Information and Computation 252 (2017) 30–47

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Lost in abstraction: Monotonicity in multi-threaded programs [☆]

Alexander Kaiser [a], Daniel Kroening [a], Thomas Wahl [b,*]

[a] University of Oxford, United Kingdom
[b] Northeastern University, Boston, United States

ARTICLE INFO

Article history:
Received 15 December 2014
Available online 4 March 2016

Keywords:
Multi-threaded software
Parameterized verification
Monotonicity
Predicate abstraction

ABSTRACT

Monotonicity in concurrent systems stipulates that, in any global state, system actions remain executable when new processes are added to the state. This concept is both natural and useful: if every thread's memory is finite, monotonicity often guarantees the decidability of safety properties even when the number of running threads is unknown. In this paper, we show that finite-data thread abstractions for model checking can be at odds with monotonicity: predicate-abstracting monotone software can result in non-monotone Boolean programs — the monotonicity is *lost in the abstraction*. As a result, pertinent well-established safety checking algorithms for infinite-state systems become inapplicable. We demonstrate how monotonicity in the abstraction can be restored, without affecting safety properties. This improves earlier approaches of enforcing monotonicity via overapproximations. We implemented our solution in the unbounded-thread model checker monabs and applied it to numerous concurrent programs and algorithms, whose predicate abstractions are often fundamentally beyond existing tools.

1. Introduction

Multi-threading is becoming a 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. This can entrap the inexperienced programmer to create code with puzzling and irreproducible behavior. The software community needs to respond to this threat in part by providing formal technology that exposes potential bugs in concurrent programs before deployment.

Towards this end, this paper proposes a safety analysis method for non-recursive procedures executed by multiple threads (e.g. dynamically generated, and possibly unbounded in number), which communicate via shared variables and higher-level mechanisms such as mutexes. 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.

The verification method we propose is based on *predicate abstraction*. The utility of this technique 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. We introduce an extension of predicate abstraction to multi-threaded programs that enables reasoning about intricate data relationships, namely

[☆] Support provided by the Toyota Motor Corporation, NSF grant 1253331, and ERC project 280053.

[*] Corresponding author.

http://dx.doi.org/10.1016/j.ic.2016.03.003
0890-5401/© 2016 Elsevier Inc. All rights reserved.

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

struct Spinlock lock;                       // shared

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

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

The ticket algorithm: Shared variable `lock` has two natural-number components: `s` holds the number of the ticket currently being served, while `t` holds the next free number (if no thread is currently served, we have `s` = `t`).
To request access to the protected region, a thread atomically retrieves the value of `t` into local variable `l` and then increments `t`. The thread then spins until `l` equals `s`. To unlock, `s` is incremented.

Fig. 1. The ticket algorithm — Our goal is to verify standard *parameterized mutual exclusion*: no matter how many threads try to acquire and release the lock concurrently, no two of them are simultaneously between the calls to functions spin_lock and spin_unlock.

shared-variable: "shared variables `s` and `t` are equal",
single-thread: "local variable `l` of thread i is less than shared variable `s`", and
inter-thread: "local variable `l` of thread i is less than variable `l` *in all other threads*".

Why such a rich predicate language? For certain concurrent algorithms such as the widely used *ticket* busy-wait lock algorithm [1] (the default locking mechanism in the Linux kernel since 2008; see Fig. 1), the verification of elementary safety properties *requires* inter-thread relationships (see Sect. 2.2). They are needed to express, for instance, that a thread holds the minimum ticket value, an inter-thread relationship.

In the main part of the paper, we address the problem of full parameterized (unbounded-thread) program verification with respect to our predicate language. Such reasoning requires first that the n-thread abstract program $\hat{\mathcal{P}}^n$, obtained by existential predicate abstraction of the n-thread concrete program $\mathcal{P}^n$, is rewritten into a generic template program $\tilde{\mathcal{P}}$ to be executed by (any number of) multiple threads. In order to capture the semantics of these programs in the template $\tilde{\mathcal{P}}$, the template programming language, too, must permit variables that refer to the currently executing thread, or to *all* passive (non-executing) threads; we call such programs *dual-reference (DR)*. We describe how to obtain $\tilde{\mathcal{P}}$, namely as an overapproximation of $\hat{\mathcal{P}}^{\mathsf{b}}$, for a constant b that grows linearly with the number of inter-thread predicates used in the abstraction.

Given a *Boolean* dual-reference program $\tilde{\mathcal{P}}$ (obtained from predicate abstraction), we might expect the unbounded-thread replicated program $\tilde{\mathcal{P}}^\infty$ to form a classical *well quasi-ordered transition system* [2], enabling the fully automated, algorithmic safety property verification in the abstract. This turns out not to be the case: the expressiveness of dual-reference programs renders parameterized 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. Since the input C programs are, by contrast, typically monotone, we say the monotonicity is *lost in the abstraction*. As a result, our abstract programs are in fact not well quasi-ordered.

Inspired by earlier work on *monotonic abstractions* [3], 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 $\tilde{\mathcal{P}}_m$ gives rise to a monotone (and thus well quasi-ordered) system. The closure operator essentially terminates passive threads that block transitions allowed by other passive threads. In contrast to earlier work [3], which enforces monotonicity in genuinely *non*-monotone systems, we exploit the monotonicity of the input programs. As a result, the monotone closure $\tilde{\mathcal{P}}_m$ can be shown to be safety-equivalent to the non-monotone program $\tilde{\mathcal{P}}$.

To summarize, the core contribution of this paper is a predicate abstraction strategy for asynchronous unbounded-thread 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. We have implemented our technique in the infinite-state model checker monabs. We observe that our tool is able to verify certain parameterized programs (such as the ticket algorithm) that are fundamentally beyond existing tools we are aware of [4–11]. The reasons vary from their inability to deal with unbounded threads, to lacking support for inter-thread predicates. We include an extensive experimental evaluation on system-level concurrent software and synchronization algorithms.

2. Inter-thread predicate abstraction

In this section we introduce single- and inter-thread predicates, with respect to which we then formalize existential predicate abstraction. Except for the extended predicate language, these concepts are mostly standard and lay the technical foundations for the contributions of this paper.

2.1. Input programs and predicate language

Asynchronous programs An *asynchronous program* $\mathcal{P}$ allows only one thread at a time to change its local state; the executability of the state change does not depend on the state of other threads. We model $\mathcal{P}$, 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 states are characterized by a formula $\mathcal{I}$ over V. We assume $\mathcal{I}$ is expressible in a suitable logic for which existential quantification is computable (required later for the abstraction).

As usual, the computation may be controlled by a local program counter $\mathtt{pc}$; it may also involve non-recursive function calls. 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 $L_i = \{\mathtt{l}_i : \mathtt{l} \in L\}$. We 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 $f\{X_1 \mathrel{|\!\triangleright} X_2\}$ short for $f\{X_1 \triangleright X_2\}\{X_1{}' \triangleright X_2{}'\}$. As an example, given $S = \{\mathtt{s}\}$ and $L = \{\mathtt{l}\}$, we have $(\mathtt{l}' = \mathtt{l} + \mathtt{s})\{L \mathrel{|\!\triangleright} L_a\} = (\mathtt{l}'_a = \mathtt{l}_a + \mathtt{s})$. Finally, let $X \overset{\circ}{=} X'$ stand for $\forall x \in X : x = x'$.

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

$$\mathcal{P}^n = (\mathcal{R}^n, \mathcal{I}^n) = \left(\bigvee\nolimits_{a=1}^{n} (\mathcal{R}_a)^n,\ \bigwedge\nolimits_{i=1}^{n} \mathcal{I}\{L \triangleright L_i\}\right) \tag{1}$$

where

$$(\mathcal{R}_a)^n :: \mathcal{R}\{L \mathrel{|\!\triangleright} L_a\} \ \wedge\ \bigwedge\nolimits_{p \in [1..n] \setminus \{a\}} L_p \overset{\circ}{=} L'_p\,. \tag{2}$$

Formula $(\mathcal{R}_a)^n$ asserts that the shared variables and the local variables of the *active* (executing) thread a are updated according to $\mathcal{R}$, while those of passive threads $p \neq a$ are unchanged. A state is *initial* if all threads are in a state satisfying $\mathcal{I}$. An *n-thread execution* is a finite sequence of n-thread program states whose first state is initial, and whose adjacent states are related by $\mathcal{R}^n$. We assume the existence of an error program location in $\mathcal{P}$; an *error state* of $\mathcal{P}^n$ is one with some thread in the error location. $\mathcal{P}$ is *safe* if no execution in $\mathcal{P}^n$, for any n, ends in an error.

Predicate language We extend the predicate language from [9] by introducing *passive-thread variables* $L_{\mathbf{P}} = \{\mathtt{l}_{\mathbf{P}} : \mathtt{l} \in L\}$; here **P** is a fixed symbol (not a variable). Each passive-thread variable represents a local variable owned by a generic passive thread. The presence of variables of various categories gives rise to the following predicate classification.

Definition 1. A predicate Q over S, L and $L_{\mathbf{P}}$ is **shared** if it contains variables from S only, **local** if it contains variables from L only, **single-thread** if it contains variables from L but not from $L_{\mathbf{P}}$, and **inter-thread** if it contains variables from L and from $L_{\mathbf{P}}$.

Assuming each predicate contains at least one variable, the classifications *shared*, *single-thread*, and *inter-thread* are mutually exclusive. Note that single- and inter-thread predicates may contain variables from S; local predicates are a special case of single-thread predicates. As an example, in the ticket algorithm (Fig. 1), with $S = \{\mathtt{s}, \mathtt{t}\}$ and $L = \{\mathtt{l}, \mathtt{pc}\}$, examples of shared, local, single- and inter-thread predicates are: $\mathtt{s} = \mathtt{t}$, $\mathtt{l} = 5$, $\mathtt{s} = \mathtt{l}$ and $\mathtt{l} \neq \mathtt{l}_{\mathbf{P}}$, respectively.

Semantics Let $Q[1], \ldots, Q[m]$ be m predicates (any class). Predicate $Q[j]$ is evaluated in a given n-thread state ($n \geq 2$) with respect to a choice of active thread a:

$$Q[j]_a :: \ \bigwedge\nolimits_{p \in [1..n] \setminus \{a\}} Q[j]_{a,p} \tag{3}$$

where

$$Q[j]_{a,p} :: \ Q[j]\{L \triangleright L_a\}\{L_{\mathbf{P}} \triangleright L_p\}\,. \tag{4}$$

As special cases, for single-thread and shared predicates (no $L_{\mathbf{P}}$ variables), we have $Q[j]_a = Q[j]\{L \triangleright L_a\}$ and $Q[j]_a = Q[j]$, respectively. We write $u \models Q[j]_a$ if $Q[j]_a$ holds in state u. Predicate $Q[j]$ gives rise to an abstraction function α, mapping each n-thread program state u to an $m \times n$ bit matrix with entries

$$\alpha(u)_{j,a} \quad = \quad u \models Q[j]_a\,. \tag{5}$$

Function α partitions the n-thread program state space via m predicates into $2^{m \times n}$ equivalence classes. Consider the inter-thread predicates $\mathtt{l} \leq \mathtt{l}_{\mathbf{P}}$, $\mathtt{l} > \mathtt{l}_{\mathbf{P}}$, and $\mathtt{l} \neq \mathtt{l}_{\mathbf{P}}$ for a local variable $\mathtt{l}$, $n = 4$ and the state $u :: (\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}_4) = (4, 4, 5, 6)$:

$$\alpha(u) = \begin{pmatrix} \mathtt{T} & \mathtt{T} & \mathtt{F} & \mathtt{F} \\ \mathtt{F} & \mathtt{F} & \mathtt{F} & \mathtt{T} \\ \mathtt{F} & \mathtt{F} & \mathtt{T} & \mathtt{T} \end{pmatrix}. \tag{6}$$

In the matrix, row $j \in \{1,2,3\}$ lists the truth of predicate $Q[j]$ for each thread $a \in \{1,2,3,4\}$ in the active role. Predicate $\mathtt{l} \leq \mathtt{l}_\mathbf{P}$ captures whether a thread owns the minimum value for local variable $\mathtt{l}$ (true for $a = 1,2$); $\mathtt{l} > \mathtt{l}_\mathbf{P}$ tracks whether a thread is the *unique* owner of the maximum value (true for $a = 4$); finally $\mathtt{l} \neq \mathtt{l}_\mathbf{P}$ captures the uniqueness of a thread's value of $\mathtt{l}$ (true for $a = 3,4$).

