Lost in Abstraction:
Monotonicity in Multi-Threaded Programs[*]

Alexander Kaiser[1], Daniel Kroening[1], and Thomas Wahl[2]

[1] University of Oxford, United Kingdom
[2] Northeastern University, Boston, United States

Abstract. *Monotonicity* in concurrent systems stipulates that, in any global state, extant system actions remain executable when new processes are added to the state. This concept is not only natural and common in multi-threaded software, but also useful: if every thread's memory is finite, monotonicity often guarantees the decidability of safety property verification even when the number of running threads is unknown. In this paper, we show that the act of obtaining finite-data thread abstractions for model checking can be at odds with monotonicity: Predicate-abstracting certain widely used monotone software results in non-monotone multi-threaded Boolean programs — the monotonicity is *lost in the abstraction*. As a result, well-established sound and complete safety checking algorithms become inapplicable; in fact, safety checking turns out to be undecidable for the obtained class of unbounded-thread Boolean programs. We demonstrate how the abstract programs can be modified into monotone ones, without affecting safety properties of the non-monotone abstraction. This significantly improves earlier approaches of enforcing monotonicity via overapproximations.

1 Introduction

This paper addresses non-recursive procedures executed by multiple threads (e.g. dynamically generated, and possibly unbounded in number), which communicate via shared variables or higher-level mechanisms such as mutexes. 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 utility of *predicate abstraction* as a safety analysis method 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 propose in this paper an extension of predicate abstraction to multi-threaded programs that enables reasoning about intricate data relationships, namely

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 [4] (the default locking mechanism in the Linux kernel since 2008; see Fig. 1), the verification of elementary safety properties **requires** single- and inter-thread relationships. 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 (un-bounded-thread) program verification with respect to our rich predicate language. Such reasoning requires first that the n-thread abstract program $\hat{\mathcal{P}}^n$, obtained by existential inter-thread predicate abstraction of the n-thread concrete program $\mathcal{P}^n$, is rewritten into a single 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 must itself permit variables that refer to the currently executing or a generic passive thread; we call such programs *dual-reference (DR)*. We describe how to obtain $\tilde{\mathcal{P}}$, namely essentially as an overapproximation of $\hat{\mathcal{P}}^b$, for a constant b that scales linearly with the number of inter-thread predicates used in the predicate abstraction.

Given the *Boolean* dual-reference program $\tilde{\mathcal{P}}$, we might now 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 and in fact block the system. Since the input C programs are, by contrast, perfectly 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 $\tilde{\mathcal{P}}_m$ engenders 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 those earlier approaches, which *enforce* (rather than restore) monotonicity in genuinely non-monotone systems, we exploit the fact that the input programs are monotone. As a result, the monotonicity closure $\tilde{\mathcal{P}}_m$ can be shown to be *safety-equivalent* to the intermediate program $\tilde{\mathcal{P}}$.

To summarize, the central contribution of this paper is a predicate abstraction strategy for 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. To this end, we first develop such a strategy for a fixed number of threads. Second, in preparation for extending it to the unbounded case, we describe how the abstract model, obtained by existential predicate abstraction for a given thread count n, can be expressed as a template program that can be multiply instantiated. Third, we show a sound and complete algorithm for reachability analysis for the obtained parameterized Boolean dual-reference programs. We

```
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 integer components: s holds the ticket currently served (or, if none, the ticket served next), while t holds the ticket to be served after all waiting threads have had access.
To request access to the locked region, a thread atomically retrieves the value of t and then increments t. The thread then busy-waits ("spins") until local variable l agrees with shared s. To unlock, a thread increments s.

See [21] for more intuition.

Fig. 1: Our goal is to verify "unbounded-thread mutual exclusion": no matter how many threads try to acquire and release the lock concurrently, no two of them should simultaneously be between the calls to functions spin_lock and spin_unlock.

overcome the undecidability of the problem by building a monotone closure that enjoys the same safety properties as the original abstract dual-reference program.

We omit in this submission practical aspects such as predicate discovery, the algorithmic construction of the abstract programs, and abstraction refinement. In our technical report [21], we provide, however, an extensive appendix, with proofs of all lemmas and theorems.

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

2.1.1 Asynchronous Programs An *asynchronous program* $\mathcal{P}$ allows only one thread at a time to change its local state. 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 $V' = \{v' : v \in V\}$. Further, the initial states are characterized by the initial 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 step).

