Delay-Bounded Scheduling Without Delay!

Andrew Johnson[1(✉)] and Thomas Wahl[1,2(✉)]

[1] Northeastern University, Boston, MA 02115, USA
aj3189@princeton.edu
[2] GrammaTech Inc., Bethesda, USA

Abstract. We consider the broad problem of analyzing safety properties of asynchronous concurrent programs under arbitrary thread interleavings. *Delay-bounded deterministic scheduling*, introduced in prior work, is an efficient bug-finding technique to curb the large cost associated with full scheduling nondeterminism. In this paper we first present a technique to *lift the delay bound* for the case of finite-domain variable programs, thus adding to the efficiency of bug detection the ability to prove safety of programs under arbitrary thread interleavings. Second, we demonstrate how, combined with predicate abstraction, our technique can both refute and verify safety properties of programs with unbounded variable domains, even for unbounded thread counts. Previous work has established that, for non-trivial concurrency routines, predicate abstraction induces a highly complex abstract program semantics. Our technique, however, never statically constructs an abstract parametric program; it only requires some abstract-states set to be closed under certain actions, thus eliminating the dependence on the existence of verification algorithms for abstract programs. We demonstrate the efficiency of our technique on many examples used in prior work, and showcase its simplicity compared to earlier approaches on the unbounded-thread Ticket Lock protocol.

1 Introduction

Asynchronous concurrent programs consist of a number of threads executing in an interleaved fashion and communicating through shared variables, message passing, or other means. In such programs, the set of states reachable by one thread depends both on the behaviors of the other threads, and on the order in which the threads are interleaved to create a global execution. Since the thread interleaving is unknown to the program designer, analysis techniques for asynchronous programs typically assume the worst case, i.e., that threads can interleave arbitrarily; we refer to this assumption as *full scheduling nondeterminism*. In order to prove safety properties of such programs, we must therefore ultimately investigate all possible interleavings.

Partially supported by the US National Science Foundation under grant #1718235.

Proposed about a decade ago, *delay-bounded deterministic scheduling* [10] is an effective technique to curb the large cost associated with exploring arbitrary thread interleavings. The idea is that permitting a limited number of scheduling *delays*—skipping a thread when it is normally scheduled to execute— in an otherwise deterministic scheduler approximates a fully nondeterministic scheduler from below. Delaying gives rise to a new thread interleaving, potentially reaching states unreachable to the deterministic scheduler. In the limit, i.e., with unbounded delays, the delaying and the fully nondeterministic scheduler permit the same set of executions and, thus, reach the same states.

Prior work has demonstrated that delay-bounded scheduling can "discover concurrency bugs efficiently" [10], in the sense that such errors are often detected for a small number of permitted delays. The key is that few delays means to explore only few interleavings. Thus, under moderate delay bounds, the reachable state space can often be explored exhaustively, resulting—if no errors are found—in a delay-bounded verification result.

We build on the empirical insight of efficient delay-*bounded* bug detection (testing) or verification, and make the following contributions.

1. Delay-bounded scheduling without delay. If no bug is found while exhaustively exploring the given program for a given delay budget, we "feel good" but are left with an uncertainty as to whether the program is indeed bug-free. We present a technique to remove this uncertainty, as follows. We prove that the set $R(d)$ of states reached under a delay bound d equals the set R of reachable states under *arbitrary* thread interleavings if two conditions are met:

– increasing the delay bound by a number roughly equal to the number of executing threads produces no additional reachable states, and
– set $R(d)$ is closed under a certain set of critical program actions.

In some cases, the set of "critical program actions" may be definable statically at the language level; in others, this must be determined per individual action. To increase the chance that the above two conditions eventually hold, we typically work with conservative abstractions of R; the (precisely computed) abstract reachability set $\overline{R}$ is then used to decide whether the program is safe.

2. Efficient delay-unbounded analysis. We translate the above foundational result into an efficient delay-unbounded analysis algorithm. It starts with a deterministic Round-Robin scheduler, parameterized by the number of rounds r it runs and of delays d it permits, and increases r and d in a delicate schedule *weak-until* the two conditions above hold (it is not guaranteed that they ever will). The key for efficiency is that the reachability sets under increasing r and d are *monotone*. We therefore can determine reachability under parameters $r' \geq r$ and $d' \geq d$ starting from a *frontier* of the states reached under bounds r and d. We present this algorithm and prove it correct. We also prove its termination (either finding a bug or proving correctness), under certain conditions.

3. Delay-unbounded analysis for general infinite-state systems. We demonstrate the power of our technique on programs with unbounded-domain variables and unbounded thread counts. The existence of integer-like variables suggests the use of a form of predicate abstraction. Prior work has shown that predicate abstraction for unbounded-thread concurrent programs leads to complex abstract program semantics [8,15], going beyond even the rich class of well-quasiordered systems [1]. Our delay-unbounded analysis technique does not require an abstract program. Instead, we add to the idea of reachability analysis under increasing r and d a third dimension n, representing increasing thread counts, enjoying a similar convergence property. Circumventing the static construction of the abstract program simplifies the verification process dramatically.

In summary, this paper presents a technique to lift the bound used in delay-bounded scheduling, while (empirically) avoiding the combinatorial explosion of arbitrary thread interleavings. Our technique can therefore find bugs as well as prove programs bug-free. We demonstrate its efficiency using concurrent pushdown system benchmarks, as well as known-to-be-hard infinite-state protocols such as the Ticket Lock [3]. We offer a detailed analysis of internal performance aspects of our algorithm, as well as a comparison with several alternative techniques. We attribute the superiority of our method to the retained parsimony of limited-delay deterministic-schedule exploration.

A full version of this paper, with proofs omitted here and other supplementary information, can be found in an accompanying Technical Report [14].

2 Delay-Bounded Scheduling

2.1 Basic Computational Model

For the purposes of introducing the idea behind delay-bounded scheduling, we define a deliberately broad asynchronous program model. Consider a multi-threaded program $\mathcal{P}$ consisting of n threads. We fix this number throughout the paper up to and including Sect. 5.1, after which we consider parameterized scenarios. Each thread runs its own procedure and communicates with others via shared program variables. A "procedure" is a collection of *actions* (such as those defined by program statements). We define a shared-states set G and, for each thread, a local-states set L_i $(0 \leq i < n)$. A global program state is therefore an element of $G \times \prod_{i=0}^{n-1} L_i$. In addition, a finite number of states are designated as *initial*. (Finiteness is required in Sect. 4 for a termination argument [Lemma 11].)

The execution model we assume in this paper is asynchronous. A *step* is a pair (s, s') of states such that there exists a thread i $(0 \leq i < n)$ such that s and s' agree on the local states of all threads $j \neq i$; the local state of thread i may have changed, as well as the shared state. We say thread i *executes* during the step, by executing some action of its procedure.[1] The execution semantics within the procedure is left to the thread (e.g., there may be multiple enabled actions in a state, an action may itself be nondeterministic, etc.). Without loss of generality for safety properties, we assume that the transition relation induced

[1] If only the shared state changes, it is possible that the identity of the executing thread is not unique. This small ambiguity is inconsequential for this paper.

by each thread's possible actions be total. That is, instead of an action x being disabled for a thread in state s, we stipulate that firing x from s results in s.

A *path* is a sequence $p = (s_0, \ldots, s_l)$ of states such that, for $0 \le i < l$, (s_i, s_{i+1}) is a step. This path has length l (= number of steps taken). A state s is *reachable* if there exists a path from some initial state to s. We denote by R the (possibly infinite) set of states reachable in $\mathcal{P}$. Note that these definitions permit arbitrary asynchronous thread interleavings.