According to Eq. (3), the semantics of inter-thread predicates is defined via universal thread quantification. As a result, our predicate language is not closed under negation. For example, the predicate formulas $\mathtt{l} \leq \mathtt{l}_\mathbf{P}$ and $\mathtt{l} > \mathtt{l}_\mathbf{P}$ both evaluate to false under (3), for $a = 3$. (On the other hand, a predicate and its formal negation cannot both evaluate to true.)

2.2. Limits of single-thread predicate abstraction

The use of the expressive and presumably expensive inter-thread predicates introduced in Sect. 2.1 is motivated: automated methods that cannot reason about them [12,9,7] fail for the ticket algorithm:

Lemma 2. *Consider the parameterized ticket algorithm (Fig. 1) where threads call* spin_lock *and* spin_unlock *arbitrarily often. There is no quantifier-free invariant over finitely many shared and single-thread predicates that implies mutual exclusion.*

Proof. We formalize the statement of the lemma. Let $Q[1], \ldots, Q[m]$ be predicates formulated over the shared variables $\mathtt{s}, \mathtt{t}$ and the local variables $\mathtt{pc}, \mathtt{l}$ of any **one** thread. Define a set of Boolean variables $\{b[j]_a : 1 \leq j \leq m \wedge a \in \mathbb{N}\}$, with semantics $b[j]_a \Leftrightarrow Q[j]\{L \triangleright L_a\}$. Suppose I is an invariant formula over the $b[j]_a$, i.e. $\mathit{Reach} \Rightarrow I$ for the (infinite) set Reach of reachable global states. We show that I does not imply safety.

For $i \in \mathbb{N}$, consider the global states u_i with

$$\begin{aligned} u_i.\mathtt{s} = 1,\ u_i.\mathtt{t} = 2,\ u_i.\mathtt{pc}_i = \ell_3,\ u_i.\mathtt{l}_i = 1, \\ u_i.\mathtt{pc}_k = \ell_1,\ u_i.\mathtt{l}_k = 0 \ \text{ for } k \neq i \end{aligned} \tag{7}$$

(where $x.y$ denotes the value of variable y in state x; an index points to a local variable owned by the corresponding thread; ℓ_3 stands for the protected code region). All u_i are reachable, witnessed by thread i pulling the first ticket and proceeding to the protected region; the other threads remain in ℓ_1.

Let $C_1, \ldots, C_w$ be the cubes in the DNF representation of I: $I = C_1 \vee \ldots \vee C_w$. Since, for each i, u_i is reachable and thus satisfies I, it satisfies at least one cube of I. Since we have infinitely many u_i to choose from, by the pigeon hole principle there exist a cube C and u_i, u_k with $i \neq k$ such that both u_i, u_k satisfy C. Now let u be the state equal to u_i except that, on thread-k variables, u agrees with u_k: $u.\mathtt{pc}_k = u_k.\mathtt{pc}_k$ $(= \ell_3)$ and $u.\mathtt{l}_k = u_k.\mathtt{l}_k$ $(= 1)$. Since $u.\mathtt{pc}_i = u_i.\mathtt{pc}_i = \ell_3$, state u violates mutual exclusion. We argue that u satisfies I, which shows that I does not imply safety.

To this end, we note that C is a conjunction of literals over the $b[j]_a$. We partition C into the sub-cubes C^i and C^k of literals containing $\mathtt{pc}_i, \mathtt{l}_i$ and $\mathtt{pc}_k, \mathtt{l}_k$, resp., and the remaining literals C^r: $C = C^i \wedge C^k \wedge C^r$. Since u is equal to u_i except for thread-k variables, and u_i satisfies C and hence C^i and C^r, and the latter two do not refer to any thread-k variables, u also satisfies C^i and C^r. It remains to show that u satisfies C^k (hence C, hence I): u agrees with u_k on all variables $\mathtt{s}, \mathtt{t}, \mathtt{pc}_k, \mathtt{l}_k$, and u_k satisfies C and hence C^k, and the latter refers only to those four variables. □

We discuss the sensitivity of this proof to the program being investigated, and to the expressiveness of the predicate language.

- The proof **generalizes** to the class of safe unbounded-thread asynchronous programs (i) with a mutual exclusion safety property that does not depend on the shared variables (as is the case for local state section reachability), and (ii) with an unbounded supply of *reachable* states u_i such that thread i is in its critical section (and all other threads are not, since the program is safe).
- The proof **generalizes** to the case of **arbitrary** c**-thread predicates** (instead of only single-thread predicates), for a constant c: these are predicates over the (shared and) local states of any c-tuple of threads. To see this, observe that formula I refers to only a finite number of predicate variables $b[j]_{a_1..a_c}$ and thus can track only a finite number of thread c-tuple relationships. More precisely, there always exists an unbounded supply of states u_i, u_k with $i \neq k$ such that no predicate j used in I (via the corresponding variable $b[j]_{a_1..a_c}$) relates threads i and k (we can pick i and k "large enough"). This enables partitioning cube C as in the proof into sub-cubes that refer to i but not k, to k but not i, and to neither. State u as in the proof now satisfies C and hence I, but violates mutual exclusion.
- The proof **breaks down** if we permit **quantifiers**, e.g. in the form of **inter-thread predicates**: individual predicates can now reason about any number of threads; the partitioning of cube C is no longer possible.
- The proof **breaks down** under a **thread bound**: in this case the "state pool" u_i is bounded, too; the pigeon hole principle may fail.

A general treatment of the limits of thread-modular and Owicki–Gries style proof systems (which do not use inter-thread predicates) is available in the literature [13].

Table 1
Abstraction $(\hat{\mathcal{R}}_1)^2$ for stmt. $\mathtt{l} := \mathtt{l} - 1$ against predicate $\mathtt{l} < \mathtt{l}_{\mathbf{P}}$ (left); concrete witness transitions, i.e. elements of $(\mathcal{R}_1)^2$ (right). The highlighted row indicates asynchrony violations.

$\mathtt{b}_1$	$\mathtt{b}_2$	$\mathtt{b}'_1$	$\mathtt{b}'_2$		$\mathtt{l}_1$	$\mathtt{l}_2$	$\mathtt{l}'_1$	$\mathtt{l}'_2$
F	F	T	F	←	1	1	0	1
F	T	F	F		1	0	0	0
F	T	F	T		2	0	1	0
T	F	T	F		1	2	0	2

2.3. *Existential inter-thread predicate abstraction*

Embedded into our formalism, the goal of *existential predicate abstraction* [14,15] is to derive an abstract program $\hat{\mathcal{P}}^n$ by treating the equivalence classes induced by Eq. (5) as abstract states. $\hat{\mathcal{P}}^n$ thus has $m \times n$ Boolean variables:

$$\hat{V}_n = \bigcup\nolimits_{a=1}^{n} \hat{L}_a = \bigcup\nolimits_{a=1}^{n} \{b[j]_a : 1 \le j \le m\}.$$

Variable $b[j]_a$ tracks the truth of predicate $Q[j]$ for active thread a (if $m = 1$ we simply write b_a). This is formalized in (8), relating concrete and abstract n-thread states (valuations of V_n and $\hat{V}_n$, resp.):

$$\mathcal{D}^n :: \bigwedge_{j=1}^{m} \bigwedge_{a=1}^{n} b[j]_a \Leftrightarrow Q[j]_a . \tag{8}$$

For a formula f, let f' denote f after replacing each variable by its primed version. We then have $\hat{\mathcal{P}}^n = (\hat{\mathcal{R}}^n, \hat{\mathcal{I}}^n) = \left(\bigvee_{a=1}^{n} (\hat{\mathcal{R}}_a)^n,\ \hat{\mathcal{I}}^n\right)$ where

$$(\hat{\mathcal{R}}_a)^n :: \exists V_n V'_n : (\mathcal{R}_a)^n \wedge \mathcal{D}^n \wedge (\mathcal{D}^n)', \tag{9}$$

$$\hat{\mathcal{I}}^n :: \exists V_n \quad : \mathcal{I}^n \wedge \mathcal{D}^n . \tag{10}$$

In the abstraction, all variables are local,[1] and variables $b[j]_a$ are owned by abstract thread a. As an example, consider the decrement operation $\mathtt{l} := \mathtt{l} - 1$ on a local integer variable $\mathtt{l}$, and the inter-thread predicate $\mathtt{l} < \mathtt{l}_{\mathbf{P}}$. Using Eq. (9) with $n = 2$, $a = 1$, we get four abstract transitions, listed in Table 1. The table shows that asynchrony is lost in the abstraction: in the highlighted transition, the executing thread 1 changes its local state by advancing its pc, and thread 2 changes its local state by changing $\mathtt{b}_2$. By contrast, on the right we have $\mathtt{l}_2 = \mathtt{l}'_2$ (and $\mathtt{pc}_2 = \mathtt{pc}'_2$) in all rows. The loss of asynchrony will become relevant in Sect. 3, where we define an abstract Boolean programming language that necessarily must accommodate non-asynchronous programs.

Proving the ticket algorithm for the fixed-thread case As in any existential abstraction, the abstract program $\hat{\mathcal{P}}^n$ overapproximates (the set of executions of) the concrete program $\mathcal{P}^n$; the former can therefore be used to verify safety of the latter. We illustrate this using the ticket algorithm (Fig. 1). Consider the predicates $Q[1] :: \mathtt{l} \neq \mathtt{l}_{\mathbf{P}}$, $Q[2] :: \mathtt{t} > \max(\mathtt{l}, \mathtt{l}_{\mathbf{P}})$, and $Q[3] :: \mathtt{s} = \mathtt{l}$. The first two are inter-thread; the third is single-thread. The predicates assert the uniqueness of a ticket $(Q[1])$, that the next free ticket is larger than all tickets currently owned by threads $(Q[2])$, and that a thread's ticket is currently being served $(Q[3])$. The (symmetry-reduced and slightly partial) abstract reachability tree for $\hat{\mathcal{P}}^n$ and these predicates is shown in Fig. 2. It tells us that mutual exclusion is satisfied: there is no state with both threads in location ℓ_3. The tree has about a dozen nodes, a number that grows exponentially with n.

3. From existential to parametric abstraction

Classical existential abstraction as described in Sect. 2.3 obliterates the parametricity present in the concrete concurrent program: the program is given as an instantiation of a template $\mathcal{P}$, while the abstraction is formulated via predicates over the explicitly expanded n-thread program $\mathcal{R}^n$. As a result, parametric reasoning over an unknown number of threads is impossible.

To overcome these problems, we now derive an overapproximation of $\hat{\mathcal{P}}^n$ via a generic program template $\tilde{\mathcal{P}}$ that can be instantiated for any n. To this end we note that the programs $\hat{\mathcal{P}}^n$ resulting from inter-thread predicate abstraction are no longer asynchronous, as Table 1 has shown. As a result, we need an abstract programming language more powerful than the asynchronous language of Sect. 2.1.

[1] If predicate $Q[j]$ is shared, the semantics of terms $Q[j]_a$ is the same for all a; we could track $Q[j]$ using a single Boolean variable. To keep the notation compact, we ignore this redundancy here

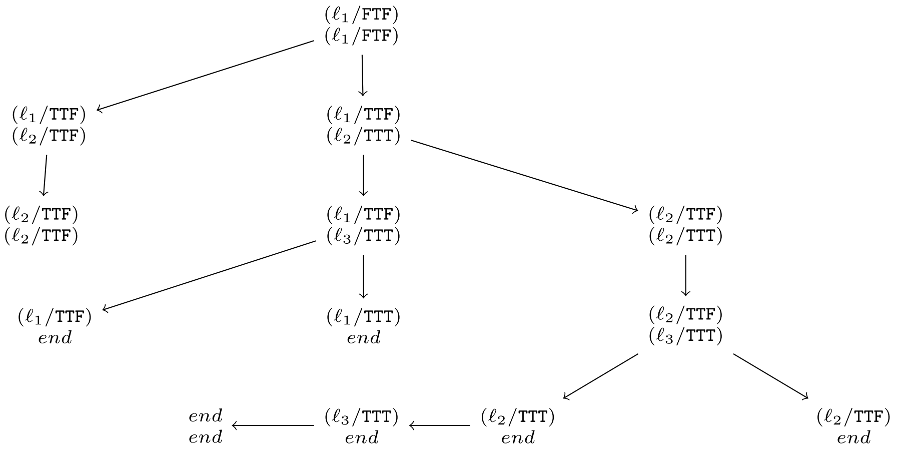

Fig. 2. Abstract reachability tree for the ticket algorithm for 2 threads. Local variables of abstract thread 1 are shown atop those of thread 2. (There are no shared abstract variables.)