As usual, the computation may be controlled by a local program counter `pc`, and 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 *uniformly indexed* if its variables either all have no index, or all have the same index. For a formula f and two uniformly-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 $f\{X_1 \triangleright\!\!\triangleright X_2\}$ 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 $(\mathtt{l}' = \mathtt{l} + \mathtt{s})\{L \triangleright\!\!\triangleright L_a\} = (\mathtt{l}'_a = \mathtt{l}_a + \mathtt{s})$. Finally, let $X \stackrel{\circ}{=} X'$ stand for $\forall x \in X : x = x'$.

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

$$\mathcal{P}^n = (\mathcal{R}^n, \mathcal{I}^n) = \left(\bigvee_{a=1}^{n} (\mathcal{R}_a)^n, \ \bigwedge_{a=1}^{n} \mathcal{I}\{L \triangleright L_a\} \right) \tag{1}$$

where
$$(\mathcal{R}_a)^n :: \mathcal{R}\{L \triangleright\!\!\triangleright L_a\} \wedge \bigwedge_{p:p \neq a} L_p \stackrel{\circ}{=} L'_p . \tag{2}$$

Formula $(\mathcal{R}_a)^n$ asserts that the shared variables, and the variables of the *active* (executing) thread a are updated according to $\mathcal{R}$, while the local variables of passive threads $p \neq a$ are not modified (p ranges over $\{1, \ldots, n\}$). A state is *initial* if all threads are in a state satisfying $\mathcal{I}$. An *n-thread execution* is a sequence of n-thread program states whose first state satisfies $\mathcal{I}^n$ and whose consecutive states are related by $\mathcal{R}^n$. We assume the existence of an error location in $\mathcal{P}$; an *error state* is one where some thread resides in the error location. $\mathcal{P}$ is *safe* if no execution exists that ends in an error state. Mutex conditions can be checked using a ghost semaphore and redirecting threads to the error location if they try to access the critical section while the semaphore is set.

2.1.2 Predicate Language We extend the predicate language from [10] to allow the use of the *passive-thread variables* $L_P = \{ \mathtt{l}_P : \mathtt{l} \in L \}$, each of which 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_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_P, and **inter-thread** if it contains variables from L and from L_P.*

Single- and inter-thread prediactes may contain variables from S. For example, in the ticket algorithm (Fig. 1), with $S = \{\mathtt{s}, \mathtt{t}\}$ and $L = \{\mathtt{l}\}$, examples of shared, local, single- and inter-thread predicates are: $\mathtt{s} = \mathtt{t}$, $\mathtt{l} = 5$, $\mathtt{s} = \mathtt{l}$ and $\mathtt{l} \neq \mathtt{l}_P$, respectively.

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

$$Q[i]_a \quad :: \quad \bigwedge_{p:p \neq a} Q[i]\{L \triangleright L_a\}\{L_P \triangleright L_p\} . \tag{3}$$

As special cases, for single-thread and shared predicates (no L_P variables), we have $Q[i]_a = Q[i]\{L \triangleright L_a\}$ and $Q[i]_a = Q[i]$, resp. We write $v \models Q[i]_a$ if $Q[i]_a$ holds in

state v. Predicates $Q[i]$ give rise to an abstraction function α, mapping each n-thread program state v to an $m \times n$ bit matrix with entries

$$\alpha(v)_{i,a} = \begin{cases} \texttt{T} & \text{if } v \models Q[i]_a \\ \texttt{F} & \text{otherwise}. \end{cases} \tag{4}$$

Function α partitions the n-thread program state space via m predicates into $2^{m \times n}$ equivalence classes. As an example, consider the inter-thread predicates $\texttt{l} \leq \texttt{l}_P$, $\texttt{l} > \texttt{l}_P$, and $\texttt{l} \neq \texttt{l}_P$ for a local variable $\texttt{l}$, $n = 4$ and the state $v :: (\texttt{l}_1, \texttt{l}_2, \texttt{l}_3, \texttt{l}_4) = (4, 4, 5, 6)$:

$$\alpha(v) = \begin{pmatrix} \texttt{T T F F} \\ \texttt{F F F T} \\ \texttt{F F T T} \end{pmatrix}. \tag{5}$$

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

Inter-thread predicates and abstraction Predicates that reason universally about threads have been used successfully as targets in (inductive) invariant generation procedures [5, 24]. In this paper we discuss their role in abstractions. The use of these fairly expressive and presumably expensive predicates is not by chance: automated methods that cannot reason about them [13, 10, 26] essentially fail for the ticket algorithm in Fig. 1: for a fixed number of threads that concurrently and repeatedly (e.g. in an infinite loop) request and release lock ownership, the inter-thread relationships need to be "simulated" via enumeration, incurring very high time and space requirements, even for a handful of threads. In the unbounded-thread case, they diverge. This is essentially due to known limits of thread-modular and Owicki-Gries style proof systems, which do not have access to inter-thread predicates [23]. In [21], we show that the number of *single-thread* predicates needed to prove correctness of the ticket algorithm depends on n, from which unprovability in the unbounded case follows.