2.2 Free and Round-Robin Scheduling

We formalize the notion of a scheduling policy indirectly, by parameterizing the concept of reachability by the chosen scheduler. A state s is *reachable under free scheduling* if there exists a path $p = (s_0, \ldots, s_l)$ from some initial state s_0 to $s_l = s$. A free scheduler is simulated in state space explorers using full nondeterminism. State s is *reachable under n-thread Round-Robin scheduling with round bound r* if there exists a path $p = (s_0, \ldots, s_l)$ from some initial state s_0 to $s_l = s$ such that

1. $\lceil l/n \rceil \le r$, and
2. for $0 \le i < l$, thread $i \pmod n$ executes during step (s_i, s_{i+1}).

2.3 Delay-Bounded Round-Robin Scheduling

We approximate the set of states reachable under free scheduling from below, using a relaxed Round-Robin scheduler. The scheduler introduced so far is, however, deterministic and thus *vastly* underapproximates the free scheduler, even for unbounded r. The solution proposed in earlier work is to introduce a limited number d of scheduling *delays* [10]. A delayed thread is skipped in the current round and must wait until the next round.

Definition 1. *State s* ***is reachable under Round-Robin scheduling with round bound*** *r* ***and delay bound*** *d (*"reachable under $RR(r, d)$ scheduling"* for short) if there exists a path $p = (s_0, \ldots, s_l)$ from some initial state s_0 to $s_l = s$ and a function $f : \{0, \ldots, l-1\} \to \{0, \ldots, n-1\}$, called* scheduling function*, such that*

1. for $d_p := f(0) + \sum_{i=1}^{l-1} \left((f(i) - f(i-1) - 1) \bmod n\right)$, we have $d_p \le d$,
2. $\lceil \frac{l+d_p}{n} \rceil \le r$ (d_p as defined in 1.), and
3. for $0 \le i < l$, thread $f(i)$ executes during step (s_i, s_{i+1}).

Variable d_p from 1. quantifies the total delay, compared to a perfect Round-Robin scheduler, that the scheduling along path p has accumulated. Consider the case of $n = 4$ threads T0,...,T3. Then the scheduling sequence $(f(0), \ldots, f(11))$ below on the left, of $l = 12$ steps and involving 13 states, follows a perfect Round-Robin schedule of $r = 3$ rounds (separated by |):

0 1 2 3 | 0 1 2 3 | 0 1 2 3   0 1 $\cancel{2}$ 3 | 0 1 2 $\cancel{3}$ | $\cancel{0}$ 1 2 3

The sequence on the right of $l = 9$ steps follows a Round-Robin scheduling of $r = 3$ rounds and a total of $d_p = 3$ delays: one after the second step (T2 is delayed: $3 - 1 - 1 \bmod 4 = 1$), another two delays after the sixth step (T3 and T0 are delayed: $1 - 2 - 1 \bmod 4 = 2$). The final state of this path is reachable under $RR(3, 3)$ scheduling. Note that delays effectively shorten rounds.

We denote by $R(r, d)$ the set of states reachable in $\mathcal{P}$ under $RR(r, d)$ scheduling. (Note that this set is finite, for any program $\mathcal{P}$.) It is easy to see that, given sufficiently large r and d, *any* schedule can be realized under $RR(r, d)$ scheduling:

Theorem 2. *State s is reachable under free scheduling iff there exist r, d such that s is reachable under $RR(r, d)$ scheduling: $R = \bigcup_{r,d \in \mathbb{N}} R(r, d)$.*

State-space exploration under free scheduling can therefore be reduced to enumerating the two-dimensional parameter space (r, d) and computing states reachable under $RR(r, d)$ scheduling. This can be used to turn a Round Robin-based state explorer into a semi-algorithm, dubbed *delay-bounded tester* in [10].

An important property of the round and delay bounds is that increasing them can only increase the reachability sets:

Property 3 (Monotonicity in r & d). *For any round and delay bounds r and d:*

$$R(r, d) \subseteq R(r + 1, d) \quad , \quad R(r, d) \subseteq R(r, d + 1) \ . \tag{1}$$

This follows from the $\ldots \leq r$ and $\ldots \leq d$ constraints in Definition 1. The property relies on r and d being external to the program, not accessible inside it. Under this provision, monotonicity in any kind of resource bound is a fairly natural *yet not always guaranteed* property; we give a counterexample in Sect. 5.2.

3 Abstract Closure for Delay-Bounded Analysis

The goal of this paper is a technique to prove safety properties of asynchronous programs under arbitrary thread schedules. Theorem 2 affords us the possibility to reduce the exploration of such arbitrary schedules to certain bounded Round-Robin schedules, but we still need to deal with those bounds. In this section we present a closure property for bounded Round-Robin explorations.

3.1 Respectful Actions

Let $\mathcal{S}$ be the set of global program states of $\mathcal{P}$, and let $\alpha \colon \mathcal{S} \to \mathcal{A}$ be an *abstraction function*, i.e., a function that maps program states to elements of some abstract domain $\mathcal{A}$. Function α typically hides certain parts of the information contained in a state, but the exact definition is immaterial for this subsection.

A key ingredient of the technique proposed in this paper is to identify actions of the program executed by a thread with the property that the abstract successor of an abstract state under such an action does not depend on concrete-state information hidden by the abstraction.

Definition 4. *Let x be a program action, and let the relation $s \xrightarrow{i\,:\,x} s'$ denote that $s \to s'$ is a step during which thread i executes x. Action x **respects** α if, for all states $s_1, s_2, s'_1, s'_2 \in \mathcal{S}$ and all $i : 0 \leq i < n$:*

$$\alpha(s_1) = \alpha(s_2) \ \wedge \ s_1 \xrightarrow{i,x} s'_1 \ \wedge \ s_2 \xrightarrow{i,x} s'_2 \quad \Rightarrow \quad \alpha(s'_1) = \alpha(s'_2) \ . \tag{2}$$

Intuitively, "x respects α" means that successors under action x of α-equivalent states all have the same unique abstraction. Note the special case $s_1 = s_2$, $s'_1 \neq s'_2$: for nondeterministic actions x to respect α, multiple successors s'_1, s'_2 of the same concrete state $s_1 = s_2$ under x also must have the same abstraction.

Example 5. *Consider n-thread* concurrent pushdown systems *(CPDS), an instance of the asynchronous computational model presented in Sect. 2.1. We have a finite set of shared states readable and writeable by each thread. Each thread also has a finite-alphabet stack, which it can operate on by (i) overwriting the top-of-the-stack element, (ii) pushing an element onto the stack, or (iii) popping an element off the top of the non-empty stack. The classic pointwise top-of-the-stack abstraction function is defined by*

$$\alpha(g, w_0, \ldots, w_{n-1}) = (g, \sigma_0, \ldots, \sigma_{n-1}) \ , \tag{3}$$

where g is the shared state (unchanged by α), w_i is the contents of the stack of thread i, and σ_i is the top of w_i if w_i is non-empty, and empty otherwise [18]. Note that the domain into which α maps is a finite set.