3.1. Dual-reference programs

In contrast to 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 $\tilde{L}_{\mathbf{P}} = \{\mathtt{l}_{\mathbf{P}} : \mathtt{l} \in \tilde{L}\}$, where again $\mathbf{P}$ is a fixed symbol. The latter set contains passive-thread variables, which, intuitively, regulate the behavior of non-executing threads. To simplify reasoning about DR programs, we exclude classical shared variables from the description: they can be simulated using the active and passive flavors of local variables, as we discuss at the end of Sect. 3.1.

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, $\tilde{\mathcal{I}}$ may contain variables from $\tilde{L}_{\mathbf{P}}$. The n-thread instantiation $\tilde{\mathcal{P}}^n$ of a DR program $\tilde{\mathcal{P}}$ is defined for $n \geq 2$ as

$$\tilde{\mathcal{P}}^n = (\tilde{\mathcal{R}}^n, \tilde{\mathcal{I}}^n) = \left(\bigvee\nolimits_{a=1}^{n} (\tilde{\mathcal{R}}_a)^n,\ \bigvee\nolimits_{a=1}^{n} (\tilde{\mathcal{I}}_a)^n\right) \tag{11}$$

where

$$(\tilde{\mathcal{R}}_a)^n :: \ \bigwedge\nolimits_{p\in[1..n]\setminus\{a\}} \tilde{\mathcal{R}}\{\tilde{L} \trianglerighteq \tilde{L}_a\}\{\tilde{L}_{\mathbf{P}} \trianglerighteq \tilde{L}_p\} \tag{12}$$

$$(\tilde{\mathcal{I}}_a)^n :: \ \bigwedge\nolimits_{p\in[1..n]\setminus\{a\}} \tilde{\mathcal{I}}\{\tilde{L} \triangleright \tilde{L}_a\}\{\tilde{L}_{\mathbf{P}} \triangleright \tilde{L}_p\} \tag{13}$$

Recall that $f\{X_1 \trianglerighteq X_2\}$ denotes index replacement of both current-state and next-state variables. Eq. (12) encodes the effect of a transition on the active thread a, and $n-1$ passive threads p. The conjunction ensures that the transition formula $\tilde{\mathcal{R}}$ holds no matter which thread $p \neq a$ takes the role of the passive thread in $\tilde{\mathcal{R}}$: transitions that effectively depend on the identity of the passive thread are rejected.

Simulating shared via local variables Our exclusion of shared variables from the description of dual-reference programs is not a restriction: such variables can be simulated via active- and passive-thread local variables, as follows. To eliminate shared variable $\mathtt{s}$, we introduce a fresh local variable $\mathtt{l} \in \tilde{L}$, and replace a statement like $\mathtt{s} := 5$ by the atomic statement $\mathtt{l} := 5$, $\mathtt{l}_{\mathbf{P}} := 5$. That is, 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 synchronized across all threads. An example of this construction is given in the proof of Theorem 6 in Sect. 4.1.

3.2. Computing an abstract dual-reference template

From the existential abstraction $\hat{\mathcal{P}}^n$ we derive a Boolean dual-reference template program $\tilde{\mathcal{P}}$ such that, for all $n \geq 2$, the n-fold instantiation $\tilde{\mathcal{P}}^n$ overapproximates $\hat{\mathcal{P}}^n$. The variables of $\tilde{\mathcal{P}}$ are $\tilde{L} = \{\mathtt{b}[j] : 1 \leq j \leq m\}$ and $\tilde{L}_{\mathbf{P}} = \{\mathtt{b}[j]_{\mathbf{P}} : 1 \leq j \leq m\}$. Intuitively, the transitions of $\tilde{\mathcal{P}}$ are those that are feasible, for **some** n, in $\hat{\mathcal{P}}^n$, given active thread 1 and passive thread 2. The semantics of dual-refer ence programs then ensures that the transitions are valid for any choice of passive thread.

We first compute the set $\tilde{\mathcal{R}}(n)$ of these transitions for fixed n. Formally, the components of $\tilde{\mathcal{P}}(n) = (\tilde{\mathcal{R}}(n), \tilde{\mathcal{I}}(n))$ are, for $n \geq 2$,

$$\tilde{\mathcal{R}}(n) :: \exists \hat{L}_3, \hat{L}_3', \ldots, \hat{L}_n, \hat{L}_n' : \ (\hat{\mathcal{R}}_1)^n\{\hat{L}_1 \trianglerighteq \tilde{L}\}\{\hat{L}_2 \trianglerighteq \tilde{L}_{\mathbf{P}}\} \tag{14}$$

$$\tilde{\mathcal{I}}(n) :: \exists \hat{L}_3, \ldots, \hat{L}_n : \qquad \hat{\mathcal{I}}^n\ \{\hat{L}_1 \triangleright \tilde{L}\}\{\hat{L}_2 \triangleright \tilde{L}_{\mathbf{P}}\} \tag{15}$$

Table 2
Part of the abstraction $(\hat{\mathcal{R}}_1)^3$ for stmt. $\mathtt{l} := \mathtt{l} - 1$ against predicate $\mathtt{l} < \mathtt{l}_\mathbf{P}$ (left); concrete witness transitions (right). The abstract transitions are inconsistent with (16) as a template.

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

We apply this strategy to the earlier example of the decrement statement $\mathtt{l} := \mathtt{l} - 1$. To compute Eq. (14) first with $n = 2$, we need $(\hat{\mathcal{R}}_1)^2$, which was enumerated in Table 1. Simplification results in a Boolean DR program with variables $\mathtt{b}$ and $\mathtt{b}_\mathbf{P}$ and transition relation

$$\tilde{\mathcal{R}}(2) = (\neg \mathtt{b} \wedge \mathtt{b}_\mathbf{P} \wedge \neg \mathtt{b}') \vee (\neg \mathtt{b}_\mathbf{P} \wedge \mathtt{b}' \wedge \neg \mathtt{b}'_\mathbf{P})\,. \tag{16}$$

Using (16) as the template $\tilde{\mathcal{R}}$ in (12) generates existential abstractions of many concrete decrement transitions; for instance, for $n = 2$ and $a = 1$ we get back the transition relation in Table 1. The question is now: does (16) suffice as a template, i.e. does $(\tilde{\mathcal{R}}(2))^n$ overapproximate $\hat{\mathcal{R}}^n$ for all n? The answer is no: the abstract 3-thread transitions shown in Table 2 are not permitted by $(\tilde{\mathcal{R}}(2))^n$ for any n, since neither $\neg \mathtt{b} \wedge \mathtt{b}_\mathbf{P}$ nor $\mathtt{b}' \wedge \neg \mathtt{b}'_\mathbf{P}$ are satisfied for all choices of passive threads. We thus increase n to 3, recompute Eq. (14), and obtain

$$\tilde{\mathcal{R}}(3) :: \tilde{\mathcal{R}}(2) \vee (\neg \mathtt{b} \wedge \neg \mathtt{b}_\mathbf{P} \wedge \neg \mathtt{b}' \wedge \neg \mathtt{b}'_\mathbf{P})\,. \tag{17}$$

The new disjunct accommodates the abstract transitions in Table 2, which were missing before.

Does $(\tilde{\mathcal{R}}(3))^n$ overapproximate $\hat{\mathcal{R}}^n$ for all n? When does the process of increasing n stop? To answer these questions, we first state the following diagonalization lemma, which helps us prove the overapproximation property for the template program.

Lemma 3. *$(\tilde{\mathcal{P}}(n))^n$ overapproximates $\hat{\mathcal{P}}^n$: For every $n \geq 2$ and every a, $(\hat{\mathcal{R}}_a)^n \Rightarrow (\tilde{\mathcal{R}}(n)_a)^n$ and $\hat{\mathcal{I}}^n \Rightarrow (\tilde{\mathcal{I}}(n)_a)^n$.*

Proof. For the initial states, by equations (10), (13) and (15) the implication becomes

$$\begin{gathered}
\hat{\mathcal{I}}^n \;\Rightarrow\; \textstyle\bigwedge_{p\in[1..n]\setminus\{a\}} \tilde{\mathcal{I}}(n)\{\tilde{L} \triangleright \tilde{L}_a\}\{\tilde{L}_\mathbf{P} \triangleright \tilde{L}_p\} \\
\equiv \\
\hat{\mathcal{I}}^n \;\Rightarrow\; \textstyle\bigwedge_{p\in[1..n]\setminus\{a\}} \left(\left(\exists \hat{L}_3,\ldots,\hat{L}_n : \hat{\mathcal{I}}^n\right)\{\hat{L}_1 \triangleright \tilde{L}\}\{\hat{L}_2 \triangleright \tilde{L}_\mathbf{P}\}\right)\{\tilde{L} \triangleright \tilde{L}_a\}\{\tilde{L}_\mathbf{P} \triangleright \tilde{L}_p\} \\
\equiv \\
\hat{\mathcal{I}}^n \;\Rightarrow\; \textstyle\bigwedge_{p\in[1..n]\setminus\{a\}} \left(\exists \hat{L}_3,\ldots,\hat{L}_n : \hat{\mathcal{I}}^n\right)\{\hat{L}_1 \triangleright \tilde{L}_a\}\{\hat{L}_2 \triangleright \tilde{L}_p\}\,.
\end{gathered}$$

The final implication is a formula over the Boolean variable set $\hat{V}_n = \{b[j]_i : 1 \leq j \leq m, 1 \leq i \leq n\}$. Consider any assignment $\hat{A}_n$ to these variables that satisfies $\hat{\mathcal{I}}^n$. Let $p \in \{1,\ldots,n\} \setminus \{a\}$ be arbitrary, and assign to the variables in $\hat{L}_3 \cup \ldots \cup \hat{L}_n = \{b[j]_i : 1 \leq j \leq m, 3 \leq i \leq n\}$ the values given by $\hat{A}_n$. Then the final implication holds since the initial condition $\mathcal{I}$ is identical for all threads (1), so replacing thread ids 1 and 2 by thread ids a and p preserves truth. The case of the transition relation is similar. □

We finally give a *linear* saturation bound for the sequence $(\tilde{\mathcal{P}}(n))$. Along with the diagonalization lemma, the bound allows us to obtain a template program $\tilde{\mathcal{P}}$ independent of n, which in turn enables parametric reasoning in the abstract.

Theorem 4. *Let $\#_{IT}$ be the number of inter-thread predicates among the $Q[j]$. Then the sequence $(\tilde{\mathcal{P}}(n))$ stabilizes at $\mathsf{b} = 2 \times (\#_{IT} + 1)$, i.e. for $n \geq \mathsf{b}$, $\tilde{\mathcal{P}}(n) = \tilde{\mathcal{P}}(\mathsf{b})$.*

Proof. It suffices to prove this theorem for the special case that all predicates are inter-thread: $m = \#_{IT}$. Sect. 3.1 explained how to eliminate shared predicates, and any single-thread predicate can syntactically be turned into an inter-thread predicate by conjoining it with the redundant expression $\mathtt{l}_\mathbf{P} = \mathtt{l}_\mathbf{P}$; applying formula (3) gives us single-thread predicate semantics.

For the duration of this proof, let $Q[1],\ldots,Q[m]$ be the list of predicates, all of which are inter-thread, and the formula $\tilde{\mathcal{R}}(\infty)$ over the variable set $\tilde{L} \cup \tilde{L}_\mathbf{P}$ denote a finite characterization of $\bigvee_{n=1}^{\infty} \tilde{\mathcal{R}}(n)$. The existence of a finite encoding of $\tilde{\mathcal{R}}(\infty)$ is guaranteed, since the sequence $(\tilde{\mathcal{P}}(n))$ is monotonously increasing and the variable sets $\tilde{L}$ and $\tilde{L}_\mathbf{P}$ are finite.

