from the outgoing transition of root node in Figure 10: the state of the thread that is passive in the transition changes in both successors from $(\ell_1/\texttt{FTF})$ to $(\ell_1/\texttt{TTF})$. The loss of asynchrony **cannot be blamed on the use of inter-thread predicates**; instances of this effect without such predicates are reported in [58].

3.2 From Existential to Parametric Abstraction

Classical existential abstraction incurs a high verification cost: because of the exponential (in the number of threads) space complexity, techniques that exhaustively explore the state space of $\hat{\mathcal{P}}^n$ will not scale beyond a very small number of threads. Moreover, such abstractions need to be computed for every n individually, rendering parametric reasoning about threads impossible.

To overcome these problems, we now derive an overapproximation of $\hat{\mathcal{P}}^n$ via a program template $\tilde{\mathcal{P}}$ that can be instantiated for any n, much like the strictly asynchronous input program $\mathcal{P}$ can via eqs. (5) to (7). As we have seen, predicate-abstracting strictly asynchronous programs gives rise to programs that themselves cannot in general be expressed in the asynchronous language defined in Section 2. Instead, our abstract programming language is the richer class of *dual-reference* programs.

3.2.1 Dual-Reference Programs

In contrast to strictly asynchronous programs, the variable set $\tilde{V}$ of a dual-reference (DR) program $\tilde{\mathcal{P}}$ is partitioned into two sets, $\tilde{L}$ (the local variables of the active thread, as before) and $\tilde{L}_P = \{\mathtt{m}_P \,:\, \mathtt{m} \in \tilde{L}\}$. The latter set contains passive-thread variables, which are owned by some generic thread that is *passive* in a transition. To simplify reasoning about DR programs, we exclude classical shared variables from the description: they can be easily simulated, as we show below after the introduction of the program semantics.

As before, we assume the statements of $\tilde{\mathcal{P}}$ are given by a transition formula $\tilde{\mathcal{R}}$ over $\tilde{V}$ and $\tilde{V}'$, now potentially including passive-thread variables. Similarly, $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{E}}$ may contain variables from $\tilde{L}_P$.

The n-thread instantiation of a DR program $\tilde{\mathcal{P}}^n$ is defined for $n \geq 2$ via

$$\tilde{\mathcal{R}}(a)^n :: \bigwedge\nolimits_{p:p\neq a} \tilde{\mathcal{R}}\{\tilde{L}\Rrightarrow\tilde{L}_a\}\{\tilde{L}_P\Rrightarrow\tilde{L}_p\} \tag{30}$$

$$\tilde{\mathcal{I}}(a)^n :: \bigwedge\nolimits_{p:p\neq a} \tilde{\mathcal{I}}\{\tilde{L}\triangleright\tilde{L}_a\}\{\tilde{L}_P\triangleright\tilde{L}_p\} \tag{31}$$

$$\tilde{\mathcal{E}}(a)^n :: \bigvee\nolimits_{p:p\neq a} \tilde{\mathcal{E}}\{\tilde{L}\triangleright\tilde{L}_a\}\{\tilde{L}_P\triangleright\tilde{L}_p\} \tag{32}$$

Unprimed and primed variables of $\tilde{L}$ and $\tilde{L}_P$ encode the effect of a transition on the single active thread a, and $n-1$ passive threads $p \neq a$, respectively. As we can see from the conjunction in, e.g., eq. (30), for a valuation to be a valid transition, formula $\tilde{\mathcal{R}}$ must be satisfied no matter which thread $p \neq a$ takes the role of the passive thread. The semantics of passive-thread variables thus resembles that of inter-thread predicates.

We can now also demonstrate how active- and passive-thread variables can simulate shared variables (which we have excluded, to simplify the notation). To eliminate shared variable $\mathtt{s}$, we instead introduce a fresh local variable $\mathtt{m} \in \tilde{L}$, and replace a statement like $\mathtt{s} := 5$ by the atomic statement $\mathtt{m} := 5, \mathtt{m}_P := 5$, i.e., each thread keeps a local copy of what used to be the shared variable; the semantics of passive-thread variables ensures that the values are synchronised across all threads.

3.2.2 Computing an Abstract Dual-Reference Template

From the existential abstraction $\hat{\mathcal{P}}^n$ we derive a Boolean dual-reference program $\tilde{\mathcal{P}}$ (the "template") such that, for all n, the n-fold instantiation $\tilde{\mathcal{P}}^n$ overapproximates $\hat{\mathcal{P}}^n$. We begin with an example: a template characterisation of the abstract transition relation enumerated in Table 2 is given by the Boolean DR program with variables $\mathtt{b}$ and $\mathtt{b}_P$, and transition relation

$$(\neg\mathtt{b} \wedge \mathtt{b}_P \wedge \neg\mathtt{b}') \vee (\neg\mathtt{b}_P \wedge \mathtt{b}' \wedge \neg\mathtt{b}'_P)\,. \tag{33}$$

More specifically, we obtain the transition relation listed in Table 2, which is for the first thread in a 2-thread instantiation (hence $a = 1$ and $n = 2$), by plugging into eq. (30) as $\tilde{\mathcal{R}}$ the relation from eq. (33). Hence, we can use DR programs to exactly capture the abstraction, despite the loss of strict asynchrony.

In general, the variables of $\tilde{\mathcal{P}}$ are $\tilde{L} = \{\mathtt{b}[c] \,:\, 1 \leq c \leq m\}$ and $\tilde{L}_P = \{\mathtt{b}[c]_P \,:\, 1 \leq c \leq m\}$. Intuitively, $\tilde{\mathcal{P}}$ is obtained by exhaustively enumerating states and transitions that are feasible in $\hat{\mathcal{P}}^2, \hat{\mathcal{P}}^3, \ldots$ from the perspective of an active and a

$\mathtt{m}_1$	$\mathtt{m}_2$	$\mathtt{m}_3$	$\mathtt{m}'_1$	$\mathtt{m}'_2$	$\mathtt{m}'_3$	$\mathtt{b}_1$	$\mathtt{b}_2$	$\mathtt{b}_3$	$\mathtt{b}'_1$	$\mathtt{b}'_2$	$\mathtt{b}'_3$
1	0	0	0	0	0	F	F	F	F	F	F
1	1	0	0	1	0	F	F	T	F	F	F
2	1	0	1	1	0	F	F	T	F	F	T

Table 3: **Abstract transitions for $n = 3$** — The abstract 2-thread transition (highlighted) cannot be observed for $n = 2$ threads

passive thread. For fixed n, the components of $\tilde{\mathcal{P}}_n$ are

$$\tilde{\mathcal{R}}_n :: \tilde{\exists}\hat{L}_3, \ldots, \hat{L}_n \,.\; \hat{\mathcal{R}}(1)^n\{\hat{L}_1 \Rsh \tilde{L}\}\{\hat{L}_2 \Rsh \tilde{L}_P\} \tag{34}$$

$$\tilde{\mathcal{I}}_n :: \exists\hat{L}_3, \ldots, \hat{L}_n \,.\; \hat{\mathcal{I}}(1)^n\{\hat{L}_1 \triangleright \tilde{L}\}\{\hat{L}_2 \triangleright \tilde{L}_P\} \tag{35}$$

$$\tilde{\mathcal{E}}_n :: \exists\hat{L}_3, \ldots, \hat{L}_n \,.\; \hat{\mathcal{E}}(1)^n\{\hat{L}_1 \triangleright \tilde{L}\}\{\hat{L}_2 \triangleright \tilde{L}_P\} \tag{36}$$

Lemma 12. *$(\tilde{\mathcal{P}}_n)^n$ overapproximates $\hat{\mathcal{P}}^n$: For every $n \geq 2$, $\hat{\mathcal{I}}(a)^n \Rightarrow \tilde{\mathcal{I}}_n(a)^n$, $\hat{\mathcal{E}}(a)^n \Rightarrow \tilde{\mathcal{E}}_n(a)^n$, and $\hat{\mathcal{R}}(a)^n \Rightarrow \tilde{\mathcal{R}}_n(a)^n$.*

Proof. For the initial states, we have by eq. (35):

$$\forall a, p \,.\; p \neq a \Rightarrow (\hat{\mathcal{I}}(a)^n \Rightarrow \tilde{\mathcal{I}}_n\{\tilde{L} \triangleright \tilde{L}_a\}\{\tilde{L}_P \triangleright \tilde{L}_p\}) \,.$$

Hence $\hat{\mathcal{I}}(a)^n$ implies each conjunct of eq. (31), and thus $\hat{\mathcal{I}}(a)^n \Rightarrow \tilde{\mathcal{I}}_n(a)^n$. Similar reasoning proves the two other cases. □

An obstacle is that Lemma 12 depends on $\tilde{\mathcal{P}}_n$, which needs to be computed for every n individually. In order to illustrate that this dependence is crucial, let us revisit the decrement operation $\mathtt{m} := \mathtt{m} - 1$ and predicate $\mathtt{m} < \mathtt{m}_P$. We have already seen the corresponding transition relation $\tilde{\mathcal{R}}_2$ obtained according to eq. (34), namely in eq. (33). As we increase n from 2 to 3, the relation, however, changes to

$$\tilde{\mathcal{R}}_3 :: \tilde{\mathcal{R}}_2 \vee (\neg\mathtt{b} \wedge \neg\mathtt{b}_P \wedge \neg\mathtt{b}' \wedge \neg\mathtt{b}'_P) \,. \tag{37}$$

The inducing concrete transitions are listed in Table 3. We thus cannot use $\tilde{\mathcal{P}}_2$ as template to check safety of $\mathcal{P}^n$ for $n \geq 3$.

To enable parameterised reasoning in the abstract, we derive a tight "saturation bound", i.e. a thread count $\mathtt{b}$ such that, for any n, the n-fold instantiation of $\tilde{\mathcal{P}}_\mathtt{b}$ overapproximates $\hat{\mathcal{P}}_n$. Such bounds are guaranteed to exist as $\tilde{\mathcal{P}}$'s data domain (Boolean) is finite (note that $\mathcal{P}$'s data domain may well be *in*finite).

Theorem 13. *Suppose there are $\#_{IT}$ inter-thread predicates. Then the abstract dual-reference program* ***stabilises*** *at $\mathsf{b} = 4 \times \#_{IT} + 2$, i.e. $\tilde{\mathcal{P}}_n = \tilde{\mathcal{P}}_\mathsf{b}$ for every $n \geq \mathsf{b}$.*

Proof. Let $Q[1], \ldots, Q[m]$ be a list of predicates, $\#_{IT}$ of which are purely inter-thread, and let $\tilde{\mathcal{R}}_\infty$ denote the formula characterising $\bigvee_{n=1}^{\infty} \tilde{\mathcal{R}}_n$ (the existence of a finite encoding is guaranteed). We show that stabilisation occurs at $\mathsf{b} = 2+4\times\#_{IT}$, i.e., $\tilde{\mathcal{R}}_\infty \Rightarrow \tilde{\mathcal{R}}_\mathsf{b}$. The proof for stabilisation of $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{E}}$ is analogous (factor 4 then reduces to 2). We first show that stabilisation occurs for m inter-thread predicates at b, and then prove that this value is insensitive to the number of single-thread predicates.

Let $m = \#_{IT} = 1$ and $i = (\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_1', \mathtt{b}_2') \in \mathbb{B}^4$ be a transition in $\tilde{\mathcal{R}}_\infty$, and $\mathcal{D}_a^n(c) :: (\mathtt{b}[c]_a \Leftrightarrow \mathcal{C}_a(c))$. Then by definition of $\tilde{\mathcal{R}}_\infty$, eqs. (24) and (34) there exists a thread number $n \geq 2$, and a valuation v of variables $\tilde{V}_n$, $\tilde{V}_n'$, V_n and V_n' such that v satisfies $\mathcal{R}(1)^n \wedge \bigwedge_{a=1}^{2}(\mathtt{b}_a \Leftrightarrow \mathcal{C}_a \wedge \mathtt{b}_a' \Leftrightarrow \mathcal{C}_a{}') \wedge \bigwedge_{a=3}^{n} \mathcal{D}_a^n \wedge \mathcal{D}_a^{n\prime}$. Let

$$v = (\mathtt{b}_1, \ldots, \mathtt{b}_n, \mathtt{b}_1', \ldots, \mathtt{b}_n', \mathtt{s}, \mathtt{m}_1, \ldots, \mathtt{m}_n, \mathtt{s}', \mathtt{m}_1', \ldots, \mathtt{m}_n') \tag{38}$$

be that valuation. Then there exists a number $q \in [2, 6]$, and an index mapping $\pi : \{1, \ldots, q\} \to \{1, \ldots, n\}$ such that $(\mathtt{b}_{\pi_1}, \ldots, \mathtt{b}_{\pi_n}, \mathtt{b}_{\pi_1}', \ldots, \mathtt{b}_{\pi_n}', \mathtt{s}, \mathtt{m}_{\pi_1}, \ldots, \mathtt{m}_{\pi_n}, \mathtt{s}', \mathtt{m}_{\pi_1}', \ldots, \mathtt{m}_{\pi_n}')$ satisfies $\mathcal{R}^n 1 \wedge \mathcal{D}^n \wedge \mathcal{D}^{n\prime}$, namely by defining $\pi_1 = 1$, $\pi_2 = 2$, and letting $\pi_3, \ldots, \pi_q$ identify passive threads that falsify a conjunct in each of the expanded $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_1{}'$, and $\mathcal{C}_2{}'$ (if any)—recall that by eq. (12) formula $\mathcal{C}_a$ evaluates false exactly if

$$\exists p \neq a \; : \; \neg Q\{L \triangleright L_a\}\{L_P \triangleright L_p\}\,. \tag{39}$$

Then by eqs. (24) and (34) i satisfies $\tilde{\mathcal{R}}_q$ (and thus $\tilde{\mathcal{R}}_6$). We generalise the argument to one for arbitrary inter-thread predicates (case $m = \#_{IT} \geq 2$) by extending π accordingly (by 4 elements per additional predicate in the worst case). It follows that stabilisation occurs at $\mathsf{b} = 2 + 4 \times \#_{IT}$ for any $m = \#_{IT} \geq 1$.

It remains to show that stabilisation is not deferred by single-thread predicates. By eq. (13) it follows that the truth of such predicates depends only on the variables in V_n that are visible by the thread it is evaluated over, hence on variables V_2 and V_2' for any transition in $\tilde{\mathcal{R}}_\infty$. Now observe that these values are maintained in the permutation $(\mathtt{b}_{\pi_1}, \ldots)$ defined above—recall $\pi_1 = 1$ and $\pi_2 = 2$, which gives the desired result. □

Corollary 14. *Let $\tilde{\mathcal{P}} := \tilde{\mathcal{P}}_\mathsf{b}$, for b as in Theorem 13. The components of $\tilde{\mathcal{P}}$ are thus $(\tilde{\mathcal{R}}, \tilde{\mathcal{I}}, \tilde{\mathcal{E}}) = (\tilde{\mathcal{R}}_\mathsf{b}, \tilde{\mathcal{I}}_\mathsf{b}, \tilde{\mathcal{E}}_\mathsf{b})$. Then, for $n \geq 2$, $\tilde{\mathcal{P}}^n$ overapproximates $\hat{\mathcal{P}}^n$.*

Concrete transition ($n = 6$)										Abstract rel. $\tilde{\mathcal{R}}$			
$\mathtt{s}$	$\mathtt{m}_1$	$\mathtt{m}_2$	$\mathtt{m}_3$	$\mathtt{m}_4$	$\mathtt{m}_5$	$\mathtt{m}_6$	$\mathtt{s}'$	$\mathtt{m}'_1$	$\mathtt{m}'_i$	$\mathtt{b}$	$\mathtt{b}_P$	$\mathtt{b}'$	$\mathtt{b}'_P$
F	1	2	3	4	5	6	T	7	$\mathtt{m}_i$	F	F	F	F
F	1	1	3	5	$*$	$*$	T	7	$\mathtt{m}_i$	F	F	F	T

Table 4: **Tightness example for one inter-thread predicate: eq. (40)** — The first abstract transition (highlighted) requires $n = 6$ threads to be discovered, the second only $n = 4$

The bound established in Theorem 13 is asymptotically tight: consider the following concocted scenario with shared and local variables $\mathtt{s} \in \mathbb{B}$ and $\mathtt{m} \in [1, 7] \cap \mathbb{N}$:

$$\begin{aligned}\mathcal{R} ::& (\neg \mathtt{s} \wedge \mathtt{m} = 1) \wedge (\mathtt{s}' \wedge \mathtt{m}' = 7)\\ Q ::& (\neg \mathtt{s} \vee \mathtt{m} \neq 7 \vee \mathtt{m}_P \neq 5) \wedge (\neg \mathtt{s} \vee \mathtt{m} \neq 2 \vee \mathtt{m}_P \neq 6) \wedge\\ & (\mathtt{s} \vee \mathtt{m} \neq 1 \vee \mathtt{m}_P \neq 3) \wedge (\mathtt{s} \vee \mathtt{m} \neq 2 \vee \mathtt{m}_P \neq 4)\,. \end{aligned} \tag{40}$$

Equation (34) then does in fact not stabilise for less than $4 \times \#_{IT} + 2 = 6$ threads. The obtained DR program has two transitions, which are enumerated with an inducing concrete transition in Table 4. The generalisation necessary to show tightness for arbitrary numbers of inter-thread predicates is straightforward.

In practice, inspecting the actual structure of the given set of predicates often allows us to reduce the number $\mathtt{b}$ of threads used to formalise the abstraction process. For instance, if the set of predicates contains only local and shared predicates, $\mathtt{b} = 1$ is sufficient (one can exploit that the abstraction is guaranteed to be strictly asynchronous). As another instance, if inter-thread predicates are symmetric or contain no shared variables, then the factor 4 in Theorem 13 can be reduced to 3. All inter-thread predicates we have seen so far have this property. Similar constraints reduce the bound further: the abstraction for the decrement operation saturates at $n = 3 < \mathtt{b} = 6$, and the ticket lock at $n = 4 < \mathtt{b} = 10$ (147, 429, and 552 abstract transitions exist for $n = 2, 3, 4$, respectively).

As a consequence of losing asynchrony in the abstraction, many existing model checkers for concurrent software become inapplicable, e.g. [167, 62, 69]. For a fixed thread count n, the problem can be circumvented by forgoing the replicated nature of the concurrent programs, as done in [58] for the model checker boom: within 30 minutes, boom proves the ticket algorithm (Figure 5) correct up to $n = 3$. This approach is not only inefficient, but in fact ruins any chance of a parametric solution. Our solution to this problem first presents us with an unpleasant surprise.

3.3 Unbounded-Thread Dual-Reference Programs

The multi-threaded Boolean dual-reference programs $(\tilde{\mathcal{P}}_{\mathrm{b}})^n$ resulting from predicate-abstracting asynchronous programs against inter-thread predicates as shown in Section 3.1 are symmetric and free of recursion. Each of the n threads acts as a finite-state machine; their symmetry can therefore be exploited using classical methods that "counterise" the state space [90]: a global state is encoded as a vector of local-state counters, each of which records the number of threads currently occupying a particular local state.

In order to apply such fixed-thread solutions to the verification problem for unbounded thread numbers, the local state counters are extended to range over unbounded natural numbers $[0, \infty[$. The fact that the program executed by each thread is finite-state and in particular recursion-free now seems to suggest that the resulting infinite-state counter systems can be modelled as Petri nets or, more generally, as *well quasi-ordered transition systems* [1]. This would give rise to sound and complete (if worst-case expensive) algorithms for local-state reachability in such programs.

This strategy is ultimately workable, but not straightforwardly so. In this section, we illustrate the problem, and discuss a simple amendment to the abstraction presented in Section 3.1 that enables us to overcome that problem.

3.3.1 Undecidability of Boolean DR Program Verification

As it turns out, the full class of Boolean DR programs is expressive enough to render safety checking for an unbounded number of threads undecidable, despite the finite-domain variables:

Theorem 15. *Program location reachability for Boolean DR programs run by an unbounded number of threads is undecidable.*

Proof sketch. By reducing the halting problem for the Turing-powerful deterministic 2-counter Minsky machines [146] with k control states, to the program location reachability problem in DR programs with 3 program locations and a local variable with k values. We demonstrate the reduction using a deterministic Minsky machine that enumerates pairs in $\mathbb{N}^2$ (Figure 11; the formalism is from [163]). The machine consists of five control states $s_0, \ldots, s_4$ (s_0 initial), two natural-number counters c_1 and c_2 (initially 0), and three increment, two decrement and two zero-test operations, denoted for a counter c by $c{+}{+}$, $c{-}{-}$ and $c \stackrel{?}{=} 0$, respectively. Each operation

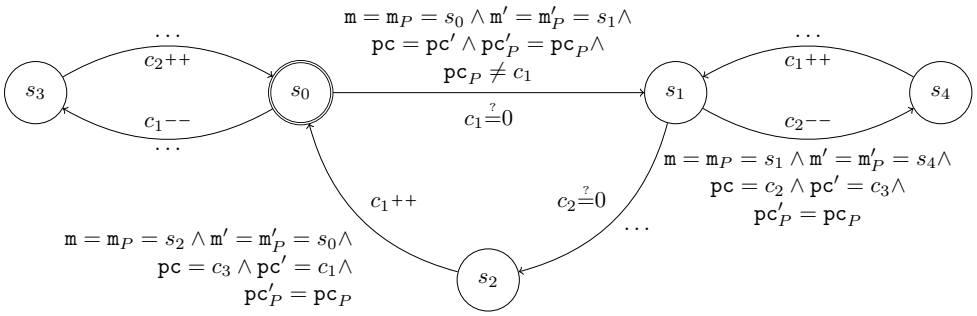

Figure 11: **Minsky machine and its DR program encoding** — (In and outside labels of control transitions, respectively.) The initial state $\mathcal{I}$ of the DR encoding is $\mathtt{m} = \mathtt{m}_P = s_0 \wedge \mathtt{pc} = \mathtt{pc}_P = c_3$

changes the control state and counter value as indicated (the decrement and zero-test operation freeze if c is zero and non-zero, respectively). Control states are encoded in local variables, and counters in program locations of the DR program $\mathcal{P}$ such that the counter value equals the number of threads in that location. Let further ℓ_e be a special program location of $\mathcal{P}$ that is reached if and only if the a local variable has the value that encodes the Minsky machine's halting state. Let c_3 be the single initial program location, thus with an unbounded number of threads; control state changes turn into local variable updates (for all threads), together with the following program counter modifications: for $c{+}{+}$ and $c{-}{-}$ a thread moves from c_3 to c and vice versa, and for $c \stackrel{?}{=} 0$ a thread in c_3 tests for the absence of a passive thread in the location associated with c. The machine halts if and only if program location ℓ_e is reached in $\mathcal{P}$. * □

Theorem 15 implies that the unbounded counter systems obtained straightforward from strictly asynchronous programs are in fact *not* well quasi-ordered. Why not? Can this problem be fixed, in order to permit a complete verification method? If so, at what cost? We answer these questions in the rest of this section.

*In the reduction used in the proof, neither the increment nor the decrement operations restrict the passive-thread variables to specific values, but the simulation of the zero test does: the test on variables c_1 and c_2 in the transition from s_0 to s_1 and s_1 to s_2 of Figure 11 are implemented through the absence of a transition for a passive thread in c_1 and c_2, respectively.

3.3.2 Monotonicity in Dual-Reference Programs

Recall that for a transition system $(\Sigma, \rightarrowtail)$ to be well-quasi ordered, we need two conditions to be in place [76, 1, 2]:

well-quasi-orderedness: there exists a relation $\leq$ on Σ that is reflexive and transitive, and such that for every infinite sequence $v_1, v_2, \ldots$ of states from Σ there exist i, j with $i < j$ and $v_i \leq v_j$.

monotonicity: for any v_1, v_1', v_2 with $v_1 \rightarrowtail v_1'$ and $v_1 \leq v_2$ there exists v_2' such that $v_2 \rightarrowtail v_2$ and $v_1' \leq v_2'$.

We now consider the case of dual-reference programs. If we represent global states of the abstract system $\tilde{\mathcal{R}}^n$ defined in Section 3.1 as counter tuples, we can define $\leq$ as the component-wise application of $\leq$, as done in Section 2 for strictly asynchronous programs.

We can now characterise monotonicity of DR programs as follows:

Lemma 16. *Let $\tilde{\mathcal{R}}$ be the transition relation of a DR program. Then the infinite-state transition system $\cup_{n=1}^{\infty}\tilde{\mathcal{R}}^n$ is monotone (with respect to $\leq$) exactly if, for all $k \geq 2$, the following formula is valid:*

$$(v, v') \in \tilde{\mathcal{R}}^k \Rightarrow \forall l_{k+1}\, \exists l'_{k+1}, \pi\,.\, \big(\langle v, l_{k+1}\rangle, \pi(\langle v', l'_{k+1}\rangle)\big) \in \tilde{\mathcal{R}}^{k+1}\,. \tag{41}$$

In eq. (41), the expression $\forall l_{k+1} \exists l'_{k+1} \ldots$ quantifies over valuations of the local variables of thread $k+1$. The notation $\langle v, l_{k+1}\rangle$ denotes a $(k+1)$-thread state that agrees with v in the first k local states and whose last local state is l_{k+1}; similarly $\langle v', l'_{k+1}\rangle$. Symbol π denotes a permutation on $\{1, \ldots, k+1\}$ that acts on states by acting on thread indices, which effectively reorders thread local states.

Proof of Lemma 16. "$\Rightarrow$": suppose $\cup_{n=1}^{\infty}\tilde{\mathcal{R}}^n$ is monotone. Let $v = (l_1, \ldots, l_k)$, $v' = (l'_1, \ldots, l'_k)$ with $(v, v') \in \tilde{\mathcal{R}}^k$, and $w = \langle v, l_{k+1}\rangle$. We have $v \leq w$, hence by the monotonicity of $\cup_{n=1}^{\infty}\tilde{\mathcal{R}}^n$ there exists w' such that (a) $(w, w') \in \cup_{n=1}^{\infty}\tilde{\mathcal{R}}^n$ and (b) $v' \leq w'$. From (a) we conclude that in fact $(w, w') \in \tilde{\mathcal{R}}^{k+1}$. From (b) we conclude that w' contains k threads in local states as in v'. Let l'_{k+1} be the local state of the additional thread (not necessarily the $k+1$st) in w', and σ be a permutation such that $(l'_1, \ldots, l'_{k+1}) = \sigma(w')$. That is, σ reorders the local states of w' such that

the k local states in v' come first, l'_{k+1} comes last. With $\pi := \sigma^{-1}$, we then have

$$\begin{aligned}(\langle v, l_{k+1}\rangle, \pi(\langle v', l'_{k+1}\rangle)) &= \Big(\langle v, l_{k+1}\rangle, \sigma^{-1}(\langle v', l'_{k+1}\rangle)\Big) \\ &= (w, w') \quad \in \quad \tilde{\mathcal{R}}^{k+1}\,.\end{aligned}$$

"$\Leftarrow$": suppose $(v, v') \in \cup_{n=1}^{\infty}\tilde{\mathcal{R}}^n$, say $(v, v') \in \tilde{\mathcal{R}}^k$, so we write $v = (l_1, \ldots, l_k)$ and $v' = (l'_1, \ldots, l'_k)$. Let further $v \leq w$. If w has k threads, like v, then $v \leq w$ implies $v \geq w$: the states are symmetry equivalent, say $w = \pi(v)$, for a permutation π on $\{1, \ldots, k\}$. In this case $w' := \pi(v')$ satisfies the monotonicity conditions.

If w has $k+1$ threads, then observe that w contains k threads in local states as in v; let l_{k+1} be the local state of the additional thread (not necessarily the $k+1$st). Let further l'_{k+1} and π be as provided in eq. (41). With $u = \langle v, l_{k+1}\rangle$ and $u' = \pi(\langle v', l'_{k+1}\rangle)$, we get $(u, u') \in \tilde{\mathcal{R}}^{k+1}$ by eq. (41). Since u and w contain the same local states, let σ be a permutation such that $\sigma(u) = w$. Define $w' = \sigma(u')$. Then $w' \sim u' = \pi(\langle v', l'_{k+1}\rangle) \geq v'$, where $\sim$ is symmetry equivalence. Further, $(u, u') \in \tilde{\mathcal{R}}^{k+1}$ implies $(\sigma(u), \sigma(u')) \in \tilde{\mathcal{R}}^{k+1}$ by symmetry, so $(w, w') \in \tilde{\mathcal{R}}^{k+1} \subseteq \cup_{n=1}^{\infty}\tilde{\mathcal{R}}^n$, demonstrating that the monotonicity conditions are satisfied.