Push and overwrite actions respect α, while pop actions disrespect it: consider the case $n = 1$ and $s_1 = (g, w_0) = (0, 10)$ and $s_2 = (0, 11)$, with stack contents 10 *and* 11*, resp. (left = top). While $\alpha(s_1) = \alpha(s_2) = (0, 1)$, the (unique) successor states of s_1 and s_2 after a* pop *are not α-equivalent: the elements* 0 *and* 1 *emerge as the new top-of-the-stack symbols, respectively, which α can distinguish.*

The notion of respectful actions gives rise to a condition on sets of abstract states that we will later use for convergence proofs:

Definition 6. *An abstract-state set A is **closed under actions disrespecting** α if, for every $\boldsymbol{a} \in A$ and every successor $\boldsymbol{a}'$ of $\boldsymbol{a}$ under a disrespectful action, $\boldsymbol{a}' \in A$.*

For maximum precision: a' is said to be a successor of a under a disrespectful action if there exist concrete states s and s', a thread id i and an action x such that $\alpha(s) = a$, $\alpha(s') = a'$, x disrespects α, and $s \xrightarrow{i\,:\,x} s'$. If abstraction α is clear from the context, we may just say "closed under disrespectful actions".

3.2 From Delay-Bounded to Delay-Unbounded Analysis

We now present our idea to turn a round- and delay-bounded tester into a (partial) verifier, namely by exploring the given asynchronous program for a number of round and delay bounds until we have "seen enough". Recall the notations R and $R(r, d)$ defined in Sect. 2. We also use $\overline{R}$ and $\overline{R}(r, d)$ short for

$\alpha(R)$ and $\alpha(R(r,d))$, i.e. the respective abstract reachability sets. (Note that $\overline{R}$ is *not* an abstract fixed point—instead, it is the result of applying α to the concrete reachability set R; see discussion in Sect. 7.)

Theorem 7. *For any $r, d \in \mathbb{N}$, if $\overline{R}(r,d) = \overline{R}(r+1, d+n-1)$ and $\overline{R}(r,d)$ is closed under actions disrespecting α, then $\overline{R}(r,d) = \overline{R}$.*

The theorem states: if the set of *abstract* states reachable under $RR(r,d)$ scheduling does not change after increasing the round bound by 1 and the delay bound by $n-1$, and it is closed under disrespectful actions, then $\overline{R}(r,d)$ is in fact the *exact* set $\overline{R}$ of abstract states reachable under a *free* scheduler: no approximation, no rounds, no delays, no Round-Robin.

Proof. of Theorem 7: we have to show that $\overline{R}(r,d)$ is closed under the abstract image function $\overline{Im}$ induced by α, defined as

$$\overline{Im}(a) \;=\; \{a' : \; \exists s, s' : \; \alpha(s) = a, \; \alpha(s') = a', \; s \rightarrow s'\} \; .$$

That is, we wish to show $\overline{Im}(\overline{R}(r,d)) \subseteq \overline{R}(r,d)$, which proves that no more abstract states are reachable. Consider $a \in \overline{R}(r,d)$ and $a' \in \overline{Im}(a)$, i.e. we have states s, s' such that $\alpha(s) = a$, $\alpha(s') = a'$, and $s \xrightarrow{i\,:\,x} s'$ for some thread i and some action x. The goal is to show that $a' \in \overline{R}(r,d)$.

To this end, we distinguish flavors of x. If x disrespects α, then $a' \in \overline{R}(r,d)$, since the set is closed under disrespectful actions.

So x respects α. Since $a \in \overline{R}(r,d)$, there exists a state $s_0 \in R(r,d)$ with $\alpha(s_0) = a$. Suppose for a moment that thread i is scheduled to run in state s_0. Then it can execute action x; any successor state s_0' satisfies $s_0' \in R(r,d)$, and:

$$a' \overset{(\text{def } a')}{=} \alpha(s') \overset{(x \text{ resp. } \alpha)}{=} \alpha(s_0') \overset{(\text{def } s_0')}{\in} \alpha(R(r,d)) = \overline{R}(r,d) \; .$$

But what if the thread scheduled to run in state s_0 under $RR(r,d)$ scheduling, call it j, **is not thread** i**?** Then we *delay* any threads that are scheduled before thread i's next turn; if $i < j$, this "wraps around", and we need to advance to the next round. The program state has not changed—we are still in s_0. Let s_0' be the successor state obtained when thread i now executes action x, and $\lambda(i,j) = 1$ if $i < j$, 0 otherwise. Then we have $s_0' \in R(r + \lambda(i,j), d + (j-i) \bmod n)$, and:

$$\begin{aligned} a' \overset{(\text{def } a')}{=} \alpha(s') \overset{(x \text{ resp. } \alpha)}{=} \alpha(s_0') &\overset{(\text{def } s_0',\, \alpha)}{\in} \overline{R}(r + \lambda(i,j), d + (j-i) \bmod n) \\ &\overset{(\text{monot. } r,d)}{\subseteq} \overline{R}(r+1, d+n-1) \overset{(\text{Thm. } 7)}{=} \overline{R}(r,d) \; . \end{aligned}$$

This concludes the proof of Theorem 7. □

Example 8. *Consider a simple 3-thread system with a shared-states set $G = \{0,1,2\}$. The local state of each thread is immaterial; function α just returns the shared state: $\alpha(g, l_0, l_1, l_2) = g$. The threads' procedures consist of the following actions, which update only the shared state:*

Thread T0: $0 \rightarrow 1$ *Thread T1:* $0 \rightarrow 1$ *Thread T2:* $0 \rightarrow 2$.

Table 1 shows the set of reachable states for different round and delay bounds. For example, with one round and zero delays, the only feasible action is T0's. The reachable states are 0 *(initial) and* 1 *(found by T0). The table shows a path to a pair* (r, d) *that meets the conditions of Theorem 7. From* $(r, d) = (1, 0)$ *we increment* r *to find a plateau in* r *of length* 1*. We then increase* d *to try to find a plateau in* d *of length* $n - 1 = 2$*. This example shows that a delay plateau of length* 1 *is not enough, as* 2 *is only reachable at least* 2 *delays. At* $(2, 2)$ *we find a new state* (2)*, so we restart the search for plateaus in* r *and* d*. At* $(3, 4)$*, the plateau conditions for Theorem 7 are met. There are no disrespectful transitions, so by Theorem 7, we know that* $\overline{R}(3, 4) = \overline{R}$*.*

Table 1. Reachable states in Example 8 under various round and delay bounds. The boxed set passes the convergence test suggested by Theorem 7

	$d = 0$	$d = 1$	$d = 2$	$d = 3$	$d = 4$
$r = 1$	{0,1} $\downarrow$	{0,1}	{0,1,2}	{0,1,2}	{0,1,2}
$r = 2$	{0,1} $\longrightarrow$	{0,1} $\longrightarrow$	{0,1,2} $\downarrow$	{0,1,2}	{0,1,2}
$r = 3$	{0,1}	{0,1}	{0,1,2} $\rightarrow$	{0,1,2} $\rightarrow$	$\boxed{\{0,1,2\}}$

4 Efficient Delay-Unbounded Analysis

Turning Theorem 7 into a reachability algorithm requires efficient computation of the sets $R(r, d)$. This section presents an approach to achieve this, by expanding only *frontier* states when either the round or the delay parameter is increased.

To this end, let C be a state property (such as an assertion) that respects α, in the sense that, for any states s_1, s_2, if $\alpha(s_1) = \alpha(s_2)$, then $s_1 \models C$ iff $s_2 \models C$. From now on, we further assume the domain $\mathcal{A}$ of abstraction function α to be finite, which will ensure termination of our algorithm (see Lemma 11 later).