2.2 Existential Inter-Thread Predicate Abstraction

Embedded into our formalism, the goal of *existential predicate abstraction* [8, 18] is to derive an abstract program $\hat{\mathcal{P}}^n$ by treating the equivalence classes induced by Eq. (4) as abstract states. $\hat{\mathcal{P}}^n$ thus has $m \times n$ Boolean variables:

$$\hat{V}_n = \bigcup_{a=1}^{n} \hat{L}_a = \bigcup_{a=1}^{n} \{b[i]_a : 1 \leq i \leq m\}.$$

Variable $b[i]_a$ tracks the truth of predicate $Q[i]$ for active thread a. This is formalized in (6), relating concrete and abstract n-thread states (valuations of V_n and $\hat{V}_n$, resp.):

$$\mathcal{D}^n :: \bigwedge_{i=1}^{m} \bigwedge_{a=1}^{n} b[i]_a \Leftrightarrow Q[i]_a. \tag{6}$$

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

$$(\hat{\mathcal{R}}_a)^n :: \exists V_n V_n' : (\mathcal{R}_a)^n \wedge \mathcal{D}^n \wedge (\mathcal{D}^n)', \tag{7}$$

$$\hat{\mathcal{I}}^n :: \exists V_n \quad : \mathcal{I}^n \wedge \mathcal{D}^n. \tag{8}$$

As an example, consider the decrement operation $\mathtt{l} := \mathtt{l} - 1$ on a local integer variable $\mathtt{l}$, and the inter-thread predicate $\mathtt{l} < \mathtt{l}_P$. Using Eq. (7) with $n = 2$, $a = 1$, we get 4 abstract transitions, which are listed in Table 1. The table shows that the abstraction is no longer asynchronous (treating $\mathtt{b}_1$ as belonging to thread 1, $\mathtt{b}_2$ to thread 2): in the highlighted transition, the executing thread 1 changes (its pc and hence) its local state, and so does thread 2. By contrast, on the right we have $\mathtt{l}_2 = \mathtt{l}_2'$ in all rows. The loss of asynchrony will become relevant in Sect. 3, where we define a suitable abstract Boolean programming language (which then necessarily must accommodate non-asynchronous programs).

$\mathtt{b}_1$	$\mathtt{b}_2$	$\mathtt{b}_1'$	$\mathtt{b}_2'$		$\mathtt{l}_1$	$\mathtt{l}_2$	$\mathtt{l}_1'$	$\mathtt{l}_2'$
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

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

Proving the ticket algorithm (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 $Q[1] :: \mathtt{l} \neq \mathtt{l}_P$, $Q[2] :: \mathtt{t} > \max(\mathtt{l}, \mathtt{l}_P)$, 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 abstract reachability tree for $\hat{\mathcal{P}}^n$ and these predicates reveals that mutual exclusion is satisfied: there is no state with both threads in location ℓ_3. The tree grows exponentially with n.

3   From Existential to Parametric Abstraction

Classical existential abstraction as described in Sect. 2.2 obliterates the symmetry present in the concrete concurrent program, which is given as the n-thread instantiation of a single-thread template $\mathcal{P}$: the abstraction is instead formulated via predicates over the explicitly expanded n-thread program $\mathcal{R}^n$. As observed in previous work [10], such a

"symmetry-oblivious" approach suffers from poor scalability for fixed-thread verification problems. Moreover, *parametric* reasoning over an unknown number of threads is impossible since the abstraction (7) directly depends on n.

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. There is, however, one obstacle: instantiating a program (such as $\mathcal{P}$) formulated over shared variables and one copy of the thread-local variables naturally gives rise to asynchronous concurrency. The programs resulting from inter-thread predicate abstraction are, however, not asynchronous, as we have seen. As a result, we need a more powerful abstract programming language.

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 $\tilde{L}_P = \{1_P : 1 \in \tilde{L}\}$. 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 (see [21]).