The case that w has more than $k+1$ threads follows by induction. □

Asynchronous programs are trivially monotone (and DR): eq. (41) is satisfied by choosing $l'_{k+1} := l_{k+1}$ and π the identity. Table 5 shows instructions found in *non*-asynchronous programs that destroy monotonicity, and why. For example, the swap instruction in the first row gives rise to a DR program with a 2-thread transition $(\mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{F}) \in \tilde{\mathcal{R}}^2$. Choosing $l_3 = \mathtt{T}$ in eq. (41) requires the existence of a transition in $\tilde{\mathcal{R}}^3$ of the form $(\mathtt{m}_1, \mathtt{m}_2, \mathtt{m}_3, \mathtt{m}'_1, \mathtt{m}'_2, \mathtt{m}'_3) = (\mathtt{F}, \mathtt{F}, \mathtt{T}, \pi(\mathtt{F}, \mathtt{F}, \mathtt{m}'_3))$, which is impossible: by eqs. (4) and (30), there must exist $a \in \{1, 2, 3\}$ such that for $\{p, q\} = \{1, 2, 3\} \setminus \{a\}$, both "$a$ swaps with p" and "a swaps with q" hold, i.e.

$$\mathtt{m}'_p = \mathtt{m}_a \wedge \mathtt{m}'_a = \mathtt{m}_p \quad \wedge \quad \mathtt{m}'_q = \mathtt{m}_a \wedge \mathtt{m}'_a = \mathtt{m}_q\,,$$

which is equivalent to $\mathtt{m}'_a = \mathtt{m}_p = \mathtt{m}_q \wedge \mathtt{m}_a = \mathtt{m}'_p = \mathtt{m}'_q$. It is easy to see that this formula is inconsistent with the state description $(\mathtt{F}, \mathtt{F}, \mathtt{T}, \pi(\mathtt{F}, \mathtt{F}, \mathtt{m}'_3))$, no matter what $\mathtt{m}'_3$.

More interesting for us is the fact that asynchronous programs (= our input language) are monotone, while their parametric predicate abstractions may not be; this demonstrates that the monotonicity is in fact *lost in the abstraction*. Consider again

Dual-reference program		Monotonicity	
instruction	variables	mon.?	assgn. violating eq. (42)
$\mathtt{m}, \mathtt{m}_P := \mathtt{m}_P, \mathtt{m}$	$\mathtt{m} \in \mathbb{B}$	no	$\mathtt{m} = \mathtt{F}, \mathtt{m}' = \mathtt{T}$
$\mathtt{m}, \mathtt{m}_P := \mathtt{m}+1, \mathtt{m}_P - 1$	$\mathtt{m} \in \mathbb{N}$	yes	
$\mathtt{m}_P := \mathtt{m}_P + \mathtt{m}$	$\mathtt{m} \in \mathbb{N}$	yes	
$\mathtt{m} := \mathtt{m} + \mathtt{m}_P$	$\mathtt{m} \in \mathbb{N}$	no	$\mathtt{m} = \mathtt{m}' = 1$
$\mathtt{m}_P := c$	$\mathtt{m}, c \in \mathbb{N}$	yes	

Table 5: **Examples of monotonicity, and violations of it** — Each row shows a single-instruction program, whether the program gives rise to a monotone system and, if not, an assignment that violates eq. (42). (Some of these programs are not finite-state.)

the decrement instruction $\mathtt{m} := \mathtt{m} - 1$, but this time abstracted against the inter-thread predicate $Q :: \mathtt{m} = \mathtt{m}_P$. Parametric abstraction results in the two-thread and three-thread template instantiations

$$\begin{aligned} \tilde{\mathcal{R}}^2 &= (\neg \mathtt{b}_1 \vee \neg \mathtt{b}'_1) \wedge \mathtt{b}_1 = \mathtt{b}_2 \wedge \mathtt{b}'_1 = \mathtt{b}'_2 \\ \tilde{\mathcal{R}}^3 &= (\neg \mathtt{b}_1 \vee \neg \mathtt{b}'_1) \wedge \mathtt{b}_1 = \mathtt{b}_2 = \mathtt{b}_3 \wedge \mathtt{b}'_1 = \mathtt{b}'_2 = \mathtt{b}'_3 \,. \end{aligned}$$

Consider the two-thread transition from $(\mathtt{F}, \mathtt{F})$ to $(\mathtt{T}, \mathtt{T})$ and the three-thread state $w = (\mathtt{F}, \mathtt{F}, \mathtt{T}) > (\mathtt{F}, \mathtt{F})$: state w clearly has no successor. It is in fact inconsistent: the symmetry of predicate Q renders the role of the active thread immaterial; it is $Q_1 = Q_2 = Q_3 = (\mathtt{m}_1 = \mathtt{m}_2 = \mathtt{m}_3)$, so w has an empty concretisation. We discuss in Section 3.3.3 what happens to the instruction with respect to predicate $\mathtt{m} < \mathtt{m}_P$.

3.3.3 Restoring Monotonicity in the Abstraction

Our goal is now to restore the monotonicity that was lost in the parametric abstraction. The standard covering relation $\leq$ defined over local state counter tuples turns **monotone** and **Boolean** DR programs into instances of well quasi-ordered transition systems. Program location reachability is then decidable, even for unbounded threads.

In order to do so, we first derive a sufficient condition for monotonicity that can be checked **locally** over $\tilde{\mathcal{R}}$, as follows.

Theorem 17. *Let $\tilde{\mathcal{R}}$ be the transition relation of a DR program. Then the infinite-state transition system $\cup_{n=1}^{\infty} \tilde{\mathcal{R}}^n$ is monotone if the following formula over $\tilde{L} \times \tilde{L}'$ is valid:*

$$\exists \tilde{L}_P \tilde{L}'_P \,.\, \tilde{\mathcal{R}} \quad \Rightarrow \quad \forall \tilde{L}_P \exists \tilde{L}'_P \,.\, \tilde{\mathcal{R}} \,. \tag{42}$$

Proof. We show monotonicity using Lemma 16. Suppose $(v, v') \in \tilde{\mathcal{R}}^k$, and let l_{k+1} be given. By eq. (4), there exists $a \in \{1, \ldots, k\}$ such that $(v, v') \in \tilde{\mathcal{R}}(a)^k$. By eq. (30), we have

$$\forall p \in \{1, \ldots, k\} \setminus \{a\}\ \tilde{\mathcal{R}}\{\tilde{L} \triangleright \tilde{L}_a\}\{\tilde{L}_P \triangleright \tilde{L}_p\}\,. \tag{43}$$

Since $k \geq 2$, the quantification in eq. (43) is not empty and hence satisfies the left-hand side of eq. (42). By the right-hand side, there exists a valuation l'_{k+1} of all $\tilde{L}'_P$ variables such that, replacing the $\tilde{L}_P$ variables by the valuation l_{k+1}, $\tilde{\mathcal{R}}$ still holds, i.e. $\tilde{\mathcal{R}}\{\tilde{L} \triangleright \tilde{L}_a\}\{\tilde{L}_P \triangleright \tilde{L}_{k+1}\}$. Merging this with crefequation: instantiated lemma, we obtain

$$\forall p \in \{1, \ldots, k+1\} \setminus \{a\}\ \tilde{\mathcal{R}}\{\tilde{L} \triangleright \tilde{L}_a\}\{\tilde{L}_P \triangleright \tilde{L}_p\}\,,$$

and thus $(\langle v, l_{k+1}\rangle, \langle v', l'_{k+1},\rangle) \in \tilde{\mathcal{R}}(a)^{k+1} \subset \tilde{\mathcal{R}}^{k+1}$, establishing the right-hand side of eq. (41) with the identity permutation π. □

Unlike the monotonicity characterisation given in Lemma 16, eq. (42) is formulated only about the template program $\tilde{\mathcal{R}}$. It suggests that, if $\tilde{\mathcal{R}}$ holds for some valuation of its variables, then no matter how we replace the current-state passive-thread variables $\tilde{L}_P$, we can find next-state passive-thread variables $\tilde{L}'_P$ such that $\tilde{\mathcal{R}}$ is still valid. This holds for asynchronous programs, since here $\tilde{L}_P = \emptyset$. It fails for the swap instruction in the first row of Table 5: the instruction gives rise to the DR program $\tilde{\mathcal{R}} :: \mathtt{m}' = \mathtt{m}_P \wedge \mathtt{m}'_P = \mathtt{m}$. The assignment on the right in the table satisfies $\tilde{\mathcal{R}}$, but if $\mathtt{m}_P$ is changed to $\mathtt{F}$, $\tilde{\mathcal{R}}$ is violated no matter what value is assigned to $\mathtt{m}'_P$.

Equation (42) is strictly stronger than monotonicity. To see this, we revisit again the decrement operation abstracted against inter-thread predicate $\mathtt{m} < \mathtt{m}_P$, and the template $\tilde{\mathcal{R}}$ of the abstraction, which we determined to be eq. (37). Condition (42) is violated for $\mathtt{b} = \mathtt{b}' = \mathtt{T}$: this partial assignment can be continued via $\mathtt{b}_P = \mathtt{b}'_P = \mathtt{F}$ to a valid assignment for $\tilde{\mathcal{R}}$, yet for $\mathtt{b}_P = \mathtt{T}$ it cannot: $(\mathtt{T}, \mathtt{T}, \mathtt{T}, \mathtt{b}'_P) \notin \tilde{\mathcal{R}}$ for any $\mathtt{b}'_P$. Intuitively, $\mathtt{b} = \mathtt{b}_P = \mathtt{T}$ asserts that both threads have the unique minimum, which is impossible. On the other hand, it is not difficult to see that $\cup_{n=1}^{\infty} \tilde{\mathcal{R}}^n$ is monotone, e.g. using Lemma 16. The reason is that the parametrization of the existential abstraction (Section 3.2) has introduced spurious transitions, which permit inconsistent states such as $(\mathtt{T}, \mathtt{T})$.

We are now ready to modify the possibly non-monotone abstract DR program $\tilde{\mathcal{P}}$ into a new, monotone abstraction $\tilde{\mathcal{P}}_m$. Our solution is similar in spirit to, but different in effect from, earlier work on *monotonic abstractions* [4], which proposes to delete processes that violate universal guards and thus block a transition. This results in an overappoximation of the original system and thus possibly spuriously reachable error states. In contrast, exploiting the monotonicity of the *concrete* program $\mathcal{P}$, we can build a monotone program $\tilde{\mathcal{P}}_m$ that is safe exactly when $\tilde{\mathcal{P}}$ is, thus fully preserving soundness and precision of the abstraction $\tilde{\mathcal{P}}$.

Definition 18. *The **non-monotone fragment (NMF)** of a DR program with transition relation $\tilde{\mathcal{R}}$ is the formula over $\tilde{L} \times \tilde{L}_P \times \tilde{L}'$:*

$$\mathcal{F}(\tilde{\mathcal{R}}) \quad :: \quad \exists \tilde{L}_P \tilde{L}'_P \,:\, \tilde{\mathcal{R}} \;\wedge\; \neg \exists \tilde{L}'_P \,:\, \tilde{\mathcal{R}}\,. \tag{44}$$

The NMF encodes partial assignments $(\mathtt{m}, \mathtt{m}_P, \mathtt{m}')$ that have no continuation (via any $\mathtt{m}'_P$) to a full assignment satisfying $\tilde{\mathcal{R}}$, but do have a continuation for some valuation of $\tilde{L}_P$ other than $\mathtt{m}_P$. We revisit the two non-monotone instructions from Table 5. The NMF of $\mathtt{m}, \mathtt{m}_P := \mathtt{m}_P, \mathtt{m}$ is $\mathtt{m}' \neq \mathtt{m}_P$: this clearly cannot be continued to an assignment satisfying $\tilde{\mathcal{R}}$, but when $\mathtt{m}_P$ is changed so that $\mathtt{m}' = \mathtt{m}_P$, we can choose $\mathtt{m}'_P = \mathtt{m}$ to satisfy $\tilde{\mathcal{R}}$. The non-monotone fragment of $\mathtt{m} := \mathtt{m} + \mathtt{m}_P$ is $\mathtt{m}' \geq \mathtt{m} \wedge \mathtt{m}' \neq \mathtt{m} + \mathtt{m}_P$.

Equation (44) is slightly stronger than the negation of eq. (42): the NMF binds the values of the $\tilde{L}_P$ variables for which a violation of $\tilde{\mathcal{R}}$ is possible. It can be used to "repair" the transition relation:

Lemma 19. *For a DR program with transition relation $\tilde{\mathcal{R}}$, the program with transition relation $\tilde{\mathcal{R}} \vee \mathcal{F}(\tilde{\mathcal{R}})$ is monotone.*

Proof. We show that $\tilde{\mathcal{R}} \vee \mathcal{F}(\tilde{\mathcal{R}})$ satisfies eq. (42), i.e.

$$\exists \tilde{L}_P \tilde{L}'_P \,.(\tilde{\mathcal{R}} \vee \mathcal{F}(\tilde{\mathcal{R}})) \quad \Rightarrow \quad \forall \tilde{L}_P \exists \tilde{L}'_P \,:\, (\tilde{\mathcal{R}} \vee \mathcal{F}(\tilde{\mathcal{R}}))\,. \tag{45}$$

Monotonicity then follows using Theorem 17. Simplifying the right-hand side of eq. (45) yields

$$\begin{aligned}
& \forall \tilde{L}_P \exists \tilde{L}'_P \,:\, (\tilde{\mathcal{R}} \vee (\exists \tilde{L}_P \tilde{L}'_P \,:\, \tilde{\mathcal{R}} \wedge \neg \exists \tilde{L}'_P \,:\, \tilde{\mathcal{R}})) \\
= \;& \forall \tilde{L}_P \,:\, (\exists \tilde{L}'_P \,:\, \tilde{\mathcal{R}} \vee (\exists \tilde{L}_P \tilde{L}'_P \,:\, \tilde{\mathcal{R}} \wedge \neg \exists \tilde{L}'_P \,:\, \tilde{\mathcal{R}})) \\
= \;& \forall \tilde{L}_P \,:\, (\exists \tilde{L}'_P \,:\, \tilde{\mathcal{R}} \vee \exists \tilde{L}_P \tilde{L}'_P \,:\, \tilde{\mathcal{R}}) \\
= \;& \forall \tilde{L}_P \,:\, (\exists \tilde{L}_P \tilde{L}'_P \,:\, \tilde{\mathcal{R}}) \\
= \;& \exists \tilde{L}_P \tilde{L}'_P \,:\, \tilde{\mathcal{R}}\,.
\end{aligned}$$

Equation (45) now becomes

$$\exists \tilde{L}_P \tilde{L}'_P . (\tilde{\mathcal{R}} \vee (\exists \tilde{L}_P \tilde{L}'_P \ : \ \tilde{\mathcal{R}} \ \wedge \ \neg \exists \tilde{L}'_P \ : \ \tilde{\mathcal{R}})) \Rightarrow \exists \tilde{L}_P \tilde{L}'_P \ : \ \tilde{\mathcal{R}}$$

which trivially reduces to $true$, in both cases of the disjunction. □

Lemma 19 suggests to add artificial transitions to $\tilde{\mathcal{P}}$ that allow arbitrary passive-thread changes in states of the non-monotone fragment, thus lifting the blockade previously caused by some passive threads. While this technique restores monotonicity, the problem is of course that such arbitrary additions will generally modify the program behavior; in particular, an added transition may lead a thread directly into an error state that used to be unreachable.

In order to instead obtain a *safety-equivalent* program, we prevent passive threads that block a transition in $\tilde{\mathcal{P}}^n$ from affecting the future execution. This can be realised by redirecting them to an auxiliary local sink state. Let ℓ_e be a fresh label non-existent in $\tilde{\mathcal{P}}$.

Definition 20. *The **monotone closure** of DR program $\tilde{\mathcal{P}} = (\tilde{\mathcal{R}}, \tilde{\mathcal{I}})$ is the DR program $\tilde{\mathcal{P}}_m = (\tilde{\mathcal{R}}_m, \tilde{\mathcal{I}})$ with transition relation $\tilde{\mathcal{R}}_m :: \tilde{\mathcal{R}} \vee (\mathcal{F}(\tilde{\mathcal{R}}) \wedge (\mathtt{pc}'_P = \ell_e))$.*

This extension of the transition relation has the following effects: (i) for any program state, if any passive thread can make a move, so can all, ensuring monotonicity, (ii) the added moves do not affect the safety of the program, and (iii) transitions that were previously possible are retained, so no behavior is removed. The following theorem summarises these claims:

Theorem 21. *Let $\mathcal{P}$ be an asynchronous program, and $\tilde{\mathcal{P}}$ its parametric predicate abstraction. The monotone closure $\tilde{\mathcal{P}}_m$ of $\tilde{\mathcal{P}}$ is monotone. Further, $(\tilde{\mathcal{P}}_m)^n$ is safe exactly if $\tilde{\mathcal{P}}^n$ is.*

Proof. Monotonicity of $\tilde{\mathcal{P}}_m$: appealing to Theorem 17, we prove that the following formula is valid:

$$\exists \tilde{L}_P \tilde{L}'_P . \tilde{\mathcal{R}}_m \quad \Rightarrow \quad \forall \tilde{L}_P \exists \tilde{L}'_P . \tilde{\mathcal{R}}_m .$$

Let $(\mathtt{l}, \mathtt{l}') \in \tilde{L} \times \tilde{L}'$ be arbitrary, and suppose there exist $(\mathtt{l}_P, \mathtt{l}'_P) \in \tilde{L}_P \times \tilde{L}'_P$ such that $(\mathtt{l}, \mathtt{l}_P, \mathtt{l}', \mathtt{l}'_P) \in \tilde{\mathcal{R}}_m$. Let further $\mathtt{m}_P \in \tilde{L}_P$. We construct $\mathtt{m}'_P \in \tilde{L}'_P$ such that $(\mathtt{l}, \mathtt{m}_P, \mathtt{l}', \mathtt{m}'_P) \in \tilde{\mathcal{R}}_m$. Since $\tilde{\mathcal{R}}_m = \tilde{\mathcal{R}} \vee (\mathcal{F}(\tilde{\mathcal{R}}) \wedge (\mathtt{pc}'_P = \ell_e))$, we have either $(\mathtt{l}, \mathtt{l}_P, \mathtt{l}', \mathtt{l}'_P) \in \tilde{\mathcal{R}}$ or $(\mathtt{l}, \mathtt{l}_P, \mathtt{l}', \mathtt{l}'_P) \in \mathcal{F}(\tilde{\mathcal{R}}) \wedge (\mathtt{pc}'_P = \ell_e)$. In both cases, $(\mathtt{l}, \mathtt{l}_P, \mathtt{l}', \mathtt{l}'_P) \in \tilde{\mathcal{R}} \vee \mathcal{F}(\tilde{\mathcal{R}})$. The latter relation is monotone by Lemma 19.

Hence there exists some $\mathtt{k}'_P$ such that $(\mathtt{l}, \mathtt{m}_P, \mathtt{l}', \mathtt{k}'_P) \in \tilde{\mathcal{R}} \vee \mathcal{F}(\tilde{\mathcal{R}})$. This element $\mathtt{k}'_P$ is almost the element $\mathtt{m}'_P \in \tilde{L}'_P$ we are looking for: if $(\mathtt{l}, \mathtt{m}_P, \mathtt{l}', \mathtt{k}'_P) \in \tilde{\mathcal{R}} \subset \tilde{\mathcal{R}}_m$, then the choice $\mathtt{m}'_P = \mathtt{k}'_P$ ensures $(\mathtt{l}, \mathtt{m}_P, \mathtt{l}', \mathtt{m}'_P) \in \tilde{\mathcal{R}}_m$. Otherwise $(\mathtt{l}, \mathtt{m}_P, \mathtt{l}', \mathtt{k}'_P) \in \mathcal{F}(\tilde{\mathcal{R}})$. Formula $\mathcal{F}(\tilde{\mathcal{R}})$ does not contain $\tilde{L}'_P$ variables, however; the latter can thus be replaced freely without affecting membership in $\mathcal{F}(\tilde{\mathcal{R}})$. Let therefore $\mathtt{m}'_P :: (\exists \mathtt{pc}'_P \mathtt{k}'_P) \wedge (\mathtt{pc}'_P = \ell_e)$. The latter expression denotes the replacement of the value of $\mathtt{pc}'_P$ in $\mathtt{k}'_P$ by ℓ_e. Now we have $(\mathtt{l}, \mathtt{m}_P, \mathtt{l}', \mathtt{m}'_P) \in \mathcal{F}(\tilde{\mathcal{R}})$ and in fact $(\mathtt{l}, \mathtt{m}_P, \mathtt{l}', \mathtt{m}'_P) \in \mathcal{F}(\tilde{\mathcal{R}}) \wedge (\mathtt{pc}'_P = \ell_e) \subset \tilde{\mathcal{R}}_m$.

Safety-equivalence: from Definition 20 (applied to $\tilde{\mathcal{P}}$) we conclude $\tilde{\mathcal{R}} \Rightarrow \tilde{\mathcal{R}}_m$ is valid, and thus every execution of $\tilde{\mathcal{P}}$ is an execution of $\tilde{\mathcal{P}}_m$. Thus if $\tilde{\mathcal{P}}_m$ is safe, so is $\tilde{\mathcal{P}}$. For the converse argument observe that every infinite trace π of $\tilde{\mathcal{P}}_m$ gives rise to a sequence of j traces of $\tilde{\mathcal{P}}$ as follows:

$$\pi = t_1, \ldots, r_1, t_2, \ldots, r_2, \ldots, t_j, \ldots$$

such that for all i, subtrace $t_i, \ldots, r_i$ is pairwise related by $\tilde{\mathcal{R}}$, $(r_i, t_{i+1}) \notin \tilde{\mathcal{R}}$, yet $(r_i, t_{i+1}) \in \tilde{\mathcal{R}}_m$. (If π is finite it is of the form $t_1, \ldots, r_1, \ldots, t_j, \ldots, r_j$; the following remains valid.) Call a state *safe* if it has no emanating execution ending in an error state. Observe that because the asynchronous input program $\mathcal{P}$ is monotone ("fewer threads can do less"), state-safety is $<$-closed for $\mathcal{P}$: if a state r is safe in $\mathcal{P}$ and $s < r$ then s is also safe. In order to see that the same is true for the (possibly non-monotone) abstract DR program $\tilde{\mathcal{P}}$, let R be the concretisation of a state r of $\tilde{\mathcal{P}}$, i.e. a set of programs states of input program $\mathcal{P}$. Then $\tilde{\mathcal{P}}$'s conservativeness (Section 3.1.2 and Corollary 14) guarantee the safety of states in R, and $<$-closedness of state-safety in $\mathcal{P}$ implies the safety of states in the $<$-downward closure of R. From the fact that s's concretisation is in that closure we can conclude that state-safety is also $<$-closed for $\tilde{\mathcal{P}}$.

Using the previous result we next show that if a subtrace $t_i, \ldots, r_i$ of π contains no error state then $t_{i+1}, \ldots, r_{i+1}$ also contains none, which (by induction) gives us the desired results; $t_1, \ldots, r_1$ contains no error state (otherwise $\tilde{\mathcal{P}}$ cannot be safe). The prove of the induction step goes by contradiction. Assume $t_i, \ldots, r_i$ contains no error state, yet $t_{i+1}, \ldots, r_{i+1}$ does so. Let r'_i be a state such that $r'_i < r_i$ and $(r'_i, t_{i+1}) \in \tilde{\mathcal{R}}$. Such a state is always guaranteed to exist.* Hence r_i is safe, $r'_i < r_i$, yet r'_i not safe, which contradicts the property that state-safety is $<$-closed for $\tilde{\mathcal{P}}$ and gives the desired result. □

*Such a state can always be obtained from r_i by removing the threads that were redirected to an auxiliary state in transition $(r_i, t_{i+1}) \in \tilde{\mathcal{R}}_m$.

Theorem 21 justifies our strategy for reachability analysis of an asynchronous program $\mathcal{P}$: form its parametric predicate abstraction $\tilde{\mathcal{P}}$ described in Sections 3.1 and 3.2, build the monotone closure $\tilde{\mathcal{P}}_m$, and analyse $(\tilde{\mathcal{P}}_m)^\infty$ using any technique for monotone systems.

Proving the parameterized ticket lock. We apply this strategy to the ticket lock algorithm. The backward reachability method described in [1] returns "uncoverable", confirming that the ticket algorithm guarantees mutual exclusion, this time for arbitrary thread counts. The search tree has 107 nodes. More compact proofs can be obtained using the inductive model-checker breach, which we will present in the second part of this thesis (Section 4); the reachability tree it generates has 16 vertices (shown in Figure 12) and thus only three states more than were obtained earlier for a constant number of $n = 2$ threads (Figure 10).

Recall that the ticket lock is challenging for existing techniques: cream [95], slab [62] and symmpa [58] handle only a fixed number of threads, and the resource requirements of these algorithms grow rapidly; none of them can handle even a handful of threads. The recent approach from [71] generates polynomial-size proofs, but again only for fixed thread counts.

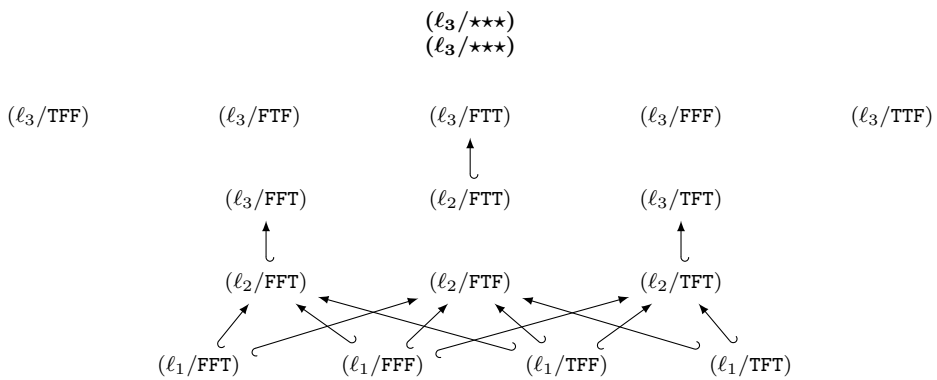

Figure 12: **Minimal uncoverability proof for the ticket lock** — The breach model-checker attempts to prove uncoverability of *smaller* ($<$) undecided elements first, which is why some (larger) elements are not expanded

3.3.4 Expressiveness of Monotone Boolean DR Programs

We show that monotone Boolean DR programs are stricly more expressiveness than strictly asynchronous Boolean programs, and illustrate why the latter are inadequate as an abstract domain model for single- and inter-thread predicate abstractions.

Lemma 22. *Monotone Boolean DR programs are strictly more expressive than strictly asynchronous Boolean programs.*

Proof. Reusing the reduction used in the proof of Theorem 15, we can conclude that strictly asynchronous Boolean programs are exactly as expressive as vector addition systems with states [121], which in turn are as powerful as *zero test-free* Minsky machines. On the other hand, *monotone* Boolean DR programs are as expressive as transfer Petri nets [39], or equivalently zero test-free Minsky machines *with* transfers operations. The latter atomically add the value of one counter to another, and then clear the former. In order to illustrate the equivalence, let us first consider the Minsky machine used in the proof of Theorem 15, yet assume the zero-test $c_1 \stackrel{?}{=} 0$ between s_0 and s_1 is replaced by a transfer operation from c_1 to c_2. The corresponding transition has the *monotone* DR encoding

$$\begin{aligned} &(\mathtt{l} = \mathtt{l}_P = s_0 \wedge \mathtt{l}' = \mathtt{l}'_P = s_1) \wedge \\ &(\mathtt{pc} \ \ = c_1 \Rightarrow \mathtt{pc}' \ \ = c_2) \wedge (\mathtt{pc} \ \ \neq c_1 \Rightarrow \mathtt{pc}' \ \ = \mathtt{pc} \ \ ) \wedge \\ &(\mathtt{pc}_P = c_1 \Rightarrow \mathtt{pc}'_P = c_2) \wedge (\mathtt{pc}_P \neq c_1 \Rightarrow \mathtt{pc}'_P = \mathtt{pc}_P) \ . \end{aligned} \tag{46}$$

In this manner, every zero test-free Minsky machines with transfers operations can be encoded as a monotone Boolean DR program. The other direction directly follows from the fact that every Boolean DR program can be transformed into one exhibiting transitions that modify at most one valuation of passive-thread variables at a time. Each of the passive-thread updates can then be represented by a transfer operation in the constructed zero test-free Minsky machines with transfers operations. The fact that transfer Petri nets are strictly more expressive than vector addition systems with states [63] concludes the proof. □

Why is this additional expressive power necessary for our abstract model? The reason is that we must sometimes *force* a passive thread in an abstract transition to leave its local state and move to another one. For example in the abstract reachability tree of ticket lock abstraction shown in Figure 10 the local state of the passive thread changes in all successors of the root state from $(\ell_1/\mathtt{FTF})$ to $(\ell_1/\mathtt{TTF})$. If the number of executing threads in unbounded, this behaviour cannot be simulated

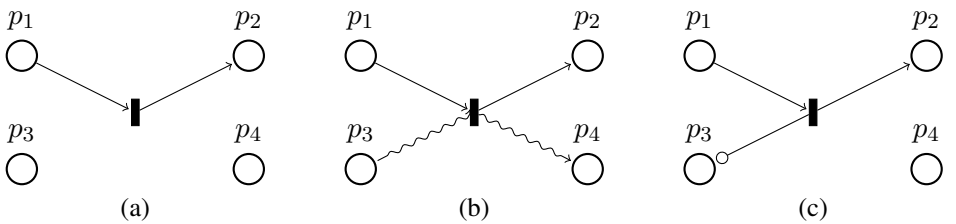

Figure 13: **Petri nets** — (a) Petri net; (b) Petri net with transfer arc; (c) Petri with inhibitor arc. Standard transitions move a finite number of tokens (here: from p_1 to p_2), transfer arcs can additionally move all tokens between place (here: from p_3 to p_4), and inhibitor arcs can block a transition depending on the existence of a tokens in some place (here: p_3)

by a strictly asynchronous Boolean program. We therefore consider our monotone Boolean dual-reference programs a "tight" abstract domain for single- and inter-thread predicate abstractions.