Our verification scheme for C is shown in Algorithm 1, which uses Algorithm 2 as a subroutine. In the rest of this paper, we also refer to Algorithm 1 as *Delay-(and round-) UnBounded Analysis*, DrUBA for short.

The main data structure used in the algorithms is that of a `State`, which stores both program variables and scheduling information, in the attributes *finder*, *rounds_taken*, and *delays_taken*. For a state s, variables s.*rounds_taken* and s.*delays_taken* represent the number of times the scheduler started a round and delayed a thread, resp., to get to s. Variable s.*finder* contains the index of the thread whose action produced s. This is enough information to continue

Algorithm 1. Verifying property C against all reachable states of program $\mathcal{P}$

Input: n-thread asynchronous program, property C
Output: "safe", "violation of C", or "unknown"

```
 1: $Reached$ := (finite) set of initial states        ▷ $Reached$: states reached so far
 2: $r$ := 0; $d$ := 0
 3: repeat
 4:     $Frontier$ := $\{s \in Reached : s.rounds\_taken = r\}$
 5:     $r$++
 6:     for $s \in Frontier$ do
 7:         $Reached$ := $Reached \cup FinishRounds(s, r+1, C)$
 8:     $r$++
 9: until round plateau of length 1
10: repeat
11:     $Frontier$ := $\{s \in Reached : s.delays\_taken = d\}$
12:     $d$++
13:     for $s \in Frontier$ do
14:         $s'$ := $s$                                   ▷ copy of state $s$
15:         $s'.delays\_taken$++
16:         $s'.finder$ := ($s'.finder$ + 1) mod n
17:         if $s'.finder$ mod n = 0 then
18:             $s'.rounds\_taken$++
19:         $Reached$ := $Reached \cup FinishRounds(s', r, C)$
20:     if new abstract state found during for loop in Line 13 then
21:         goto 3                  ▷ abort second repeat loop; go back to first
22: until delay plateau of length $n-1$
23: if $\alpha(Reached)$ is closed under disrespectful actions then
24:     return "safe"
25: else
26:     return "unknown"
```

Algorithm 2. $FinishRounds(s, r, C)$

Input: s: state, r: round bound, C: state property
Output: states reachable from s up to round bound r, without delaying

```
1: Set<State> $Unexplored$ := $\{s\}$, $Reached$ := $\{\}$
2: while $Unexplored$ != $\{\}$ do
3:     select and remove some state $u$ from $Unexplored$
4:     if $u$ violates $C$ then
5:         throw "violation of $C$ (witnessed by reaching state $u$)"
6:     $Reached$ := $Reached \cup \{u\}$
7:     if $u.finder < n-1$ or $u.rounds\_taken < r$ then        ▷ if $u$ schedulable
8:         $Unexplored$ := $Unexplored \cup (Image(u) \setminus Reached)$
9: return $Reached$
```

the execution from s later, starting with the thread after *finder*. For the initial states, *rounds_taken* and *delays_taken* are zero, and *finder* is $n-1$ (the latter so that expanding the initial states starts with thread $(n-1)+1 \bmod n = 0$). For set membership testing, two states are considered equal when they agree on their finders and on program variables. The *rounds_taken* and *delays_taken* variables are for scheduling purposes only and ignored when checking for equality.

As mentioned in Prop. 3, the sequence of reachability sets is monotone with respect to both rounds and delays, for any program. This entails two useful properties for Algorithm 1. First, we can increase the bounds in any order and at individual rates. Second, it suffices to expand states at the frontier of the exploration, without missing new schedules. When adding a new delay, we only need to delay those states that were (first) found in schedules using the maximum delays. When adding a round, we only need to expand states that were (first) found in the last round of a schedule.

Algorithm 1 first advances the round parameter r until a round plateau has been reached (Lines 3–9). It does so by running the *FinishRounds* function on *frontier states* s: those that were reached in the final round r of the previous round iteration. *FinishRounds* (Algorithm 2) explores from the given state s, Round-Robin style, up to the given round, without delaying any thread. The actual expansion of a state happens in function *Image* (Line 8 of Algorithm 2), which computes a state's successors and initializes their scheduling variables: *rounds_taken* and *delays_taken* are copied from u, the *finder* of the successor is the next thread (+1 mod n). If this wraps around, *rounds_taken* is incremented as well.

Back to the main Algorithm 1: we have reached a round plateau of length 1 if the entire **for** loop in Line 6 sees no new *abstract* states (no new elements in $\alpha(\mathit{Reached})$). If so, we are not ready yet to perform the convergence test (recall Example 8). Instead, Algorithm 1 now similarly advances the delay parameter d (Lines 10–22). For each frontier state ($\mathit{delays\_taken} = d$), we delay the thread scheduled to execute from this state (by incrementing (mod n) the *finder* variable), and record the taken delay (Line 15). Then we again call the *FinishRounds* function and merge in the states found. Importantly, these merges preserve states already in *Reached*, meaning that the algorithm will keep states found earlier in the exploration (with smaller r, d).

The loop beginning in Line 10 repeats until a delay plateau of length $n-1$ is encountered (as required by Theorem 7). This means that during $n-1$ consecutive **repeat** iterations, the **for** loop in 13 did not find any new abstract states. When the round and delay plateaus have the required lengths (1 and $n-1$, resp.), we invoke the convergence test (Line 23), which amounts to applying Theorem 7. If the test fails, Algorithm 1 returns "unknown".

Towards proving partial correctness of Algorithm 1, we first show that the states eventually collected in set *Reached* by the algorithm correspond exactly to the round- and delay-bounded reachability sets $R(r, d)$, and that—after the two main **repeat** loops—a plateau of sufficient length has been generated. As a corollary, the algorithm is partially correct, i.e. it returns correct answers if it terminates.

Lemma 9. *If Algorithm 1 reaches Line 23, the current values of r and d satisfy: (i) Reached $= R(r,d)$, and (ii) $\overline{R}(r-1, d-(n-1)) = \overline{R}(r,d)$.*

Corollary 10. *The answers "safe" and "violation of C" returned by Algorithm 1 are correct.*

The algorithm won't return either "safe" or "violation of C" in one of two situations: when the convergence test fails in Line 23 (it gives up), and when it fails to ever reach this line. The latter can be prevented using a finite-domain α:

Lemma 11. *If the domain $\mathcal{A}$ of abstraction function α is finite, Algorithm 1 terminates on every input.*

Since abstraction α approximates the information contained in a state, a plateau may be *intermediate*, e.g. $\overline{R}(1,0) \subsetneq \overline{R}(1,1) = \overline{R}(2,2) \subsetneq \overline{R}(2,3)$. Thus, stopping the exploration simply on account of encountering a plateau—even of lengths $(1, n-1)$—is unsound. Intermediate plateaus make our algorithm (unavoidably) incomplete: if the test in Line 23 fails, then there are known-to-be-reachable abstract states with abstract successors whose reachability cannot be decided at that moment. If we knew the plateau to be intermediate, we could keep exploring the sets $R(r,d)$ for larger values of r and d until the next plateau emerges, hoping that the convergence test succeeds at that time. In general, however, we cannot distinguish intermediate from final plateaus.

5 DrUBA with Unbounded-Domain Variables

In addition to unbounded control structures like stacks, which come up in pushdown systems and were discussed in Ex 5, infinite state spaces in programs are often due to (nominally) unbounded-domain program variables. This presents no problem for the computation of the concrete reachability sets *Reached* in Algorithm 1: for any round and delay bounds (r,d), the set of concrete reachable states $RR(r,d)$ is finite and thus explicitly computable (no symbolic data structures are needed).[2] On the other hand, termination of the same algorithm requires that it eventually reach a plateau in r and d of sufficient length. This is guaranteed by an abstraction function α that maps concrete states into a finite abstract space. A finite abstract domain is therefore highly desirable.