We show that stabilization occurs at $\mathsf{b} = 2 \times (m + 1)$, i.e., $\tilde{\mathcal{R}}(\infty) = \tilde{\mathcal{R}}(\mathsf{b})$. The proof is by induction on m, we begin with $m = 1$. Let $t = (\mathtt{b}, \mathtt{b}_\mathbf{P}, \mathtt{b}', \mathtt{b}'_\mathbf{P})$ be some abstract transition in the dual-reference relation $\tilde{\mathcal{R}}(\infty)$, and $\hat{w}$ and w be the abstract and concrete n-thread witness transitions for t, respectively, and finally $u = (\hat{w}, w)$. Hence u is — in analogy with the presentation in Table 2 — of the form

Table 3
Tightness example: The input transition relation (18) with inter-thread predicate (19). The obtained abstract dual-reference program template $\tilde{\mathcal{R}}(\infty)$ is shown on the left, with the concrete part of a minimal witness from V_4 and V'_4 for each abstract transition in the middle ($\star =$ unconstrained), and on the right the mapping π as used in the proof of Theorem 4.

t				$w^\triangle$						π			
$\mathtt{b}$	$\mathtt{b}_\mathbf{P}$	$\mathtt{b}'$	$\mathtt{b}'_\mathbf{P}$	$\mathtt{l}_1$	$\mathtt{l}_2$	$\mathtt{l}_3$	$\mathtt{l}_4$	$\mathtt{l}'_1$	$\mathtt{l}'_{i\neq 1}$	π_1	π_2	π_3	π_4
F	F	F	F	1	1	2	3	4	$\mathtt{l}_i$	3	3	4	3
F	F	T	F	1	1	2	$\star$	4	$\mathtt{l}_i$	3	3	–	3
T	F	F	F	1	4	2	$\star$	4	$\mathtt{l}_i$	2	–	3	–
F	T	F	T	1	2	3	$\star$	4	$\mathtt{l}_i$	2	–	–	–
F	T	T	T	1	2	$\star$	$\star$	4	$\mathtt{l}_i$	–	3	3	3
T	T	F	T	1	3	$\star$	$\star$	4	$\mathtt{l}_i$	–	–	2	–
T	T	T	T	1	5	$\star$	$\star$	4	$\mathtt{l}_i$	–	–	–	–

$$u = (\overbrace{\underbrace{\mathtt{b}_1, \ldots, \mathtt{b}_n}_{\hat{V}_n}, \underbrace{\mathtt{b}'_1, \ldots, \mathtt{b}'_n}_{\hat{V}'_n}}^{\hat{w}}, \overbrace{\underbrace{\mathtt{l}_1, \ldots, \mathtt{l}_n}_{V_n}, \underbrace{\mathtt{l}'_1, \ldots, \mathtt{l}'_n}_{V'_n}}^{w})$$

and such that it satisfies $(\mathcal{R}_1)^n \wedge \mathcal{D}^n \wedge (\mathcal{D}^n)'$ of the expanded $\tilde{\mathcal{R}}(n)$; see (9) and (14).[2] We call u an *(n-thread) witness* for t, and in particular a *minimal* one if no $(n-1)$-thread witness exists. Due to the final substitution in (14) it is $t = (\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}'_1, \mathtt{b}'_2)$, and due to asynchrony $\mathtt{l}_i = \mathtt{l}'_i$ for $i \in [2..n]$.

Next, we proof by cases that we can derive for any t with given witness u an ≤ 4-thread witness $u^\triangle = (\hat{w}^\triangle, w^\triangle)$, and thus that $t \in \tilde{\mathcal{R}}(4)$. Case $t = (\mathtt{T}, \mathtt{T}, \mathtt{T}, \mathtt{T})$. According to Eq. (3), $Q[1]_1$ evaluates to true (and hence $\mathtt{b}_1$) if and only if *all* its conjuncts $Q[1]_{1,p}$ evaluate to true (analog for $\mathtt{b}_2$ and primed versions). As in this case, removing any of the conjuncts preserves the overall's truth value, $\hat{w}^\triangle = (\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}'_1, \mathtt{b}'_2)$ and $w^\triangle = (\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}'_1, \mathtt{l}'_2)$ induce the 2-thread witness $u^\triangle$ for t.

Case $t = (\mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{F})$. According to Eq. (3), $Q[1]_1$ evaluates false (and hence $\mathtt{b}_1$) if and only if there *exists* a $p \in [2..n]$ such that each of the $Q[1]_{1,p}$ evaluates false (again, analog for $\mathtt{b}_2$ and primed versions). Without loss of generality, let $\pi : \{1, \ldots, 4\} \to \{3, \ldots, 6\}$ identify 4 passive threads in w that falsify each of the 4 conjuncts $Q[1]_1$, $Q[1]_2$, $Q[1]'_1$ and $Q[1]'_2$, i.e., π is such that $Q[1]_{1,\pi_1}$, $Q[1]_{2,\pi_2}$, $Q[1]'_{1,\pi_3}$, and $Q[1]'_{2,\pi_4}$ evaluate false. Then a 6-thread witness $u^\triangle$ for t is induced by $\hat{w}^\triangle = (b, b')$ with $b = (\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_{\pi_1}, .., \mathtt{b}_{\pi_4})$ and $w^\triangle = (l, l')$ with $l = (\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_{\pi_1}, .., \mathtt{l}_{\pi_4})$.

For the inductive step from m to $m+1$ predicates, we extend the obtained permutation by (at most) $4 = 6 - 2$ elements (we can reuse the first and second thread). We can conclude that stabilization occurs at $2 \times (2 \times m + 1)$ for any $m \geq 1$. Moreover, since for each single-thread predicate that was syntactically turned into inter-thread one, just the corresponding thread's local state (1st or 2nd) determines the truth of Eq. (3), and their state is untouched, we obtain that stabilization occurs at $2 \times (2 \times \#_{IT} + 1)$, and hence single-thread predicates do not "count".

Note that the previous bound holds for *non-asynchronous* input programs as well, as we did not assume any immutability of passive-thread variables so far. For asynchronous input programs, which exhibit this property, we can do better: First, we eliminate the reference to thread π_2 by defining $\pi_2 := \pi_1$ and setting $\mathtt{l}_2 := \mathtt{l}_1$. Then, if $Q[1]_{1,\pi_1}$ evaluates false, $Q[1]_{2,\pi_2}$ still evaluates false. Second, we eliminate the need for thread π_4. Due to asynchrony, which implies $\mathtt{l}_2 = \mathtt{l}'_2$ and $\mathtt{l}_{\pi_2} = \mathtt{l}'_{\pi_2}$, by defining $\pi_4 := \pi_2$, it still holds that if $Q[1]_{2,\pi_2}$ evaluates false, so does $Q[1]'_{2,\pi_4}$. Hence, $\hat{w}^\triangle = (b, b')$ with $b = (\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_{\pi_1}, \mathtt{b}_{\pi_2})$, and $w^\triangle = (l, l')$ with $l = (\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_{\pi_1}, \mathtt{l}_{\pi_2})$ induce a 4-thread witness $u^\triangle$ for t. The proofs for the remaining 14 cases is analogous, always yielding < 4-thread witnesses.

With the same inductive step as before we can conclude that stabilization occurs at $\mathsf{b} = 2 \times (\#_{IT} + 1)$ for any $m \geq 1$. The proof for the stabilization of $\tilde{\mathcal{I}}$ is analogous. $\square$

The bound established in Theorem 4 is asymptotically tight: consider the following concocted scenario with local variables $\mathtt{l} \in \{1, \ldots, 5\}$:

$$\mathcal{R} :: \ (\mathtt{l} = 1) \wedge (\mathtt{l}' = 4) \tag{18}$$

$$Q :: \ (\mathtt{l} \neq 1 \vee \mathtt{l}_\mathbf{P} \neq 2) \wedge (\mathtt{l} \neq 4 \vee \mathtt{l}_\mathbf{P} \neq 3)\,. \tag{19}$$

Eq. (14) does not stabilize for less than 4 threads. The obtained DR program has 7 transitions, which are enumerated in Table 3. The generalization that shows tightness for arbitrary numbers of inter-thread predicates is straightforward.

Corollary 5. *Let $\tilde{\mathcal{P}} := \tilde{\mathcal{P}}(\mathsf{b})$, for b as in Theorem 4. The components of $\tilde{\mathcal{P}}$ are thus $(\tilde{\mathcal{R}}, \tilde{\mathcal{I}}) = (\tilde{\mathcal{R}}(\mathsf{b}), \tilde{\mathcal{I}}(\mathsf{b}))$. Then, for $n \geq 2$, $\tilde{\mathcal{P}}^n$ overapproximates $\hat{\mathcal{P}}^n$.*

[2] Expanding Eq. (14) yields $\tilde{\mathcal{R}}(n) :: \exists V_n, V'_n, \hat{L}_3, \hat{L}'_3..\hat{L}'_n : \ (\mathcal{R}_1)^n \wedge \mathcal{D}^n \wedge (\mathcal{D}^n)'\{\hat{L}_1 \trianglerighteq \tilde{L}\}\{\hat{L}_2 \trianglerighteq \tilde{L}_\mathbf{P}\}$.

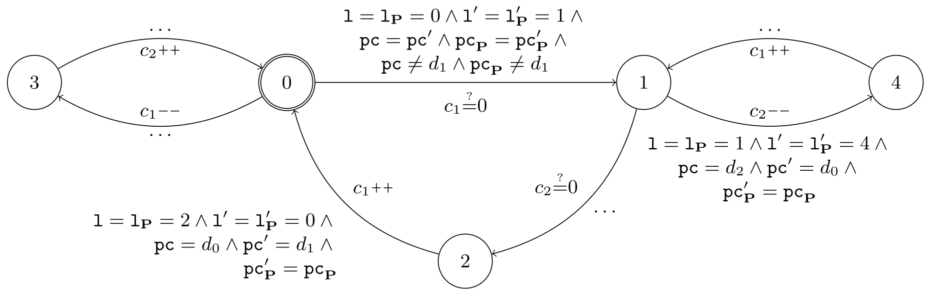
$\dots$
$c_2{+}{+}$
$c_1{-}{-}$
$\dots$
$\mathtt{l} = \mathtt{l}_{\mathbf{P}} = 0 \wedge \mathtt{l}' = \mathtt{l}'_{\mathbf{P}} = 1 \wedge$
$\mathtt{pc} = \mathtt{pc}' \wedge \mathtt{pc}_{\mathbf{P}} = \mathtt{pc}'_{\mathbf{P}} \wedge$
$\mathtt{pc} \neq d_1 \wedge \mathtt{pc}_{\mathbf{P}} \neq d_1$
$c_1 \overset{?}{=} 0$
$\dots$
$c_1{+}{+}$
$c_2{-}{-}$
$\mathtt{l} = \mathtt{l}_{\mathbf{P}} = 1 \wedge \mathtt{l}' = \mathtt{l}'_{\mathbf{P}} = 4 \wedge$
$\mathtt{pc} = d_2 \wedge \mathtt{pc}' = d_0 \wedge$
$\mathtt{pc}'_{\mathbf{P}} = \mathtt{pc}_{\mathbf{P}}$
3
0
1
4
2
$c_1{+}{+}$
$c_2 \overset{?}{=} 0$
$\dots$
$\mathtt{l} = \mathtt{l}_{\mathbf{P}} = 2 \wedge \mathtt{l}' = \mathtt{l}'_{\mathbf{P}} = 0 \wedge$
$\mathtt{pc} = d_0 \wedge \mathtt{pc}' = d_1 \wedge$
$\mathtt{pc}'_{\mathbf{P}} = \mathtt{pc}_{\mathbf{P}}$

Fig. 3. Minsky machine and (part of) its DR program encoding, shown as formulas labeling control transitions. The initial state $\tilde{\mathcal{I}}$ of the DR encoding is $\mathtt{l} = \mathtt{l}_{\mathbf{P}} = 0 \wedge \mathtt{pc} = \mathtt{pc}_{\mathbf{P}} = d_0$.