In fact it is not difficult to show that strictly asynchronous Boolean programs can at most simulate "optional" passive-thread updates, i.e. where every passive thread is left with the possibility to retain its local state. For example all passive-thread updates in the abstraction enumerated in Table 2 are in this sense optional.

The following example sketches the equivalence of (i) strictly asynchronous Boolean programs and Petri nets, (ii) monotone Boolean DR programs and Petri nets with transfer arcs, and (iii) non-monotone Boolean DR programs and Petri nets with inhibitor arcs; we refer the reader to [39] for details on the Petri net formalisms.

Example 23. *Figure 13 depicts for each of the three models a simple (1-transition) net over places p_1 to p_4.* We can encode them as DR programs over two local Boolean variables $\mathtt{b}[1]$ and $\mathtt{b}[2]$ (no shared variables), i.e. $L = \{\mathtt{b}[1], \mathtt{b}[2]\}$ and $S = \emptyset$. In particular, let the places be represented as valuations in this order:*

$$\begin{aligned} p_1 &:: \neg\mathtt{b}[1] \wedge \neg\mathtt{b}[2] \\ p_2 &:: \neg\mathtt{b}[1] \wedge \mathtt{b}[2] \\ p_3 &:: \mathtt{b}[1] \wedge \neg\mathtt{b}[2] \\ p_4 &:: \mathtt{b}[1] \wedge \mathtt{b}[2] \end{aligned}$$

*More complex transitions can be split into uninterruptible sequences of such simple transitions using fresh intermediate shared states.

Equipped with these place encodings, Figures 13a to 13c have the following in common: The active-thread variables $\mathtt{b}[1]$ *and* $\mathtt{b}[2]$ *are updated in the same way, and the passive-thread variable* $\mathtt{b}[1]_P$ *does not change, which we record by the following relation:*

$$f :: (\neg\mathtt{b}[1] \wedge \neg\mathtt{b}[2]) \wedge (\neg\mathtt{b}[1]' \wedge \mathtt{b}[2]') \wedge (\mathtt{b}[1]'_P \Leftrightarrow \mathtt{b}[1]_P)\,.$$

The influence of, and effect on the passive-thread variable $\mathtt{b}[2]$ *is what distinguishes the three transitions: For Figure 13a it does not change, hence*

$$\mathcal{R} :: f \wedge (\mathtt{b}[2]'_P \Leftrightarrow \mathtt{b}[2]_P)\,. \tag{47}$$

For Figure 13b passive threads residing in local state $\mathtt{b}[1]\wedge\neg\mathtt{b}[2]$ *move to* $\mathtt{b}[1]\wedge\mathtt{b}[2]$*, hence*

$$\mathcal{R} :: f \wedge (\mathtt{b}[2]'_P \Leftrightarrow (\mathtt{b}[1]_P \vee \mathtt{b}[2]_P))\,. \tag{48}$$

Finally, for Figure 13c the transition of the active thread is restricted to cases where the passive-thread variable $\mathtt{b}[2]$ *evaluates true (in that case* $\mathtt{b}[2]$ *does not change), hence*

$$\mathcal{R} :: f \wedge (\mathtt{b}[1]_P \Rightarrow \mathtt{b}[2]_P) \wedge (\mathtt{b}[2]'_P \Leftrightarrow \mathtt{b}[2]_P)\,. \tag{49}$$

Clearly, eq. (47) *does not change passive-thread variables and hence is strictly a-synchronous. Furthermore, it is not difficult to check via eq.* (44) *that eqs.* (48) *and* (49) *are monontone and non-monotone, respectively.*

3.4 Extension to Programs with Nonblocking Condition Variables

The key property that makes our approach work is the monotonicity of the input programs. Strictly asynchronous programs are monotone, and many human-written programs are naturally strictly asynchronous. However, the approach remains valid with certain non-asynchronous monotone input programs, such as our monotone DR programs; notice that the latter generalisation does not affect the proof of Theorem 13. The shift from strictly asynchronous to monotone DR programs as input has a benefit: It enables us to reason about programs with more complex synchronisation primitives, notably *condition variables* [129].

Condition variables enable threads to wait until a designated condition occurs (e.g., I/O completion), and permit the implementation of high-level synchronisation mechanisms like bounded circular buffers. They are by now available in all major operating systems, e.g. Windows, UNIX and Mac OS. A condition variable offers the following routines:

- `cv_init` for initialisation,
- `cv_broadcast` to wake up all passive threads that currently wait on it, and
- `cv_wait` to block the caller until the condition is broadcast.

To avoid race conditions, `cv_wait` is used in conjunction with a mutex that is released and reacquired on function entry and return, respectively. An example of condition variables in action is shown in Figure 1. Calls to function `cv_broadcast` are *non-blocking*: The calling thread does not relinquish control, yet as a side effect "causes execution of [threads] waiting on the condition to resume at some convenient future time" [129]. In particular, the call returns without side effect if no thread currently blocks on the condition variable. Typically, they are implemented via lists of waiting threads, where a call to `cv_wait` adds the executing thread to the respective list, and `cv_broadcast` makes all threads runnable, before the list is cleared. Notice that thread communication mechanisms such as locking and unlocking operations on mutexes are *blocking*.

The non-blocking semantics of `cv_broadcast` makes them intricate to model. In fact, if every thread's memory is finite, yet the number of executing threads is unbounded, it provably *cannot* in general be encoded in a strictly asynchronous program [53]; note that in such a setting lists of thread identifiers could, e.g. become arbitrarily long. On the other hand, modelling a condition variable using a local Boolean variable in a dual-reference program is straightforward, as shown in Figure 14:

(i) function `cv_init` sets the local flag of *all* threads to `PEN` (e.g. false), thereby indicating that the condition has not yet been broadcast,
(ii) `cv_broadcast` sets the local flag of *all passive* threads to `BRC` (e.g. true), and
(iii) `cv_wait` is implemented by busy-waiting until the owned local flag is broadcast, i.e. set to `BRC` (i.e. true).

Modifications of passive-thread variables are required in function `cv_init`, and again in `cv_broadcast`. However, since both DR instructions are monotone (cp. Table 5) our approach remains valid. (We use this encoding for the programs with broadcasts in Section 6.)

```
enum SIG_t { PEN, /*condition is pending*/
             BRC }; /*condition has been broadcast*/

struct cv{ thread_local SIG_t l; }; /*local flag l*/

void cv_init(struct cv *cvp, /*...*/) {
  (*cvp).l := PEN; /*set active thread pending*/
▶ (*cvp).m_P := PEN; } /*set all passive threads pending*/

void cv_wait(struct cv *cvp, struct mtx *mp) {
  mtx_unlock(mp); /*release protection mutex*/
  while((*cvp).l ≠ BRC)
    ; /*wait on local flag*/
  (*cvp).l := PEN; /*thread is awake; reset flag*/
  mtx_lock(mp); /*reacquire before return*/}

void cv_broadcast(struct cv *cvp) {
▶ (*cvp).m_P := BRC; /*wake up all passive threads*/ }
```

Figure 14: **Expressing broadcast operations in monotone DR programs** — The local flag `l` in structure `cv` stores whether the owning thread has been broadcast to or not (`BRC` and `PEN`, respectively). The instructions accessing the passive-thread variable $\texttt{l}_P$ are marked

4 A Widening Approach to Multi-Threaded Program Verification*

In this section, we present an algorithmic solution for verifying safety properties of monotone Boolean dual-reference programs, such as obtained by our predicate abstraction technique introduced in Section 3 after applying the monotone closure operator. The approach is, however, applicable to the more general class of well quasi-ordered systems, which subsume many other concurrency models like Petri nets [157] (or, equivalently Vector Addition Systems [121, 97]), Broadcast protocols [65, 39] and Lossy counter machines [139]. Due to our focus on verifying shared-memory programs, and despite the generality of our method, we will primarily use strictly asynchronous programs to illustrate the concepts and intuition behind our approach. We first recap relevant concepts [1], then we illustrate and formalise our target set widening approach.

4.1 Coverability Analysis by Target Set Widening

The method presented in this section pursues a strategy that may seem counter-intuitive at first: instead of focusing on the original input program state e, whose coverability we wish to determine, the algorithm builds a hierarchy of elements for which coverability is tested; state e itself may well never be directly investigated. The motivation is that uncoverability of strictly covered elements, which suffices to prove the uncoverability of e, is likely to terminate more quickly, as it involves fewer threads. Our approach thus biases the search algorithm towards proving *un*coverability of elements. An external coverability generator, which will typically trade in termination in favour of efficiency, intervenes in case the search attempts to prove uncoverability of coverable states.

In the remainder of this section, we review the most general solution to coverability analysis in well quasi-ordered systems [2], then we illustrate and formalise the idea sketched above, and finally describe our algorithm in detail. We use the program in Figure 15 as a running example; Figure 16 enumerates its transitions.

*The main content of this chapter was previously presented in [117, 118, 119].

shared s := 0; *//shared; range {0,1,2,3}*
local pc := 0; *//(implicit) local instruction counter; range* $\{0, 1, 2\}$

0: **if** (s $\neq$ 0){ **goto** 0; } **else** { s := 3; }
1: **if** (s $\neq$ 3){ **goto** 1; }
2: **if** (s = 2){ **goto** 2; } s = (s + 1) mod 4; **goto** 0;

Figure 15: **Running example** — A strictly asynchronous program with 3 atomic statements, labelled 0, 1 and 2, and a finite-domain shared variable `s`. Each thread can perform only one execution sequence: from the initial state $(\mathtt{s}, \mathtt{pc}) = (0, 0)$ to $(3, 1)$ to $(3, 2)$, and finally back to $(0, 0)$. Hence, only states with $s \in \{0, 3\}$ are reachable, and the jump instructions `goto 0`, `goto 1` and `goto 2` in locations $\mathtt{pc} = 0$, $\mathtt{pc} = 1$ and $\mathtt{pc} = 2$, respectively, are never executed

Transition	s	pc	$\mathtt{pc}_P$	s′	pc′	$\mathtt{pc}'_P$
t_1	1	2	ℓ	2	0	ℓ
t_2	0	2	ℓ	1	0	ℓ
t_3	3	2	ℓ	0	0	ℓ
t_4	3	1	ℓ	3	2	ℓ
t_5	0	0	ℓ	3	1	ℓ

Figure 16: **Explicit dual-reference transitions** — Transitions induced by the program in Figure 15; because the input program is strictly asynchronous, the passive-thread variable remain unchanged: $\ell \in \{0, 1, 2\}$. The target state is $(2 \,|\,)$, i.e. we want to check if $\mathtt{s} = 2$ for some reachable state

Algorithm 1 $\mathsf{bc}(I, e)$

Require: initial state set I, query $e \notin I$

```
1: W := {e} ; U := {e}
2: while ∃w ∈ W do
3:     W := W \ {w}
4:     for all p ∈ C-Pre(w) \ ↑U do
5:         if p ∈ I then
6:             return "e ∈ Cover"
7:         W := min(W ∪ {p})
8:         U := min(U ∪ {p}))
9: return "e ∉ Cover"
```

4.1.1 Review: Backward Coverability Analysis

Given a state $v \in \Sigma$, the set of all predecessors of elements in its upward closure $\uparrow v$ is again upward-closed and can therefore be represented by its minimal elements. We call these minimal elements the *cover predecessors* of v and denote them by $\mathsf{C\text{-}Pre}(v)$:

$$\mathsf{C\text{-}Pre}(v) = \min \bigcup_{r \geq v} \{p \in \Sigma \;:\; p \rightarrowtail r\}\,.$$

We write $p \hookrightarrow v$ for $p \in \mathsf{C\text{-}Pre}(v)$. Note that the number of threads in a program state and its cover predecessors can differ: we will see many examples later where a thread can, for instance, enter a particular program state $(s \mid c)$ only if another thread in some other local state h "helps", by setting the shared state to s. The 1-thread state $(s \mid c)$ then has a 2-thread cover predecessor.

Algorithm 1 shows a version of the *classical backward search* to decide coverability for well quasi-ordered systems [2, 1]. Input is a set of initial states $I \subseteq \Sigma$, and a target state $e \notin I$. The algorithm maintains a set $U \subseteq \Sigma$ of minimal encountered states, and a *work set* $W \subseteq U$ of unprocessed states. It successively computes cover predecessors, starting from e, and terminates either by backward-reaching an initial state (thus proving coverability of e), or when no unprocessed vertex remains (thus proving uncoverability; this will happen eventually since $\leq$ is a well-quasi-ordering).

4.1.2 Target Set Widening: The Idea

In case Algorithm 1 terminates in Line 9, set U contains the minimal program states backward-reachable from $\uparrow e$, i.e. the minimal elements of the set $\mathsf{C\text{-}Pre}^*(e)$ of states that have a path to an element in $\uparrow e$. Figure 17 shows an example of minimal backward-reachable states computed by Algorithm 1. The uncoverability of e follows from the disjointness of $\mathsf{C\text{-}Pre}^*(e)$ and the initial states I; set $\mathsf{C\text{-}Pre}^*(e)$ thus serves as an *uncoverability proof* for the target state e.

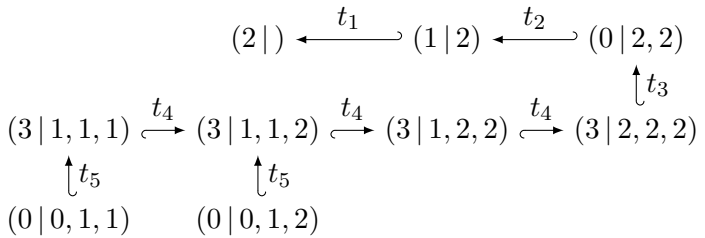

Figure 17: **Uncoverability proof of the classical backward search** — Shown are the states in the final set U computed by Algorithm 1 for the strictly asynchronous program in Figure 15. The target set is $\{(2\,|\,)\}$, i.e. we wish to check whether shared state $s = 2$ is reachable in the program in Figure 15. Edges denote cover predecessor relations and are labelled by the inducing thread transition (from Figure 15).

Instead of computing this set we can, however, also prove q uncoverable using an overapproximation of $\mathsf{C\text{-}Pre}^*(e)$ that is closed under $\mathsf{C\text{-}Pre}$ and does not intersect with the initial states. The potential benefit of proving uncoverability via such an overapproximation is that overapproximating $\mathsf{C\text{-}Pre}^*(e)$ by appropriately pushing down ("$<$") its minimal elements leads to more compact, or ideally, *minimal* uncoverability proofs.

Definition 24. *An **uncoverability proof** for e is an upward-closed set UCP that*

(i) contains e;
(ii) is closed under pre-images; and
(iii) is disjoint from the initial state set I.

*Further, the uncoverability proof is **minimal** if*

(iv) for any $r \in \min UCP$ and any $v < r$, v is coverable; and
(v) no proper subset of UCP satisfies the foregoing conditions (i)–(iv).

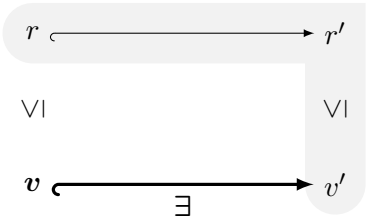

Figure 18: **Monotonicity upside down** — Smaller states have smaller cover predecessors

The first three conditions imply that all elements of UCP are uncoverable. The conditions capture the fact that, to prove uncoverability of e, it suffices to compute an overapproximation of $\mathsf{C\text{-}Pre}^*(e)$ that is disjoint from I. For example, if e is uncoverable, then the upward closure of $\mathsf{C\text{-}Pre}^*(e)$ is an uncoverability proof, as is the complement of the set Cover of coverable states. Condition (iv) of a *minimal* uncoverability proof UCP states that every minimal element of UCP is also a minimal element of the complement of Cover: $\min \mathrm{UCP} \subseteq \min(\neg\, \mathsf{Cover})$.

The notion of an uncoverability proof is sound, in that $\mathrm{UCP} \cap \mathsf{Cover} = \emptyset$; in particular, the query state e is uncoverable. The notion is also complete: our algorithm can construct a minimal uncoverability proof (if e is uncoverable). In contrast to the set $\mathsf{C\text{-}Pre}^*(e)$, which is unique for a given e, multiple minimal uncoverability proofs may exist, and the upward closure of $\mathsf{C\text{-}Pre}^*(e)$ and the complement of Cover do not in general represent minimal uncoverability proofs.

Example 25. *We illustrate these claims with an example. Consider the well quasi-ordered system with states $\{i, e, x_1, x_2\}$; i and e are the initial and query states, respectively. The ordering $\leq$ is the reflexive closure of $\{(x_1, e), (x_2, e)\}$. Further, the system has no transitions (hence condition (ii) holds trivially). Query e is therefore uncoverable. There are four uncoverabililty proofs of this fact:*

- $\uparrow\{e\} = \{e\}$ *is the upward closure of* $\mathsf{C\text{-}Pre}^*(e)$*; it violates condition (iv) for* $v \in \{x_1, x_2\}$ *and* $r = e$ *and is hence not minimal.*
- $\uparrow\{x_1\} = \{x_1, e\}$ *and* $\uparrow\{x_2\} = \{x_2, e\}$ *both satisfy conditions (iv) (vacuously) and (v), and are therefore minimal uncoverability proofs for* e*.*
- $\uparrow\{x_1, x_2\} = \{x_1, x_2, e\}$ *is the complement of* Cover*; it satisfies (iv) but violates (v) by the previous item.*

Uncoverability proof construction. The idea underlying our widening approach is that "cover predecessors of smaller elements are smaller" (an observation that also appears in [1]):

Lemma 26. *For states r, r', v', if r is a cover predecessor of r' and $v' \leq r'$, then there exists a cover predecessor v of v' such that $v \leq r$. (Figure 18)*

Proof. Let $C_1 = \{w \,:\, w \hookrightarrow v'\}$ and $C_2 = (\downarrow r) \cap C_1$. The latter set is non-empty, since $r \in C_2$. Let therefore v be a minimal element of C_2. Then $v \in \downarrow r$, so $v \leq r$. Also, $v \in C_1$. It remains to be shown that v is a *minimal* element of C_1, since then $v \in \mathsf{C\text{-}Pre}(v')$. To this end, let $w \in C_1$ be arbitrary. If $w < v$, then $w \leq r$, hence $w \in \downarrow r$, hence $w \in C_2$. This is not possible, however, since v (supposedly $> w$) is a minimal element of C_2. Thus $w \not< v$. Since $w \in C_1$ was arbitrary, it follows that v is minimal in C_1. □

The lemma suggests: before expanding an element, we first select smaller elements for expansion; the resulting cover predecessors will be smaller as well. Since smaller elements have fewer (cover) predecessors, this leads to earlier termination along the path and thus faster decisions.

We can observe the impact of these observations on the performance of Algorithm 1, using the program in Figure 15. Let the query target be $(2\,|\,)$, i.e. we want to determine whether shared state 2 is reachable. Figure 19 sketches the process.

Example 27. *We start at target $e = (2\,|\,)$. Before expanding it into cover predecessors, we check whether a widening candidate exists. As this is not the case ($\downarrow e = \{e\}$), we proceed by obtaining cover predecessor $(1\,|\,2)$, which covers $(1\,|\,)$, a widening candidate. From $(1\,|\,)$ we obtain cover predecessor $(0\,|\,2)$, which covers $(0\,|\,)$. The latter state is not considered for expansion, as it is initial and hence coverable. Thus we expand $(0\,|\,2)$ to obtain $(3\,|\,2,2)$, which covers $(3\,|\,)$; the latter becomes a new target state and is expanded. We now find that $(3\,|\,)$ is coverable, with initial cover predecessor $(0\,|\,0)$. We thus have to cancel the backward search from $(3\,|\,)$ and mark it as coverable. Similarly, trying to add $(3\,|\,2)$ as a new target fails, since its cover predecessor $(3\,|\,1)$ is coverable.*

The algorithm thus resorts to expanding $(3\,|\,2,2)$ to $(3\,|\,1,2)$. All 3 states strictly covered by $(3\,|\,1,2)$ have previously been shown coverable and are thus not added to the target set. The same is true for its predecessor $(3\,|\,1,1)$, which is hence expanded to $(0\,|\,0,1)$; the latter state strictly covers $(0\,|\,1)$. The only cover predecessor of $(0\,|\,1)$ is $(3\,|\,1,2)$, which is not new, so the search terminates.

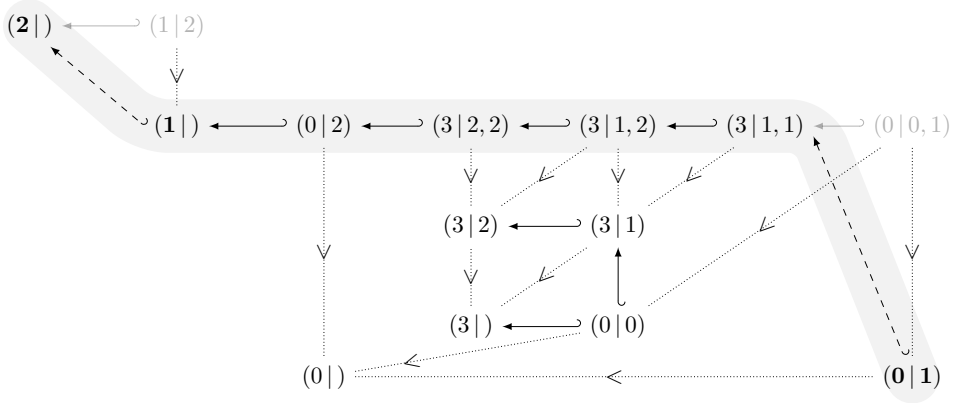

Figure 19: **Minimal uncoverability proof construction** — The initial stateset is given by $(0\,|\,0^*) = \{(0\,|\,0^i) : i \in \mathbb{N}\}$. We write $t \hookleftarrow\!\!\dashrightarrow v$ if $p \hookrightarrow v$ for some $p > t$. The (initially singleton) set of coverability targets is widened on-the-fly by so far undecided elements of the downward closure of encountered states (indicated via edges $\hookleftarrow\!\!\dashrightarrow$). If an element turns out coverable we backtrack and mark it and other coverable states, so they are not reselected

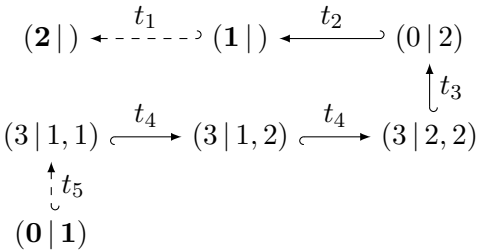

Figure 20: **Minimal uncoverability proof** — States in the widened target set in bold

Figure 20 shows the uncoverability proof that is actually generated for the example of Figure 15 and target $(2\,|\,)$. State $(2\,|\,)$ is expanded to $(1\,|\,2)$ followed by widening to $(1\,|\,)$. Comparing this proof with that in Figure 17, we observe reductions in the number of minimal states (9 vs. 7), in the longest traversed path (7 vs. 6), and in the maximum thread count (3 vs. 2). The uncoverability proof is even minimal: every proper subset is not an uncoverability proof, and every state that is strictly covered by one of the 7 states in Figure 20 turns out to be coverable.

We can think of the cancellation of the backward search from a state that turns out coverable as *backtracking out of the widening*: we made a bad choice that needs to be rolled back. In contrast, if the termination condition is satisfied for a state v obtained by widening p (such as for state $(0\,|\,1)$ above), $v \in {\downarrow}p$, we do not need to go back to the pre-widening query state p: any cover predecessor of p is also a cover predecessor of v and would thus have been "caught" while expanding v.

In Section 6 we present experimental evidence showing the potential of compressing the proof size by target set widening in practice: for our concurrent C program benchmarks we observed reductions in the numbers of proof nodes, in the longest traversed paths, and in the maximum thread counts across the proofs by 95%, 67% and 50%, respectively.

4.1.3 The Target Set Widening Algorithm

We now present our algorithm in detail. In addition to the data structures used by Algorithm 1, namely a set $U \subseteq \Sigma$ with *vertices* that are labelled and identified with encountered states, and a work set $W \subseteq U$ of *unprocessed vertices*, our algorithm maintains

(i) a set E of directed *edges* between vertices: $E \subseteq U \times U$,
(ii) a *function* $\zeta\colon U \to U$, and
(iii) a downward-closed set $D \subset U$ of states that have been shown coverable: $D \subseteq \mathsf{Cover}$.

We write $u \hookrightarrow_E r$ for $(u, r) \in E$, and $\hookrightarrow_E^*$ for the reflexive transitive closure of $\hookrightarrow_E$. Intuitively, E stores cover predecessor relationships that were expanded: $\hookrightarrow_E \subseteq \hookrightarrow$. For a vertex r, $\zeta(p) = r$ indicates that r was either present initially (if $e = r$), or otherwise during target set widening—we call such a vertex a *(widening) candidate vertex*—and p was encountered after backward-expanding r. We have $\zeta(v) = v$ precisely for candidate vertices v; other vertices are called *predecessor vertices*. *Graph* structure (U, E) gives rise to paths ending with a candidate vertex,

i.e. a vertex in $\zeta(U) = \{\zeta(u) \;:\; u \in U\}$. The mapping ζ defines an equivalence relation: vertices v and r are *equivalent* if $\zeta(v) = \zeta(r)$; there is one equivalence class (partition) per candidate vertex. The set D stores states that were shown to be coverable, and hence is an underapproximation of the coverability set: $D \subseteq \mathsf{Cover}$.

Example 28. *We illustrate these definitions using the graph in Figure 19. The set U contains 9 elements: $e = (2\,|\,)$ is the query candidate vertex, $(1\,|\,)$ and $(0\,|\,1)$ are widening candidates, and $(1\,|\,2)$, $(0\,|\,2)$, $(3\,|\,2,2)$, $(3\,|\,1,2)$, $(3\,|\,1,1)$ and $(0\,|\,0,1)$ are predecessor vertices; $(1\,|\,2)$ and $(0\,|\,0,1)$ are non-minimal in U. (All other vertices in the figure have turned out coverable and have been removed from U.) Solid harpoon arrows determine the edges in the set E. The mapping ζ induces three partitions, one for each of the candidate vertices $(2\,|\,)$, $(1\,|\,)$ and $(0\,|\,1)$ (with 2, 6 and 1 element(s), resp.); e.g. $(1\,|\,2)$ was encountered after backward-expanding $(2\,|\,)$, and hence $\zeta\,(1\,|\,2) = (2\,|\,)$. The set D is $\downarrow\{(3\,|\,1),(3\,|\,2)\} \cup (0\,|\,0^*)$, witnessing failed widening attempts, which slow down the algorithm. (We discuss in Section 4.3.1 how we remedy this problem.)*

The algorithm takes a set of initial states I, and a non-initial target $e \notin I$ as input and maintains the invariant that the subgraph of (U, E) over vertices from the same partition forms a tree, with the candidate vertex as root. Each tree represents an attempt to prove the corresponding candidate uncoverable. The algorithm consists of three routines: widen tries to add new candidate vertices, backtrack prunes partitions whose candidate vertex proved coverable, and Cat is the main routine.