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

$$\tilde{\mathcal{P}}^n = (\tilde{\mathcal{R}}^n, \tilde{\mathcal{I}}^n) = \left(\bigvee_{a=1}^n (\tilde{\mathcal{R}}_a)^n, \ \bigvee_{a=1}^n (\tilde{\mathcal{I}}_a)^n \right) \tag{9}$$

where
$$(\tilde{\mathcal{R}}_a)^n :: \bigwedge_{p:p \neq a} \tilde{\mathcal{R}} \{\tilde{L} \rhd\!\!\!\rhd \tilde{L}_a\} \{\tilde{L}_P \rhd\!\!\!\rhd \tilde{L}_p\} \tag{10}$$

$$(\tilde{\mathcal{I}}_a)^n :: \bigwedge_{p:p \neq a} \tilde{\mathcal{I}} \{\tilde{L} \rhd \tilde{L}_a\} \{\tilde{L}_P \rhd \tilde{L}_p\} \tag{11}$$

Recall that $f\{X_1 \rhd\!\!\!\rhd X_2\}$ denotes index replacement of both current-state and next-state variables. Eq. (10) 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: transitions that "work" only for select passive threads are rejected.

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, the n-fold instantiation $\tilde{\mathcal{P}}^n$ overapproximates $\hat{\mathcal{P}}^n$. The variables of $\tilde{\mathcal{P}}$ are $\tilde{L} = \{b[i] : 1 \leq i \leq m\}$ and $\tilde{L}_P = \{b[i]_P : 1 \leq i \leq m\}$. 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. 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 $n \geq 2$,

$$\tilde{\mathcal{R}}(n) :: \exists \hat{L}_3, \hat{L}_3', \ldots, \hat{L}_n, \hat{L}_n' : (\hat{\mathcal{R}}_1)^n \{\hat{L}_1 \rhd\!\!\!\rhd \tilde{L}\} \{\hat{L}_2 \rhd\!\!\!\rhd \tilde{L}_P\} \tag{12}$$

$$\tilde{\mathcal{I}}(n) :: \exists \hat{L}_3, \ldots, \hat{L}_n : \qquad \hat{\mathcal{I}}^n \ \{\hat{L}_1 \rhd \tilde{L}\} \{\hat{L}_2 \rhd \tilde{L}_P\} \tag{13}$$

We apply this strategy to the earlier example of the decrement statement $\mathtt{l} := \mathtt{l} - 1$. To compute Eq. (12) first with $n = 2$, we need $(\hat{\mathcal{R}}_1)^2$, which was enumerated previously in Table 1. Simplification results in a Boolean DR program with variables $\mathtt{b}$ and $\mathtt{b}_P$ and transition relation

$$\tilde{\mathcal{R}}(2) = (\neg \mathtt{b} \wedge \mathtt{b}_P \wedge \neg \mathtt{b}') \vee (\neg \mathtt{b}_P \wedge \mathtt{b}' \wedge \neg \mathtt{b}'_P). \tag{14}$$

Using (14) as the template $\tilde{\mathcal{R}}$ in (10) 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 (14) 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 $\neg \mathtt{b} \wedge \mathtt{b}_P$ nor $\mathtt{b}' \wedge \neg \mathtt{b}'_P$ are satisfied for all choices of passive threads (violations highlighted in the table).

We thus increase n to 3, recompute Eq. (12), and obtain

$$\tilde{\mathcal{R}}(3) :: \tilde{\mathcal{R}}(2) \vee (\neg \mathtt{b} \wedge \neg \mathtt{b}_P \wedge \neg \mathtt{b}' \wedge \neg \mathtt{b}'_P). \tag{15}$$

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

$\mathtt{b}_1$	$\mathtt{b}_2$	$\mathtt{b}_3$	$\mathtt{b}'_1$	$\mathtt{b}'_2$	$\mathtt{b}'_3$		$\mathtt{l}_1$	$\mathtt{l}_2$	$\mathtt{l}_3$	$\mathtt{l}'_1$	$\mathtt{l}'_2$	$\mathtt{l}'_3$
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

Table 2: Part of the abstraction $(\hat{\mathcal{R}}_1)^3$ for stmt. $\mathtt{l} := \mathtt{l} - 1$ against predicate $\mathtt{l} < \mathtt{l}_P$ (left); concrete witness transitions (right). The highlighted elements are inconsistent with (14) as a template

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 2 $(\tilde{\mathcal{P}}(n))^n$ *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 $\hat{\mathcal{I}}^n \Rightarrow (\tilde{\mathcal{I}}(n)_a)^n$.*

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

Theorem 3 *Let $\#_{IT}$ be the number of inter-thread predicates among the $Q[i]$. Then the sequence $(\tilde{\mathcal{P}}(n))$ stabilizes at $\mathtt{b} = 4 \times \#_{IT} + 2$, i.e. for $n \geq \mathtt{b}$, $\tilde{\mathcal{P}}(n) = \tilde{\mathcal{P}}(\mathtt{b})$.*

Corollary 4 (from L. 2, T. 3) *Let $\tilde{\mathcal{P}} := \tilde{\mathcal{P}}(\mathtt{b})$, for $\mathtt{b}$ as in Thm. 3. The components of $\tilde{\mathcal{P}}$ are thus $(\tilde{\mathcal{R}}, \tilde{\mathcal{I}}) = (\tilde{\mathcal{R}}(\mathtt{b}), \tilde{\mathcal{I}}(\mathtt{b}))$. Then, for $n \geq 2$, $\tilde{\mathcal{P}}^n$ overapproximates $\hat{\mathcal{P}}^n$.*

Building a template DR program thus requires instantiating the existentially abstracted transition relation for a number b of threads that is linear in the number of inter-thread predicates with respect to which to abstraction is built.