Proof. We have $\tilde{\mathcal{P}}^n = (\tilde{\mathcal{P}}(\mathsf{b}))^n \overset{(*)}{\sqsupseteq} (\tilde{\mathcal{P}}(n))^n \overset{(**)}{\sqsupseteq} \hat{\mathcal{P}}^n$, where $\sqsupseteq$ means "overapproximates". Step $(*)$ follows, for $\mathsf{b} > n$, from the monotonicity of sequence $(\tilde{\mathcal{P}}(n))$ and, for $\mathsf{b} \leq n$, from Theorem 4 (in which case $\sqsupseteq$ is $=$). Step $(**)$ is Lemma 3. □

As a consequence of losing asynchrony in the abstraction, many existing model checkers for concurrent software are not applicable to $\tilde{\mathcal{P}}$ [16,17,10]. For a fixed thread count n, the problem can be circumvented by forgoing the replicated nature of the concurrent programs, as done in [9] in the boom tool: it proves the ticket algorithm correct up to $n = 3$, but takes a defeating 30 minutes. The goal of the following section is to design an efficient and fully parametric solution.

4. Unbounded-thread dual-reference programs

4.1. Undecidability of boolean DR program verification

The multi-threaded Boolean dual-reference programs $\tilde{\mathcal{P}}^n$ resulting from predicate-abstracting asynchronous programs against inter-thread predicates are symmetric and free of recursion. The symmetry can be exploited using classical methods that counter-abstract the state space [18]: a global state is encoded as a vector of counter variables, each recording the number of threads currently occupying a particular local state.

These methods are applicable to unbounded thread numbers as well, in which case the local state counters are unbounded. The fact that the DR program executed by each thread is finite-state might suggest that the resulting infinite-state counter systems can be modeled as vector addition systems (as done in [18]) or, more generally, as *well quasi-ordered transition systems* [19,20] (defined below). This would give rise to sound and complete algorithms for local-state reachability in such programs.

This strategy turns out to fail: the full class of Boolean DR programs is expressive enough to render safety checking for an unbounded number of threads undecidable:

Theorem 6. *Program location reachability for Boolean DR programs run by an unbounded number of threads is undecidable.*

Proof: by reducing the Halting problem for the Turing-powerful deterministic 2-counter Minsky machines [21] with k control states to the program location reachability problem in DR programs with 3 locations and a local variable with k values.

We demonstrate the reduction using the Minsky machine in Fig. 3; the formalism is from [22]. The machine consists of five control states $\{0, \dots, 4\}$ $(0 = \text{initial})$, two natural-number counters c_1 and c_2 (initially 0), and increment, decrement, and zero-test operations, denoted by $c_i{+}{+}$, $c_i{-}{-}$ and $c_i \overset{?}{=} 0$, respectively. Each operation changes the control state and counter value as indicated in the figure (the decrement and zero-test operation block if c is zero and non-zero, respectively).

Control states are encoded in a local variable $\mathtt{l}$ of $\tilde{\mathcal{P}}$ ranging over $0, \dots, 4$. (This makes $\tilde{\mathcal{P}}$ non-Boolean but still finite-domain.) As can be seen from the figure, these local variables are synchronized across the threads: they simulate a single shared variable that tracks the control state (see end of Sect. 3.1). Control state changes thus turn into synchronized local variable updates.

The two counters are encoded in program locations d_1, d_2 of the DR program $\tilde{\mathcal{P}}$ such that counter value c_i equals the number of threads in location d_i, for $i \in \{1, 2\}$. In addition, location d_0 is the single initial program location, thus with an unbounded number of threads; it serves as thread pool. For program counter modifications $c_i{+}{+}$ and $c_i{-}{-}$, a thread moves from d_0 to d_i and vice versa, respectively. To simulate $c_i \overset{?}{=} 0$, we test whether any active *or any passive thread* resides in location d_i.

Let finally ℓ_e be a special program location of $\tilde{\mathcal{P}}$ that is reached if and only if a local variable has the value that encodes the Minsky machine's halting control state. The machine halts if and only if program location ℓ_e is reached in $\tilde{\mathcal{P}}$. □

Note how program $\tilde{\mathcal{P}}$ used in the above proof is highly non-asynchronous. Theorem 6 implies that its unbounded-thread instantiation is not even well quasi-ordered. Can this problem be fixed for the unbounded-counter systems obtained from asynchronous programs, in order to permit a complete verification method? If so, at what cost?

4.2. Monotonicity in dual-reference programs

For a transition system $(\Sigma, \rightarrowtail)$ to be well quasi-ordered, we need two conditions to be in place [19,20,2]:

well quasi-orderedness*:* there exists a reflexive and transitive binary relation $\preceq$ on Σ such that for every infinite sequence $u_0, u_1, \ldots$ of states in Σ there exist i, j with $i < j$ and $u_i \preceq u_j$.
monotonicity*:* for any u, u', w with $u \rightarrowtail u'$ and $u \preceq w$ there exists w' such that $w \rightarrowtail w'$ and $u' \preceq w'$.

We apply this definition to the case of dual-reference programs. Representing global states of the abstract system $\tilde{\mathcal{P}}^n$ defined in Sect. 3 as counter tuples, we define $\preceq$ as

$$(n_1, \ldots, n_\Lambda) \preceq (n'_1, \ldots, n'_\Lambda) \;::\; \forall i = 1..\Lambda : n_i \leq n'_i$$

where Λ is the total number of thread-local states, i.e. the number of local variable evaluations. We now call a DR program $\tilde{\mathcal{P}}$ *monotone* if the induced infinite-state transition system $\cup_{i=1}^{\infty} \tilde{\mathcal{P}}^i$ is monotone with respect to $\preceq$. Monotonicity of $\tilde{\mathcal{P}}$ can be characterized as follows:

Lemma 7. *A DR program with transition set $\tilde{\mathcal{R}}$ is monotone exactly if, for all $n \geq 2$:*

$$(u, u') \in \tilde{\mathcal{R}}^n \quad \Rightarrow \quad \forall l_{n+1}\, \exists l'_{n+1}, \pi : \big(\langle u, l_{n+1}\rangle, \pi(\langle u', l'_{n+1}\rangle)\big) \in \tilde{\mathcal{R}}^{n+1} \,. \tag{20}$$

In (20), the quantifiers $\forall l_{n+1}, \exists l'_{n+1}$ range over local states (i.e. valuations of the local variables). The notation $\langle u, l_{n+1}\rangle$ denotes an $(n+1)$-thread state that agrees with u in the first n local states and whose last local state is l_{n+1}; similarly $\langle u', l'_{n+1}\rangle$. Symbol π denotes a permutation on $\{1, \ldots, n+1\}$ that acts on states by acting on thread indices, which effectively reorders local states.

Proof of Lemma 7. "$\Rightarrow$": suppose $\cup_{i=1}^{\infty} \tilde{\mathcal{P}}^i$ is monotone. Let $u = (l_1, \ldots, l_n)$, $u' = (l'_1, \ldots, l'_n)$ with $(u, u') \in \tilde{\mathcal{R}}^n$, and $w = \langle u, l_{n+1}\rangle$. We have $u \preceq w$, hence by the monotonicity of $\cup_{i=1}^{\infty} \tilde{\mathcal{R}}^i$ there exists w' such that (a) $(w, w') \in \cup_{i=1}^{\infty} \tilde{\mathcal{R}}^i$ and (b) $u' \preceq w'$. From (a) we conclude that in fact $(w, w') \in \tilde{\mathcal{R}}^{n+1}$. From (b) we conclude that w' contains n threads in local states as in u'. Let l'_{n+1} be the local state of the additional thread (not necessarily the $(n+1)$st) in w', and σ be a permutation such that $(l'_1, \ldots, l'_{n+1}) = \sigma(w')$. That is, σ reorders the local states of w' such that the n local states in u' come first, l'_{n+1} comes last. With $\pi := \sigma^{-1}$, we then have

$$\big(\langle u, l_{n+1}\rangle, \pi(\langle u', l'_{n+1}\rangle)\big) \;=\; \Big(\langle u, l_{n+1}\rangle, \sigma^{-1}(\langle u', l'_{n+1}\rangle)\Big) \;=\; (w, w') \;\in\; \tilde{\mathcal{R}}^{n+1} \,.$$

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

If w has $n+1$ threads, then observe that w contains n threads in local states as in u; let l_{n+1} be the local state of the additional thread (not necessarily the $n+1$st). Let further l'_{n+1} and π be as provided in (20). With $y = \langle u, l_{n+1}\rangle$ and $y' = \pi(\langle u', l'_{n+1}\rangle)$, we get $(y, y') \in \tilde{\mathcal{R}}^{n+1}$ by (20). Since y and w contain the same local states, let σ be a permutation such that $\sigma(y) = w$. Define $w' = \sigma(y')$. Then $w' \sim y' = \pi(\langle u', l'_{n+1}\rangle) \succeq u'$, where $\sim$ is symmetry equivalence. Further, $(y, y') \in \tilde{\mathcal{R}}^{n+1}$ implies $(\sigma(y), \sigma(y')) \in \tilde{\mathcal{R}}^{n+1}$ by symmetry, so $(w, w') \in \tilde{\mathcal{R}}^{n+1} \subseteq \cup_{i=1}^{\infty} \tilde{\mathcal{R}}^i$, demonstrating that the monotonicity conditions are satisfied.

The case that w has more than $n+1$ threads follows by induction. $\square$

Asynchronous programs are trivially monotone (and DR): Eq. (20) is satisfied by choosing $l'_{n+1} := l_{n+1}$ and π the identity. Table 4 shows instructions found in *non*-asynchronous programs that destroy monotonicity. For example, the swap instruction in the first row gives rise to a DR program with a 2-thread transition $(0, 0, 0, 0) \in \tilde{\mathcal{R}}^2$. Choosing $l_3 = 1$ in (20) requires the existence of a transition in $\tilde{\mathcal{R}}^3$ of the form $(\mathtt{l}_1, \mathtt{l}_2, \mathtt{l}_3, \mathtt{l}'_1, \mathtt{l}'_2, \mathtt{l}'_3) = (0, 0, 1, \pi(0, 0, \mathtt{l}'_3))$. By equations (11) and (12), there must further 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.

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

which is equivalent to $\mathtt{l}'_a = \mathtt{l}_p = \mathtt{l}_q \wedge \mathtt{l}_a = \mathtt{l}'_p = \mathtt{l}'_q$. This formula is inconsistent with the partial assignment $(0, 0, 1, \pi(0, 0, \mathtt{l}'_3))$, no matter what π and $\mathtt{l}'_3$.

Table 4
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. (21). (For ease of illustration, some of these programs are not finite-state.)

Dual-reference program		Monotonicity	
instruction	variables	mon.?	assgn. violating (21)
$\mathtt{l}, \mathtt{l}_{\mathbf{P}} := \mathtt{l}_{\mathbf{P}}, \mathtt{l}$	$\mathtt{l} \in \mathbb{B}$	no	$\mathtt{l} = 0,\ \mathtt{l}' = 1$
$\mathtt{l}, \mathtt{l}_{\mathbf{P}} := \mathtt{l} + 1, \mathtt{l}_{\mathbf{P}} - 1$	$\mathtt{l} \in \mathbb{N}$	yes	
$\mathtt{l}_{\mathbf{P}} := \mathtt{l}_{\mathbf{P}} + 1$	$\mathtt{l} \in \mathbb{N}$	yes	
$\mathtt{l} := \mathtt{l} + \mathtt{l}_{\mathbf{P}}$	$\mathtt{l} \in \mathbb{N}$	no	$\mathtt{l} = \mathtt{l}' = 1$
$\mathtt{l}_{\mathbf{P}} := c$	$\mathtt{l}, c \in \mathbb{N}$	yes	

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

$$\tilde{\mathcal{R}}^2 = \left(\neg \mathtt{b}_1 \vee \neg \mathtt{b}_1'\right) \wedge (\mathtt{b}_1 = \mathtt{b}_2) \wedge \left(\mathtt{b}_1' = \mathtt{b}_2'\right)$$
$$\tilde{\mathcal{R}}^3 = \left(\neg \mathtt{b}_1 \vee \neg \mathtt{b}_1'\right) \wedge (\mathtt{b}_1 = \mathtt{b}_2 = \mathtt{b}_3) \wedge \left(\mathtt{b}_1' = \mathtt{b}_2' = \mathtt{b}_3'\right) .$$

Consider the transition $(0, 0) \to (1, 1) \in \tilde{\mathcal{R}}^2$ and the $\tilde{\mathcal{R}}^3$-state $u = (0, 0, 1) \succ (0, 0)$: u has no successor (it is in fact inconsistent), violating monotonicity.