Widening. The widen routine takes a vertex n and tries to widen the target set by an element in $\downarrow n$. More precisely, if the set $\mathcal{C}(n) = (\downarrow n) \setminus (\{n\} \cup D)$ of candidate elements is non-empty, we select a *minimal* element u from it; note that candidates must not come from the set D of elements known to be coverable. If u is new ($u \notin U$), it is inserted in the work and vertex sets; we set $\zeta(u) := u$. Otherwise, u's new role as candidate vertex and partition root must be acknowledged: the graph is *repartitioned* by modifying ζ for all vertices in the subtree with root u, i.e. the set

$$\Lambda(u) = \{r \in U \,|\, r \hookrightarrow_E^* u \wedge \zeta(r) = \zeta(u)\}\,.$$

All vertices in u's subtree are removed from their old partition and form a new partition with u as root: we set $\zeta(r) := u$ for all $r \in \Lambda(u)$. For the example in Figure 19, the widen routine is successfully called four times, with vertices $(1\,|\,2)$, $(3\,|\,2,2)$, $(3\,|\,2,2)$ again (D has changed in the meantime), and $(0\,|\,0,1)$ as input.

Note that in the definition of the widen routine we assume that the downward closure of a finite set is finite (which ensures that the candidates set $\mathcal{C}(\cdot)$ is finite). This is guaranteed for the well quasi-ordered systems induced by strictly asynchronous programs according to Section 3.2.1, and generally for Petri nets, Broadcast protocols and Lossy counter machines.

Backtracking. The backtrack routine (Algorithm 2) prunes partitions represented by candidate vertices from a set P; this happens whenever some vertex in such a partition is found coverable. An obstacle is that some edges may point from the partiton to be removed into another partition. Such edges (and the adjacent vertices) must be preserved, since otherwise paths generated by the other partition are destroyed.

Definition 29. *Given a set P of candidate vertices, an edge $(r, s) \in E$ is called **P-conflicting** if $\zeta(r) \in P$ and $\zeta(s) \notin P$.*

The backtrack routine first resolves all conflicts (Lines 1–3): for a conflicting edge $r \hookrightarrow_E s$, we re-associate vertices in $\Lambda(r)$ to $\zeta(s)$. Remaining vertices and edges of partitions in P can now be pruned (Lines 4–7). It suffices to prune edges *ending* in $\zeta(r)$: edges starting from $\zeta(r)$ are pruned when their target vertices are processed; note that, after resolving P-conflicts, those target vertices also have their roots in P. Figure 21 illustrates both steps. In the example in Figure 19, routine backtrack is called on candidate vertices $(3\,|\,)$ and $(3\,|\,2)$: the former is pruned alone, while the latter is pruned along with its cover predecessor $(3\,|\,1)$ (in both cases no P-conflicting edges exist).

Algorithm 2 backtrack(P)

Require: Set P with states to be removed, $P \subseteq \zeta(U)$

1: **for all** $(r, s) \in E$ such that (r, s) is P-conflicting **do**
2: **for all** $t \in \Lambda(r)$ **do**
3: $\zeta(t) := \zeta(s)$
4: **for all** $r \in U : \zeta(r) \in P$ **do**
5: $W := W \setminus \{r\}\,;\ U := U \setminus \{r\}$
6: **for all** $(t, r) \in E$ **do**
7: $E := E \setminus \{(t, r)\}$

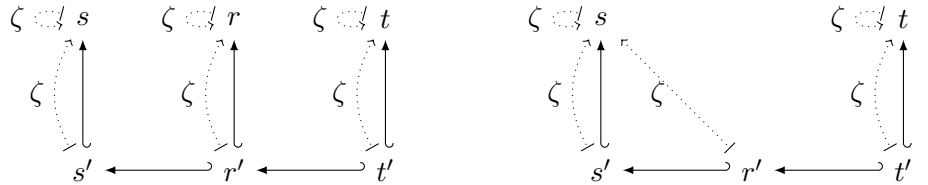

Figure 21: **Backtracking** — (a) A partitioned graph structure with three candidate vertices, s, r and t, each with a single cover predecessor (primed states) in its partition. The partition underneath r is to be pruned, but edge $r' \hookrightarrow_E s'$ is $\{r\}$-conflicting. (b) The structure obtained after calling backtrack($\{r\}$). Node r' belongs to s's partition, and r was pruned

Main routine. We introduce some terminology. A state v is *u-minimal* if it covers none of the vertices in u's partition, nor any immediate predecessor of such a vertex (the predecessors may lie outside of this partition):

Definition 30. *Let $v \in \Sigma$ and $u \in \zeta(U)$. State v is **u-minimal** if $v \not\geq u$ and for all $s, t \in U$ such that $s \hookrightarrow_E t$ and $\zeta(t) = u$, we have $v \not\geq s$.*

A set is *lower successor-closed* if it is closed both under $\hookrightarrow_E$ successors and downward:

Definition 31. *Let $X \subseteq \Sigma$. Set X is **lower successor-closed** if, for every $p \in X$, all vertices in $\downarrow p$ and all $\hookrightarrow_E$ successors of p belong to X.*

We write $\lfloor v$ for the *least* lower successor-closed set containing v. This set is obtained by closing $\{v\}$ downward and under $\hookrightarrow_E$ successors until fixed point. The point of this definition is that, if v is coverable, so is every vertex in $\lfloor v$: the set Cover is lower successor-closed.

Algorithm 3 shows the main routine Cat (for "**C**overability **A**nalysis via **T**arget set widening"). Input is a set I of initial states (downward closed by definition), and a non-initial target query $e \notin I$. At the outset, W and U contain one candidate vertex, the target e. The set D of elements found coverable contains the initial states, the set E of edges is empty, and ζ maps e to itself (Line 1). Then we try to widen the target set by elements smaller than e.

The algorithm terminates with "e uncoverable" if no minimal unprocessed vertex remains. Otherwise it selects and removes a minimal such vertex w. The **for**

Algorithm 3 $\mathsf{Cat}(I, e)$: Coverability Analysis by Target Set Widening

Require: Set I of initial states, query target $e \notin I$

```
 1: W := {e} ; U := {e} ; D := I ; E := ∅ ; ζ : e ↦ e
 2: widen(e)  // add smaller candidate if possible
 3: while W ∩ min U ≠ ∅ do
 4:     select w ∈ W ∩ min U  // select minimal unprocessed vertex
 5:     W := W \ {w}
 6:     for all p ∈ C-Pre(w): p is ζ(w)-minimal do
 7:         if p ∈ D then
 8:             if e ∈ ⌊p then
 9:                 return "e coverable"
10:             else
11:                 D := D ∪ ⌊p  // mark coverable states
12:                 backtrack(ζ(⌊p))  // call backtrack routine
13:                 for all u ∈ min U do
14:                     widen(u)
15:                 break  // skip forward to next iteration of while in Line 3
16:         else
17:             E := E ∪ {(p, w)}
18:             if p ∉ U then
19:                 W := W ∪ {p} ; U := U ∪ {p} ; ζ(p) := ζ(w)  // add
20:                 widen(p)  // add smaller candidate if possible
21: return "e uncoverable"
```

loop in Line 6 now steps through all cover predecessors p of w that are $\zeta(w)$-minimal, and processes them as follows:

Line 7 If p is in D, then p and all elements in $\lfloor p$ are known to be coverable. We distinguish:

Lines 8–9 If the query e is among the elements in $\lfloor p$, it is coverable; the algorithm terminates.

Lines 10–15 If e is not among the elements in $\lfloor p$, then the search must go on. We add all of $\lfloor p$ to D and invoke the backtrack routine to remove partitions of coverable candidate vertices. Since this may remove candidate vertices of remaining predecessor vertices, we have to ensure that their downward closure is further searched for candidates. We therefore create new minimal candidate vertices (Lines 13–14). The **break** instruction then skips forward to the next iteration of the **while** loop (Line 3). As a consequence of backtracking, unprocessed vertices that were previously not minimal may now be.

Lines 16–20 If p is not in D and hence not currently known to be coverable, then the graph is *expanded*. If p is new (Line 18), then we add it to our work and undecided lists, and add it as predecessor vertex to w's partition. We also call the widen routine to (try to) add smaller target elements.

4.2 Correctness, Complexity and Efficiency

We prove our target set widening algorithm correct, study its time complexity, and give a preview of its compact operation. We begin with a classical termination + partial correctness argument.

4.2.1 Correctness

Algorithm 3 manipulates a node set U by adding elements to it in Line 19 and during widening, but also by pruning elements from it during backtracking. As a result, set U does not grow monotonically. We will first prove termination of a variant of Algorithm 3 *without* backtracking. Let therefore Algorithm 3' be the same as Algorithm 3, except that Line 12 is replaced by a no-op.

Theorem 32. *Algorithm 3' terminates on all inputs.*

We first show the following property:

Lemma 33. *Line 4 of Alg. 3' never selects an element twice.*

Proof. In this proof, we write "$\in_t$" for "element of, at time t". Let w be an element selected in Line 4—say at time t_0; we show it will never be selected again. In Line 5, w is removed from W. Since Line 4 selects from $W \cap \min U$, element w needs to be added back into W—say at time $t_1 > t_0$—before it can be selected a second time. Element w can be added to W in Line 19, or during widening. In both cases, only elements not in U are added to W. Since $w \in_0 U$ (in fact, $w \in_0 \min U$), and—in Algorithm 3'—elements are never removed from U, we have $w \in_1 U$, so the addition of w to W at time t_1 is not possible. □

Now the proof of Theorem 32.

Proof (Theorem 32). The only loop that could cause non-termination is the **while** loop in Line 3. We show that the set $\min U$ will eventually be depleted, terminating the loop. Consider the sequence $p_1, p_2, \ldots$ of elements p added to U (keep in mind that, in Algorithm 3', these are never removed from U). Let $J = \{j \in \mathbb{N} : \neg(\exists i : i < j \wedge p_i \leq p_j)\}$. The elements p_j with $j \in J$ constitute a "bad" sequence of states: one that never increases. Since $\geq$ is a well-quasi-ordering, J is finite, and since $1 \in J$, it is also non-empty. Let therefore $j = \max J$. After adding element p_j, only elements p_k ($k > j$) that satisfy $p_k \geq p_i$ for at least one previously added element p_i ($i < k$) are added to U. We now distinguish: (i) If there exists $i < k$ such that $p_k > p_i$, then $\min U$ is not changed by the addition of p_k: p_k is guaranteed not to be a minimal element of U. (ii) Otherwise we have, for all $i < k$, $p_k \not> p_i$. By the definition of $>$, this means that, for all $i < k$, $p_k \not\geq p_i \vee p_i \geq p_k$. Now pick $i < k$ such that $p_k \geq p_i$ (existence guaranteed since $k > j$). For this i, we have $p_k \equiv p_i$. This in turn implies that $p_i \in \min U$: otherwise there would exist $p_r \in U$ with $p_r < p_i \leq p_k$, contradicting the condition leading to case (2). Element p_k is thus equivalent to ($\equiv$) some minimal element (p_i) of U. The addition of elements p_k to U thus either does not modify $\min U$ (1), or at most modifies $\min U$ by adding elements to $\min U$ whose local state vector is a permutation of the local state vector of some existing element of $\min U$ (2). The latter can happen only finitely many times, as there are only finitely many permutations of elements in $\min U$. Eventually $\min U$ therefore stops changing. By Lemma 33, the **while** loop in Line 3 of Algorithm 3' eventually terminates. □

Corollary 34. *Algorithm 3 terminates on all inputs.*

Proof. We need to determine the impact (on termination) of the backtracking. Algorithm 2 mostly removes elements that are also contained in D, namely all elements in $\downarrow p \subseteq D$, for some $p \in D$ (cf. Line 12). Set D grows monotonically: elements are never removed from it. Further, elements in D are never added to U, this is guaranteed both in Line 19, and in the calls to the widening routine. As a result, the elements in D that fall victim to pruning during backtracking will never be added into U again, so pruning these elements can only accelerate termination.

On the other hand, Algorithm 2 may also remove some element $s \notin D$, whose coverability has not been decided yet. Element s's root, in contrast, does belong to D: in Line 12 of Algorithm 2, we have $\zeta(\downarrow p) \subseteq \downarrow p$, and all elements in $\downarrow p$ were added to D in Line 11. Root $\zeta(s)$, which is removed in the same call to Algorithm 2, will thus never be added to U again. When and if s later reappears in U, it must therefore be associated with another root node (s may itself be a root at that time). Element s can therefore be removed from and reintroduced into U only finitely many times, namely at most whenever we add query elements during widening, which happens finitely often. □

Intuitively, pruning and reintroducing an element $s \notin D$ causes finite delay of the termination of the algorithm, but keeps the search graph compact. We continue by proving partial correctness.

Theorem 35. *If control reaches Line 9 in Algorithm 3, the query target e is coverable.*

Proof. We show that $D \subseteq \mathsf{Cover}$ is an invariant of the algorithm. The theorem then follows: in Line 9, we have $p \in D$ (hence $p \in \mathsf{Cover}$) and $e \in \downarrow p \subseteq \mathsf{Cover}$ (since $p \in \mathsf{Cover}$, and coverability is closed under $\hookrightarrow_E$ successors and downward).

We prove $D \subseteq \mathsf{Cover}$ by induction over the number of modifications made to D in Line 11. Before any such modifications, we have, as per Line 1, $D = I \subseteq \mathsf{Cover}$. Now assume $D \subseteq \mathsf{Cover}$. In Line 11, we have $p \in D$, hence $p \in \mathsf{Cover}$ by the induction hypothesis. Again by the above closure property of coverability, we obtain $D' = D \cup \downarrow p \subseteq \mathsf{Cover}$, which completes the step. □

Theorem 36. *If control reaches Line 21 in Algorithm 3, the query target e is uncoverable.*

Proof. We show that, if control reaches Line 21, the upward-closed set $\uparrow U_{fin}$ is an *uncoverability proof* for e (Definition 24), for the final value U_{fin} of variable U at Line 21. The fact that e is uncoverable then follows immediately by that definition: all elements of U_{fin} are uncoverable.

We prove the first three conditions of Definition 24: **Condition (i):** $\uparrow U_{fin}$ contains e, in fact $e \in U_{fin} \subseteq \uparrow U_{fin}$: e is added to U in Line 1, and the call to backtrack (the only chance for e to be removed) is guarded by the condition $e \notin \lfloor p$ (Line 10), which implies $e = \zeta(e) \notin \zeta(\lfloor p)$, so e is never removed. **Condition (ii):** $\uparrow U_{fin}$ is closed under pre-images: (i) We show: cover predecessors of elements of $\min U_{fin}$ are in U_{fin}: $\mathsf{C\text{-}Pre}(\min U_{fin}) \subseteq U_{fin}$: Suppose $w \in \min U_{fin}$, hence $w \in U_{fin}$. At the time w was added to U, it was also added to W (this is true in Line 19, and for all calls to the widening routine). When the algorithm terminates in Line 21, we have $W \cap \min U_{fin} = \emptyset$. Hence w was, at some point, removed from W but not from U, which only happens in Line 5, following which w's cover predecessors p are processed. If $p \notin D$ (and new to U), it is added to U. If $p \in D$ and $e \in \lfloor p$, then the algorithm terminates in Line 9; this case does not apply to Theorem 36. If let $p \in D$ and $e \notin \lfloor p$ (Line 10), then p is not added to U, but we backtrack: we also remove the successor w, so the closure property is preserved. (ii) Let now (i) $w \in \uparrow U_{fin}$ and (ii) $p \rightarrowtail w$; we show $p \in \uparrow U_{fin}$: Due to (i), there exists $v \in U_{fin}$ such that $v \leq w$. Let further $t \in \min U_{fin}$ such that $t \leq v$. Due to (ii), there exists $o \in \mathsf{C\text{-}Pre}(w)$ such that $o \leq p$. Applying Lemma 26 to o, w and $t(\leq w)$ tells us that there exists $m \in \mathsf{C\text{-}Pre}(t)$ such that $m \leq o$. By (i), we conclude $m \in U_{fin}$. Since $p \geq m$, we also have $p \in \uparrow U_{fin}$. **Condition (iii):** $\uparrow U_{fin}$ is disjoint from the initial state set I: we first show $U_{fin} \cap I = \emptyset$: (i) By Algorithm 3's precondition, $e \notin I$. Hence $U_0 \cap I = \emptyset$, for the initial value $U_0 = \{e\}$ of U. (ii) Only elements not in D are ever added to U (this is true in Line 19, and for all calls to the widening routine); since $I \subseteq D$ is an invariant of the algorithm, such additions exclude initial states. Let now $y \in \uparrow U_{fin}$, i.e. there exists $x \in U_{fin}$ with $y \geq x$. Since $U_{fin} \cap I = \emptyset$, $x \notin I$. This implies $y \notin I$, since I is downward closed. As a result, $\uparrow U_{fin} \cap I = \emptyset$. □

4.2.2 Complexity and Space Efficiency

The coverability problem is decidable for transfer and rendezvous Petri nets (see for example [63, 2]), and more generally for well quasi-ordered systems provided some natural conditions on the computability of the well-quasi-ordering relation hold [130]. The misery is in the complexity: coverability over transfer Petri nets or, equivalently monotone Boolean DR programs, was shown to be *Ackermann-complete* [164], which means that the complexity grows as fast as the Ackermann function. If the Petri net is hard-wired into the algorithm (the input consists only of the query), then the complexity subsides gracefully to primitive-recursive [164, 72].

In the following, for states $v \in V$ let $|v|_\infty$ denote the infinity norm

$$|(s \,|\, l_1, \ldots, l_n)|_\infty = \begin{cases} 1 & \text{if } n = 0 \\ \max\{\#i \;:\; l_i = l\} & \text{otherwise.} \end{cases}$$

Hence, the infinity norm of a state is the largest number of threads that reside in the *same* local state, or *one* if no thread exists. Examples: $|(2\,|\,)|_\infty = 1$ and $|(2 \,|\, 0, 3, 3, 4)|_\infty = 2$.

We use the so-called Fast Growing Hierarchy [133], $(\mathfrak{F}_k)_k$ for finite ordinals k for comparing the order of growth of functions. For the rest of this section we fix a monotone Boolean DR program $\mathcal{P}$ over shared variables S and local variables L. Furthermore, recall from Definition 1 that every infinite sequence $x = x_0, x_1, \ldots$ over V then contains an *increasing pair* $x_i \leq x_j$ for some $i < j$. The following theorem from [72] plays the central role in our complexity analysis, as it permits us to establish bounds on the length of paths traversed by both Algorithms 1 and 3.

Theorem 37 (adapted from [72])**.** *Given a sequence $x = x_0, x_1, \ldots$ over V with **no** increasing pair, a natural number $t \in \mathbb{N}$, and a linear function $f : \mathbb{N} \to \mathbb{N}$ such that $|x_i|_\infty < f(i + t)$ for all i. Then $x_i \leq x_{\gamma+1}$ for some $i \leq \gamma$, where γ is bounded by a function in $\mathfrak{F}_k$ for $k = 2^{|S|} + 2^{|L|}$.*

Theorem 37 bounds the length of state sequences under certain conditions. In order to use it to derive complexity bounds for our algorithms, it remains to establish a constant t, and a linear function f that meet the imposed requirements. This can easily be done by exploiting the fact that due to Line 6 of Algorithm 3 paths cannot "jump" to arbitrarily large thread dimensions: $x_i \in \min \mathsf{C\text{-}Pre}(x_{i+1})$ for all i of a traversed path x. Moreover, it is not difficult to show that the infinity norm for two neighbours in x can differ at most by one, i.e. $|x_{i+1}|_\infty \leq |x_i|_\infty + 1$. This estimation

gives us the desired t and f, namely

$$t = |x_0|_\infty \quad \text{and} \quad f(x) = x + 1\,. \tag{50}$$

Equipped with t and f from eq. (50), Theorem 37 implies (i) the existence of a bound γ in $\mathfrak{F}_k$ such that $x_i \leq x_{\gamma+1}$ for some $i \leq \gamma$, and hence the maximum *length* of a sequence with no increasing pair is bounded by γ, and (ii) that γ is *primitive-recursive* for fixed k, but *Ackermannian* when k is part of the input. We derive complexity bounds for our algorithm.

Lemma 38. *The complexity of Algorithm 3 is primitive-recursive for fixed S and L, and Ackermannian when S and L are part of the input.*

Proof. We establish a bound on the number of iterations that Algorithm 3 performs in the worst case. Let $|\mathcal{R}|$ for the transition relation of the monotone Boolean DR program $\mathcal{P}$ denote the the number of valuations that satisfy $\mathcal{R}$. First note that for a DR transition t and a state v the set of cover predecessors $\mathsf{C\text{-}Pre}(v)$ obtained with respect to t contains at most $|v| + 1$ states. Hence, the partition of a (widening) candidate q has at most $g_i = g_{i-1} \times (m_{i-1} + 1) \times |\mathcal{R}|$ vertices at level i, where $m_i = |q| + i$ denotes the maximum size of a state at level i, or equivalently

$$g_i = |\mathcal{R}|^i \times \prod_{n=1}^{i} (|q| + n) \leq |\mathcal{R}|^i \times (|q| + i)^i. \tag{51}$$

Now observe that value γ from Theorem 37 bounds the *maximum height* of the tree $\Lambda(q)$ rooted in q: $|\Lambda(q)| \leq g_0 + g_1 + \ldots + g_\gamma$. By using estimations $|\mathcal{R}| \leq \gamma$ and $|q| \leq \gamma$ the bound further simplifies to

$$|\Lambda(q)| \leq \sum_{i=0}^{\gamma} g_i \leq \gamma \times |\mathcal{R}|^\gamma \times (|q| + \gamma)^\gamma \leq 2\gamma^{2\gamma+1}. \tag{52}$$

Treating all candidates q, of which no more than γ can be introduced during widening, separately then gives the desired space and time upper bound. □

The complexity result is not surprising: the coverability problem for the equivalent transfer Petri nets was shown to be Ackermann-complete [164, 162], and Algorithm 1 is well-known to actually operate in Ackermannian time (see, e.g. [72]). Note also that these complexity results permit no conclusion about practical advantages. In fact, a general comparison of Algorithms 1 and 3 with this aim is doomed

to fail: there exist instances where the former converges faster, and instances where the opposite is true. A program that illustrates the former case is the one we have seen in Figure 15. As can easily be seen from Figures 17 and 19, Algorithm 1 requires 9 iterations—one less than Algorithm 3. In the sequel we will see that in practice this unfavorable case is very rare.

Space efficiency. We give a preview of our algorithm's compact operation, by comparing the widening and backtracking strategy to the classical backward exploration without it (a comprehensive empirical evaluation of the algorithm, including runtime performance, is presented in Section 6). The example at hand concerns a mutual exclusion property for an implementation of an Apple Bus protocol in the FreeBSD operating system (benchmark BSD-AK2; a code fragment was shown earlier in Figure 1).

Example 39. *Figure 22 plots the size of U on the vertical axis against the iterations of the main **while** loop. The backtracking nature of the widening approach is evident from the fact that the curve occasionally drops, namely whenever candidates with large partitions in their backward expansion were found to be coverable and pruned.*

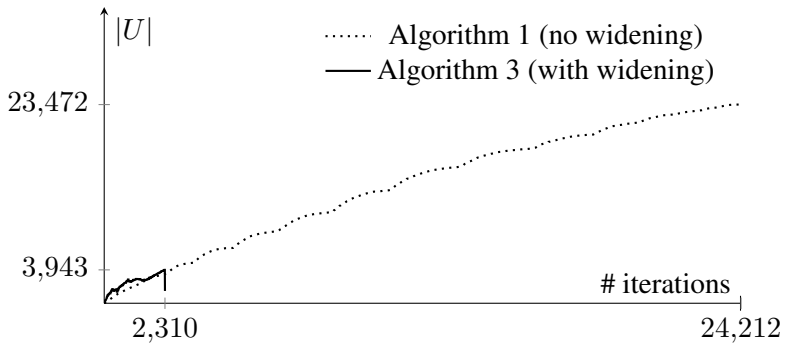

Figure 22: **Impact of widening on the number of iterations and graph vertices** — The target set widening strategy reduces the iteration count for the Apple Bus protocol benchmark from 24,212 to 2,310, with a maximum of 3,943 explored vertices compared to 23,472

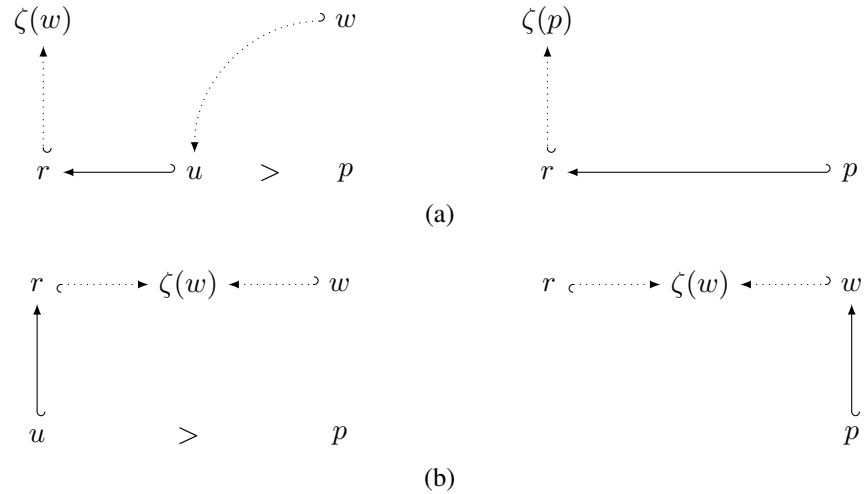

Figure 23: **Consolidation steps** — (a) consolidation step if w and u belong to the same path; (b) consolidation if w and u reside on different paths. Each subfigure shows the graph before and after the non-minimal vertex u is removed; vertex p was encountered by backward-expanding w

Proof minimality. Algorithm 3 computes uncoverability proofs that satisfy minimality requirement (iv) of Definition 24: for $v \notin U$ and $r \in U$, if $v < r$ then v is coverable. To see this, note first that $r \in U$ and $v < r$ implies that, at some point during the run of the algorithm, v was explored and therefore an element of U: the widen routine is called eagerly on the entire downward closure of encountered states such as r. Since at the end $v \notin U$, state v was removed some time thereafter. This only happens in the backtrack routine, and only to elements found coverable.

Our current implementation does not satisfy requirement (v) of Definition 24, stating that the uncoverability proof be "least": no proper subset satisfies conditions (i)–(iv) of that definition. It is easy to enforce this requirement, by adding redundancy checks to the set U at certain points where the algorithm modifies it. We found that this adversely affects the runtime of the algorithm, while the benefit in reduced proof size does not compensate for this effect. As an example, whenever the graph is expanded by adding the new cover predecessor p of w, we can purge vertices $u \in \Lambda(\zeta(w))$ satisfying $u > p$ from the partition of candidate vertex $\zeta(w)$ (together with their subtrees); illustrated in Figure 23.

4.3 Optimisations

We describe two optimisations of our widening approach. First, we counter the negative impact of "bad guesses", i.e. states that turn out coverable by supporting the search with a parallel engine that aims at reporting coverable states swiftly, and then show how to obtain a compacter search structure by tracking only vertices that are minimal with respect to all encountered vertices.

4.3.1 External Coverability Results

The widening employed in Algorithm 3 never adds candidates that have already been found coverable. The reason is that coverable elements do not contribute towards the coverability decision for larger elements; expanding them would only thus waste time. The added candidates may, however, be found coverable later, during the subsequent expansion, in which case we find in retrospect that we have unnecessarily expanded those states, and now incur extra work to prune them.

The determination of what elements are coverable is, of course, the ultimate goal of our algorithm, so we cannot assume to have such information when deciding whether to add a potential widening candidate. If, however, we happen to know that a particular element is coverable, we will not add it to the candidate set. Such incidental information can come from an external source. We call such a source a *coverability oracle*. It must (i) soundly report coverable states, but is (ii) not required to report *all* coverable states. (i) suggests that the oracle perform a *forward*-directed search, which generates coverability information on the fly. As a consequence of (ii), our algorithm must be correct even in the presence of unreported coverable elements: they will be pruned during backtracking.

The coverability oracle and Algorithm 3 run in parallel and synchronize via the set D: the oracle populates this set with coverable elements. Receiving such updates, Algorithm 3 terminates if the input query belongs to D, or otherwise invokes the backtrack routine on now known-to-be-coverable candidate vertices in regular intervals to restore disjointness of the sets U and D.