A generic abstraction that reduces an unbounded data domain to a finite one is predicate abstraction [4,13, see [14] for a short primer]. The goal in this section is to demonstrate how the simple scheme of delay-unbounded analysis can be combined with predicate abstraction to verify unbounded-thread programs.

[2] Contrast this to a *context-switch* bound, under which reachability sets can be infinite.

5.1 The Fixed-Thread Case

Consider program $\mathbb{P}$ in Fig. 1 on the left [8, page 4: program $\mathbb{P}''$]. Intuitively, variable m counts the number of threads spawned to execute $\mathbb{P}$ concurrently. It is easy to see that "the assertion in [$\mathbb{P}$] cannot be violated, no matter how many threads execute [$\mathbb{P}$], since no thread but the first will manage to" [8] enter the *true* branch of the **if** statement and reach the assertion.

```
Program ℙ
────────────────────
    shared int m := 0
    shared int s := 0
    local int l := 0
0:  m++
1:  if m = 1 then
2:      s++, l++
3:      assert s = l
4:      goto 2
```

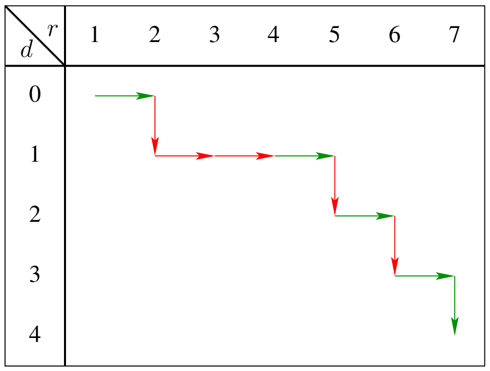

Fig. 1. Left: program $\mathbb{P}$; **Right:** how Algorithm 1 operates on (an abstraction of) it

Previous work has shown that even the 1-thread version of this program cannot be proved correct using predicate abstraction **unless** we permit predicates that depend on both shared and local variables [9, for the unprovability result], which have been referred to as *mixed* [8]. An example is the predicate $p :: (s = l)$, which comes up in the assertion. The dependence of p on both shared and thread-local data causes standard solutions that track the truth value of p in a shared or local Boolean variable to be unsound. The solution proposed in [8] is to use *broadcast* instructions to have the executing thread notify all other threads whenever the truth value of p changes. This solution comes with two disadvantages: (i) the resulting Boolean broadcast programs are more expensive to analyze than strictly asynchronous Boolean programs, and (ii) the solution cannot be extended to the unbounded-thread case.

Let us consider how we can verify this program using Algorithm 1, for the fixed-thread case; we consider $n = 2$ threads. We will have to use mixed predicates as in [8], but since we never execute the abstract Boolean program, there is no need for constructing it. As a result, there is no need for broadcast instructions.

The program generates an unbounded number of reachable concrete states, but we explore it only under round and delay bounds r and d. As per Algorithm 1, we increase these bounds until we have reached plateaus of lengths 1 and $n - 1 = 1$, resp. Plateaus are determined over the abstract-state set, so we need a function α.

First attempt: a single predicate. We define α as follows, for a concrete state c:

$$\alpha_1(c) \quad = \quad (c.pc_0,\ c.s = c.l_0) \in 0, \ldots, 4 \times \{0, 1\} \ ,$$

where $c.pc_0$, $c.l_0$, and $c.s$ are the values of thread 0's pc and local variable l, and of shared variable s, in state c, respectively. The function extracts from a concrete state the current program location of thread 0 and the value of predicate $p :: (s = l)$ for thread 0.[3] The only statement *not* respecting α_1 is the **if** statement in Line 1: here, the new value of the pc cannot be determined from the current values of pc and predicate p alone. All other statements respect α_1.

We can now perform an iterative exploration of this program—bounded but exhaustive within each bound. In Fig. 1 on the right, red arrows denote "new abstract state reached". A red horizontal arrow (r++) means: "keep increasing r". A red vertical arrow (d++) means: "switch to increasing r". In other words, following a red arrow—no matter the direction—we always go "right" (r++). The green horizontal arrow followed by a green vertical arrow at the end indicates that we have reached the first plateaus of length 1 in *both* directions: at $(r, d) = (7, 4)$.

At this point we have reached a total of 7 abstract states. State $(3, 0)$ ($pc = 3,\ s \neq l$) is not among them, so the assertion has not been violated so far. We run the convergence test, to determine whether set $\overline{R}(7, 4)$ is closed under disrespectful actions. Since the **if** in Line 1 is the only disrespectful statement, we only need to check successors of abstract states of the form $(1, ?)$ (i.e., with $pc = 1$). Unfortunately, $\overline{R}(7, 4)$ contains abstract state $(1, 0)$ (a reachable abstract state) but not its abstract successor $(2, 0)$. This state is unreachable, but we do not know that at this point. This causes Algorithm 1 to return "unknown".

Second attempt: two predicates. The disrespectful action causing the failure suggests that we need to keep track of whether the branch in Line 1 can be taken, i.e. whether $m = 1$. We refine our abstraction using this (non-mixed) predicate:

$$\alpha_2(c) \quad = \quad (c.pc_0,\ c.s = c.l_0,\ c.m = 1) \in 0, \ldots, 4 \times \{0, 1\}^2 \ . \tag{4}$$

The abstract successors of the **if** statement can now be decided based only on knowledge provided by α_2, i.e. the statement respects α_2. There is, however, another statement disrespecting α_2, and only one: the increment m++ in Line 0. If $m \neq 1$, we cannot decide whether $m = 1$ will be true after the increment.

We again perform our iterative exploration of this program, and find the first suitable plateau at the same point $(r, d) = (7, 4)$. This time, however, we have reached a total of 12 abstract states (all of them "safe"). We run the convergence test: we only need to check already reached abstract states of the form $(0, ?, 0)$ ($pc = 0,\ m \neq 1$). Set $\overline{R}(7, 4)$ contains exactly one state of this form: $(0, 1, 0)$, which m++ can turn into $(1, 1, 0)$ and $(1, 1, 1)$—note that the next pc value is unambiguous (1), and predicate $s = l$ is not affected. **The good news** is now that both abstract states $(1, 1, 0)$ and $(1, 1, 1)$ are contained in $\overline{R}(7, 4)$. This proves this set closed under disrespectful actions; Algorithm 1 terminates: the assertion is safe for any execution schedule, for the case of $n = 2$ threads.

[3] Tracking these values for thread 0 suffices: the multi-threaded program is symmetric.

We summarize that, in our solution above, we assumed a lucky hand in picking predicates—the question of predicate discovery is orthogonal to the delay-unbounded analysis scheme. However, the proof obtained using Algorithm 1 does not involve costly broadcast operations, previously proposed as an ingredient to extend predicate abstraction to concurrent programs. A second, more powerful advantage is that, unlike the earlier broadcast solution, Algorithm 1 extends gracefully to the unbounded-thread case. This is the topic of the rest of this section.

5.2 The Unbounded-Thread Case

The goal now is to investigate whether an asynchronous unbounded-domain variable program is safe for *arbitrary* thread counts (and thread interleavings).