The monotonicity can be lost even without inter-thread predicates: the assignment $\mathtt{s} := \mathtt{l}$ abstracted with respect to the predicates $Q[1] :: \mathtt{s} = \mathtt{l}$ (single-thread) and $Q[2] :: \mathtt{l} = 4$ (local) gives rise to a non-asynchronous abstraction that is not even monotone: while $(1, 1) \to (1, 1) \in \tilde{\mathcal{R}}^1$ is a valid transition, the $\tilde{\mathcal{R}}^2$-state $(1, 1, 1, 0) \succ (1, 1)$ is inconsistent and hence has no successors.

4.3. Restoring monotonicity in the abstraction

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

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

$$\exists \tilde{L}_{\mathbf{P}} \exists \tilde{L}_{\mathbf{P}}' : \tilde{\mathcal{R}} \quad \Rightarrow \quad \forall \tilde{L}_{\mathbf{P}} \exists \tilde{L}_{\mathbf{P}}' : \tilde{\mathcal{R}} . \tag{21}$$

Unlike the monotonicity characterization given in Lemma 7, Eq. (21) is formulated only over the template program $\tilde{\mathcal{R}}$. It suggests that, if $\tilde{\mathcal{R}}$ holds for some variable assignment, then no matter how we replace the current-state passive-thread variables $\tilde{L}_{\mathbf{P}}$, we can find next-state passive-thread variables $\tilde{L}_{\mathbf{P}}'$ such that $\tilde{\mathcal{R}}$ still holds. This is true for asynchronous programs, since here $\tilde{L}_{\mathbf{P}} = \emptyset$. It fails for the swap instruction in the first row of Table 4: the instruction gives rise to the DR program $\tilde{\mathcal{R}} :: \mathtt{l}' = \mathtt{l}_{\mathbf{P}} \wedge \mathtt{l}_{\mathbf{P}}' = \mathtt{l}$. The assignment $\mathtt{l} = 0, \mathtt{l}' = 1$ in the table satisfies $\exists \tilde{L}_{\mathbf{P}} \tilde{L}_{\mathbf{P}}' \tilde{\mathcal{R}}$, but for $\mathtt{l}_{\mathbf{P}} = 0$, $\tilde{\mathcal{R}}$ is violated no matter what value is assigned to $\mathtt{l}_{\mathbf{P}}' \in \tilde{L}_{\mathbf{P}}'$.

Proof of Theorem 8. We show monotonicity using Lemma 7. Suppose $(u, u') \in \tilde{\mathcal{R}}^n$, and let l_{n+1} be given. By (11), there exists $a \in \{1, \ldots, n\}$ such that $(u, u') \in (\tilde{\mathcal{R}}_a)^n$. By (12), we have

$$\forall p \in \{1, \ldots, n\} \setminus \{a\}\ \tilde{\mathcal{R}}\{\tilde{L} \trianglerighteq \tilde{L}_a\}\{\tilde{L}_{\mathbf{P}} \trianglerighteq \tilde{L}_p\} . \tag{22}$$

Since $n \geq 2$, the quantification in Eq. (22) is not empty and hence satisfies the left-hand side of (21). By the right-hand side, there exists a valuation l_{n+1}' of all $\tilde{L}_{\mathbf{P}}'$ variables such that, replacing the $\tilde{L}_{\mathbf{P}}$ variables by the valuation l_{n+1}, $\tilde{\mathcal{R}}$ still holds, i.e. $\tilde{\mathcal{R}}\{\tilde{L} \trianglerighteq \tilde{L}_a\}\{\tilde{L}_{\mathbf{P}} \trianglerighteq \tilde{L}_{n+1}\}$. Merging this with (22), we obtain

$$\forall p \in \{1, \ldots, n+1\} \setminus \{a\}\ \tilde{\mathcal{R}}\{\tilde{L} \trianglerighteq \tilde{L}_a\}\{\tilde{L}_{\mathbf{P}} \trianglerighteq \tilde{L}_p\} ,$$

and thus $(\langle u, l_{n+1} \rangle, \langle u', l_{n+1}', \rangle) \in (\tilde{\mathcal{R}}_a)^{n+1} \subseteq \tilde{\mathcal{R}}^{n+1}$, establishing the right-hand side of (20) with the identity permutation π. □

We are now ready to transform the possibly non-monotone abstract DR program $\tilde{\mathcal{P}}$ into a monotone abstraction $\tilde{\mathcal{P}}_m$. Our solution is similar in spirit to, but different in effect from, earlier work on *monotonic abstractions* [3]. The idea is to terminate processes that violate universal guards and thus block a transition. 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 9. The **non-monotone fragment (NMF)** of a DR program with transition relation $\tilde{\mathcal{R}}$ is the following formula $\mathcal{F}(\tilde{\mathcal{R}})$ over $\tilde{L} \times \tilde{L}_{\mathbf{P}} \times \tilde{L}'$:

$$\mathcal{F}(\tilde{\mathcal{R}}) \quad :: \quad \neg\exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \;\wedge\; \exists \tilde{L}_{\mathbf{P}} \exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \,. \tag{23}$$

The NMF encodes partial assignments $(\mathtt{l}, \mathtt{l}_{\mathbf{P}}, \mathtt{l}', -)$ that cannot be extended to a full assignment satisfying $\tilde{\mathcal{R}}$, but such that $(\mathtt{l}, -, \mathtt{l}', -)$ *can* be. We observe that Eq. (23) is slightly stronger than the negation of (21): the NMF binds the values of the $\tilde{L}_{\mathbf{P}}$ variables for which a violation of $\tilde{\mathcal{R}}$ is possible.

We revisit the two non-monotone instructions from Table 4. The NMF of $\mathtt{l}, \mathtt{l}_{\mathbf{P}} := \mathtt{l}_{\mathbf{P}}, \mathtt{l}$ is $\mathtt{l}' \neq \mathtt{l}_{\mathbf{P}}$: this clearly cannot be extended to an assignment satisfying $\tilde{\mathcal{R}}$, but when $\mathtt{l}_{\mathbf{P}}$ is changed to $\mathtt{l}'$, we can choose $\mathtt{l}'_{\mathbf{P}} = \mathtt{l}$ to satisfy $\tilde{\mathcal{R}}$. The non-monotone fragment of $\mathtt{l} := \mathtt{l} + \mathtt{l}_{\mathbf{P}}$ is $\mathtt{l}' \geq \mathtt{l} \wedge \mathtt{l}' \neq \mathtt{l} + \mathtt{l}_{\mathbf{P}}$.

The NMF can be used to "repair" $\tilde{\mathcal{R}}$: the program with transition relation $\tilde{\mathcal{R}} \vee \mathcal{F}(\tilde{\mathcal{R}})$ is monotone, as we will see shortly. This suggests to add transitions $(\mathtt{l}, \mathtt{l}_{\mathbf{P}}, \mathtt{l}', \mathtt{l}'_{\mathbf{P}})$ to $\tilde{\mathcal{R}}$ that allow arbitrary passive-thread state changes whenever $(\mathtt{l}, \mathtt{l}_{\mathbf{P}}, \mathtt{l}')$ belongs to the non-monotone fragment, thus lifting the blockade previously caused by some passive threads. The problem is of course that such additions will generally modify the program behavior; in particular, an added transition might 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 $\tilde{\mathcal{P}}$ from affecting the future execution. This can be realized by redirecting them to an auxiliary sink state. Let $\ell_\perp$ be a fresh program location.

Definition 10. 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 the transition relation $\tilde{\mathcal{R}}_m \;::\; \tilde{\mathcal{R}} \vee (\mathcal{F}(\tilde{\mathcal{R}}) \wedge \mathtt{pc}'_{\mathbf{P}} = \ell_\perp)$.

Lemma 11. *The monotone closure $\tilde{\mathcal{P}}_m$ of a DR program $\tilde{\mathcal{P}}$ is monotone.*

Proof. Appealing to Theorem 8, we show that $\tilde{\mathcal{R}}_m$ satisfies (21), i.e.

$$\exists \tilde{L}_{\mathbf{P}} \exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}}_m \quad \Rightarrow \quad \forall \tilde{L}_{\mathbf{P}} \exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}}_m \,. \tag{24}$$

We first simplify the following expression occurring on both sides of (24):

$$\begin{aligned}
\exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}}_m &= \exists \tilde{L}'_{\mathbf{P}} : (\tilde{\mathcal{R}} \vee (\neg\exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \;\wedge\; \exists \tilde{L}_{\mathbf{P}} \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \;\wedge\; \mathtt{pc}'_{\mathbf{P}} = \ell_\perp)) \\
&= \exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \vee (\neg\exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \;\wedge\; \exists \tilde{L}_{\mathbf{P}} \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \;\wedge\; \exists \tilde{L}'_{\mathbf{P}} : \mathtt{pc}'_{\mathbf{P}} = \ell_\perp) \\
&= \exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \vee (\neg\exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \;\wedge\; \exists \tilde{L}_{\mathbf{P}} \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}}) \\
&= \exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \vee \exists \tilde{L}_{\mathbf{P}} \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \\
&= \exists \tilde{L}_{\mathbf{P}} \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \,.
\end{aligned}$$

Eq. (24) is now seen to be valid: the above derivation shows that the expression $\exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}}_m$ is semantically independent of $\tilde{L}_{\mathbf{P}}$ variables. Hence the quantifications $\exists \tilde{L}_{\mathbf{P}}$ and $\forall \tilde{L}_{\mathbf{P}}$ on both sides of (24) can be omitted, making the two sides equal. □

Sect. 4 so far reports results on arbitrary DR programs. We now return to DR programs $\tilde{\mathcal{P}}$ that are obtained via predicate abstraction from asynchronous input programs, as described in Sect. 3. It is these types of DR programs for which we can guarantee that the monotone closure is not only monotone, but equivalent to $\tilde{\mathcal{P}}$ in the sense of safety preservation.

To state this result, we stipulate that the abstract program $\tilde{\mathcal{P}}$ inherit the error location from $\mathcal{P}$ (predicate abstraction preserves control locations). Also note that location $\ell_\perp$ is "fresh" and hence different from the error location.

Theorem 12. *Let $\mathcal{P}$ be an asynchronous program, and $\tilde{\mathcal{P}}$ its parametric predicate abstraction (Corollary 5). Given $\tilde{\mathcal{P}}$'s monotone closure $\tilde{\mathcal{P}}_m$, $(\tilde{\mathcal{P}}_m)^n$ is safe exactly if $\tilde{\mathcal{P}}^n$ is.*

Proof. Since $\tilde{\mathcal{R}} \Rightarrow \tilde{\mathcal{R}}_m$, if $(\tilde{\mathcal{P}}_m)^n$ is safe, so is $\tilde{\mathcal{P}}^n$. For the converse argument, we assume $\tilde{\mathcal{P}}^n$ is safe and consider a path π in $(\tilde{\mathcal{P}}_m)^n$, split into segments as follows:

$$\pi \;=\; t_1, \ldots, r_1 \,,\; t_2, \ldots, r_2 \,,\; \ldots \,,\; t_j, \ldots, r_j \,,\; \ldots \tag{25}$$

where the transitions in each segment $t_j, \ldots, r_j$ belong to relation $\tilde{\mathcal{R}}$, and the transitions (r_j, t_{i+1}) belong to $\mathcal{F}(\tilde{\mathcal{R}}) \wedge \mathtt{pc}'_{\mathbf{P}} = \ell_\perp$.

Call a state *safe* if there is no path from that state to an error state. State safety is closed under reachability: if state s is safe, any state reachable from s is also safe. Moreover, since the asynchronous input program $\mathcal{P}$ is monotone, state safety is downward-closed for $\mathcal{P}$: if state r is safe in $\mathcal{P}$ and $s \preceq r$ then s is also safe. This property also holds for the (possibly non-monotone) abstract DR program $\tilde{\mathcal{P}}$: let R be the concretization of state $\tilde{r}$ of $\tilde{\mathcal{P}}$, i.e. a set of programs states of input program $\mathcal{P}$. Then $\tilde{\mathcal{P}}$'s overapproximation properties (Sect. 2.3 and Corollary 5) guarantee the safety of states in R, and downward-closedness of state safety in $\mathcal{P}$ implies the safety of states in the downward closure of R. From the fact that s's concretization is in that closure we conclude s is safe, too.