The flexibility afforded by the above two conditions allows us to use any fast but possibly underapproximating forward-directed search: a random or enumerative reachability analysis like boom [24] works just as well as generalizations of the Karp-Miller procedure to broadcast synchronization. In our experiments we use a version of Karp-Miller that never accelerates across broadcast transitions. As a result, this version of the procedure is sound for broadcasts but may not terminate.

In contrast, the standard Karp-Miller procedure is *not* suitable as an oracle, since—in the presence of broadcasts—it may return an overapproximation of the set of coverable states. This can be observed for the strictly asynchronous program on the left of Figure 24; the corresponding Karp-Miller tree is shown on the right. The acceleration that yields $(0 \mid 0^\omega, 1^\omega, 2^\omega)$ introduces imprecision: due to the broadcast transition, only one thread can reside in local state $c = 2$ at any time, which the algorithm misses. As a result, it falsely reports e.g. state $(0 \mid 2, 2)$ coverable. Receiving this false report, Algorithm 3 will remove this state from U and never expand it.

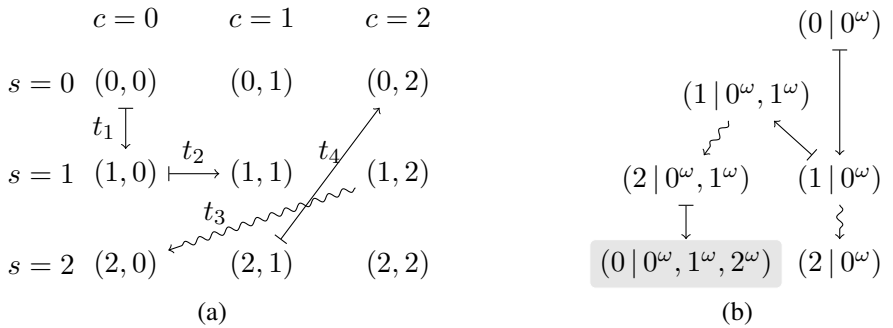

Figure 24: **Karp-Miller overapproximates for broadcasts** — (a) strictly asynchronous program with initial thread state $(0, 0)$ and broadcast transition t_3; (b) the corresponding Karp-Miller tree, which contains omega state $(0 \mid 0^\omega, 1^\omega, 2^\omega)$ (gray) and thus falsely reports $(0 \mid 2, 2)$ as coverable

Example 40. *Figure 25 presents a preview of how our algorithm benefits from external coverability results, again for the Apple Bus protocol benchmark. The plot reveals a significant reduction in work related to coverable candidates. The oracle reports roughly 90% of the coverable candidates ahead of our approach without oracle.*

These observations indicate that while forward directed search may not be complete for checking programs with arbitrary thread numbers, it is good at reporting coverable states rapidly, which in the context of our algorithm is all that is needed.

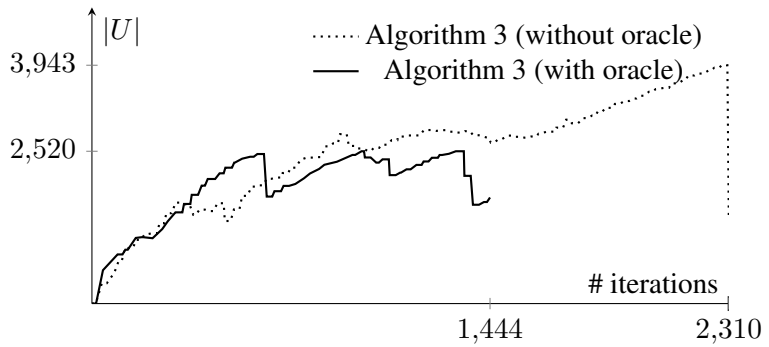

Figure 25: **Benefitting from external coverability results** — Algorithm 3 without and with support by a generalization of the Karp-Miller procedure as oracle (cf. Figure 22). The oracle significantly reduces the work related to coverable candidates, both in terms of iteration count and maximum graph size

4.3.2 Optimistic Backtracking

Given a minimal unprocessed vertex w, Algorithm 3 steps through those of w's cover predecessors that are $\zeta(w)$-minimal and, if new, adds them to the node set U. In contrast to the classical backward search (Algorithm 1), our widening algorithm may therefore keep some vertices p that are not minimal in U. This is necessary because vertices strictly smaller than p may belong to the partition of a bad candidate choice that later needs to be rolled back: a non-minimal cover predecessor p may eventually become minimal in U. We call this strategy *pessimistic* since it is effective if the algorithm makes a high number of bad candidate choices. On the other hand, the worst case scenario for such an approach is that only good (i.e., uncoverable) candidates are selected for widening, and thus non-minimal cover predecessors p are kept without ever being used.

We therefore propose an *optimistic* variant CatOpt of our widening approach Cat: in agreement with the classical backward search, CatOpt only keeps minimal vertices, i.e. it maintains the invariant $\min U = U$. This reduces the node set size, but takes more effort to achieve than for the classical backward search: simply discarding non-minimal predecessors p from U is not enough, for the reason mentioned in the previous paragraph. Instead, CatOpt (i) maintains, in addition to the work set, a *list* of processed vertices in the processing order, and (ii) after each pruning step, marks as *unprocessed* those (unpruned) vertices that were discovered *after* some pruned vertex. We experimentally compare both variants in Section 6.

5 Implementation

We implemented our abstraction technique from Section 3 in the CEGAR-driven monabs verifier for C programs, and the two target set widening algorithms (Section 4) in the infinite-thread model checker breach, which we use as back-end model checker for monabs; both tools are available online, see Section 1. We provide a short summary of both tools.

5.1 The monabs Verifier for Concurrent C Programs

For a non-recursive shared-memory program written in C, monabs automatically generates and checks a variety of correctness conditions like absence of array bound errors, division by zero, unlocks of not-locked mutexes and mutexes held by a different thread. Pthread-style thread creation and joining, and synchronisation objects (mutexes and condition variables) are supported via designated wrapper functions (function calls are handled by inlining). For mechanisation, monabs implements counterexample-guided abstraction refinement-style reasoning [15, 126] in a way that goes back to early work by Das and Dill [51], and has become known as *transition refinement*. It is by now standard in CEGAR techniques for *sequential* software and, e.g. implemented in Microsoft's slam verifier [16].

Monotonicity-aware Das/Dill-style refinement. Figure 26 depicts our *monotonicity-aware Das/Dill-style refinement* scheme for multi-threaded programs, which operates as follows: ① With a strictly asynchronous program $\mathcal{P}$ and a possibly empty set of predicates as input, the algorithm constructs a Boolean DR program $\tilde{\mathcal{A}}$ that overapproximates $\mathcal{P}$'s existential inter-thread predicate abstraction template $\tilde{\mathcal{P}}$. ② To facilitate coverability analysis of $\tilde{\mathcal{A}}$, which may be hindered due to the loss of monotonicity, the algorithm restores monotonicity using the closure operator introduced in Section 3.3.3; the obtained program $\tilde{\mathcal{A}}_m$ is monotone, and as suitable as $\tilde{\mathcal{A}}$ for verifying or falsifying $\mathcal{P}$'s safety. ③ Then $\tilde{\mathcal{A}}_m$ is model checked for *unbounded* threads through coverability analysis, which is decidable. ❶,❷ If the abstract program is provably safe, or an erroneous execution reported by a coverability checker reveals a bug in the input program, the loop terminates with "$\mathcal{P}$ is safe" and "$\mathcal{P}$ is unsafe", respectively. ❸ Otherwise the erroneous execution found in $\tilde{\mathcal{A}}_m$ and thus in $\tilde{\mathcal{A}}$ is an artefact of the abstraction, which is subsequently examined for the existence of infeasible transitions in the existential abstraction $\tilde{\mathcal{P}}$. ④ If such infeasible transitions exist, we refine the transition relation appropriately and proceed with the

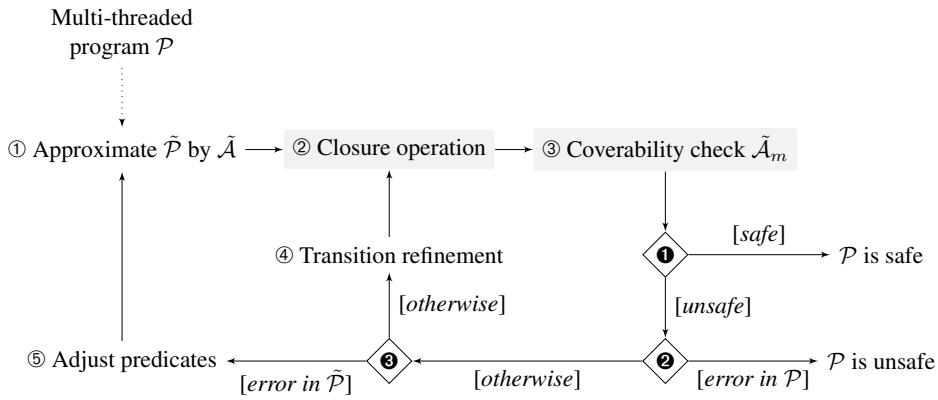

Figure 26: **Monotonicity-aware Das/Dill algorithm** — The approach uses two nested loops: the outer loop adjusts the predicate set, while the inner loop refines the transition relation on-demand. Our extension applies the monotone closure operator right before coverability checking the abstraction for unbounded threads (highlighted blocks)

more accurate, yet possibly non-monotone, approximation by re-applying the closure operator. ⑤ On the other hand, if all transitions are feasible in $\tilde{\mathcal{P}}$ we adjust the predicate set and enter the next iteration by regenerating an overapproximation.

Like any other technique, the loop may in general not terminate since checking safety in strictly asynchronous programs, run by even a single thread, is undecidable [170]. Nonetheless, the method can successfully prove properties for a large number of realistic programs. In fact, it is not difficult to prove that the algorithm is (i) *complete* if in the input program every thread's memory is finite, and (ii) may not halt once we permit infinite-domain variables, yet terminate normally otherwise (no false positives). Finally, for non-monotone DR programs as input (see Section 3.4), the algorithm may not halt or terminate *ab*normally*.

Design decisions. We briefly summarise the design decisions monabs makes: ① Medium-precision Cartesian abstractions [20] with cube length 3 are used to approximate the existential inter-thread predicate abstraction (cube enumeration is done with respect to abstract shared, active- and passive-thread variables). ② The

*Abnormal termination occurs whenever the model checker reports an erroneous execution that is feasible in $\tilde{\mathcal{A}}_m$ but not in $\tilde{\mathcal{A}}$ (Theorem 21 guarantees safety-equivalence only if $\mathcal{P}$ is monotone).

abstract monotone closure operation is performed symbolically. ③ The monotone abstraction is model checked for unbounded threads with the optimistic variant of our widening approach sketched in Section 4.3.2 and implemented in our breach tool (see details below). ④ Constraints on passive-thread variables are generated during transition refinement only for transitions that are not spurious for all valuations of passive-thread variables. ⑤ Predicates are discovered by weakest precondition propagation.

Preparing input programs. A software engineering practice that has nowadays become standard in programming [80] is the specification of error conditions via assertion predicates. These assertions allow programmers to instrument their code with test probes indicating intended behaviour. The C language defines a header file `assert.h`, which provides the macro `assert(pred)` for this purpose. When a statement like `assert(p! = NULL)` is executed, the execution is aborted with an error message if the condition evaluates to false, e.g. if `p` is `NULL` in the previous case. The monabs verifer can check the correctness of such code-annotations statically, i.e. without explicitly executing the code. More specifically, monabs checks that such assertions hold for any nondeterministic choice that the executing threads as well as the scheduler can make.

Sources for *nondeterministic input* in monabs are volatile-qualified objects, unspecified and indeterminate values, and several build-in library functions (nondeterminism can also be used to overapproximate the program behaviour). To restrict nondeterministic choices made by a program, monabs provides the built-in function `__CPROVER_assume(pred)`. As an example, if we wish to model a function that nondeterministically returns a value between 50 and 60, this can be achieved as shown in Figure 27. Although monabs performs counterexample-guided abstraction refinement and hence can prove many program fully automatically, it can sometimes be useful to specify predicates manually. This can be achieved using the following built-in functions:

(i) `__CPROVER_predicate(cond)` adds user-defined predicates;
(ii) `__CPROVER_parameter_predicates()` adds call-site parameter predicates;
(iii) `__CPROVER_return_predicates()` adds call-site relating return value predicates.

In addition to these *sequential* primitives, monabs supports the following *concurrency* features:

```
unsigned int nondet_uint(); //undefined functions implement nondeterministic choice

unsigned int get_range() {
  unsigned int r=nondet_uint();
  __CPROVER_assume(r>=50 && r<=60);
  return r; }
```

Figure 27: **Modelling nondeterminism in monabs** — The function nondeterministically returns a value between 50 and 60; the side-effect of `__CPROVER_assume(pred)` in monabs can equivalently be defined as `((pred)||(exit(0),1))` in the C language

(i) Magic labels in the form of `__CPROVER_ASYNC_1`, `__CPROVER_ASYNC_2` etc. to *dynamically spawn* new threads.
(ii) `__CPROVER_atomic_begin()` and `__CPROVER_atomic_end()` functions to declare *transactions* with strong atomicity semantics, i.e. executions of transactions cannot interleave with non-transactional code [137].
(iii) Magic label `__CPROVER_passive_broadcast` to mark statements that shall be evaluated over passive- instead of active-thread variables.*
(iv) Storage-class specifier `_Thread_local` for declaring local variables at global-scope.

The test-and-set lock shown in Figure 28 demonstrates how to use some of these features. The lock employs a busy-wait to access the Boolean flag `LOCK` that indicates whether the lock is already held by a thread. Notice how

(i) the keyword `_Thread_local` is used to define the variable `LOCK` as local,
(ii) an unbounded number of threads is spawned in the `main` function, the starting point of execution of the *single initial* thread, by repeatedly calling function `thr1` from the magic label `__CPROVER_ASYNC_1`, and
(iii) atomicity of the test-and-set instruction is modelled by the enclosing calls to functions `__CPROVER_atomic_begin()` and `__CPROVER_atomic_end()`, i.e. no thread can interrupt the execution of the comma-separated assignments in the macro.

Figure 29 illustrates how broadcasts can be modelled using an instruction with the special label `__CPROVER_passive_broadcast`.

*If `__CPROVER_passive_broadcast` is used, the program gives rise to a monotone dual-reference program rather than a strictly asynchronous program; see also Section 3.4.

```
#define unlocked 0
#define locked 1
volatile int LOCK = unlocked; //shared lock
_Thread_local int delay = 1, COND; //local
      variable

#define TAS(val,old) { \
  __CPROVER_atomic_begin(); \
  old = val, val = 1; \ //executed atomically
  __CPROVER_atomic_end(); }

void acquire_lock(){
  TAS(LOCK,COND);
  while(COND == locked){
    pause(delay);
    if(delay∗2 > delay)
      delay ∗= 2;
    TAS(LOCK,COND); }
  assert(COND != LOCK); }

void release_lock(){
  assert(LOCK != unlocked); //check for
       unlocks of not−locked mutexes
  LOCK = unlocked; }

int c = 0; //auxiliary shared variable
void thr1(){
  while(1){
    acquire_lock(); //enter the critical section
    c++;
    assert(c == 1); //can the critical section be
          accessed by two or more threads?
    c−−;
    release_lock(); } } //leave the critical
          section

int main(){
  while(1) {
    __CPROVER_ASYNC_1: thr1(); }}
```

Figure 28: **Test-and-set lock with exponential backoff [140]** — Each thread repeatedly calls the `TAS` instruction in an attempt to flip the flag `LOCK` from $false$ to $true$, indicating that acquisition of the lock has been achieved; the lock is released by resetting the flag

```
_Bool s = false; //shared flag
_Thread_local _Bool l = false; //local flag

void thr(){
  assert(!l || s);
  s = s || true;
  __CPROVER_passive_broadcast: l = true; } //execute assignment ''m_P=true'', i.e. set
       variable l of all passive thread to true

void main(){
  while(1) __CPROVER_ASYNC_01: thr(); }
```

Figure 29: **Modelling broadcasts** — The local flag `l` indicates whether a thread has been broadcast to ($\texttt{l} = true$) or not ($\texttt{l} = false$); the program assertion cannot be violated because an attempt to set the flags passive-thread flags is preceded by the flip of `s` from $false$ to $true$

Command line interface. For instructions on how to use monabs, execute:

```
monabs --help
```

For example to check the user-specified assertions in the example in Figure 28, file `tsl.c` say, with unbounded threads execute

```
monabs --build-tts --concurrency tsl.c
```

The first option activates the explicit monotone dual-reference program language interface extension of monabs, while the second one selects our monotonicity-aware Das/Dill algorithm (Figure 26) and hence enables the support for the above concurrency primitives. In total, monabs requires 4 CEGAR iterations to certify program safety for an unbounded number of executing threads. In particular four shared, one local and one single-thread predicates are discovered: predicates $\texttt{c} == 1$, $\texttt{LOCK} == false$ and $\texttt{COND} == \texttt{LOCK}$ are added initially, while $\texttt{c} == 0$, $\texttt{LOCK} == true$ and $\texttt{COND} == 1$ are added in the last iteration (the two remaining iterations give rise to 3 transition refinements).

Cartesian abstraction with maximum cube length approximation [20] can be enabled by passing option `--abstractor cartesian`; the maximum length of cubes can then be adjusted via option `--max-cube-length` (default: 3).

$\langle \#S \rangle$ ::= '1' | '2' | ...
$\langle \#L \rangle$ ::= '1' | '2' | ...
$\langle s \rangle$ ::= '0' | '1' | ... | $\langle \#S \rangle$
$\langle l \rangle$ ::= '0' | '1' | ... | $\langle \#L \rangle$

$\langle trans \rangle$ ::= $\langle s \rangle$ $\langle l \rangle$ '->' $\langle s \rangle$ $\langle l \rangle$ [$\langle ptrans \rangle$]*
$\langle ptrans \rangle$ ::= $\langle l \rangle$ '~>' $\langle l \rangle$

$\langle prog \rangle$ ::= $\langle \#S \rangle$ $\langle \#L \rangle$ [$\langle trans \rangle$]*

Figure 30: **The breach input language: EMDR programs** — Syntax of explicit monotone dual-reference programs

5.2 The Infinite-Thread Model Checker breach

The infinite-state model checker breach implements Algorithm 3 and the optimistic variant as described in Section 4.3.2, equipped with an incomplete generalisation of the Karp-Miller procedure as coverability oracle. The backward search runs in parallel with the oracle, which reports coverability results to a shared data pool that the backward search taps into at regular intervals.

In order to measure the impact of our new approach, the widening and oracle can be deactivated, turning breach into the refined version of the classical backward search (Algorithm 1). As a trade-off between efficiency and proof compaction, breach does by default not add candidate vertices that involve two threads or more. To store and manipulate upward-closed sets, breach uses a proprietary tree-based data structure for Algorithm 3, while hash tables are used in the optimistic variant. In the latter case, we implement the time-critical test for upward-closed set inclusion by standard set inclusion tests on covered states.

Input format. As input, breach takes *explicit monotone dual-reference* (EMDR) programs as defined by the grammar in Figure 30; this format is also generated by monabs in step ② of our refinement algorithm shown in Figure 26.

An EMDR program declares the total number of shared and local states, followed by a sequence of transitions. The first component of a transition specifies a designated update of the shared and active-thread local state, i.e. in part $\langle s \rangle$ $\langle l \rangle$ '->' $\langle s \rangle$ $\langle l \rangle$ of $\langle trans \rangle$, while the rest lists simultaneously executed passive-

Abstract transition				
`b[1]`	$\texttt{b[1]}_P$	`b[1]'`	$\texttt{b[1]}'_P$	EMDR transition
false	*false*	*true*	*false*	
false	*false*	*true*	*true*	`0 0 -> 0 1 0 ~> 0 0 ~> 1 1 ~> 0`
false	*true*	*true*	*false*	
true	*false*	*true*	*false*	
true	*true*	*true*	*true*	`0 1 -> 0 1`

Table 6: **DR transitions and their encoding** — Transitions that modify the shared and active-thread state in the same way (delimited by a horizontal line) are represented by a single transition in the EMDR program

thread local state updates. A local state '`1`', say, that does not appear on the left-hand side of any of the rules in the second part is subject to special treatment: passive threads residing in such states are not affected by the transition, i.e. they are treated as if '`1~>1`' appeared in the list. This treatment ensures that every EMDR program characterises a *monotone* dual-reference program, and in particular one that is strictly asynchronous if all ⟨*ptrans*⟩-parts are empty.

Table 6 illustrates how the format relates to our model introduced in Section 3.2.1.

Command line interface. For instructions on how to use breach, run

```
breach --help
```

For example to check whether a program state in $\uparrow(0 \mid 16, 16)$ can be reached in the explicit monotone dual-reference program obtained for the ticket lock (shown in Figure 31), file `ticket.EMDR` say, from an initial state in $\downarrow(0 \mid 2, 2, \ldots)$ execute

```
breach --init "0/2" --target "0|16,16" ticket.EMDR
```

The argument to the first option indicates that an unbounded number of threads initially reside in local state 2, while the second argument `0|16,16` specifies the minimal element of the target states. breach can generate the forward search tree and the backward graph in the form of `.dot` files that serve as input for the `graphviz` package [81].

```
#init 0/2
#target 1|25,25
2 28
0 0->0 8 0~>0 0~>4 2~>27 3~>27 4~>0 4~>4 6~>27 7~>27 8~>8 8~>12 10~>27 11~>27 12~>8 12~>12 14~>27
      15~>27 16~>16 16~>20 18~>27 19~>27 20~>16 20~>20 22~>27 23~>27
0 0->0 9 0~>0 0~>4 2~>27 3~>27 5~>1 6~>27 7~>27 8~>8 8~>12 10~>27 11~>27 13~>9 14~>27 15~>27 16~>16
      16~>20 18~>27 19~>27 21~>17 22~>27 23~>27
0 0->0 10 0~>2 0~>6 1~>3 2~>27 3~>27 4~>2 4~>6 5~>7 6~>27 7~>27 8~>10 8~>14 9~>11 10~>27 11~>27
      12~>10 12~>14 13~>15 14~>27 15~>27 16~>18 16~>22 17~>19 18~>27 19~>27 20~>18 20~>22 21~>23
      22~>27 23~>27
0 0->0 11 0~>2 0~>6 1~>3 2~>27 3~>27 4~>6 5~>3 6~>27 7~>27 8~>10 8~>14 9~>11 10~>27 11~>27 12~>14
      13~>11 14~>27 15~>27 16~>18 16~>22 17~>19 18~>27 19~>27 20~>22 21~>19 22~>27 23~>27
0 0->0 12 0~>0 0~>4 2~>27 3~>27 6~>27 7~>27 8~>8 8~>12 10~>27 11~>27 14~>27 15~>27 16~>16 16~>20
      18~>27 19~>27 22~>27 23~>27
0 0->0 13 0~>0 0~>4 1~>27 2~>27 3~>27 5~>27 6~>27 7~>27 8~>8 8~>12 9~>27 10~>27 11~>27 13~>27 14~>27
      15~>27 16~>16 16~>20 17~>27 18~>27 19~>27 21~>27 22~>27 23~>27
0 1->0 8 1~>1 1~>5 2~>27 3~>27 4~>0 4~>4 5~>27 6~>27 7~>27 9~>9 9~>13 10~>27 11~>27 12~>8 12~>12
      13~>27 14~>27 15~>27 17~>17 17~>21 18~>27 19~>27 20~>16 20~>20 21~>27 22~>27 23~>27
0 1->0 9 2~>27 3~>27 5~>27 6~>27 7~>27 10~>27 11~>27 13~>27 14~>27 15~>27 18~>27 19~>27 21~>27 22~>27
       23~>27
0 1->0 10 0~>2 1~>3 1~>7 2~>27 3~>27 4~>2 4~>6 5~>27 6~>27 7~>27 8~>10 9~>11 9~>15 10~>27 11~>27
      12~>10 12~>14 13~>27 14~>27 15~>27 16~>18 17~>19 17~>23 18~>27 19~>27 20~>18 20~>22 21~>27
      22~>27 23~>27
0 1->0 11 0~>2 1~>3 2~>27 3~>27 4~>6 5~>27 6~>27 7~>27 8~>10 9~>11 10~>27 11~>27 12~>14 13~>27 14~>27
       15~>27 16~>18 17~>19 18~>27 19~>27 20~>22 21~>27 22~>27 23~>27
0 1->0 12 1~>1 1~>5 2~>27 3~>27 5~>27 6~>27 7~>27 9~>9 9~>13 10~>27 11~>27 13~>27 14~>27 15~>27
      17~>17 17~>21 18~>27 19~>27 21~>27 22~>27 23~>27
0 2->0 14 0~>27 1~>27 2~>2 2~>6 4~>27 5~>27 8~>27 9~>27 10~>10 10~>14 12~>27 13~>27 16~>27 17~>27
      18~>18 18~>22 20~>27 21~>27
0 2->0 15 0~>27 1~>27 2~>2 2~>6 3~>27 4~>27 5~>27 7~>27 8~>27 9~>27 10~>10 10~>14 11~>27 12~>27
      13~>27 15~>27 16~>27 17~>27 18~>18 18~>22 19~>27 20~>27 21~>27 23~>27
0 3->0 14 0~>27 1~>27 3~>3 3~>7 4~>27 5~>27 7~>27 8~>27 9~>27 11~>11 11~>15 12~>27 13~>27 15~>27
      16~>27 17~>27 19~>19 19~>23 20~>27 21~>27 23~>27
0 4->0 8 2~>27 3~>27 4~>0 4~>4 6~>27 7~>27 10~>27 11~>27 12~>8 12~>12 14~>27 15~>27 18~>27 19~>27
      20~>16 20~>20 22~>27 23~>27
0 4->0 9 2~>27 3~>27 5~>1 6~>27 7~>27 10~>27 11~>27 13~>9 14~>27 15~>27 18~>27 19~>27 21~>17 22~>27
      23~>27
0 4->0 10 0~>2 1~>3 2~>27 3~>27 4~>2 4~>6 5~>7 6~>27 7~>27 8~>10 9~>11 10~>27 11~>27 12~>10 12~>14
      13~>15 14~>27 15~>27 16~>18 17~>19 18~>27 19~>27 20~>18 20~>22 21~>23 22~>27 23~>27
```