As a consequence of losing asynchrony in the abstraction, many existing model checkers for concurrent software become inapplicable [25, 11, 12]. For a fixed thread count n, the problem can be circumvented by forgoing the replicated nature of the concurrent programs, as done in [10] for boom tool: it proves the ticket algorithm correct up to $n = 3$, but takes a disappointing 30 minutes. The goal of the following section is to design an efficient and, more importantly, fully parametric solution.

4 Unbounded-Thread Dual-Reference Programs

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 "counterize" the state space [17]: a global state is encoded as a vector of local-state counters, each of which records the number of threads currently occupying a particular local state.

These methods are applicable to unbounded thread numbers as well, in which case the local state counters range over unbounded natural numbers $[0, \infty[$. The fact that the abstract program executed by each thread is finite-state now might suggest that the resulting infinite-state counter systems can be modeled as vector addition systems (as done in [17]) or, more generally, as *well quasi-ordered transition systems* [15, 1] (defined below). This would give rise to sound and complete algorithms for local-state reachability in such programs.

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

Theorem 5 *Program location reachability for Boolean DR programs run by an unbounded number of threads is undecidable.*

The proof reduces the halting problem for 2-counter machines to a reachability problem for a DR program $\tilde{\mathcal{P}}$. Counter values c_i are reduced to numbers of threads in program locations d_i of $\tilde{\mathcal{P}}$. A zero-test for counter c_i is reduced to testing the *absence of any thread* in location d_i. This condition can be expressed using passive-thread variables, but not using traditional single-thread local variables. (Details of the proof in [21].)

Thm. 5 implies that the unbounded-counter systems obtained from asynchronous programs are in fact *not* well quasi-ordered. How come? Can this problem be fixed, in order to permit a complete verification method? If so, at what cost?

4.1 Monotonicity in Dual-Reference Programs

For a transition system $(\Sigma, \rightarrowtail)$ to be well-quasi ordered, we need two conditions to be in place [15, 1, 2]:

well quasi-orderedness: there exists a reflexive and transitive binary relation $\preceq$ on Σ such that for every infinite sequence $v, w, \ldots$ of states in Σ there exist i, j with $i < j$ and $v_i \preceq v_j$.

monotonicity: for any v, v', w with $v \rightarrowtail v'$ and $v \preceq w$ there exists w' such that $w \rightarrowtail w'$ and $v' \preceq w'$.

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 can define $\preceq$ as

$$(n_1, \ldots, n_k) \preceq (n'_1, \ldots, n'_k) \ :: \ \forall i = 1..k : n_i \leq n'_i$$

where k is the number of thread-local states. We can now characterize monotonicity of DR programs as follows:

Lemma 6 *Let $\tilde{\mathcal{R}}$ be the transition relation of a DR program. Then the infinite-state transition system $\cup_{n=1}^{\infty} \tilde{\mathcal{R}}^n$ is monotone (with respect to $\preceq$) exactly if, for all $k \geq 2$:*

$$(v, v') \in \tilde{\mathcal{R}}^k \quad \Rightarrow \quad \forall l_{k+1} \, \exists l'_{k+1}, \pi : \big(\langle v, l_{k+1} \rangle, \pi(\langle v', l'_{k+1} \rangle)\big) \in \tilde{\mathcal{R}}^{k+1} \, . \qquad (16)$$

In (16), the expression $\forall l_{k+1} \exists l'_{k+1} \ldots$ quantifies over valuations of the local variables of thread $k+1$. The notation $\langle v, l_{k+1} \rangle$ denotes a $(k+1)$-thread state that agrees with v in the first k local states and whose last local state is l_{k+1}; similarly $\langle v', l'_{k+1} \rangle$. Symbol π denotes a permutation on $\{1, \ldots, k+1\}$ that acts on states by acting on thread indices, which effectively reorders thread local states.