Existing solutions. We are aware of only one general technique that combines predicate abstraction with unbounded-thread concurrency [15]. That technique can achieve the above goal, roughly as follows. In addition to standard and mixed predicates used also in the fixed-thread case, we now permit *inter-thread* predicates, which quantify over all threads other than the executing one. Such predicates allow us to express, for example, that a thread's local variable l's value is larger than that of any other thread: $\forall i : i \neq \mathit{self} : l > l_i$. Predicates of this type are provably required during predicate abstraction to verify the safety of the Ticket Lock algorithm [3,15].

Abstraction against inter-thread predicates leads to a *dual-reference program* [15], a process that is already far more complex than standard sequential or even fixed-thread predicate abstraction. But we pay another price for using these predicates: namely, the loss of *monotonicity* of the transition relation w.r.t. a standard well-quasiordering $\preccurlyeq$ on infinite state sets of unbounded-thread Boolean programs. In this context, monotonicity states, roughly, that adding passive threads to a valid transition keeps the transition intact.

This price is heavy, since monotonicity w.r.t. $\preccurlyeq$ would have given us a well-quasiordered infinite-state transition system, for which local-state reachability properties are decidable [1]; working implementations exist. The above-mentioned prior work attempts to salvage the situation, by adding a set of transitions (the *non-monotone fragment*) to the dual-reference program that restore monotonicity and further overapproximate but without affecting the reachability of unsafe states [15].

Alternative solution. We now propose a solution that uses the same type of inter-thread predicates (this is inevitable), but renders dual-reference programs, the monotone closure of the transition relation and all other "overhead" introduced in [15] unnecessary. We will use Algorithm 1 as a sub-routine.

The idea is as follows. Sect. 5.1 suggests a way to verify fixed-thread asynchronous programs, using a combination of predicate abstraction and Algorithm 1 . To handle the unbounded-thread case, we wrap another layer of incremental resource bounding around this combined algorithm—the "resource" this time is the number n of threads executing the program. For each member of a sequence of increasing fixed thread counts we compute the set of abstract states reachable under arbitrary thread interleavings. This is purely a sub-routine; we will use the method proposed in Sect. 5.1 (others are possible, e.g. [8]).

The incremental (in n) analysis proceeds until we have reached a thread plateau *of length 1*, and then run the convergence test: we check the current abstract reachability set for closure under disrespectful actions. This time, the abstract transitions must take into account that the number of executing threads is unknown. It is easy to see that a plateau of length 1 is sufficient: we compute the set of abstract states reachable under *arbitrary* thread schedules; thus, the obstacle of non-schedulability of thread i in the proof of Theorem 7 that forced us to wait for a (delay) plateau of length $n-1$ does not apply here.

A non-monotone resource parameterization

Before we demonstrate this idea on program $\mathbb{P}$, we justify our strategy of combining resource bounds. The idea presented above can be viewed as a multi-resource analysis problem where we increment r and d in an "inner loop" (represented by Algorithm 1 as a sub-routine to compute fixed-thread reachability sets), and n in an outer loop. Both loops compute monotonously increasing reachability sequences: for "inner" this is Prop. 3; for "outer" this is easy to see. Theorem 7 relies upon the monotonicity: without it, the test $\overline{R}(r,d) = \overline{R}(r+1, d+n-1)$ makes the algorithm unsound.

The way we nest the three involved resource parameters is not arbitrary: Round-Robin reachability under an increasing thread count is not monotone. More precisely, making the thread-count parameter n explicit, let $R(r,d,n)$ denote the set of states reachable in the n-thread program $\mathcal{P}$ under $RR(r,d)$ scheduling. Then $R(r,d,n) \subseteq R(r,d,n+1)$ is **not** valid. The following example illustrates this (at first counter-intuitive) monotonicity violation:

Example 12. *Consider the asynchronous Boolean program over shared variables s and t on the right. Here we have $R(3,0,1) \not\subseteq R(3,0,2)$: given 1 thread (sequential execution), a state with*

```
   shared bool s := 0, t := 0
0: t := !t
1: if t then
2:     s := 1
```

$s = 1$ is reachable. With 2 symmetric threads, under delay-free Round-Robin scheduling ($d = 0$), the first and second thread will repeatedly flip t to 1 *and back to* 0*, resp., before either one has a chance to get past the guard in* Line 1.

A stronger result is: for all $r \in \mathbb{N}$, $R(3,0,1) \not\subseteq R(r,0,2)$, i.e. we cannot make up for the poor scheduling of the second thread by adding more rounds.

The consequence for us is that we cannot compute, for fixed r, d, the sets $R(r,d,\infty)$, using the closure-under-disrespectful-actions paradigm. Instead we must, for each n, compute $R(\infty,\infty,n)$ (using Algorithm 1 or otherwise) and increase n in the outer loop.

Verifying program $\mathbb{P}$ for unbounded thread count

We recall that, given the two predicates shown in Eq. (4) and the pc, we were able to verify program $\mathbb{P}$ correct (under arbitrary thread interleavings) for $n = 2$

threads; a total of 12 abstract states were reached (out of $5 \cdot 2^2 = 20$ possible). Advancing the outer loop, we invoke Algorithm 1 for $n = 3$ threads. This reveals another reachable abstract state, namely $pc = 0$, $s \neq l$, $m \neq 1$. Unfortunately, this state causes Algorithm 1 to return "unknown": under α_2, one currently unreached abstract successor is $pc = 1$, $s \neq l$, $m = 1$, violating closure. Observing that a thread executing Line 1 with $m = 1$ must be the first thread executing, we try tracking the initial value of m:

$$\alpha_3(c) \quad = \quad (c.pc_0,\ c.s = c.l_0,\ c.m = 1,\ c.m = 0) \in 0, \ldots, 4 \times \{0,1\}^3 \ . \qquad (5)$$

Interestingly, *all actions (statements) of program $\mathbb{P}$ respect abstraction α_3*. This means that the test for closure under disrespectful actions is *vacuously true*—we can stop as soon as we have reached a plateau in n of length 1. We don't have to wait long for this plateau: we invoke Algorithm 1 for $n = 3$ and $n = 4$ under abstraction α_3. (Note that $n = 4$ requires a longer plateau than $n = 3$.) The abstract reachability sets consist of the same 14 abstract states in both cases. We report the program safe, for arbitrary interleavings and arbitrary thread counts. We can also report the exact set of 14 reachable abstract states.

We again summarize that, while we still (and unavoidably) use mixed predicates, we do not construct a thread-parameterized abstract program, which would require broadcast statements [8] and a rather involved dual-reference transition semantics [15]. In fact, we did not even need to test for closure under any abstract images, since the chosen abstraction enjoys respect from all actions.

6 Evaluation

Our goal for the evaluation of DrUBA was to answer the following questions:

1. How does DrUBA compare to abstract fixed-point computation ("AI")?
2. How does DrUBA compare to the approach from [18] ("CUBA")?
3. How expensive is the state exploration along a plateau in Algorithm 1?
4. What is the performance benefit of the frontier optimization in Algorithm 1?

Questions 1 and 2 serve to compare DrUBA against other techniques; Questions 3 and 4 investigate features of Algorithm 1.

To this end we implemented, in Java 11, a verifier using Algorithm 1 that takes concurrent pushdown systems as input; we refer to this verifier as DrUBA in this section.[4] We also implemented the AI approach in Java 11. For the comparison with the context-unbounded approach, we used a publicly available tool[5]. Our experiments are based on the concurrent benchmark programs also used in [18]. The experiments are performed on a 3.20GHz Intel i5 PC. The memory limit was 8GB, with a timeout of 1 h.