Figure 31: **Ticket lock abstraction** — To map variable valuations to integers, we pick up our notation from the end of Section 3.1.2, and furthermore associate a local state $(\ell_i/\mathtt{b}[1]\mathtt{b}[2]\mathtt{b}[3])$ with value $8(i-1)+4\mathtt{b}[1]+2\mathtt{b}[2]+1\mathtt{b}[3]$, e.g. $(\ell_1/010) = 14$. The initial state is $(0 \mid 2, 2)$, and the set of error states is given by $\{(0 \mid i, j) \ : \ i, j \in \{16, \ldots, 23\}\}$

```
0 4->0 11 0~>2 1~>3 2~>27 3~>27 4~>6 5~>3 6~>27 7~>27 8~>10 9~>11 10~>27 11~>27 12~>14 13~>11 14~>27
      15~>27 16~>18 17~>19 18~>27 19~>27 20~>22 21~>19 22~>27 23~>27
0 4->0 12 2~>27 3~>27 6~>27 7~>27 10~>27 11~>27 14~>27 15~>27 18~>27 19~>27 22~>27 23~>27
0 4->0 13 1~>27 2~>27 3~>27 5~>27 6~>27 7~>27 9~>27 10~>27 11~>27 13~>27 14~>27 15~>27 17~>27 18~>27
      19~>27 21~>27 22~>27 23~>27
0 4->0 14 0~>2 1~>3 2~>27 3~>27 4~>6 5~>7 6~>27 7~>27 8~>10 9~>11 10~>27 11~>27 12~>14 13~>15 14~>27
      15~>27 16~>18 17~>19 18~>27 19~>27 20~>22 21~>23 22~>27 23~>27
0 4->0 15 0~>2 1~>27 2~>27 3~>27 4~>6 5~>27 6~>27 7~>27 8~>10 9~>27 10~>27 11~>27 12~>14 13~>27
      14~>27 15~>27 16~>18 17~>27 18~>27 19~>27 20~>22 21~>27 22~>27 23~>27
0 5->0 8 1~>27 2~>27 3~>27 4~>0 4~>4 5~>27 6~>27 7~>27 9~>27 10~>27 11~>27 12~>8 12~>12 13~>27 14~>27
       15~>27 17~>27 18~>27 19~>27 20~>16 20~>20 21~>27 22~>27 23~>27
0 5->0 10 0~>2 1~>27 2~>27 3~>27 4~>2 4~>6 5~>27 6~>27 7~>27 8~>10 9~>27 10~>27 11~>27 12~>10 12~>14
      13~>27 14~>27 15~>27 16~>18 17~>27 18~>27 19~>27 20~>18 20~>22 21~>27 22~>27 23~>27
0 5->0 12 1~>27 2~>27 3~>27 5~>27 6~>27 7~>27 9~>27 10~>27 11~>27 13~>27 14~>27 15~>27 17~>27 18~>27
      19~>27 21~>27 22~>27 23~>27
0 5->0 13 1~>27 2~>27 3~>27 5~>27 6~>27 7~>27 9~>27 10~>27 11~>27 13~>27 14~>27 15~>27 17~>27 18~>27
      19~>27 21~>27 22~>27 23~>27
0 5->0 14 0~>2 1~>27 2~>27 3~>27 4~>6 5~>27 6~>27 7~>27 8~>10 9~>27 10~>27 11~>27 12~>14 13~>27
      14~>27 15~>27 16~>18 17~>27 18~>27 19~>27 20~>22 21~>27 22~>27 23~>27
0 5->0 15 0~>2 1~>27 2~>27 3~>27 4~>6 5~>27 6~>27 7~>27 8~>10 9~>27 10~>27 11~>27 12~>14 13~>27
      14~>27 15~>27 16~>18 17~>27 18~>27 19~>27 20~>22 21~>27 22~>27 23~>27
0 6->0 14 0~>27 1~>27 4~>27 5~>27 8~>27 9~>27 12~>27 13~>27 16~>27 17~>27 20~>27 21~>27
0 6->0 15 0~>27 1~>27 3~>27 4~>27 5~>27 7~>27 8~>27 9~>27 11~>27 12~>27 13~>27 15~>27 16~>27 17~>27
      19~>27 20~>27 21~>27 23~>27
0 7->0 14 0~>27 1~>27 3~>27 4~>27 5~>27 7~>27 8~>27 9~>27 11~>27 12~>27 13~>27 15~>27 16~>27 17~>27
      19~>27 20~>27 21~>27 23~>27
0 9->0 17 2~>27 3~>27 5~>27 6~>27 7~>27 10~>27 11~>27 13~>27 14~>27 15~>27 18~>27 19~>27 21~>27
      22~>27 23~>27
0 11->0 19 0~>27 1~>27 4~>27 5~>27 7~>27 8~>27 9~>27 12~>27 13~>27 15~>27 16~>27 17~>27 20~>27 21~>27
       23~>27
0 13->0 21 1~>27 2~>27 3~>27 5~>27 6~>27 7~>27 9~>27 10~>27 11~>27 13~>27 14~>27 15~>27 17~>27 18~>27
       19~>27 21~>27 22~>27 23~>27
0 15->0 23 0~>27 1~>27 3~>27 4~>27 5~>27 7~>27 8~>27 9~>27 11~>27 12~>27 13~>27 15~>27 16~>27 17~>27
      19~>27 20~>27 21~>27 23~>27
0 16->0 26 0~>0 0~>1 1~>0 2~>27 3~>27 4~>4 4~>5 5~>4 6~>27 7~>27 8~>8 8~>9 9~>8 10~>27 11~>27 12~>12
      12~>13 13~>12 14~>27 15~>27 16~>16 16~>17 17~>16 18~>27 19~>27 20~>20 20~>21 21~>20 22~>27
      23~>27
0 16->1 25 16~>25 17~>25 20~>25 21~>25
0 17->0 26 0~>0 0~>1 1~>0 2~>27 3~>27 4~>4 4~>5 5~>27 6~>27 7~>27 8~>8 8~>9 9~>8 10~>27 11~>27 12~>12
       12~>13 13~>27 14~>27 15~>27 16~>16 16~>17 17~>16 18~>27 19~>27 20~>20 20~>21 21~>27 22~>27
      23~>27
0 17->1 25 16~>25 17~>25 20~>25
0 18->0 26 0~>27 1~>27 2~>2 2~>3 3~>2 4~>27 5~>27 6~>6 6~>7 7~>6 8~>27 9~>27 10~>10 10~>11 11~>10
      12~>27 13~>27 14~>14 14~>15 15~>14 16~>27 17~>27 18~>18 18~>19 19~>18 20~>27 21~>27 22~>22
      22~>23 23~>22
0 18->1 25 18~>25 19~>25 22~>25 23~>25
0 19->0 26 0~>27 1~>27 2~>2 2~>3 3~>2 4~>27 5~>27 6~>6 6~>7 7~>27 8~>27 9~>27 10~>10 10~>11 11~>10
      12~>27 13~>27 14~>14 14~>15 15~>27 16~>27 17~>27 18~>18 18~>19 19~>18 20~>27 21~>27 22~>22
      22~>23 23~>27
0 19->1 25 18~>25 19~>25 22~>25
0 20->0 26 0~>0 0~>1 1~>0 2~>27 3~>27 4~>4 4~>5 5~>4 6~>27 7~>27 8~>8 8~>9 9~>8 10~>27 11~>27 12~>12
      12~>13 13~>12 14~>27 15~>27 16~>16 16~>17 17~>16 18~>27 19~>27 20~>20 20~>21 21~>20 22~>27
      23~>27
0 20->1 25 16~>25 17~>25 20~>25 21~>25
0 21->0 26 0~>0 0~>1 1~>27 2~>27 3~>27 4~>4 4~>5 5~>27 6~>27 7~>27 8~>8 8~>9 9~>27 10~>27 11~>27
      12~>12 12~>13 13~>27 14~>27 15~>27 16~>16 16~>17 17~>27 18~>27 19~>27 20~>20 20~>21 21~>27
      22~>27 23~>27
0 21->1 25 16~>25 20~>25
0 22->0 26 0~>27 1~>27 2~>2 2~>3 3~>2 4~>27 5~>27 6~>6 6~>7 7~>6 8~>27 9~>27 10~>10 10~>11 11~>10
      12~>27 13~>27 14~>14 14~>15 15~>14 16~>27 17~>27 18~>18 18~>19 19~>18 20~>27 21~>27 22~>22
      22~>23 23~>22
0 22->1 25 18~>25 19~>25 22~>25 23~>25
0 23->0 26 0~>27 1~>27 2~>2 2~>3 3~>27 4~>27 5~>27 6~>6 6~>7 7~>27 8~>27 9~>27 10~>10 10~>11 11~>27
      12~>27 13~>27 14~>14 14~>15 15~>27 16~>27 17~>27 18~>18 18~>19 19~>27 20~>27 21~>27 22~>22
      22~>23 23~>27
0 23->1 25 18~>25 22~>25
```

Figure 31 (continued)

Implementation details. The breach tool is implemented in C++ 11 [113] and was developed using Microsoft's Visual Studio IDE; in addition, we provide a `makefile` for Linux and Mac OS (`gcc` ≥ 4.7 required). The source code, as well as binaries for Windows, Linux and Mac OS are available online (see Section 1 for the link). In total the project consists of 37 files (8880 lines of code with a comment ratio of 23.0%), 43 classes (avg. 5 methods per class) and 120 functions. The core algorithm of the optimistic variant as described in Section 4.3.2 without data structures has 638 lines (comment ratio: 16.0%) and 11 functions, while the pessimistic variant (Algorithm 3) is slightly larger: 1192 lines with comment ratio 28.5% and 24 functions.

The following libraries are used by breach: (i) Standard Template Library, notably TR1's `unordered_set` hash tables; (ii) Combination and Permutation library by Howard Hinnant [31], a library to iterate over combinations or permutations of a set of objects (it is used to implement our upward-closed set data structure); (iii) Boost C++ library, a collection of free, peer-reviewed libraries. More specifically, we use these Boost libraries: a) `DateTime` for time-spent profiling of different program blocks, b) `Filesystem` for portable file manipulation and path handling, c) `ProgramOptions` to read user input from the command line, d) `Thread` for multiple threads of execution (one for the backward-directed search and one for the oracle), e) `Tokenizer` to parse the input programs and f) `Bimap`, a bidirectional map library, to implement our priority queues.

We employ the following testing methods to improve software quality: (i) Random testing to check functional correctness; (ii) Regression testing to ensure that code change do not reintroduce previously known bugs (58 regression tests at the date of writing). Finally, we used Intel's VTune* profiler for performance analysis and tuning, and Intel's ParallelInspector tool to do memory and thread checking (memory leaks, dangling pointers, uninitialised variables, deadlocks etc.).

*`http://software.intel.com/en-us/intel-vtune-amplifier-xe`

6 Experimental Evaluation

In this section, we evaluate our program verifier monabs and the (back-end) model checker breach on a set of 37 non-recursive shared-memory C programs. Section 6.1 provides a detailed performance analysis of monabs. A comparison of monabs and breach against existing techniques is given in Section 6.2 and Section 6.3, respectively.

The experiments are performed on a 3 GHz Intel Xeon machine with 20 GB memory, running 64-bit Linux, with a timeout of 30 minutes. We first provide an overview of the benchmark set.

Benchmark selection. For experimental evaluation, we selected a representative set of multi-threaded C programs with the following characteristics: (i) parameterised and non-parameterised programs; (ii) programs that exhibit a variety of synchronisation primitives; (iii) programs with intricate functional properties such as the success of concurrent push operations on a stack, as well as more control-driven properties such as to establish mutual exclusion; (iv) programs that exhibit basic C features including flow of control primitives (`if`, `while`, `switch`, `break`, `goto` etc.), arithmetical, logical and bit-level operators ($+$, $||$, $|$ etc.), assignments and (non-recursive) function calls, arrays, and enumerated (`enum`) types.

Due to front-end limitations of monabs, we do not consider programs written in other languages than C, and furthermore focus on intricate concurrency aspects, rather than intricate language features such as pointers. Finally, we avoid to use too closely related programs, and exclude those with EMDR abstractions of 10 megabyte or more. Note that the latter restriction implicitly limits the number predicates that can be used in a proof (this restriction can be avoided by applying breach on symbolic EMDR encodings, which is one aspect of future work; see the discussion in Section 8 on this subject). For example, our approach cannot handle benchmark QRCU-4 from [96], which requires more than 10 predicates.

With this focus in mind, we finally selected 37 programs from these sources: (i) classical text books on concurrency (e.g. [154] and [7]), (ii) open source projects such as [9], (iii) Unix like operating system code [131], (iv) classical mutex algorithms including [140, 155, 166]), and (v) recent works on the automated verification of multi-threaded programs such as [96, 71, 95].

Program benchmarks. In the 37 non-recursive concurrent C programs we use, threads communicate through shared-memory and synchronise via locks, or in a lock-free manner via atomic compare-and-swap instructions, and condition variables. For each program, we verify a safety property, specified as an assertion. To measure performance for unbounded-thread verification, we chose primarily assertions that hold. In total, the programs comprise 4583 lines of code, featuring 2.5 shared and 4.6 local variables on average. Five programs use broadcast operations on condition variables; the programs are*:

1–10: thread-safe algorithms ([154, 7, 9]): atomic counters (1–2); find the maximum element in an array (3–6); concurrent pseudo-random number generators (7–8); stack data structure with concurrent pushes and pops (9–10) (STACK-C is adapted from an Open Source IBM implementation). For each type, we consider a version with **L**ocks, and a lock-free variant with **C**ompare-and-swap primitives (indicated in Table 7 by the suffix).

11–15: OS code ([131]): code from the FreeBSD (11–12), NetBSD (13) and Solaris (14) open-source operating systems that is related to RDMA ZFS file system support and interface/system monitoring (multiple kernel threads are simultaneously unblocked via condition variables); Linux driver skeleton that mimics concurrent open, close and ioctl calls (15).

16–26: mutex algorithms ([77, 140, 155, 166]): multiple locks control access to a shared resource (16–18); the ticket algorithm as in Figure 5 (19); classical algorithms due to Szymanski, Peterson and Dekker (20–22); a readers-writer and timed mutex (23–24); high-contention ticket algorithm with proportional backoff (25); test-and-set lock (26).

27–32: misc ([58, 78, 135, 118]): two programs that require single-thread predicates (27-28); threads synchronise via broadcasts (29); a program amenable to thread-modular verification (30); a vulnerability fix from the Mozilla repository (31); a program used to illustrate incremental coverability proofs (32).

33–37: pthread programs ([118]): several programs that use the C POSIX Threads library.

Of these, 34 were used for evaluation purposes in prior work [58, 118, 96, 71, 95]. The remaining programs are 25, 29 and 32, namely (i) a variant of the ticket algorithm, where each thread acquires the lock infinitely often, and a thread waits the

*They are available online (see Section 1 for the link), and were submitted as concurrency benchmarks to the 3rd International Competition on Software Verification held at TACAS 2014 in Grenoble, France.

longer to check if access is granted, the more distant the served ticket is; (ii) a simple program, where a thread can control a local Boolean variables of the passive threads; (iii) our running example from Section 4. Most programs contain procedures executed by an arbitrary number of threads, which are dynamically spawned by the (single) initial thread. Exceptions are 20–24 and 31 from [95], which are designed for a fixed thread count of two; the program behaviour stabilises for $n \geq 2$. These examples do not even exploit the power of our approach to deal with unbounded threads.

Program assertions. We briefly describe the properties we check:

1–10: thread-safe algorithms: each call to the increment function strictly increases the counter (1–2); a function returns the maximum element of an array (3–6); two subsequent calls to the generator never yield the same value (7–8); after a push, the stack is non-empty (9–10).

11–15: OS code: routines are mutually accessed and a condition holds after broadcasts (11–14); the device's use counter is zero when it is unregistered (15).

16–26: mutex algorithms: mutual exclusion is correctly established (16–26).

27–32: misc: a shared and a local variable are equal after some local computation (27–28); a property that depends on a condition variable (29); a shared flag is not reset by another thread in a certain code region (30); two operations are executed in the correct order (31); the value two is never assigned to a shared variable (32).

33–37: pthread programs: values in a triple nested loop remain in a certain range (33); two variables equal each other (34); a shared variable accessed concurrently by multiple functions remains in a certain range (35–37).

6.1 Detailed Evaluation of monabs

Table 7 shows detailed experimental results obtained for monabs. Within 187s and at most 8 abstraction-refinement iterations, monabs succeeds in certifying correctness of all 34 safe programs for arbitrary thread counts, and reporting counterexamples for the three remaining buggy instances BOOP, BS-LOOP and PTHREAD. As usual, the model-checking time dominates the total run time. The cost of the monotone closure operation is negligible and not shown.

Shared and single-thread predicates were required to succeed in 28 of the 37 cases, and inter-thread predicates for the two ticket lock algorithms. Roughly every

other benchmark exhibits abstractions that are "truly DR", i.e. they permit no strictly asynchronous encoding. As a consequence, existing model checkers for concurrent software become inapplicable. In five of these cases, passive-thread variable updates are used in the input program to model broadcast operations. These are then passed on to the abstraction via local predicates: programs with a mark in column *Cnd*. For the other 11, originally strictly asynchronous programs, strict asynchrony is lost after inter- or non-local single-thread predicates are discovered during refinement: programs with $IT \neq 0$ or $ST \neq L$. Only 5 programs whose abstraction is "truly" DR come out monotone; the clear majority, namely 11, *require* the application of the closure operation from Definition 20 in order to be passed to the back-end model checker. Single-thread predicates of the form `s = 1` are, e.g., required to track the success of the compare-and-swap primitive for programs INC-C, MAXSIMP-C and MAXOPT-C. The bug in BOOP can be detected in depth 45 with at least 3 threads and involves 4 context switches, that in BS-LOOP manifests in depth 17 and even with a single executing thread, and that in PTHREAD emerges in depth 56 with at least 3 threads and 5 context switches.

The results also demonstrate the adequacy of our predicate language: banning any one of the predicate types supported by our approach renders some of our C programs unprovable. On the other hand, we are not aware of any program that would require an even richer predicate language, underpinning the claim made in [44] that "properties involving three or more [threads] at a time are rare".

6.2 Comparison with symmpa and cream

Existing tools for source code programs with unbounded threads, such as duet, are insufficient for our benchmarks; Section 7 has details. We therefore evaluate at which point the search for monotone proofs pays off compared to checking increasing constant thread counts. We do so using three recent fixed-thread approaches:

symmpa *([58, 57]):* predicate abstraction for fixed threads.
cream−rely *([95, 94]):* cream with rely-guarantee proofs.
cream−owicki *([96, 94]):* cream with Owicki-Gries proofs.

Front-end capabilities of cream and symmpa are similar to that of monabs, facilitating a comparison. A notable difference is that cream relies on natural integers for representing numeric C types, whereas monabs and symmpa support machine integers. Neither cream nor symmpa support broadcasts, as used by 5 of our bench-

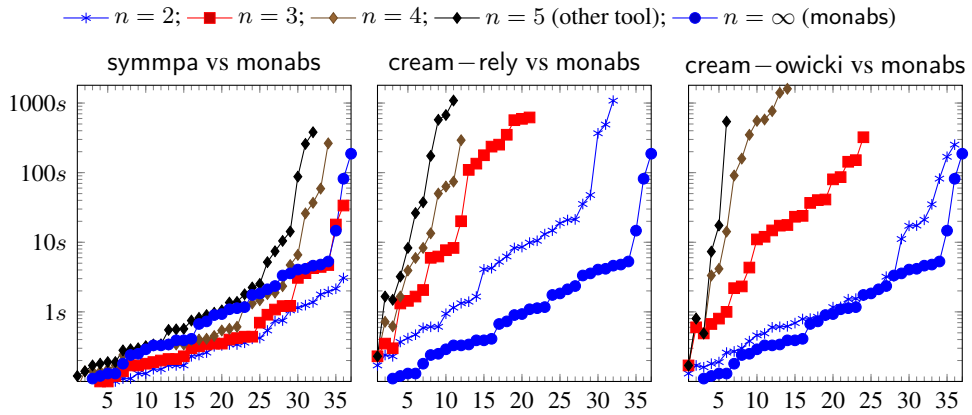

Figure 32: **Comparison with fixed-thread tools** — Cactus plots comparing monabs with three recent fixed-thread proof methods; Table 8 provides the per-program run times

marks, which is why we instead apply them to slightly modified and broadcast-free overapproximations.

Figure 32 plots the fraction of programs checked successfully by different methods for given thread numbers. Each subfigure shows five curves: one for monabs and *unbounded* thread count ($n = \infty$), and four corresponding to the respective *fixed*-thread tool with $n = 2, \ldots, 5$ concurrent threads: an entry of the form (k, t) shows the time t it took to solve the k easiest (for the given method) of the C programs. The results reveal the superiority of our unbounded approach over each of the fixed-thread verifiers, even for trivial thread counts. For symmpa and cream the proof time grows exponentially with the thread count. The single timeout for symmpa with $n = 2$ is for the high-contention variant of the ticket lock (TICKET-HC): symmpa is unable to track the uniqueness of a ticket, and as a result times out while attempting to enumerate the possible ticket values. For the simpler variant, where every thread acquires and releases the lock only once (TICKET), symmpa succeeds for up to 3 threads.

6.3 Comparison with Coverability Checkers

To compare with existing coverability checkers, we focus on programs with strictly asynchronous abstractions (no mark in column "*DR?*" of Table 7), and the OS code (benchmarks 11–15), i.e. 25 of the 37 C programs in total. The monabs verifier

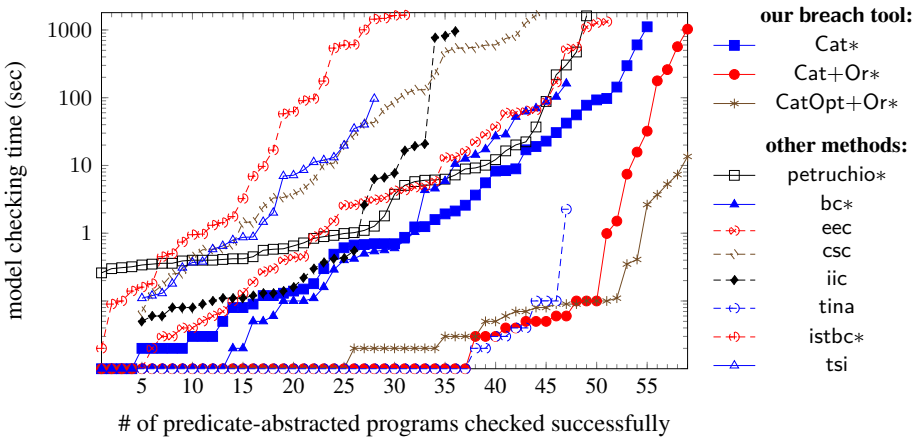

Figure 33: **Comparison for CEGAR abstractions** — Cactus plot comparing our implementation of the target set widening algorithm in the breach tool with existing coverability methods. An entry of the form (k, t) for some curve shows the time t it took to solve the k easiest—for the method associated with that curve—predicate-abstracted C programs (the order thus varies across methods). Four benchmarks feature transfer transitions, namely those obtained from programs with condition variables (cf. Table 7). Tools supporting transfers are marked $*$. The curves for the other tools start at $k = 5$, skipping the four transfer benchmarks

requires 59 CEGAR iterations until the abstractions for this set of programs stabilise; e.g. 3 iterations are necessary to prove the program in Figure 15 correct (MINUCP-EX). Because we use a preliminary version of monabs this value differs from the sum of the iteration counts, as listed in Table 7 for these programs.

The obtained abstractions feature up to 34880 thread states (for FUNC-P), and 746770 transitions (for DOUBLE-2). Figure 33 plots the total model checking run times (scaled logarithmically) for all methods. The curves in the graph correspond to the following checkers ($*$ indicates that the tool supports transfer transitions):

Cat$*$*:* Our Algorithm 3 with no coverability oracle (v1.0).
Cat+Or$*$*:* Our Algorithm 3 equipped with the coverability oracle (v1.0).
CatOpt+Or$*$*:* Optimistic variant of Algorithm 3 (Section 4.3.2) with oracle (v2.0).
petruchio$*$ *([141]):* Backward search with several heuristics (v0.1).
bc$*$ *([2, 1]):* Backward reachability analysis (Algorithm 1).
eec *([88]):* Forward analysis with enumerative refinement (v1.03).
csc *([89]):* A refined Karp-Miller procedure (v0.1).

iic *([123]):* Incremental, inductive coverability algorithm.
tina *([26]):* The classic Karp-Miller tree construction (v3.0).
istbc∗ *([85]):* Standard backward reachability analysis (v1.03).
tsi *([83]):* A variant of eec with backward refinement (v1.03).

Our methods based on widening achieve significantly better results compared to existing methods. The improvement of about two orders of magnitude over the best previous method petruchio, which in turn is roughly two orders of magnitude faster than the remaining methods, shows that our widening approach has far more impact than, e.g. structural invariant heuristics. Particularly for program abstractions we found that statically precomputed overapproximations tend to be irrelevant for the safety property or too imprecise, underpinning the claim made in [75]. On the other hand, the inferiority of the pure backward analysis bc compared to petruchio indicates that the observed improvements are a consequence of our widening technique.

To measure the difference between standard and minimal uncoverability proofs, we removed the self-imposed upper bound on the number of threads in candidate vertices. In this setup, we observed the following reductions (averaged): the length of the longest traversed path goes down from 28 to 14 (-50%), the thread count appearing in the proof from 6 to 2 (-67%), and the number of minimal states ($=$ proof size) from 22,518 to 1,222 (-95%). While the classical backward approach involves up to eight threads in a proof, our approach generates a minimal uncoverability proofs with 3 threads for MINUCP-EX (cf. Figure 19), and proofs with no more than two threads for the other 24 programs. The bound on the thread dimension in candidate vertices mentioned above diminishes these improvements somewhat but marginally (we enforced the bound for the results in Figure 33 as it results in much reduced runtimes).

		Characteristics						Predicates				CEGAR phases and times (in sec.)					
Program		*S*	*L*	*LOC*	*Mtx?*	*Cnd?*	*Safe?*	*SP*	*ST*	*L*	*IT*	*Its*	*DR?*	*Mon?*	*Abs*	*Ref*	*Chk*	*Total*
1/INC-L	thread-safe algorithms	2	1	46	●	○	●	2	2	1	0	4	●	○	.7	.1	.4	1.2
2/INC-C		1	3	57	○	○	●	0	5	4	0	8	●	○	4.0	.11	74.5	187.1
3/MAXSIMP-L		3	3	59	●	○	●	1	1	0	0	2	●	○	.1	.0	.2	.3
4/MAXSIMP-C		2	5	79	○	○	●	0	3	2	0	3	●	○	.5	.0	.6	1.1
5/MAXOPT-L		3	4	69	●	○	●	1	2	1	0	2	●	○	.4	.1	.5	1.1
6/MAXOPT-C		2	6	86	●	○	●	0	4	3	0	3	●	○	1.9	.1	2.4	4.6
7/PRNGSIMP-L		2	4	63	●	○	●	2	2	2	0	4	○	–	.0	.0	.2	.3
8/PRNGSIMP-C		1	5	95	○	○	●	0	2	2	0	2	○	–	.0	.0	.1	.1
9/STACK-L		4	2	79	●	○	●	2	1	1	0	3	○	–	.0	.0	.2	.2
10/STACK-C		3	3	89	○	○	●	1	2	2	0	3	○	–	.0	.0	.2	.2
11/BSD-AK2	OS code	1	7	516	●	●	●	1	1	1	0	1	●	●	.0	.0	4.8	4.8
12/BSD-RA2		2	21	413	●	●	●	1	1	0	0	1	●	●	.0	.0	1.8	1.8
13/NETBSD-SP2		1	28	1045	●	●	●	2	1	1	0	1	●	●	.0	.0	81.7	81.7
14/SOLARIS-SM2		1	56	616	●	●	●	4	1	1	0	1	●	●	.0	.0	2.3	2.4
15/BOOP		5	2	89	○	○	○	5	2	2	0	8	○	–	.0	.1	.7	.9
16/DOUBLE-1	mutex algorithms	3	0	70	●	○	●	8	0	0	0	7	○	–	.7	.2	2.3	3.6
17/DOUBLE-2		3	0	73	●	○	●	7	0	0	0	7	○	–	1.4	.2	1.5	3.3
18/DOUBLE-3		3	0	66	●	○	●	5	0	0	0	5	○	–	.2	.1	.5	.9
19/TICKET		3	1	46	○	○	●	0	1	0	2	4	●	○	.2	.1	3.9	4.2
20/SZYMANSKI		3	0	54	○	○	●	4	0	0	0	4	○	–	.0	.0	.3	.3
21/PETERSON		4	0	37	○	○	●	6	0	0	0	6	○	–	.0	.0	.3	.4
22/DEKKER		4	0	50	○	○	●	4	0	0	0	4	○	–	.0	.0	.2	.3
23/RW-LOCK		4	0	58	○	○	●	5	0	0	0	6	○	–	.0	.0	.4	.5
24/TIMED-MUTEX		5	0	63	○	○	●	5	0	0	0	4	○	–	.0	.0	.3	.3
25/TICKET-HC		3	1	61	○	○	●	0	1	0	2	4	●	○	.2	.0	5.1	5.3
26/TAS-L		2	2	58	○	○	●	4	2	1	0	4	●	○	.2	.0	1.5	2.1
27/UNVEREX	misc	2	1	25	○	○	●	4	2	0	0	6	●	●	.9	.1	2.5	4.1
28/MIXEDASGN		1	1	16	○	○	●	0	2	1	0	3	●	○	.1	.0	.2	.3
29/CV-VAR		1	1	14	○	●	●	1	1	1	0	1	●	○	.0	.0	.0	.1
30/SPIN		2	0	37	●	○	●	2	0	0	0	2	○	–	.0	.0	.1	.1
31/MOZILLA-VF		4	3	82	●	○	●	5	0	0	0	6	○	–	.0	.0	.3	.4
32/MINUCP-EX		1	0	80	○	○	●	2	0	0	0	2	○	–	.0	.0	.1	.1
33/BS-LOOP	pthread prog.	0	6	24	○	○	○	0	7	7	0	1	○	–	7.3	.0	7.3	14.7
34/COND		1	3	56	●	○	●	0	2	2	0	2	○	–	.0	.0	.1	.1
35/S-LOOP		5	0	60	●	○	●	3	0	0	0	3	○	–	.0	.0	.1	.1
36/FUNC-P		2	1	67	●	○	●	2	4	4	0	6	○	–	.5	.2	1.0	1.8
37/PTHREAD		5	0	85	●	○	○	6	0	0	0	6	○	–	.1	.0	.6	.7