Asynchronous programs are trivially monotone (and DR): Eq. (16) is satisfied by choosing $l'_{k+1} := l_{k+1}$ and π the identity. Table 3 shows instructions found in *non*-asynchronous programs that destroy monotonicity, and why. For example, the swap instruction in the first row gives rise to a DR program with a 2-thread transition $(0, 0, 0, 0) \in \tilde{\mathcal{R}}^2$. Choosing $l_3 = 1$ in (16) requires the existence of a transition in $\tilde{\mathcal{R}}^3$ of the form $(1_1, 1_2, 1_3, 1'_1, 1'_2, 1'_3) = (0, 0, 1, \pi(0, 0, 1'_3))$, which is impossible: by equations (9) and (10), there must exist $a \in \{1, 2, 3\}$ such that for $\{p, q\} = \{1, 2, 3\} \setminus \{a\}$, both "$a$ swaps with p" and "a swaps with q" hold, i.e.

$$1'_p = 1_a \wedge 1'_a = 1_p \quad \wedge \quad 1'_q = 1_a \wedge 1'_a = 1_q \, ,$$

which is equivalent to $1'_a = 1_p = 1_q \wedge 1_a = 1'_p = 1'_q$. It is easy to see that this formula is inconsistent with the partial assignment $(0, 0, 1, \pi(0, 0, 1'_3))$, no matter what $1'_3$.

More interesting for us is the fact that asynchronous programs (= our input language) are monotone, while their parametric predicate abstractions may not be; this demonstrates that the monotonicity is in fact *lost in the abstraction*. Consider again the decrement instruction $1 := 1 - 1$, but this time abstracted against the inter-thread predicate $Q :: 1 = 1_P$. Parametric abstraction results in the two-thread and three-thread template instantiations

$$\tilde{\mathcal{R}}^2 = (\neg b_1 \vee \neg b'_1) \wedge b_1 = b_2 \wedge b'_1 = b'_2$$
$$\tilde{\mathcal{R}}^3 = (\neg b_1 \vee \neg b'_1) \wedge b_1 = b_2 = b_3 \wedge b'_1 = b'_2 = b'_3 \, .$$

Dual-reference program		Monotonicity	
instruction	variables	mon.?	assgn. violating (17)
$\mathtt{l},\mathtt{l}_P := \mathtt{l}_P,\mathtt{l}$	$\mathtt{l} \in \mathbb{B}$	no	$\mathtt{l}=0, \mathtt{l}'=1$
$\mathtt{l},\mathtt{l}_P := \mathtt{l}+1,\mathtt{l}_P-1$	$\mathtt{l} \in \mathbb{N}$	yes	
$\mathtt{l}_P := \mathtt{l}_P + 1$	$\mathtt{l} \in \mathbb{N}$	yes	
$\mathtt{l} := \mathtt{l} + \mathtt{l}_P$	$\mathtt{l} \in \mathbb{N}$	no	$\mathtt{l}=\mathtt{l}'=1$
$\mathtt{l}_P := c$	$\mathtt{l},c \in \mathbb{N}$	yes	

Table 3: Each row shows a single-instruction program, whether the program gives rise to a monotone system and, if not, an assignment that violates Eq. (17). (Some of these programs are not finite-state.)

Consider the transition $(0,0) \to (1,1) \in \tilde{\mathcal{R}}^2$ and the three-thread state $w = (0,0,1) \succ (0,0) : w$ clearly has no successor in $\tilde{\mathcal{R}}^3$ (it is in fact inconsistent), violating monotonicity. We discuss in Sect. 4.2 what happens to the decrement instruction with respect to predicate $\mathtt{l} < \mathtt{l}_P$.

4.2   Restoring Monotonicity in the Abstraction

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

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

Theorem 7 *Let $\tilde{\mathcal{R}}$ be the transition relation of a DR program. Then the infinite-state transition system $\cup_{n=1}^{\infty}\tilde{\mathcal{R}}^n$ is monotone if the following formula over $\tilde{L} \times \tilde{L}'$ is valid:*

$$\exists \tilde{L}_P \tilde{L}'_P : \tilde{\mathcal{R}} \quad \Rightarrow \quad \forall \tilde{L}_P \exists \tilde{L}'_P : \tilde{\mathcal{R}}. \tag{17}$$

Unlike the monotonicity characterization given in Lemma 6, Eq. (17) is formulated only about the template program $\tilde{\mathcal{R}}$. It suggests that, if $\tilde{\mathcal{R}}$ holds for some valuation of its passive-thread variables, then no matter how we replace the current-state passive-thread variables $\tilde{L}_P$, we can find next-state passive-thread variables $\tilde{L}'_P$ such that $\tilde{\mathcal{R}}$ still holds. This is true for asynchronous programs, since here $\tilde{L}_P = \emptyset$. It fails for the swap instruction in the first row of Table 3: the instruction gives rise to the DR program $\tilde{\mathcal{R}} :: \mathtt{l}' = \mathtt{l}_P \wedge \mathtt{l}'_P = \mathtt{l}$. The assignment on the right in the table satisfies $\tilde{\mathcal{R}}$, but if $\mathtt{l}_P$ is changed to 0, $\tilde{\mathcal{R}}$ is violated no matter what value is assigned to $\mathtt{l}'_P$.