[4] DrUBA implementation available at https://doi.org/10.5281/zenodo.4726301.
[5] https://github.com/lpzun/cuba.

6.1 Results

Table 2 reports the benchmark names, the thread counts, and the size of the reachable abstract state space (columns 1–3). The second part of the table shows the time it took each verifier to fully explore the state space and confirm convergence. For the AI approach, we check whether the abstract state space is closed under *all* operations each time either r or d is incremented. Algorithm 1 was faster than "AI" on every example except Stefan-4,5. Stefan is the only program that actually does not require any delays to discover all reachable abstract states. The results indicate that the AI approach spends approximately half of its computation time doing repeated convergence tests after each bound increment. Furthermore, as state sets increase in size, AI seems to take even longer, as with the Bluetooth3 (2+3) example. The convergence test needed for "AI" includes checking closure under both respectful and disrespectful actions, making it more costly than the one used in Algorithm 1.

Algorithm 1 also improved on the results with "CUBA". For examples that took longer than a few seconds, DrUBA was able to run in less time on the same benchmark. The difference on small examples is likely due to a different implementation language (C++ vs. Java). DrUBA does not explore as many schedules, and explores fewer as the delay and round bounds approach their cutoff values (as noted below). Additionally, DrUBA was less memory-intensive for large examples for which the CUBA approach cannot prove that the set of reachable states per context bound is finite. In this case, "CUBA" requires the use of more expensive symbolic representations of states sets. Algorithm 1 does not suffer from this problem—the reachability sets in each iteration are finite. For the Stefan-5 example, "CUBA" ran out of memory after 23 min. DrUBA was able to prove convergence for this example (as was "AI").

Table 3 reports the number of times Algorithm 1 computed the image (successors) of a state until reaching the final r-d-plateau (Col. 3) and during the final plateau (Col. 4), as well as the total number of image computations without the *frontier* optimization (Col. 5). The table offers convincing evidence to support our heuristic that waiting for a long d-plateau at the end of exploration is not costly, answering Question 3.. On most benchmarks, the amount of computation done during the plateau (Col. 4) was negligible. This included our largest example, Bluetooth3 (2+3). The exception to this is the Stefan examples, which—as mentioned earlier—do not require any delays to reach the full abstract state set (the d-plateau starts at ($r_{\max}$,0)). Finally, a naive implementation that does not take advantage of monotonicity, forgoing the frontier approach to expanding the state set, was orders of magnitude worse. This is because it has to recompute the whole set for every iteration of r or d. This answers Question 4..

Comparing Col. 7 in Table 2 to the cutoff context-switch bounds from [18], we find that, while the r and d bounds were large, not all programs that needed large bounds took a long time to verify. For example, the Bluetooth3 (2+1) example took much less time than Stefan-5, despite requiring 21 more delays (with similar rounds). A hint for the reason can be found in Table 3. Once the set of abstract states is close to the $\overline{R}$, there are very few new states on the

Table 2. Benchmark description and running times for different algorithms. Threads: # of threads ($a + b$: the respective numbers of threads from two different templates); $\overline{R}$: number of reachable abstract states; Time: running time (sec) for each algorithm ("—": timeout or memory-out); $r_{\max}, d_{\max}$: round and delay counts at the *end* of each plateau when convergence was detected.

	1	2	3	4	5	6	7
	Benchmark	Threads	$\lvert\overline{R}\rvert$	Time: Algorithm 1	Time: "AI"	Time: "CUBA"	$r_{\max}, d_{\max}$ Algorithm 1
1	Bluetooth1	1+1	1010	.69	.92	.32	23, 15
		1+2	5468	3.32	6.79	2.25	32, 29
		2+1	18972	8.52	16.69	13.60	35, 26
2	Bluetooth2	1+1	1018	.71	.98	.29	23, 15
		1+2	5468	3.60	6.68	2.62	32, 29
		2+1	18972	8.81	16.68	13.97	35, 26
3	Bluetooth3	1+1	1018	.72	1.23	.41	23, 15
		1+2	5468	3.61	6.40	2.79	32, 29
		2+1	19002	9.97	16.27	14.50	35, 26
		2+2	94335	70.71	136.31	343.05	44, 40
		2+3	460684	654.47	2084.76	TO	56, 56
4	BST-Insert	1+1	272	.36	.49	.14	31, 16
		2+1	6644	3.62	5.25	10.09	49, 32
		2+2	14256	8.12	14.87	99.94	50, 38
5	Filecrawler	1+2	246	.37	.54	.05	20, 12
6	K-Induction	1+1	130	.51	.74	.48	20, 09
7	Proc-2	2+2	352	.56	.77	2.05	19, 20
8	Stefan	2	31	.24	.36	.04	13, 02
		4	687	13.99	13.82	20.33	32, 04
		5	3085	428.22	295.02	OOM	35, 05
		8	—	OOM	OOM	OOM	—
9	Dekker	2	1507	.82	1.62	.39	37, 16

frontier. We can see this in the small numbers in Col. 4, but it also applies to the round bound. If a state is rediscovered, it is not expanded in further round increments. Once the round bound is large enough, there are few deep schedules of maximum possible length (nr) that produce new concrete states.

6.2 Unbounded-Thread Experiments

We implemented Algorithm 1 in combination with predicate abstraction as detailed in Sect. 5.2 to check the effectiveness of our technique on a tricky concurrent program that requires unbounded variable domains. The Ticket Lock

Table 3. Detailed analysis of Algorithm 1, measuring the number of times the program computed the successors of a state. Col. 3 reports the image operations Algorithm 1 performed before reaching the FP (final plateau), Col. 4—the number of additional image operations computed until the program ended. Col. 5 shows the image operations without the *frontier* improvement, requiring recomputing each $\overline{R}(r, d)$ from the initial states.

	1	2	3	4	5
	Benchmark	Threads	Calls to *Image* → begin FP	Calls to *Image* Begin → end FP	Calls to *Image* w/o frontier
		1+1	4,034	1	339,261
1	Bluetooth1	1+2	23,441	3	4,758,084
		2+1	80,283	19	23,199,458
		1+1	4,103	1	350,587
2	Bluetooth2	1+2	23,493	3	4,780,778
		2+1	80,714	19	23,290,556
		1+1	4,096	8	348,851
		1+2	23,493	3	4,786,950
3	Bluetooth3	2+1	80,834	19	23,467,470
		2+2	478,426	2	283,910,446
		2+3	2,766,625	6	—
		1+1	780	1	82,130
4	BST-Insert	2+1	29,802	6	17,785,065
		2+2	62,190	25	34,335,106
5	Filecrawler	1+2	1,056	4	202,074
6	K-Induction	1+1	5,636	974	218,715
7	Proc-2	2+2	2,501	1,298	578,099
		2	367	59	500,494
8	Stefan	4	658,696	261,881	—
		5	10,299,293	6,621,157	—
		8	—	—	—
9	Dekker	2	3,636	2	688,836

protocol [3] and the predicates used to prove its correctness are shown in Fig. 2. In Line 0, threads wait to enter the Critical Section, whose code is at the beginning of Line 1; the rest of Line 1 is exit code to prepare the thread for re-entry. In the predicates on the right, subscript i denotes thread i's copy of a local variable.