Using this result we now show that all states along trace π in Eq. (25) are safe. The proof is by induction on the number of $\tilde{\mathcal{R}}$-segments in π. The initial segment $t_1, \ldots, r_1$ is a path in $\tilde{\mathcal{P}}^n$ and hence no state in it leads to an error state ($\tilde{\mathcal{P}}^n$ is safe). For the inductive step, consider segment $t_i, \ldots, r_i$ and transition (r_i, t_{i+1}). By the definition of segments, state t_{i+1} contains at least one passive thread "stuck" in $\mathtt{pc}'_{\mathbf{P}} = \ell_\perp$. Let r'_i be the state obtained from r_i by removing stuck threads; hence $r'_i \prec r_i$. Since safety is downward-closed in $\tilde{\mathcal{P}}$ and r_i is safe by the induction hypothesis, r'_i is safe, too, and hence all states in segment $t_{i+1}, \ldots, r_{i+1}$, by reachability closure. □

Theorem 12 justifies our strategy for reachability analysis of an asynchronous program $\mathcal{P}$: form its parametric predicate abstraction $\tilde{\mathcal{P}}$ described in Sections 2 and 3, build the monotone closure $\tilde{\mathcal{P}}_m$, and analyze $(\tilde{\mathcal{P}}_m)^\infty$ using any technique for monotone systems. Applying this strategy to the ticket algorithm yields a well quasi-ordered transition system of about 28 local states and 50 transitions (see www.cprover.org/bfc/ticketlock/ticketabs.tts for the full spec). The backward reachability method described in [20] returns "uncoverable" on this system, confirming that the ticket algorithm ensures mutual exclusion, this time *for arbitrary thread counts*.

As a more compact example, we revisit again the decrement instruction $\mathtt{l} := \mathtt{l} - 1$, abstracted against the inter-thread predicate $Q :: \mathtt{l} = \mathtt{l}_{\mathbf{P}}$. The DR template program is:

$$\tilde{\mathcal{R}} \quad :: \quad \mathtt{b} = \mathtt{b}_{\mathbf{P}} \wedge \mathtt{b}' = \mathtt{b}'_{\mathbf{P}} \wedge (\neg \mathtt{b} \vee \neg \mathtt{b}') \,. \tag{26}$$

As illustrated near the end of Sect. 4.2, $\tilde{\mathcal{R}}$ does not give rise to a monotone system: we have $(0,0) \to (1,1) \in \tilde{\mathcal{R}}^2$, but the $\tilde{\mathcal{R}}^3$-state $(0,0,1) \succ (0,0)$ has no successor: $\mathtt{b} = \mathtt{b}_{\mathbf{P}}$ is violated for some passive threads, no matter who is active. The NMF $\neg \exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}} \;\wedge\; \exists \tilde{L}_{\mathbf{P}} \exists \tilde{L}'_{\mathbf{P}} : \tilde{\mathcal{R}}$ simplifies to $\mathtt{b} \neq \mathtt{b}_{\mathbf{P}}$. The closure operation adds to $\tilde{\mathcal{R}}$ all transitions satisfying $\mathtt{b} \neq \mathtt{b}_{\mathbf{P}} \wedge \mathtt{pc}'_{\mathbf{P}} = \ell_\perp$. Now we have for example $(0,0,1) \to (\perp,\perp,0) \in \tilde{\mathcal{R}}^3$ ($\perp$ = any state satisfying $\mathtt{pc}'_{\mathbf{P}} = \ell_\perp$) – the monotonicity is restored.

5. Practical evaluation

We evaluate our approach on a set of 37 non-recursive shared-memory C programs. 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 description Threads communicate asynchronously through shared memory and synchronize via locks or atomic compare-and-swap instructions. We also experimented with monotone input programs that are not asynchronous, featuring more complex synchronization primitives such as condition variables and broadcasts.[3]

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. (Those that fail usually fail for small thread counts, requiring little verification resources.) 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. We briefly describe the program benchmarks:

1–10: *thread-safe algorithms:* atomic counters (1–2); operations to 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). For each type, we consider a version with Locks, and one with Compare-and-swap primitives (marked -L or -C in Table 5, resp.).

11–15: *OS code:* 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:* multiple locks protecting a shared resource (16–18); the ticket algorithm in Fig. 1 (19); classical algorithms due to Szymanski, Peterson and Dekker (20–22); a readers-writers and timed mutex (23–24); high-contention ticket algorithm with proportional back-off (25); test-and-set lock (26).

[3] All technical results described in this paper hold in fact for all monotone DR programs; note that the proofs of the key results in Lemma 11 and Theorem 12 depend on monotonicity but not on asynchrony. For non-asynchronous input programs the stabilization bound $\mathtt{b}$ increases from $2 \times (\#_{IT} + 1)$ to $2 \times (2 \times \#_{IT} + 1)$, which is the first bound established in the proof of Theorem 4.

27–32: *misc:* two programs that require single-thread predicates (27–28); threads synchronizing 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 using the C POSIX Thread library.

Most programs contain procedures run by an arbitrary number of threads dynamically spawned by the (single) initial thread. Exceptions are 20–24 and 31 from [8], which are designed for a fixed thread count of two; the program behavior stabilizes for $n \geq 2$. These examples do not exploit the power of our approach to deal with unbounded threads.

Implementation We implemented our abstraction method in the verifier monabs. The verifier takes a shared-memory program annotated with assertions in the C language, which uses the pthreads library for dynamic thread creation and synchronization objects (mutexes and condition variables). Our tool monabs combines medium-precision Cartesian abstractions [23] (default cube length is 3) and handles function calls by inlining. If the abstractions lack precision, monabs performs a monotonicity-preserving variant of Das/Dill-style refinement. This is achieved by constraining passive-thread variables in the abstraction only if needed, i.e. if the spurious transition is not spurious for all valuations of passive-thread variables. Predicate discovery is done in a CEGAR loop: initially the predicate set is empty; the Boolean program represents only the control flow. Predicates are then discovered based on spurious counterexamples; basic discovery heuristics are sufficient for the benchmarks. Since existing well quasi-ordered system model checkers do not support monotone Boolean DR programs, we use an extension of the breach tool [24] as verification back-end.

5.1. Detailed evaluation of monabs

Table 5 gives detailed experimental results. 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, among the various CEGAR phases. The cost of the monotone closure computation is negligible and not shown.

Shared and single-thread predicates were needed to succeed in 28 of the 37 cases, and inter-thread predicates for the two ticket algorithms. Roughly half of the benchmarks exhibit abstractions that are "truly DR", i.e. they permit no asynchronous encoding. As a consequence, existing model checkers for concurrent software are inapplicable. In five of these cases (marked in the *Cnd?* column), 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. For the other 11, originally asynchronous programs, asynchrony is lost after inter- or single-thread predicates are discovered during refinement: programs with $IT \neq 0$, and programs with $ST > L$. Only 5 programs whose abstraction is "truly DR" turned out monotone; the majority, namely 11, *require* the application of the closure operation from Definition 10 in order to be passed to the back-end model checker (recall that loss of monotonicity is possible even without any inter-thread predicates). Single-thread predicates of the form $\mathtt{s} = \mathtt{1}$ are, e.g., required to track the success of the compare-and-swap primitive for programs Inc-C, MaxSimp-C and MaxOpt-C.

The results also demonstrate the appropriateness of our predicate language: banning any one of the predicate types supported by our approach renders some of our C programs unprovable.

5.2. Comparison with symmpa *and* cream

Existing tools for programs with unbounded threads, such as duet, are insufficient for our benchmarks (Sect. 6 has details). Instead, we compare against four recent fixed-thread approaches and measure at which point the search for monotone, unbounded-thread proofs pays off compared to checking increasing constant thread counts. Our points of comparison are:

symmpa ([9]): predicate abstraction for fixed threads.
cream-mono ([25,26]): cream with monolithic proofs.
cream-rely ([8,26]): cream with rely-guarantee proofs.
cream-owicki ([25,26]): cream with Owicki–Gries proofs.

Front-end capabilities of cream and symmpa are similar to that of monabs, facilitating a comparison. Neither cream nor symmpa support broadcasts, as used by 5 of our benchmarks; we instead apply them to broadcast-free overapproximations.

Fig. 4 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) gives the time t it took to solve the k easiest (for the given method) of the C programs. The results show that our unbounded approach quickly outperforms each of the fixed-thread verifiers, even for very small 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 algorithm (Ticket-Hc): symmpa is unable to track the uniqueness of a ticket, and as a result times out while attempting to enumerate

Table 5

Benchmark characteristics and results — *S*, *L* = # of shared, local vars; *LOC* = lines of code; *Mtx?*, *Cnd?*, *Safe?* = mutexes present?, condition variables?, program safe? (● = "yes"); *SP*, *ST*, *L*, *IT* = # of shared, single-thread, local, inter-thread predicates; *Its* = # of CEGAR iterations; *DR?*, *Mon?* = abstractions truly DR?, monotone?; *Abs*, *Ref*, *Chk* = time for abstraction, refinement, model checking; *Total* = total time.

Program		Characteristics						Predicates				CEGAR phases and times (in sec.)						*Total*
		S	*L*	*LOC*	*Mtx?*	*Cnd?*	*Safe?*	*SP*	*ST*	*L*	*IT*	*Its*	*DR?*	*Mon?*	*Abs*	*Ref*	*Chk*	
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	.1	174.5	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	mutex algorithms	5	2	89	○	○	○	5	2	2	0	8	○	–	.0	.1	.7	.9
16/Double-1		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

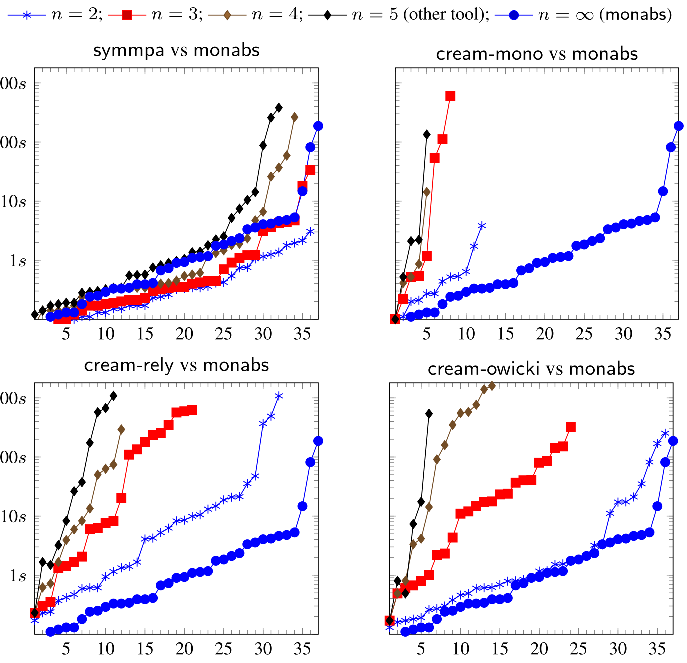

Fig. 4. Comparison with fixed-thread tools — Cactus plots comparing monabs with four recent fixed-thread proof methods.

Table 6
Comparison with existing methods — Input features: ∞, *CV*, *SM*, *ASST* = 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, **absence** of false positives, asymptotic run time for the ticket algorithm.

Verifier	Input				Relationships			Output		
	∞	*CV*	*SM*	*ASST*	*SP*	*ST*	*IT*	*CEX*	¬*FP*	*TA*(n)
cream [8,25]	○	○	●	●	●	●	●	●	●	expon.
symmpa [9]	○	○	●	●	●	●	○	●	●	expon.
iDFG (unimpl.) [11]	○	○	●	●	●	●	●	●	●	quadr.
duet [10]	●	○	●	●	●	○	○	○	○	false pos.
boppo/satabs [6]	●	○	●	●	●	○	○	●	○	N/A
ddv/satabs [7]	○	○	●	●	●	○	○	●	●	N/A
blast [4]	○	○	●	○	N/A	N/A	N/A	●	●	N/A
magic [5]	○	○	○	●	N/A	N/A	N/A	●	●	N/A
monabs (this work)	●	●	●	●	●	●	●	●	●	constant

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.

The complete set of benchmark programs used in this section is available online, at www.cprover.org/bfc. Our tool monabs is tightly integrated into the satabs verifier (www.cprover.org/satabs). For information on the novel satabs features, and on using breach as model-checker back-end, visit www.cprover.org/bfc.

6. Comparison with related work

Algorithmic solutions for verifying safety properties of multi-threaded programs have been intensively studied. We survey work related to concurrent program verification in general, and monotonicity in particular.