We are now ready to modify the possibly non-monotone abstract DR program $\tilde{\mathcal{P}}$ into a new, monotone abstraction $\tilde{\mathcal{P}}_m$. Our solution is similar in spirit to, but different in effect from, earlier work on *monotonic abstractions* [3], which proposes to delete

processes that violate universal guards and thus block a transition. This results in an overapproximation of the original system and thus possibly spuriously reachable error states. By contrast, exploiting the monotonicity of the *concrete* program $\mathcal{P}$, we can build a monotone program $\tilde{\mathcal{P}}_m$ that is safe exactly when $\tilde{\mathcal{P}}$ is, thus fully preserving soundness and precision of the abstraction $\tilde{\mathcal{P}}$.

Definition 8 *The **non-monotone fragment (NMF)** of a DR program with transition relation $\tilde{\mathcal{R}}$ is the formula over $\tilde{L} \times \tilde{L}_P \times \tilde{L}'$:*

$$\mathcal{F}(\tilde{\mathcal{R}}) \quad :: \quad \neg \exists \tilde{L}'_P : \tilde{\mathcal{R}} \;\wedge\; \exists \tilde{L}_P \tilde{L}'_P : \tilde{\mathcal{R}}. \tag{18}$$

The NMF encodes partial assignments $(\mathtt{l}, \mathtt{l}_P, \mathtt{l}')$ that cannot be extended, via any $\mathtt{l}'_P$, to a full assignment satisfying $\tilde{\mathcal{R}}$, but can be extended for some valuation of $\tilde{L}_P$ other than $\mathtt{l}_P$. We revisit the two non-monotone instructions from Table 3. The NMF of $\mathtt{l}, \mathtt{l}_P := \mathtt{l}_P, \mathtt{l}$ is $\mathtt{l}' \neq \mathtt{l}_P$: this clearly cannot be extended to an assignment satisfying $\tilde{\mathcal{R}}$, but when $\mathtt{l}_P$ is changed to $\mathtt{l}'$, we can choose $\mathtt{l}'_P = \mathtt{l}$ to satisfy $\tilde{\mathcal{R}}$. The non-monotone fragment of $\mathtt{l} := \mathtt{l} + \mathtt{l}_P$ is $\mathtt{l}' \geq \mathtt{l} \wedge \mathtt{l}' \neq \mathtt{l} + \mathtt{l}_P$.

Eq. (18) is slightly stronger than the negation of (17): the NMF binds the values of the $\tilde{L}_P$ variables for which a violation of $\tilde{\mathcal{R}}$ is possible. It can be used to "repair" $\tilde{\mathcal{R}}$:

Lemma 9 *For a DR program with transition relation $\tilde{\mathcal{R}}$, the program with transition relation $\tilde{\mathcal{R}} \vee \mathcal{F}(\tilde{\mathcal{R}})$ is monotone.*

Lemma 9 suggests to add artificial transitions to $\tilde{\mathcal{P}}$ that allow arbitrary passive-thread changes in states of the non-monotone fragment, thus lifting the blockade previously caused by some passive threads. While this technique restores monotonicity, the problem is of course that such arbitrary changes will generally modify the program behavior; in particular, an added transition may lead a thread directly into an error state that used to be unreachable.

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

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 $\tilde{\mathcal{R}}_m :: \tilde{\mathcal{R}} \vee (\mathcal{F}(\tilde{\mathcal{R}}) \wedge (\mathtt{pc}'_P = \ell_\bot))$.*

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

Theorem 11 *Let $\mathcal{P}$ be an asynchronous program, and $\tilde{\mathcal{P}}$ its parametric abstraction. The monotone closure $\tilde{\mathcal{P}}_m$ of $\tilde{\mathcal{P}}$ is monotone. Further, $(\tilde{\mathcal{P}}_m)^n$ is safe exactly if $\tilde{\mathcal{P}}^n$ is.*

Thm. 11 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.