Table 7: **Benchmark characteristics and results** — S, L, *LOC* = # of shared vars, local vars, lines of code; *Mtx?*, *Cnd?*, *Safe?* = presence of mutexes, condition variables, program safety (black disc = "yes"); *SP*, *ST*, *L*($\leq$*ST*), *IT* = # of shared, single-thread, local, inter-thread predicates; *Its*, *DR?*, *Mon?* = # of CEGAR iterations, abstractions are truly DR, if so whether they are monotone; *Abs*, *Ref*, *Chk*, *Total* = abstraction, refinement, model checking (breach), total time

Program		symmpa $n=2$	symmpa $n=3$	symmpa $n=4$	symmpa $n=5$	cream−owicki $n=2$	cream−owicki $n=3$	cream−owicki $n=4$	cream−owicki $n=5$	cream−rely $n=2$	cream−rely $n=3$	cream−rely $n=4$	cream−rely $n=5$	monabs $n=\infty$
1/INC-L	thread-safe algorithms	1.1	1.1	1.3	2.2	1.1	42.1	*time out*		12.3	*time out*	*time out*	*time out*	1.2
2/INC-C		.8	.9	4.7	87.6	.9	12.5	557.0	*time out*	1.5	3.1	7.7	39.5	187.1
3/MAXSIMP-L		.2	.4	1.2	5.2	2.6	81.7	*time out*	*time out*	21.2	*time out*	*time out*	*time out*	2.1
4/MAXSIMP-C		.3	.4	.5	1.0	18.1	*time out*	*time out*	*time out*	37.5	*time out*	*time out*	*time out*	.3
5/MAXOPT-L		2.0	4.4	263.9	*time out*	36.4	*time out*	*time out*	*time out*	2.6	7.9	52.3	1092.1	1.1
6/MAXOPT-C		.5	.7	1.9	10.5	84.2	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	1.1
7/PRNGSIMP-L		2.2	3.6	58.9	*time out*	256.2	*time out*	*time out*	*time out*	6.0	22.4	296.5	*time out*	4.6
8/PRNGSIMP-C		.2	.2	.4	1.1	1.1	18.4	351.6	*time out*	6.5	113.2	*time out*	*time out*	4.0
9/STACK-L		.1	.1	.1	.2	.3	.8	4.4	17.9	1.3	7.9	78.8	682.0	.1
10/STACK-C		.1	.2	.2	.3	1.5	*time out*	*time out*	*time out*	5.3	*time out*	*time out*	*time out*	.4
11/BSD-AK2	OS code	.2	.3	.4	.6	3.5	*time out*	*time out*	*time out*	23.2	*time out*	*time out*	*time out*	.3
12/BSD-RA2		.2	.2	.2	.3	.6	*time out*	*time out*	*time out*	1.7	*time out*	*time out*	*time out*	.4
13/NETBSD-SP2		.4	.2	.3	.3	.8	17.6	770.6	*time out*	10.0	352.6	*time out*	*time out*	.9
14/SOLARIS-SM2		.1	.1	.2	.1	.3	1.0	3.6	7.6	.2	.4	1.0	3.4	14.7
15/BOOP		.1	.1	.1	.2	1.7	144.4	*time out*	*time out*	.6	2.1	14.8	175.8	.1
16/DOUBLE-1	mutex algorithms	.3	.4	.6	1.4	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	1.8
17/DOUBLE-2		.1	.1	.1	.2	.1	.6	14.4	542.4	14.8	624.2	*time out*	*time out*	.1
18/DOUBLE-3		.3	.4	.5	.6	.1	11.2	1611.4	*time out*	.7	254.3	*time out*	*time out*	.7
19/TICKET		1.2	1.2	1.5	2.5	.6	24.2	*time out*	*time out*	6.3	570.3	*time out*	*time out*	3.6
20/SZYMANSKI		1.8	4.7	6.6	7.4	.4	37.1	*time out*	*time out*	8.7	596.7	*time out*	*time out*	3.3
21/PETERSON		.4	.4	.6	1.4	.2	4.4	582.5	*time out*	1.4	179.5	*time out*	*time out*	.9
22/DEKKER		.1	.4	2.3	14.3	1.9	87.3	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	.2
23/RW-LOCK		.1	.1	.3	.8	1.5	152.2	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	.2
24/TIMED-MUTEX		.1	.2	.3	.8	1.6	324.2	*time out*	*time out*	.4	1.4	4.4	26.8	.1
25/TICKET-HC		.1	.1	.3	.6	3.3	*time out*	*time out*	*time out*	21.1	237.8	*time out*	*time out*	.3
26/TAS-L		.7	3.1	26.0	258.7	11.6	*time out*	*time out*	*time out*	376.5	*time out*	*time out*	*time out*	4.8
27/UNVEREX	misc	1.4	4.3	36.9	381.1	21.9	*time out*	*time out*	*time out*	500.7	*time out*	*time out*	*time out*	1.8
28/MIXEDASGN		3.1	33.7	*time out*	*time out*	171.4	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	*time out*	81.7
29/CV-VAR		1.3	1.2	1.8	1.8	18.2	*time out*	*time out*	*time out*	1100.6	*time out*	*time out*	*time out*	2.4
30/SPIN		.2	.2	.4	.9	.4	2.5	91.6	*time out*	.4	.7	1.2	2.1	.3
31/MOZILLA-VF		.0	.0	.1	.1	1.5	41.3	*time out*	*time out*	1.9	9.9	67.4	581.5	.1
32/MINUCP-EX		*time out*	*time out*	*time out*	*time out*	1.0	24.1	*time out*	*time out*	50.1	*time out*	*time out*	*time out*	5.3
33/BS-LOOP	pthread prog.	.4	18.0	*time out*	*time out*	.8	15.4	1408.6	*time out*	15.0	*time out*	*time out*	*time out*	4.2
34/COND		.3	.4	.3	.3	.5	.5	.5	.5	1.7	1.7	1.7	1.7	.3
35/S-LOOP		.3	.4	.3	.3	.2	.2	.2	.2	.2	.2	.2	.2	.7
36/FUNC-P		.2	.2	.2	.2	.8	.8	.8	.8	8.3	8.3	8.3	8.3	.4
37/PTHREAD		.2	.2	.3	.3	.2	2.2	159.0	*time out*	.4	134.7	*time out*	*time out*	.1

Table 8: **Per-program results** — Run times of symmpa, cream−owicki and cream−rely for $n = 2 \ldots 5$, and our monabs verifier for $n = \infty$ threads

7 Related Work

We begin with a brief overview of the history of concurrent software verification, and then compare with more recent works in the area.

7.1 A Brief Literature Survey

The origins of program verification research can be traced back to publications by Turing [170] in 1936, whose undecidability results ruled out the existence of general sound and complete algorithmic solutions for proving program correctness, and Goldstine and Neumann [92] in 1947, who argued that programming "has to be viewed as a logical problem" and saw the potential of assertional reasoning. The first attempts that aimed at devising practical formal reasoning methods for *sequential* programs are due to Floyd and Hoare [79, 102], who also popularised research in this area (see, e.g. [104]). In 1967, Floyd suggested to assign with each program location a predicate or, to put it another way, annotate it with an assertion, that implies the target property, and should be true whenever control reaches that location; Hoare later gave a formulation in a logical framework. The conceptual idea of both approaches is induction: one first shows that a target property is true initially, and second, if it is true at some point of computation, it is also true after the next step.

In 1975, Ashcroft [12] generalised Floyd's approach to *multi-threaded* programs with a fixed number of threads. Ashcroft realised that in the presence of multiple threads of execution, Turing's observation [169] that checks of assertions can be done independently in any order, no longer applies. As a counter measure, he proposed a more complex induction step that ensures a thread cannot invalidate predicates associated with other threads by executing an instruction. Technically, Ashcroft achieved this by assigning shared predicates* with *sets* of instruction labels rather than with the individual labels, as Floyd did.

Just one year later in 1976, Owicki and Gries [152], and independently Lamport [127], proposed to construct decompositions of such "global invariants" in terms of vectors with inductive single-thread predicates. The decomposition came at the price of a test for *interference freedom*, which additionally required proofs for each of the single-thread predicates under interference from other threads, and hence with respect to every other single-thread predicate in the vector. As such decompositions may not exist, Owicki and Gries introduced auxiliary shared variables as a workaround, while Lamport preferred to enrich the predicate language such

*Ashcroft did not consider programs with local variables other than the program counter.

that predicates could be formulated over local variables of all the threads. Lamport called such predicates "control predicates" as he considered—like Ashcroft, Owicki and Gries—programs with no local variables other than the program counter [152, 127, 128].

Both workarounds conflict with their initial objective of "thread modularity", and reveal the underlying machinery: Ashcroft's global invariant method. For this reason, Roever et al conclude in [52] that these methods can technically be seen as a "mere reformulation" of Ashcroft's method, yet their objective to exploit thread modularity prepared the ground for compositional approaches, which were pioneered by Jones, the inventor of *rely-guarantee* reasoning [115, 116]. Jones proposed a variant of Owicki and Gries' approach that achieved compositionality by associating with every (ideally) single-thread invariant in the vector a *rely*, and a *guarantee* condition, which specify the interference a thread can tolerate *from*, and imposes *on* other threads, respectively; one can think of such a condition as a *stub* that soundly substitutes yet-to-be-developed, or more complex real code. These condition pairs enable the decoupling of proof efforts, and hence compositional proofs.

Already before Floyd published his work on systematic methods for program verification, a somewhat distinct development took place in modelling, and analysing communication and synchronisation in concurrent systems. In 1962 Petri introduced a model to describe processes as concurrent and interacting machines, where he used multisets over a finite alphabet ("places") to represent system states [157, 156] (see also [67] and [52]). Later in 1969 Karp and Miller [120, 121] studied an equivalent model* with the emphasis on decision procedures, and proposed an algorithm to construct global inductive invariants for Petri nets, which is still frequently used including by our coverability oracle. Much later in 1994, Ciardo [39] extended Petri's model with "transfer arcs", which is the theoretical basis for the abstract domain model we introduced in this thesis. The added expressiveness had its price: Dufourd, Finkel and Schnoebelen [63] proved in 1998 that Karp and Miller's method could not be turned into a complete decision procedure for Ciardo's extension by showing undecidability of the place-boundedness problem. A major breakthrough, which solved this issue and made our widening approach possible, was the discovery of a complementary decision procedure by Abdulla et al [2] in 1996.

Even until the mid-90s the two communities developed rather in isolation, but

*The main difference is that Petri used multisets, whereas Karp and Miller preferred the equivalent representation in terms of Parikh vectors [153], which count the occurrences of elements.

since then have come closer. This change was particularly promoted by Finkel, and later by Abdulla, Cerans, Jonsson and Tsay [73, 2] who formulated a unifying framework. Today, both areas are linked by the theory of well quasi-ordered systems, which is based on a very simple, yet powerful concept, and that both Petri's model and Ciardo's extension belong to. The strong technical connection between Ashcroft's concurrency model and that of Petri was nonetheless ignored for a long time, despite the observation he made in [12]

> *" Since [local] states are defined as subsets of [instruction labels] we will consider a program "illegal" if it allows any [instruction to be reached by two threads simultaneously]. [...] In fact it would not be difficult to remove this restriction and allow [local] states to be* ***multisets*** *instead of* ***sets*** *of labels. "*

Ashcroft hence already knew about the advantages of using multisets (like Petri did) to represent program states, which means a shift from finite to possibly *infinite* threads of execution, and Karp and Miller had an algorithm to construct Ashcroft's global invariants even without the above restriction to finite thread numbers.* The clue, why such a proof generalises to arbitrary thread numbers is the monotonicity of Ashcroft's model. But this property became a subject of academic research interest only much later.

7.2 Comparison with Related Work

Algorithmic solutions for verifying safety properties of multi-threaded programs with possibly infinite data and threads has been intensively studied in the last years. We survey work related to concurrent program verification in general, coverability analysis in various kinds of well quasi-ordered systems, restoring monotonicity, and our dual-reference program model.

Concurrent program verification. Developers are offered a large choice of programming approaches when it comes to designing concurrent software. The most widespread paradigms can roughly be classfied on whether they permit communication through shared memory, or message passing. In shared memory programs (the focus of this thesis), communication is done directly through variables that are

*Technically, the shared predicates Ashcroft assigns to sets (or multisets) of instruction labels are given by (largest) nodes of the tree generated by Karp and Miller's algorithm.

Table 9: **Comparison with existing methods** — Input features: ∞, *CV*, *SM*, *ASSERT* = unbounded threads, condition variables, shared-memory, assertions; variable relationship in the generated proofs: *SP*, *ST*, *IT* = shared, single-thread, inter-thread; Output: *CEX*, *FP*, *TA*(n) = counterexamples, false positives, asymptotic run time for the ticket algorithm

	Input				Relationships			Output		
Verifier	∞	*CV*	*SM*	*ASSERT*	*SP*	*ST*	*IT*	*CEX*	*FP*	*TA*(n)
cream [95, 96]	○	○	●	●	●	●	●	●	○	exponential
symmpa [58]	○	○	●	●	●	●	○	●	○	exponential
iDFG (unimpl.) [71]	○	○	●	●	●	●	●	●	○	quadratic
duet [69]	●	○	●	●	●	○	○	○	●	false positive
boppo/satabs [47]	●	○	●	●	●	○	○	●	●	N/A
ddv/satabs [176]	○	○	●	●	●	○	○	●	○	N/A
blast [100]	○	○	●	○	N/A	N/A	N/A	●	○	N/A
magic [37]	○	○	○	●	N/A	N/A	N/A	●	○	N/A
monabs (this work)	●	●	●	●	●	●	●	●	○	constant

accessible by different threads of execution as e.g. in C, C++, C# and Java. On the other hand, in message passing programs processes communicate by sending and receiveing messages, e.g. signals and broadcast. Popular examples with direct language support are Ada, Erlang and Occam. The link to more academic process algebras is seamless, e.g. the Occam language developed by David May [138] builds upon the *CSP* process algebra, originally proposed by Tony Hoare [103]. Finally, in asynchronous programs tasks are interleaved in a single instruction stream.

Shared-memory programs. Existing approaches for verifying strictly asynchronous programs that communicate through shared memory, such as [100, 37, 47, 176, 40, 95, 58, 69, 71], ignore the monotone structure multi-threaded programs naturally exhibit. None of these methods shifts the exponential-space burden (in the number of threads) towards a problem on well-founded orderings. Table 9 provides a feature comparison with our work.

The approach in [95], implemented in cream, generates Owicki-Gries and rely-guarantee type proofs. In contrast to our work, it uses predicate abstraction in a CEGAR loop to generate environment invariants for fixed thread counts, whereas our approach directly checks the interleaved state space and exploits monotonicity. Whenever possible, cream generates thread-modular rely-guarantee proofs by prioritising predicates that do not refer to the local variables of other threads. For the parametric benchmarks we used in Section 6 this was never successful. A

CEGAR approach for symmetric concurrent programs has been implemented in symmpa [58]. It uses predicate abstraction to generate a Boolean Broadcast program (a special case of DR program), which is then checked with the symmetry-exploiting boom model checker [24, 25]. In contrast to our work, their technique generates non-monotone abstractions, which cannot be approached with well quasi-ordered systems technology to cope with unbounded thread counts. Moreover, their approach cannot reason about relationships between local variables of different threads, which are crucial for verifying well-known algorithms such as the ticket lock. For cream and symmpa we observe the exponential time curve for the ticket algorithm. We have compared monabs with both tools in Section 6.2.

Recent work on data flow graph representations of fixed-thread concurrent programs has been applied to safety property verification [71]. The inductive data flow graphs can serve as succinct correctness proofs for safety properties; for the ticket example they generate correctness proofs of size quadratic in n. Similar to [71], the technique in [69] uses data flow graphs to compute invariants of concurrent programs with unbounded threads (implemented in duet). In contrast to monabs which uses an expressive predicate language, proofs in duet are constructed from relationships between either solely shared, or solely local variables. The consequence is *un*provability of most benchmarks we used in Section 6: duet reports false positives on 15 of the 31 parametric benchmarks (including the ticket algorithm). In contrast, our approach is complete for strictly asynchronous programs, if every thread's memory is finite. Other model checkers with some support for concurrent software (finite thread numbers only) include blast, which cannot handle general assertions when concurrency is enabled [100], the tools ddv/satabs and boppo/satabs, which restrict predicates to track either shared or local relationships and therefore fail (like duet) for every second program used in Section 6, and magic [37], which does not support shared variables.

Although the previous works deal with multi-threaded programs in general, their implementations tackle subsets of C and C++. Since multi-threading features were added in the latest C and C++ standards [112, 113] their concurrency features are now comparable with those of C# and Java; e.g. all support mutexes, condition variables, thread creation and management, shared and thread-local storage, and atomic primitives—hence all those used by our benchmark programs. Moreover, with the language feature limitations of the above tools in mind, it is fair to say that converting C# or Java code with similar constructs to C or C++ (and vice versa) is straightforward. Approaches that directly operate on multi-threaded Java programs include the Java Pathfinder tool developed by the NASA Ames Research Cen-

ter [99], Bandera [98, 111], the spin model checker by Holzmann [105, 106, 107], and the component-based method from [134]. Another Owicki-Gries based approach that operates on SMV models is [46].

The previous approaches have little or no support for unbounded data structures on the heap. A promising approach in this area is concurrent separation logic [151], an adaption of separation logic [160] for dealing with heap-manipulating programs; separation logic is an extension of classical Hoare logic [102]. An advantage of separation logic is the possibility to succinctly specify manipulations on parts of a heap, which then permit to derive properties on the entire heap; Vafeiadis and Parkinson [172] proposed a variant that adapts ideas from rely-guarantee methods. The automation of concurrent separation logics is, however, still in its infancy [45]. In [55], Distefano et al present a memory safety checker for concurrent programs with singly linked lists (implemented in SmallfootRG), and a fine-grained concurrency verifier was proposed in [34]. In [171], Vafeiadis evaluates the Cave tool, an implementation of the extension from [172] that is able to verify memory safety of intricate heap-manipulating data stuctures such as Treiber's stack algorithm [168], lock-free queues by Michael and Scott, and Doherty et al [143, 56], and set algorithms [174]. None of these tools can handle general assertions, which preclude any useful comparison.

Message passing programs. Methods for checking Ada programs have also been proposed (see, e.g. [33, 68, 32, 64, 149, 165]). The major drawback of most of these methods is that they are incapable of dealing with the precise program semantics. For example the Quasar tool by Evangelista et al [68] extracts a synchronisation skeleton [41] from an Ada program, which is subsequently checked using Petri net techniques. Synchronization skeletons are program abstractions that suppress details that are irrelevant to synchronization. Adapted to our setting, this means to compute and check the predicate abstractions obtained by tracking, for every shared (Boolean) mutex variable `s`, a shared predicate $\mathtt{s} = \mathtt{T}$, and for every local condition flag `l` a local predicate $\mathtt{l} = \mathtt{BRC}$ (cp. Figure 14). Only 4 of the programs we used in the evaluation in Section 6 can be proved correct with such a simplistic approach.

Erlang is a declarative concurrent programming language, originally developed by Joe Armstrong [175, 11]. The challenges in Erlang verification are similar to those for shared-memory programs, such as infinite-data domains (though, no shared state is accessible), recursion, unbounded concurrent threads of execution, but also the communication over unbounded-buffer mailboxes makes them hard to

check. Several approaches to verify Erlang programs have been proposed. In [110], Fredlund and Svensson present the model checker mcerlang, which was greatly inspired by the spin tool by Holzmann [105]. A case-study that uses mcerlang to verify an implementation of the supervisor behaviour of Erlang/OT is described in [36]. Model checking-based approach are [108, 61]. In contrast to [108, 110], the technique in [61] can cope accurately with unbounded-buffer mailboxes and an unbounded number of concurrent threads. The idea is to encode Erlang program abstractions as so-called *Actor Communicating Systems*, which have natural Petri net encodings. Their approach has been realised in the soter tool [60], which uses our breach tool as back-end model checker.

Links between well quasi-ordered systems and other concurrency formalisms such as the π-calculus, or asynchronous programs, continuously evolve. The π-calculus [144, 161] is a Turing-powerful process algebra, where threads communicate via synchronous message exchange. In [142] Meyer et al present a polynomial-size translation of *finite control processes* [50], a π-calculus fragment, into *safe* Petri nets, i.e. nets where a place can contain at most one token. Safe Petri nets are finite-state, and equivalent to strictly asynchronous programs with finite thread data, where at most one thread can occupy a certain local state at a time (infinity norm of reachable states is always 1; cp. Section 4.2.2). Very recently, Meyer et al [109] proposed another reduction for a strictly larger π-calculus fragment, so-called *name-bounded processes*, which yields bisimilar *in*finite-state Petri nets of non-primitive recursive size in the worst case. Finally, a translation of *asynchronous* π-calculus to *transfer* Petri nets, or equivalently monotone Boolean DR programs, is presented in [8]. Translations of CSP into Petri nets have also been proposed, e.g. in [122, 132].

Asynchronous programs. In asynchronous programs, threads interleave in a single instruction stream and run until they *explicitly* relinquish control, e.g. by calling a designated function like `yield`, or when their program stack becomes empty. Hence, in contrast our model, threads are not suspended nondeterministically. For certain *finite-data* asynchronous programs with unbounded program stack and unbounded task buffers, Ganty and Majumdar [84] recently proved the verification problem to be complete in exponential space by providing a polynomial-time reduction to Petri Nets (and vice versa). A reduction for a more general class was proposed in [124]. Several earlier approaches to programs with unordered message buffers are [148, 147, 114].

Coverability analysis. Algorithmic solutions to coverability analysis were first proposed for *vector addition systems* in a landmark paper by Karp and Miller [121] (implemented in tina [26]). The solution constructs a pseudo-reachability tree by forward exploration and replaces newly discovered states that are strictly greater than predecessors using an infinity measure. The approach has non-primitive recursive worst-case complexity [158]. Due to the undecidability of the place-boundedness problem [63], extensions to broadcast operations are inevitably incomplete. An example is the Covering Graph procedure from [65], which was shown to fail to terminate on certain systems [66]. An improvement of the Karp-Miller procedure that computes minimal coverability sets is [89] (implemented in csc), and the Karp and Miller algorithm with pruning from [159] (not available online). An approach that constructs compact, yet not necessarily minimal coverability sets is [173]. Their algorithm prioritises unexplored states with larger thread numbers to speed up convergence (not available online). We experimented with various selection heuristics for the Karp-Miller like coverability oracle. In our setup, treating states with *smallest* thread numbers performed best by far. One explanation is that dealing with such states is algorithmically easier, and that our widening algorithm does not benefit from large states being reported by the oracle. In our experiments the largest thread count that appeared in a standard uncoverability proof is 6.

To afford more flexibility in modelling parameterized programs, various algorithms were later proposed for well quasi-ordered systems, originally in a pure backward fashion [2] (implemented in istbc and petruchio [141]), later as forward exploration [74, 177], or as a backward and forward unfolding algorithm [6]. The paradigm presented in [88] is also a pure forward algorithm; it constructs abstractions of increasing precision (implemented in eec). The recent approach from [5] is based on cutoffs and exploits the fact that analysing a small number of threads is often sufficient to expose errors (not available online). It performs parameterized verification by inspecting a small set of instances of a system to show correctness. Such an approach has two drawbacks. First, it is likely to fail when systems require the inspection of large thread numbers. Second, performance degrades when many transitions do not affect the correctness of a property, such as those induced by detached processes that operated on a disjoint set of shared variables. As an example, their approach fails for the Kanban system from [82], whereas Karp-Miller like approaches (including our coverability oracle) report the error almost instantaneously.

Solutions combining forward and backward exploration are rare; we are only aware of the methods described in [75] and [86], and the very recent approach from [123]. The authors of [75] propose to use a csc-like approach to compute

overapproximations of the coverability set, which are then used in a *subsequent* backward exploration to prune the search space. Our experimental results reported in Section 6.3 demonstrate, however, that this approach cannot cope with programs of the sizes we consider, simply because their computation is too expensive. In [86], the authors combine overapproximations computed in a forward fashion, which are refined by using backward underapproximations (implemented in tsi). On an abstract level, our algorithm can be seen as the dual of this approach. Interestingly, performance-wise tsi cannot benefit from this similarity; tsi performed worst in our experimental comparison. Finally, the algorithm in [123] (implemented in iic) computes an inductive invariant by maintaining a list of overapproximations of forward-reachable states, and strengthening them (in a backward manner) using counterexamples to inductiveness. Note that our uncoverability proofs introduced in Definition 24 are inductive proofs of the backward-nonreachability of the initial states from e.

Other work that, like ours, takes parameterized system level software as input includes earlier work on multi-threaded Java programs [53], which in fact uses a set of communication primitives and derived semantics very similar to ours, and rewrites them into *multi-transfer Petri nets* using a form of counter abstraction. Our earlier cutoff based approach [117] combines *finite-state* forward exploration with infinite-state backward exploration. Recent work [69, 71] over data flow graph representations of parameterized concurrent programs has been applied to safety property verification. These methods do not explore, in a model checking fashion, replicated finite-state procedures, but instead aim to find (possibly inductive) program invariants. We have compared with petruchio, eec, csc, tina, istbc, tsi and iic in Section 6.3.

Dual-reference programs. *Boolean programs* [18] are a popular abstract domain for model checking *sequential* software, as pioneered by the slam project at Microsoft [19]. Various adaptions to the multi-threaded case have been proposed. Geeraerts and Van Begin [87] introduce a formal language called *Concurrent Boolean Programs* to describe predicate abstractions of concurrent programs with atomic sections and broadcasts communication. In contrast to dual-reference programs, which capture the essence of such primitives in a "relational" way through active- and passive-thread variable updates, their model features specific primitives on designated lock, message and thread-type variables. A very similar but slightly less expressive model that lacks support for broadcasts is proposed in [22, 47], and

used as input format for the getafix, boom and duet model checkers [167, 23, 69].

All these attempts to adapt Boolean programs for proving multi-threaded programs share the same disadvantage: they cannot be used to verify elementary safety properties requiring single-, or inter-thread predicates, and are therefore inadequate as an abstract domain for model checking concurrent software. They are, e.g. insufficient for 16 out of the 31 parametric benchmarks we used in Section 6.

On the other hand, there exist only few works that deal with data-symbolic encodings of Petri net like formalisms (see, e.g. [28] for the relationship between Petri nets and Boolean programs). For so-called *safe* Petri nets, i.e. nets where at most one token can reside in each place, an approach is described in [150]. Adapted to our setting, a restriction to safe Petri net means that every local state is only mutually exclusively accessible, and hence only the 6 *non*-parametric programs we used in Section 6 can be encoded. Moreover, the authors encode each place using a single Boolean variable, whereas we encode each place by a unique *valuation* of the Boolean variables.

Restoring monotonicity. Our solution to restore monotonicity in predicate abstractions is closely related to the reductions presented in [27, 3, 4, 136, 91], which *enforce* monotonicity in certain protocols with conjunctive guards that are not well quasi-ordered. Bingham and Hu deal with "global conditions", i.e. guards that require universal quantification over (thread) indices, by transforming such systems into Broadcast protocols [65]. This is achieved by replacing conjunctively guarded actions by transitions that, instead of checking a universal condition, execute it assuming that any thread not satisfying it "resigns" by entering (through a broadcast) a designated local state that isolates it from participation in future computation steps. The same idea was further developed by Abdulla et al. in the context of *monotonic abstractions* (see, e.g. [3, 4]); an application to array-based systems is [91].

All these approaches can be seen as adaptations of the *stopping failures model*, which assumes that a thread can suddenly halt forever (originally intended to study the consensus problem in the presence of unpredictable processor crashes) [136]. Despite technical similarities, our approach differs fundamentally: Instead of brute-forcing monotonicity in genuinely non-monotone systems (and putting up with the ensuing overapproximation), we *restore* the monotonicity that was originally present in the input program, and are able to avoid introducing spurious errors.

8 Conclusion

Although model checking was originally designed for analysing *concurrent* systems, there has been little evidence of fruitful applications of predicate abstraction to realistic shared-variable concurrent software written in mainstream languages such as C. The techniques presented in this thesis represent an initial step towards a continuation of the success-story of predicate abstraction in sequential software.