Existing approaches for verifying asynchronous shared-memory programs typically do not exploit the monotone structure that source-level multi-threaded programs often naturally exhibit [4–11]. Table 6 provides a feature comparison of our work with these methods. For example, the constraint-based approach in [8], implemented in cream, generates Owicki–Gries and rely-guarantee type proofs. 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 proofs by prioritizing predicates that do not refer to the local variables of other threads. For the parametric benchmarks we used in Sect. 5, however, this was never successful.

A CEGAR approach for fixed-thread symmetric concurrent programs has been implemented in symmpa [9]. It uses predicate abstraction to generate a Boolean broadcast program (a special case of DR program). Their approach cannot reason about relationships between local variables across threads, which is crucial for verifying the ticket lock algorithm. Nevertheless, even the restricted predicate language of [9] can give rise to non-asynchronous programs. (This possibility is acknowledged but not addressed in [9].) As a result, their technique cannot be extended to unbounded thread counts with well quasi-ordered systems techniques.

Recent work on data flow graph representations of fixed-thread concurrent programs has been applied to safety property verification [11]. 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 [11], the technique in [10] uses data flow graphs to compute invariants of concurrent programs with unbounded threads (implemented in duet). In contrast to monabs, duet constructs proofs from relationships between either solely shared or solely local variables. These are insufficient for most benchmarks we used in Sect. 5.

The ticket lock algorithm is unbounded both in the number of participating threads, and in the domain of program variables; a program class targeted in early work by Bozzano and Delzanno [27]. They present a sound symbolic backward search method that does not rely on monotonicity of the input programs and thus is not guaranteed to terminate. An abstraction can guarantee termination at the cost of overapproximation; a refinement loop can presumably re-introduce non-termination issues. The technique works well for the ticket algorithm, but has apparently not been applied to other programs; an implementation is not available.

Predicates that, like our inter-thread predicates, reason over all participating processes/threads have been used extensively in invariant generation methods [28–30]. As a recent example, an approach that relies on abstract interpretation instead of model checking is [31]. Starting with a set of candidate invariants (assertions), the approach builds a *reflective abstraction*, from which invariants of the concrete system are obtained in a fixed point process. These approaches and ours share the insight that complex relationships over all threads may be required to prove easy-to-state properties such as mutual exclusion. They differ fundamentally in the way these relationships are used: abstraction with respect to a given set $\mathcal{Q}$ of quantified predicates determines the strongest invariant expressible as a Boolean formula over the set $\mathcal{Q}$; the result is unlikely to be expressible in the language that defines $\mathcal{Q}$.

The monotonicity property of asynchronous (and other) programs is often exploited in infinite-state search algorithms to ensure termination. In the absence of monotonicity, an option is to detect *cutoffs* [32], i.e. bounds on the number of processes sufficient to guarantee correctness for the unbounded case. The option of "making" systems monotone was pioneered in earlier work [33,3]. Bingham and Hu deal with guards that require universal quantification over thread indices, by transforming such systems into broadcast protocols. 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". This happens via a designated local state that isolates such threads from participation in the future computation. The same idea was further developed by Abdulla et al. in the context of *monotonic abstractions*. Our solution to the loss of monotonicity was in some way inspired by these works, but differs in two crucial aspects: first, our concrete input systems are asynchronous and thus monotone, so our incentive to *preserve* monotonicity in the abstract is strong. Second, exploiting the input monotonicity, we can achieve a monotonic abstraction that is safety-equivalent to the non-monotone abstraction and thus not merely an error-preserving approximation. This is essential, to avoid spurious counterexamples in addition to those unavoidably introduced by the predicate abstraction.

7. Concluding remarks

We presented in this paper a comprehensive verification method for arbitrarily-threaded asynchronous shared-variable programs. Our method is based on predicate abstraction and permits expressive universally quantified *inter-thread* predicates, which track relationships such as "my ticket number is the smallest among all threads". Such predicates are required to verify, via predicate abstraction, the widely used ticket lock algorithm. We found that the abstractions with respect to these predicates result replicated Boolean programs lacking *monotonicity*, a property often relied upon during infinite-state system verification. To fix this problem, we strengthened the earlier method of monotonic abstractions such that it does not introduce spurious errors into the abstraction. We have implemented our technique in the monabs verifier and experimentally demonstrated the efficiency of our unbounded-thread analysis compared to several earlier methods, both unbounded and fixed-thread.

We view the treatment of monotonicity as the major contribution of this work. Program design often naturally gives rise to "monotone concurrency", where adding passive components cannot disable existing actions. Abstractions that interfere with this feature are limited in usefulness. Our paper shows how the feature can be inexpensively restored, allowing such abstraction methods and powerful infinite-state verification methods to coexist peacefully.

References

[1] G.R. Andrews, Concurrent Programming: Principles and Practice, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[2] P.A. Abdulla, K. Cerans, B. Jonsson, Y.-K. Tsay, General decidability theorems for infinite-state systems, in: LICS, 1996, pp. 313–321, http://dblp.uni-trier.de/db/conf/lics/lics96.html#AbdullaCJT96.

[3] P.A. Abdulla, G. Delzanno, A. Rezine, Monotonic abstraction in parameterized verification, Electron. Notes Theor. Comput. Sci. 223 (2008) 3–14, http://dblp.uni-trier.de/db/journals/entcs/entcs223.html#AbdullaDR08.

[4] T.A. Henzinger, R. Jhala, R. Majumdar, Race checking by context inference, in: PLDI, 2004, pp. 1–13, http://dblp.uni-trier.de/db/conf/pldi/pldi2004.html#HenzingerJM04.

[5] S. Chaki, E.M. Clarke, N. Kidd, T.W. Reps, T. Touili, Verifying concurrent message-passing C programs with recursive calls, in: TACAS, 2006, pp. 334–349, http://dblp.uni-trier.de/db/conf/tacas/tacas2006.html#ChakiCKRT06.

[6] B. Cook, D. Kroening, N. Sharygina, Verification of Boolean programs with unbounded thread creation, Theor. Comput. Sci. 388 (1–3) (2007) 227–242, http://dblp.uni-trier.de/db/journals/tcs/tcs388.html#CookKS07.

[7] T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher, Model checking concurrent Linux device drivers, in: ASE, 2007, pp. 501–504, http://dblp.uni-trier.de/db/conf/kbse/ase2007.html#WitkowskiBKW07.

[8] A. Gupta, C. Popeea, A. Rybalchenko, Predicate abstraction and refinement for verifying multi-threaded programs, in: POPL, 2011, pp. 331–344, http://dblp.uni-trier.de/db/conf/popl/popl2011.html#GuptaPR11.

[9] A.F. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, T. Wahl, Counterexample-guided abstraction refinement for symmetric concurrent programs, Form. Methods Syst. Des. 41 (1) (2012) 25–44, http://dblp.uni-trier.de/db/journals/fmsd/fmsd41.html#DonaldsonKKTW12.

[10] A. Farzan, Z. Kincaid, Verification of parameterized concurrent programs by modular reasoning about data and control, in: POPL, 2012, pp. 297–308, http://dblp.uni-trier.de/db/conf/popl/popl2012.html#FarzanK12.

[11] A. Farzan, Z. Kincaid, A. Podelski, Inductive data flow graphs, in: POPL, 2013, pp. 129–142, http://dblp.uni-trier.de/db/conf/popl/popl2013.html#FarzanKP13.

[12] A. Farzan, Z. Kincaid Duet, Static analysis for unbounded parallelism, in: CAV, 2013, pp. 191–196, http://dblp.uni-trier.de/db/conf/cav/cav2013.html#FarzanK13.

[13] A. Malkis, Cartesian abstraction and verification of multithreaded programs, Ph.D. thesis, University of Freiburg, 2010, http://www.freidok.uni-freiburg.de/volltexte/7356/.

[14] E.M. Clarke, O. Grumberg, D.E. Long, Model checking and abstraction, ACM Trans. Program. Lang. Syst. 16 (5) (1994) 1512–1542, http://dblp.uni-trier.de/db/journals/toplas/toplas16.html#ClarkeGL94.

[15] S. Graf, H. Saïdi, Construction of abstract state graphs with PVS, in: CAV, 1997, pp. 72–83, http://dblp.uni-trier.de/db/conf/cav/cav97.html#GrafS97.

[16] S.L. Torre, P. Madhusudan, G. Parlato, Model-checking parameterized concurrent programs using linear interfaces, in: CAV, 2010, pp. 629–644, http://dblp.uni-trier.de/db/conf/cav/cav2010.html#TorreMP10.

[17] K. Dräger, A. Kupriyanov, B. Finkbeiner, H. Wehrheim, SLAB: a certifying model checker for infinite-state concurrent systems, in: TACAS, 2010, pp. 271–274, http://dblp.uni-trier.de/db/conf/tacas/tacas2010.html#DragerKFW10.

[18] S.M. German, A.P. Sistla, Reasoning about systems with many processes, J. ACM 39 (3) (1992) 675–735, http://dblp.uni-trier.de/db/journals/jacm/jacm39.html#GermanS92.

[19] A. Finkel, P. Schnoebelen, Well-structured transition systems everywhere!, Theor. Comput. Sci. 256 (1–2) (2001) 63–92, http://dblp.uni-trier.de/db/journals/tcs/tcs256.html#FinkelS01.

[20] P.A. Abdulla, Well (and better) quasi-ordered transition systems, Bull. Symb. Log. 16 (4) (2010) 457–515, http://dblp.uni-trier.de/db/journals/bsl/bsl16.html#Abdulla10.

[21] M.L. Minsky, Computation: Finite and Infinite Machines, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[22] P. Schnoebelen, Lossy counter machines decidability cheat sheet, in: RP, 2010, pp. 51–75, http://dblp.uni-trier.de/db/conf/rp/rp2010.html#Schnoebelen10.

[23] T. Ball, A. Podelski, S.K. Rajamani, Boolean and Cartesian abstraction for model checking C programs, in: TACAS, 2001, pp. 268–283, http://dblp.uni-trier.de/db/conf/tacas/tacas2001.html#BallPR01.

[24] A. Kaiser, D. Kroening, T. Wahl, Efficient coverability analysis by proof minimization, in: CONCUR, 2012, pp. 500–515, http://dblp.uni-trier.de/db/conf/concur/concur2012.html#KaiserKW12, 2012.

[25] A. Gupta, C. Popeea, A. Rybalchenko, Threader: a constraint-based verifier for multi-threaded programs, in: CAV, 2011, pp. 412–417, http://dblp.uni-trier.de/db/conf/cav/cav2011.html#GuptaPR11.

[26] A. Gupta, C. Popeea, A. Rybalchenko, The cream tool, www.model.in.tum.de/~popeea/research/threader.html.

[27] M. Bozzano, G. Delzanno, Beyond parameterized verification, in: Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8–12, 2002, 2002, pp. 221–235.

[28] T. Arons, A. Pnueli, S. Ruah, J. Xu, L.D. Zuck, Parameterized verification with automatically computed inductive assertions, in: CAV, 2001, pp. 221–234, http://dblp.uni-trier.de/db/conf/cav/cav2001.html#AronsPRXZ01.

[29] C. Flanagan, S. Qadeer, Predicate abstraction for software verification, in: POPL, 2002, pp. 191–202, http://dblp.uni-trier.de/db/conf/popl/popl2002.html#FlanaganQ02.

[30] S.K. Lahiri, R.E. Bryant, Constructing quantified invariants via predicate abstraction, in: VMCAI, 2004, pp. 267–281, http://dblp.uni-trier.de/db/conf/vmcai/vmcai2004.html#LahiriB04.

[31] A. Sánchez, S. Sankaranarayanan, C. Sánchez, B.-Y.E. Chang, Invariant generation for parametrized systems using self-reflection (extended version), in: SAS, 2012, pp. 146–163, http://dblp.uni-trier.de/db/conf/sas/sas2012.html#SanchezSSC12.

[32] P.A. Abdulla, F. Haziza, L. Holík, All for the price of few, in: Verification, Model Checking, and Abstract Interpretation, Proceedings of the 14th International Conference, VMCAI 2013, Rome, Italy, January 20–22, 2013, 2013, pp. 476–495.

[33] J.D. Bingham, A.J. Hu, Empirically efficient verification for a class of infinite-state systems, in: TACAS, 2005, pp. 77–92, http://dblp.uni-trier.de/db/conf/tacas/tacas2005.html#BinghamH05.