Proving the parameterized ticket algorithm Applying this strategy to the ticket algorithm yields a well quasi-ordered transition system for which the backward reachability method described in [1] returns "uncoverable", confirming that the ticket algorithm guarantees mutual exclusion, this time *for arbitrary thread counts*. We remind the reader that the ticket algorithm is challenging for existing techniques: cream [19], slab [11] and symmpa [10] handle only a fixed number of threads, and the resource requirements of these algorithms grow rapidly; none of them can handle even a handful of threads. The recent approach from [14] generates polynomial-size proofs, but again only for fixed thread counts.

5   Comparison with Related Work

Existing approaches for verifying asynchronous shared-memory programs typically do not exploit the monotone structure that source-level multi-threaded programs often naturally exhibit [20, 7, 9, 26, 19, 10, 12, 14]. For example, the constraint-based approach in [19], 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.

A CEGAR approach for fixed-thread symmetric concurrent programs has been implemented in symmpa [10]. 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 algorithms such as the ticket lock. Nevertheless, even the restricted predicate language of [10] can give rise to non-asynchronous programs. As a result, their technique cannot be extended to unbounded thread counts with well quasi-ordered systems technology.

Recent work on data flow graph representations of fixed-thread concurrent programs has been applied to safety property verification [14]. 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 [14], the technique in [12] uses data flow graphs to compute invariants of concurrent programs with unbounded threads (implemented in duet). In contrast to our approach, which uses an expressive predicate language, duet constructs proofs from relationships between either solely shared or solely local variables. These are insufficient for many benchmarks such as the parameterized ticket algorithm.

Predicates that, like our inter-thread predicates, reason over all participating processes/threads have been used extensively in invariant generation methods [5, 16, 22]. As a recent example, an approach that relies on abstract interpretation instead of model checking is [24]. 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 $\mathcal{Q}$. Future work will investigate how invariant generation procedures can be used towards *predicate discovery* in our technique.

The idea of "making" systems monotone, in order to enable wqo-based reasoning, was pioneered in earlier work [6, 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 future the 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.

6   Concluding Remarks

We have 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, some widely used algorithms like the ticket lock. We found that the abstractions with respect to these predicates result in non-monotone finite-data replicated programs, for which reachability is in fact undecidable. To fix this problem, we strengthened the earlier method of monotonic abstractions such that it does not introduce spurious errors into the abstraction.

We view the treatment of monotonicity as the major contribution of this work. Program design often naturally gives rise to "monotone concurrency", where adding components cannot disable existing actions, up to component symmetry. 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. P. A. Abdulla. Well (and better) quasi-ordered transition systems. *B SYMB LOG*, 2010.
2. P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems of infinite-state systems. In *LICS*, 1996.
3. P. A. Abdulla, G. Delzanno, and A. Rezine. Monotonic abstraction in parameterized verification. *ENTCS*, 2008.
4. G. R. Andrews. *Concurrent programming: principles and practice*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

5. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *CAV*, 2001.
6. J. D. Bingham and A. J. Hu. Empirically efficient verification for a class of infinite-state systems. In *TACAS*, 2005.
7. S. Chaki, E. M. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, 2006.
8. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *TOPLAS*, 1994.
9. B. Cook, D. Kroening, and N. Sharygina. Verification of Boolean programs with unbounded thread creation. *Theoretical Comput. Sci.*, 2007.
10. A. F. Donaldson, A. Kaiser, D. Kroening, M. Tautschnig, and T. Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *FMSD*, 2012.
11. K. Dräger, A. Kupriyanov, B. Finkbeiner, and H. Wehrheim. SLAB: A certifying model checker for infinite-state concurrent systems. In *TACAS*, 2010.
12. A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, 2012.
13. A. Farzan and Z. Kincaid. Duet: static analysis for unbounded parallelism. In *CAV*, 2013.
14. A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, 2013.
15. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Comput. Sci.*, 2001.
16. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202. ACM, 2002.
17. S. German and P. Sistla. Reasoning about systems with many processes. *JACM*, 1992.
18. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*. Springer, 1997.
19. A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, 2011.
20. T. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI*, 2004.
21. A. Kaiser, D. Kroening, and T. Wahl. Lost in abstraction: Monotonicity in multi-threaded programs (extended technical report). *CoRR*, 2014.
22. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, volume 2937 of *LNCS*, pages 267–281. Springer, 2004.
23. A. Malkis. *Cartesian Abstraction and Verification of Multithreaded Programs*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2010.
24. A. Sánchez, S. Sankaranarayanan, C. Sánchez, and B.-Y. E. Chang. Invariant generation for parametrized systems using self-reflection. In *SAS*, pages 146–163, 2012.
25. S. L. Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *CAV*, 2010.
26. T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent Linux device drivers. In *ASE*, 2007.