We have introduced a novel algorithmic solution for verifying non-recursive shared-memory programs executed by an unbounded number of threads, which synchronise via higher-level mechanisms, such as mutexes, broadcasts and conditional waits. System code, including UNIX and Mac OS device drivers, make frequent use of such concurrency APIs, whose correct use is therefore critical to ensure a reliable programming environment. Our method succeeds in putting the well-established predicate abstraction technique to work by exploiting the characteristic symmetry and monotonicity exhibited by these programs. We defined a new class of concurrent programs called *dual-reference programs*, which we use to represent abstractions of the shared-memory programs we are concerned with. In dual-reference programs, transitions of an active thread and its environment are specified as actions over the executing active and a generic passive thread. We presented the relevant concepts, a formalisation, and an implementation of a *CEGAR strategy* applicable to strictly asynchronous programs run by arbitrarily many threads. Our predicate abstraction strategy supports an adequately expressive predicate language as is needed for verifying intricate properties such as mutual exclusion in the ticket busy-wait lock algorithm. Predicate-abstracting such programs results in Boolean dual-reference programs, for which we proved—despite the finite variable domain—*undecidability* of even the simplest program reachability problems. We have overcome this problem by defining a closure operator that *restores monotonicity* (and thus decidability).

Checking safety in monotone Boolean dual-reference programs incurs a high computational cost; it is Ackermann-complete. To alleviate this rise in complexity, we have proposed a solution to the equivalent coverability problem in well quasi-ordered systems that turns into account that (as we demonstrated empirically) infeasible thread constellations tend to exhibit *compact* explanations of infeasibility in terms of constellations with *fewer* threads. To this end, our algorithm combines forward propagation of *under*approximations with backward propagation of *over*approximations (widenings). Namely, it identifies and compactly represents the uncoverable elements backward-reachable from a given query target, by widening

the target set by smallest-possible elements whose backward expansion can be expected to terminate quickly. The imposition to first settle the coverability of these elements, side-stepping the original query, not only accelerates the search, but also makes the search structure more compact.

We have implemented our techniques in the monabs program verifier, and the widening algorithm in our breach tool. This allowed us to experimentally validate the superiority (already for trivial thread numbers) of our unbounded thread analysis over several existing fixed-thread verifiers on a large set of realistic shared-memory C programs. Moreover, we demonstrated experimentally that our widening algorithm, as implemented in breach, outperforms the best known coverability approach by orders of magnitude, enabling the verification of programs currently well beyond the limits of existing tools. While the presented specific implementation of our widening ideas in breach has proven to be very successful and efficient in solving real verification problems, the purpose of this thesis is also to propose our search organisation and the cooperation between the backward and forward components of the algorithm as a new general paradigm for tackling non-recursive shared-memory programs executed by an unbounded number of threads.

We conclude by discussing several issues left open by this dissertation, and outline possible approaches to enhance the practical potential and relevance of our approach.

Open issues and future work. Programs with broadcast sychronisation such as transfer Petri nets have been intensively studied in recent years from a theoretical viewpoint. Practical solutions for tackling safety of such programs remain, however, rare. We presume the main reason is their rather limited practical importance: compared with, e.g. mutexes and compare-and-swap operations they still represent a rather "exotic" thread coordination method. Our verification approach sheds new light on such programs, namely by promoting them as an abstract domain for model checking mainstream, *broadcast-free* concurrent programs. It would be interesting to study the performance of forward-directed methods that combine Karp-Miller like search with the acceleration technique from [65] in such a setting; although the basic technique has been around for more than a decade, we are not aware of any implementation. Can these accelerations be computed efficiently? Does the incompleteness matter in practice, or does it occur "just" in theory? If the answer to the former question is yes, such a technique would also be an ideal candidate to speed up our widening approach.

We have—for the purpose of this thesis' objectives—used predicate abstraction as the basis. Our approach can, however, just as well be applied on top of other relational abstract domains, such as the polyhedral abstract domain by Cousot and Halbwachs [49], or the octagon domain by Miné [145]. In such a setting, relational abstract domains would track constraints between shared (global-scope) and local variables of a given thread, or local variables of a thread and all local variables of other threads. While the adaption of such domains is rather straightforward, the question whether the resulting methods are compatible with monotonicity needs clarification. Also unclear is whether one can come up with practical examples of strictly asynchronous programs that require even more expressive predicates to be proved correct than the ones we permit. If answered in the affirmative, a natural extension of our approch would be to permit predicates to reference more than two generic threads of execution. As an example, predicates like $\mathtt{l} = \mathtt{l}_T \neq \mathtt{l}_P$ could be used to track for a given thread's local variable `l` whether there exists *exactly one* other thread with the same value, while all other threads have a different value for `l`.

A further aspect regards the encoding of program states in our breach and monabs tools. In order to check Boolean DR programs, we reverted to an explicit format that enumerates individual transitions. Devising coverability algorithms that directly operate on our data-symbolic (Boolean) encoding will significantly improve the scalability of our approach when many predicates are involved. We are, however, not aware of any coverability checker, or even a data structure for handling upward closed sets that fits our demands. A possible starting point would be to adapt BDD encodings similar to that proposed in [24], which uses propositional formulas to symbolically represent the shared state, and the local states of the respective threads. Significant work remains to be done in order to adapt such techniques to the coverability problem, such as to find effective and efficient mechanisms for handling data-symbolic representations of upward closed sets, and exploiting local computation paths through partial-order reduction strategies in the presence of broadcasts.

We have assumed a very strict and unrealistic memory model that guarantees atomicity at the statement level, although multiprocessors such as Intel x86 and ARM implement more relaxed models that feature techniques like instruction reordering, and store buffering to boost performance. One can work soundly with our assumption by pre-processing input programs such that the shared state is accessed only via word-length reads and writes, which guarantees that all computation is performed using only local variables. To obtain a more accurate analysis one could, however, also directly embed weak memory semantics into the program, or alter-

natively encode the induced program executions via partial orders. Extending our approach to such models would be a natural continuation of this thesis' research. It would be interesting to see, whether modelling the exact memory-level semantics preserves the scalability of our approach.

In Section 4, we mentioned that smaller states have smaller cover predecessors, and hence induce backward-reachability trees with smaller *height*. While our experimental results confirm that for the strictly asynchronous and monotone Boolean DR programs we are concerned with, a decrease in height goes hand in hand with faster overall convergence, we do not know if there exist programs where the opposite is true. In particular, for some well quasi-ordered systems such as Petri nets this is not the case.* We conjecture, however, that for monotone Boolean DR programs smaller states in general yield *smaller* backward reachabililty trees, and hence widening a backward-directed search by uncoverable states that are strictly smaller than cover predecessors can at most speed up convergence.

The classical backward coverability algorithm due to Abdulla et al. [2] is well known to operate in Ackermannian time for our monotone Boolean DR programs, or equivalently, transfer Petri nets (see, e.g. [162] for a recent summary). Moreover, for (plain) Petri nets recent results prove certain breadth-first backward methods to operate in 2ExpTime [29]. It is unclear whether it is possible to come up with a *depth*-first search of similar complexity. Recent developments in the complexity analysis of well quasi-ordered system models and algorithms such as [38, 164, 72] use bounds on the length of increasing-pair-free sequences that do not warrant such a conclusion. Interestingly, our experience has shown that breadth-first methods are tremendously faster in practice.

In order to construct minimal uncoverability proofs, we exploit the fact that the downward closure of a finite set is finite, a property which holds for many well quasi-ordered systems including Petri nets, Broadcast protocols, Lossy counter machines and our monotone Boolean DR programs. We do not know, however, whether minimal proofs are effectively computable for the entire class of well quasi-ordered systems. An example, where our algorithm fails to construct such a proof is the following (taken from [123]): the set of states is $\mathbb{N} \cup \{\omega\}$, the transitions are $v \rightarrowtail v'$ if either $v' = v + 1$, or $v = v' = \omega$, the ordering is $0 < 1 < \ldots < \omega$,

*Consider the 2-place Petri net with one transition that consumes a token from place two (no production), and another that consumes two tokens from the second place, and produces two in the first and one in the second. In this case, we get for $x = (\,|\,0, 2)$, $y = (\,|\,1, 1)$ and $z = (\,|\,1, 0)$ cover predecessors $\mathsf{C\text{-}Pre}(\{x, y\}) = \{x\}$ and $\mathsf{C\text{-}Pre}(\{x, z\}) = \{x, y\}$; observe $z < y$. (Many thanks to Stefan Kiefer for this example.)

and the initial and target states are 0 and ω, respectively. The reason why our approach fails is that there exists an infinite number of *coverable* widening candidates for target state ω, namely every natural number (the downward closure of ω is not finite). Interestingly, for the same reason the incremental approach from [123] does not terminate on this example.

A recent and very promising verification method for analysing sequential circuits is IC3 due to Aaron Bradley [30]. Much like our approach, IC3 incrementally overapproximates the state space and, if an overapproximation is not inductive, strengthens it to inductiveness. By replacing our notion of the "covers" relation by implication (a formula f "covers" formula g if $f \Rightarrow g$ is valid), our widening approach can directly be applied to sequential circuits. In this setting our approach seems to operate much like the IC3 procedure. Finally, [71] shows that their notion of inductive data flow graphs in the context of concurrent programs can serve as succinct correctness proofs for safety properties, much like the minimal uncoverability proofs our algorithm tries to find. We will leave the precise relationship between these recent works and ours for future investigation.

Bibliography

[1] Parosh Aziz Abdulla. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic*, 2010. 3, 30, 32, 39, 45, 47, 50, 86

[2] Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems of infinite-state systems. In *Logic in Computer Science*, 1996. 3, 11, 12, 32, 45, 47, 60, 86, 92, 93, 98, 104

[3] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Parameterized verification of infinite-state processes with global conditions. In *Computer Aided Verification*, 2007. 4, 100

[4] Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. Monotonic abstraction in parameterized verification. *Electronic Notes in Theoretical Computer Science*, 2008. 4, 35, 100

[5] Parosh Aziz Abdulla, Frédéric Haziza, and Lukás Holík. All for the price of few. In *Verification, Model Checking and Abstract Interpretation*, 2013. 98

[6] Parosh Aziz Abdulla, S. Purushothaman Iyer, and Aletta Nylén. SAT-solving the coverability problem for Petri nets. In *Formal Methods in System Design*, 2004. 98

[7] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, CA, USA, 2002. 81, 82

[8] Roberto M. Amadio and Charles Meyssonnier. On decidability of the control reachability problem in the asynchronous *pi*-calculus. *Nordic Journal of Computing*, 2002. 97

[9] Amino Concurrent Building Blocks. amino-cbbs.sourceforge.net. 81, 82

[10] Gregory R. Andrews. *Concurrent programming: principles and practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991. 23

[11] Joe Armstrong. Erlang. *Communications of the ACM*, 2010. 96

[12] Edward A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 1975. 91, 93

[13] James Aspnes, Faith Ellen Fich, and Eric Ruppert. Relationships between broadcast and shared memory in reliable anonymous distributed systems. *Distributed Computing*, 2006. 14

[14] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004. 14

[15] Felice Balarin and Alberto L. Sangiovanni-Vincentelli. An iterative approach to language containment. In *Computer Aided Verification*, 1993. 69

[16] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *European professional society in Systems*, 2006. 69

[17] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation*, 2001. viii, 17

[18] Thomas Ball and Sriram Rajamani. Bebop: A symbolic model checker for Boolean programs. In *Model Checking and Software Verification*, 2000. 99

[19] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. In *Computer Aided Verification*, 2001. 99

[20] Tom Ball, Andreas Podelski, and Sriram Rajamani. Boolean and Cartesian abstractions for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2001. 70, 74

[21] D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey. The main features of CPL. *Computer Journal*, 1963. 12

[22] Gérard Basler. *Model Checking Boolean Programs*. PhD thesis, ETH Zurich, 2010. 99

[23] Gérard Basler, Matthew Hague, Daniel Kroening, Luke Ong, Thomas Wahl, and Haoxian Zhao. Boom: Taking Boolean program model checking one step further. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2010. 99

[24] Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Symbolic counter abstraction for concurrent software. In *Computer Aided Verification*, 2009. 65, 95, 103

[25] Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. Context-aware counter abstraction. *Formal Methods in System Design*, 2010. 95

[26] Bernard Berthomieu, François Vernadat, Pierre-Olivier Ribet, and Florent Peres. The Tina tool (version 3.0). projects.laas.fr/tina. 86, 98

[27] Jesse D. Bingham and Alan J. Hu. Empirically efficient verification for a class of infinite-state systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2005. 4, 100

[28] Ahmed Bouajjani and Javier Esparza. Rewriting models of Boolean programs. In *Term Rewriting and Applications*, 2006. 100

[29] Laura Bozzelli and Pierre Ganty. Complexity analysis of the backward coverability algorithm for VASS. In *Workshop on Reachability Problems*, 2011. 104

[30] Aaron R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking and Abstract Interpretation*, 2011. 105

[31] Hervé Brönnimann. *Algorithms for permutations and combinations, with and without repetitions*. Polytechnic University and Bloomberg L.P., 2011. Implementation by Howard Hinnant. 79

[32] Eric Bruneton and Jean-François Pradat-Peyre. Automatic verification of concurrent Ada programs. In *Reliable Software Technologies*, 1999. 96

[33] Alan Burns, Andy J. Wellings, Albert M. Koelmans, Maciej Koutny, Alexander Romanovsky, and Alex V. Yakovlev. On developing and verifying design abstractions for reliable concurrent programming in Ada. In *International Real-Time Ada Workshop*, 2001. 96

[34] Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *International Static Analysis Symposium*, 2007. 96

[35] Edward W. Cardoza, Richard J. Lipton, and Albert R. Meyer. Exponential space complete problems for Petri nets and commutative semigroups: Preliminary report. In *Symposium on Theory of Computing*, 1976. 5

[36] David Castro, Victor Gulias, Clara Earle, Lars íke Fredlund, and Samuel Rivas. A case study on verifying a supervisor component using McErlang. *Electronic Notes in Theoretical Computer Science*, 2011. 97

[37] Sagar Chaki, Edmund M. Clarke, Nicholas Kidd, Thomas Reps, and Tayssir Touili. Verifying concurrent message-passing C programs with recursive calls. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2006. 6, 94, 95

[38] Pierre Chambart and Ph. Schnoebelen. The ordinal recursive complexity of lossy channel systems. In *Logic in Computer Science*, 2008. 104

[39] Gianfranco Ciardo. Petri nets with marking-dependent arc cardinality: Properties and analysis. In *Application and Theory of Petri Nets*, 1994. 40, 41, 45, 92

[40] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Verifying SystemC: a software model checking approach. In *Formal Methods in Computer-Aided Design*, 2010. 6, 94

[41] Edmund M. Clarke and Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, 1981. 96

[42] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 1994. 16, 17, 22, 23

[43] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2005. 6

[44] Edmund M. Clarke, Muralidhar Talupur, and Helmut Veith. Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008. 84

[45] Ariel Cohen and Kedar S. Namjoshi. Local proofs for global safety properties. In *Computer Aided Verification*, 2007. 96

[46] Ariel Cohen, Kedar S. Namjoshi, and Yaniv Sa'ar. Split: A compositional LTL verifier. In *Computer Aided Verification*, 2010. 96

[47] Byron Cook, Daniel Kroening, and Natasha Sharygina. Verification of Boolean programs with unbounded thread creation. *Theoretical Computer Science*, 2007. 6, 94, 99

[48] Jonathan Corbet. Ticket spinlocks. *Linux Weekly News (online)*, 2008. 23

[49] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages*, 1978. 103

[50] Mads Dam. Model checking mobile processes. *Information and Computation*, 1996. 97

[51] Satyaki Das and David L. Dill. Successive approximation of abstract transition relations. In *Logic in Computer Science*, 2001. 69

[52] Willem P. de Roever, Frank S. de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001. 92

[53] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Towards the automated verification of multithreaded Java programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2002. 43, 99

[54] Leonard E. Dickson. Finiteness of the odd perfect and primitive abundant numbers with distinct factors. *American Journal of Mathematics*, 1913. 9

[55] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2006. 96

[56] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *Formal Methods for Networked and Distributed Systems*, 2004. 96

[57] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. The SymmPa tool (integrated with SatAbs 3.1). www.cprover.org/SymmPa. 84

[58] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 2012. 5, 6, 19, 21, 25, 29, 39, 82, 84, 94

[59] Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *Computer Aided Verification*, 2011. 5, 19

[60] Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Soter: an automatic safety verifier for Erlang. In *Workshop on Programming based on Actors, Agents, and Decentralized Control*, 2012. 97

[61] Emanuele D'Osualdo, Jonathan Kochems, and Luke Ong. Automatic verification of Erlang-style concurrency. In *International Static Analysis Symposium*, 2013. 97

[62] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner, and Heike Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2010. 29, 39

[63] Catherine Dufourd, Alain Finkel, and Philippe Schnoebelen. Reset nets between decidability and undecidability. In *Colloquium on Automata, Languages and Programming*, 1998. 40, 60, 92, 98

[64] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Using state space reduction methods for deadlock analysis in Ada tasking. In *International Symposium on Software Testing and Analysis*, 1993. 96

[65] Allen Emerson and Kedar Kedar Namjoshi. On model checking for nondeterministic infinite-state systems. In *Logic in Computer Science*, 1998. 45, 98, 100, 102

[66] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *Logic in Computer Science*, 1999. 98

[67] Javier Esparza and Mogens Nielsen. Decidability issues for Petri nets—a survey. *Bulletin of the European Association for Theoretical Computer Science*, 1994. 92

[68] Sami Evangelista, Claude Kaiser, Jean-François Pradat-Peyre, and Pierre Rousseau. Quasar: A new tool for concurrent Ada programs analysis. In *Ada-Europe*, 2003. 96

[69] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Principles of Programming Languages*, 2012. 6, 29, 94, 95, 99

[70] Azadeh Farzan and Zachary Kincaid. Duet: static analysis for unbounded parallelism. In *Computer Aided Verification*, 2013. 21

[71] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *Principles of Programming Languages*, 2013. 6, 39, 81, 82, 94, 95, 99, 105

[72] Diego Figueira, Santiago Figueira, Sylvain Schmitz, and Philippe Schnoebelen. Ackermann and primitive-recursive bounds with Dickson's lemma. In *Logic in Computer Science*, 2011. 61, 62, 104

[73] Alain Finkel. A generalization of the procedure of Karp and Miller to well structured transition systems. In *Colloquium on Automata, Languages and Programming*, 1987. 93

[74] Alain Finkel and Jean Goubault-Larrecq. Forward analysis for WSTS, Part II: Complete WSTS. In *Colloquium on Automata, Languages and Programming*, 2009. 98

[75] Alain Finkel, Jean-François Raskin, Mathias Samuelides, and Laurent Van Begin. Monotonic extensions of Petri nets: Forward and backward search revisited. *Electronic Notes in Theoretical Computer Science*, 2002. 5, 87, 98

[76] Alain Finkel and Philip Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 2001. 32

[77] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *European Symposium on Programming*, 2002. 82

[78] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *Model Checking and Software Verification*, 2003. 82

[79] Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 1967. 91

[80] Harry Foster. Assertion-based verification: Industry myths to realities (invited tutorial). In *Computer Aided Verification*, 2008. 71

[81] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 2000. 76

[82] Pierre Ganty, Laurent Van Begin, and Anthony Piron. The MIST2 tool (version 0.1). github.com/pierreganty/mist. 98

[83] Pierre Ganty, Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Méthodes algorithmiques pour l'analyse des réseaux de Petri. *Techniques et Sciences Informatiques*, 2009. 86

[84] Pierre Ganty and Rupak Majumdar. Algorithmic verification of asynchronous programs. *ACM Transactions on Programming Languages and Systems*, 2012. 97

[85] Pierre Ganty, Cédric Meuter, Giorgio Delzanno, Gabriel Kalyon, Jean-François Raskin, and Laurent Van Begin. Symbolic data structure for sets of k-uples. Technical report, Université Libre de Bruxelles, 2007. 86

[86] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. A complete abstract interpretation framework for coverability properties of WSTS. In *Verification, Model Checking and Abstract Interpretation*, 2006. 5, 98, 99

[87] Gilles Geeraerts and Laurent Van Begin. Concurrent Boolean Programs. Technical report, ULB, 2003. 99

[88] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, enlarge and check: New algorithms for the coverability problem of WSTS. In *Journal of Computer and System Sciences*, 2006. 5, 86, 98

[89] Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set for Petri nets. In *Automated Technology for Verification and Analysis*, 2007. 86, 98

[90] Steven German and Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 1992. 30

[91] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards SMT model checking of array-based systems. In *International Joint Conference on Automated Reasoning*, 2008. 4, 100

[92] Herman H. Goldstine and John von Neumann. Planning and coding problems for an electronic computer. In *Collected Works of John von Neumann*. 1963. 91

[93] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification*. 1997. 16, 22

[94] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. The Cream tool. www.model.in.tum.de/~popeea/research/threader.html. 84

[95] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *Principles of Programming Languages*, 2011. 6, 39, 81, 82, 83, 84, 94

[96] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *Computer Aided Verification*, 2011. 81, 82, 84, 94

[97] Michael Hack. *Decidability Questions for Petri Nets*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1975. 45

[98] John Hatcliff and Matthew B. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In *Concurrency Theory*, 2001. 95

[99] Klaus Havelund. Java PathFinder, a translator from Java to Promela. In *Model Checking and Software Verification*, 1999. 95

[100] Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Programming Language Design and Implementation*, 2004. 6, 94, 95

[101] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 1952. 9

[102] Charles A. R. Hoare. *Communications of the ACM*, 1969. 91, 96

[103] Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 1978. 94

[104] Charles A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 2003. 91

[105] Gerard Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. 95, 97

[106] Gerard Holzmann. The model checker SPIN. *Transactions of Software Engineering*, 1997. 95

[107] Gerard Holzmann. *The Spin model checker: Primer and reference manual*. Addison-Wesley Professional, 2003. 95

[108] Frank Huch. Verification of Erlang programs using abstract interpretation and model checking. In *International Conference on Functional Programming*, 1999. 97

[109] Reiner Hüchting, Rupak Majumdar, and Roland Meyer. A theory of name boundedness. In *Concurrency Theory*, 2013. 97

[110] Lars íke Fredlund and Hans Svensson. McErlang: a model checker for a distributed functional programming language. *International Conference on Functional Programming*, 2007. 96, 97

[111] Radu Iosif, Matthew B. Dwyer, and John Hatcliff. Translating Java for multiple model checkers: The Bandera back-end. *Formal Methods in System Design*, 2005. 95

[112] ISO. *ISO/IEC 9899:2011 Information technology—Programming languages—C*. International Organization for Standardization, Geneva, Switzerland, 2011. 95

[113] ISO. *ISO/IEC 14882:2011 Information technology—Programming languages—C++*. International Organization for Standardization, Geneva, Switzerland, 2012. 79, 95

[114] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. In *Principles of Programming Languages*, 2007. 97

[115] Cliff B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, 1981. 92

[116] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 1983. 92

[117] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *Computer Aided Verification*, 2010. 5, 45, 99

[118] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Efficient coverability analysis by proof minimization. In *Concurrency Theory*, 2012. 5, 45, 82

[119] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. A widening approach to multi-threaded program verification. *ACM Transactions on Programming Languages and Systems*, 2013. 5, 45

[120] Richard M. Karp and Raymond E. Miller. Parallel program schemata: A mathematical model for parallel computation. In *Symposium on Switching and Automata Theory*, 1967. 92

[121] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 1969. 40, 45, 92, 97

[122] Krishna Kavi and Bill Buckles. Formal methods for the specification and analysis of concurrent systems. In *International Conference on Parallel Processing*, 1993. 97

[123] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. In *Computer Aided Verification*, 2013. 86, 98, 99, 104, 105

[124] Jonathan Kochems and Luke Ong. Safety verification of asynchronous pushdown systems with shaped stacks. In *Concurrency Theory*, 2013. 97

[125] Joseph B. Kruskal. The theory of well-quasi-ordering: A frequently discovered concept. *Journal of Combinatorial Theory, Series A*, 1972. 9

[126] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994. 69

[127] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 1977. 91, 92

[128] Leslie Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 1988. 92

[129] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 1980. 42

[130] Michael Leuschel and Helko Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In *Transactions on Computational Logic*, 2000. 60

[131] FreeBSD/Linux Kernel Cross Reference. svn.freebsd.org. 81, 82

[132] Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. Generating a Petri net from a CSP specification: A semantics-based method. *Advances in Engineering Software*, 2012. 97

[133] M. Löb and S. Wainer. Hierarchies of number theoretic functions, I. In *Archive for Mathematical Logic*, 1970. 61

[134] Brad Long, Paul A. Strooper, and Luke Wildman. A method for verifying concurrent Java components based on an analysis of concurrency failures. *Concurrency and Computation: Practice and Experience*, 2007. 96

[135] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages and Operating Systems*, 2008. 82

[136] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996. 4, 100

[137] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. 2006. 72

[138] David May. Occam. *ACM SIGPLAN Notices*, 1983. 94

[139] Richard Mayr. Undecidable problems in unreliable computations. In *Latin American Symposium on Theoretical Informatics*, 2000. 45

[140] John Mellor-Crummey and Michael Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *Transactions on Computer Systems*, 1991. viii, 73, 81, 82

[141] Richard Meyer and Tim Strazny. Petruchio: From dynamic networks to nets. In *Computer Aided Verification*, 2010. 86, 98

[142] Roland Meyer, Victor Khomenko, and Reiner Hüchting. A polynomial translation of *pi*-calculus (fcp) to safe petri nets. In *Concurrency Theory*, 2012. 97

[143] Maged Michael and Michael Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Principles of Distributed Computing*, 1996. 96

[144] Robin Milner. *Communicating and mobile systems—the pi-calculus*. Cambridge University Press, 1999. 97

[145] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006. 103

[146] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967. 30

[147] Madhavan Mukund, Narayan Kumar, Jaikumar Radhakrishnan, and Milind Sohoni. Robust asynchronous protocols are finite-state. In *Colloquium on Automata, Languages and Programming*, 1998. 97

[148] Madhavan Mukund, Narayan Kumar, Jaikumar Radhakrishnan, and Milind Sohoni. Towards a characterisation of finite-state message-passing systems. In *Asian Computing Science Conference*, 1998. 97

[149] Tadao Murata, Boris Shenker, and Sol M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Transactions on Software Engineering*, 1989. 96

[150] Shougo Ogata, Tatsuhiro Tsuchiya, and Tohru Kikuno. SAT-based verification of safe Petri nets. In *Automated Technology for Verification and Analysis*, 2004. 100

[151] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 2007. 96

[152] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 1976. 91, 92

[153] Rohit J. Parikh. On context-free languages. *Journal of the ACM*, 1966. 15, 92

[154] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005. viii, 4, 81, 82

[155] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 1981. 81, 82

[156] Carl A. Petri. Fundamentals of a theory of asynchronous information flow. In *International Federation for Information Processing*, 1962. 92

[157] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institute for Instrumental Mathematics, Bonn, 1962. 45, 92

[158] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 1978. 98

[159] Pierre-Alain Reynier and Frédéric Servais. Minimal coverability set for Petri nets: Karp and Miller algorithm with pruning. In *Petri Nets*, 2011. 98

[160] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, 2002. 96

[161] Davide Sangiorgi and David Walker. *The pi-Calculus—A Theory of Mobile Processes*. Cambridge University Press, 2001. 97

[162] Sylvain Schmitz and Philippe Schnoebelen. The power of well-structured systems. In *Concurrency Theory*, 2013. 62, 104

[163] Philippe Schnoebelen. Lossy counter machines decidability cheat sheet. In *Workshop on Reachability Problems*, 2010. 30

[164] Philippe Schnoebelen. Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In *Mathematical Foundations of Computer Science*, 2010. 5, 60, 61, 62, 104

[165] Sol M. Shatz, Shengru Tu, Tadao Murata, and Sastry Duri. An application of Petri net reduction for Ada tasking deadlock analysis. *IEEE Transactions on Parallel and Distributed Systems*, 1996. 96

[166] Boleslaw K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *International Conference on Supercomputing*, 1988. 81, 82

[167] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *Computer Aided Verification*, 2010. 29, 99

[168] R. K. Treiber. Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center, 1986. 96

[169] Alan Turing. On checking a large routine. In *Report of a conference on high-speed automatic calculating machines*, 1949. 91

[170] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1936. 70, 91

[171] Viktor Vafeiadis. RGSep action inference. In *Verification, Model Checking and Abstract Interpretation*, 2010. 96

[172] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *Concurrency Theory*, 2007. 96

[173] Antti Valmari and Henri Hansen. Old and new algorithms for minimal coverability sets. In *Petri Nets*, 2012. 98

[174] Martin Vechev and Eran Yahav. Deriving linearizable fine-grained concurrent objects. In *Programming Language Design and Implementation*, 2008. 96

[175] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in Erlang (2nd ed.)*. Prentice Hall International, 1996. 96

[176] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent Linux device drivers. In *Automated Software Engineering*, 2007. 6, 21, 94

[177] Damien Zufferey, Thomas Wies, and Thomas Henzinger. Ideal abstractions for well-structured transition systems. In *Verification, Model Checking and Abstract Interpretation*, 2012. 98