This example has been shown to require significant adjustments to predicate abstraction to accommodate fixed-thread concurrency [8], and has been claimed to require an entirely new theory to cope with the unbounded-thread case [15]. We rely on the same predicates used in earlier work, and it is clear what motivates each predicate. P1 ensures t is a"new" ticket larger than previous ones, P2 is

```
   shared int $s$ := 0, $t$ := 0
   local int $l$ := fetch_and_add($t$)
0: while $s \neq l$ do;          ▷ wait for $s = l$
1: critical-section code here
   inc($s$)
   $l$ := fetch_and_add($t$)
   goto 0
```

P1: $\forall i : t > l_i$
P2: $|\{i : pc_i = 1\}| \geq 2$
P3: $s = l$
P4: $\forall i : i \neq \mathit{self} : l \neq l_i$

Fig. 2. Left: the Ticket Lock protocol; **Right:** four predicates used to prove it correct

used to check the safety property, P3 tracks the condition in Line 0, and P4 means that the operating thread's l is unique. DrUBA finds four abstract states for both 2 threads and 3 threads using Algorithm 1. This is an n-plateau of length 1.

To prove convergence for both Algorithm 1 and the "outer loop" incrementing n, we used the ACL2s theorem prover [7]. We specified the data in a concrete state, and the four abstract states that were found. Only the second statement disrespects this abstraction w.r.t. r, d and n, as we know the value of the test in the first statement for an abstract state. Given these, ACL2s was able to verify that the set of abstract states is closed under the semantics of statement 1. As a result, we can report that Ticket Lock is safe (P2 is invariantly false), for an arbitrary number of threads and arbitrary thread interleavings.

7 Discussion of Related Work

This work is inspired from two angles. The first is clearly the delay-bounded scheduling (DBS) technique [10]. The authors formalize this concept and show its effectiveness as a testing scheme. Their computational model of a dynamic task buffer is somewhat different from ours. We have not discussed dynamic thread creation here; it can be simulated by creating threads up-front and delaying them until such time as they are supposed to come into existence. The DBS paper also presents a sequentialization technique that can be turned into a symbolic verifier via verification-condition generation and SMT solving. This, however, requires bounding loops and recursion. Our approach combines exhaustive finite-state model exploration with convergence detection and thus does not suffer from these restrictions.

The second inspiration comes from an earlier context-unbounded analysis technique [18]. Similar in spirit to the present work, [18] started from a yet earlier context-bounded analysis technique and describes a condition under which a chosen context bound is sufficient to reach all states reachable under some abstraction. For the case of concurrent pushdown systems (CPDS)—the verification target of [18]—, the pop operation plays a crucial role in establishing this condition; note that, in our work, pop actions disrespect the top-of-the-stack abstraction commonly used for CPDS.

Our work has a number of advantages over [18]. First, and crucially, the set of states reachable under a context bound can be infinite (a single context can already generate infinitely many states); its determination thus requires more expensive symbolic reachability methods. In contrast, the reachability set under Round-Robin scheduling with a round- and a delay bound *is always finite*; moreover, it can be computed very easily, even for complex programs. This makes our technique a prime choice for lifting existing testing schemes to verifiers. A second advantage over [18] is that we retain much of the efficiency of the "almost deterministic" exploration delay-bounded scheduling, as demonstrated in Sect. 6. A downside of our work is that our convergence condition is sound only after a plateau has emerged of length roughly equal to the number of running threads; this is not required in [18]. However, as also demonstrated in Sect. 6, our efforts to compute reachable states for increasing r, d in a *frontier-driven* way nearly annihilates this drawback: in most cases, only a small number of image computations happen along the plateau.

An alternative to our verification approach is a classical analysis based on abstract interpretation [6]. Given function α, such analysis interprets the entire program abstractly, and then computes a fixed point under the abstract program's transition relation. This fixed point, if it exists, overapproximates the set of reachable abstract states. Hence, the absence of error states in the fixed point implies safety, but the presence of errors does not immediately permit a conclusion. In contrast, our technique interleaves *concrete* state space exploration (enabling genuine testing) with *abstraction-based* convergence detection. We believe this to be a useful approach in practical programming environments, where abstract proof engines with poorly understood bug-finding capabilities may be met with skepticism. A more detailed discussion of DrUBA vs. Abstract Interpretation can be found in the Appendix of [14].

Underapproximating program behaviors using bounding techniques is a widespread solution to address undecidability of safety verification problems. Examples include depth- [12] and context-bounding [16,17,20], delay-bounding [10], bounded asynchrony [11], preemption-bounding [19], and phase-bounded analysis [2,5]. Many of these bounding techniques admit decidable analysis problems [16,17,20] and thus have been successfully used in practice for bug finding. Round- and delay-bounded Round-Robin scheduling trivially renders safety decidable, since the delay-program is finite-state. In addition, it is very easy to implement, avoiding, for example, the need for symbolic data structures and algorithms to represent and process intermediate reachability sets.

8 Conclusion

We have presented an approach to enhancing delay-bounded scheduling in asynchronous programs with a convergence test that, if successful, certifies that all states from some chosen abstract domain have been reached. The resulting algorithm inherits from earlier work the capability to detect bugs efficiently, but can also prove safety properties, under arbitrary thread interleavings. It exploits the monotonicity of delay-bounded reachability sets to expand states and test for convergence only when needed. We have further demonstrated that, combined with predicate abstraction using powerful predicates, tricky unbounded-thread routines over unbounded data, such as the Ticket Lock, can be verified using substantially less machinery than proposed in earlier work. We have shown the experimental competitiveness of our approach against several related techniques.

References

1. Abdulla, P.A.: Well (and better) quasi-ordered transition systems. Bull. Symb. Logic **16**(4), 457–515 (2010)
2. Abdulla, P.A., Atig, M.F., Cederberg, J.: Analysis of message passing programs using smt-solvers. In ATVA, pp. 272–286 (2013)
3. Andrews, G.R.: Concurrent programming: Principles and practice. Benjamin-Cummings Publishing Co. (1991)
4. Ball, T., Majumdar, R., Millstein, T.D., Rajamani, S.K.: Automatic predicate abstraction of C programs. In PLDI, pp. 203–213 (2001)
5. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. Int. J. Softw. Tools Technol. Transf. **16**(2), 127–146 (2013). https://doi.org/10.1007/s10009-013-0276-z
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL, pp. 238–252 (1977)
7. Dillinger, P.C., Manolios, P., Vroon, D., Moore, J.S.: ACL2s: "the ACL2 Sedan". Electron. Notes Theor. Comput. Sci. **174**(2), 3–18 (2007)
8. Donaldson, A., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs. In: Computer Aided Verification (CAV), pp. 356–371 (2011)
9. Donaldson, A.F., Kaiser, A., Kroening, D., Wahl, T.: Symmetry-aware predicate abstraction for shared-variable concurrent programs (Extended Technical Report). CoRR, abs/1102.2330 (2011)
10. Emmi, M., Qadeer, S., Rakamaric, Z.: Delay-bounded scheduling. In: Principles of Programming Languages (POPL), pp. 411–422 (2011)
11. Fisher, J., Henzinger, T.A., Mateescu, M., Piterman, N.: Bounded asynchrony: Concurrency for modeling cell-cell interactions. In: Formal Methods in Systems Biology, pp. 17–32 (2008)
12. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL, pp. 174–186 (1997)
13. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: CAV, pp. 72–83 (1997)
14. Johnson, A., Wahl, T.:Delay-bounded scheduling without delay! (Extended Technical Report). CoRR, abs/2105.07277 (2021)
15. Kaiser, A., Kroening, D., Wahl, T.: Lost in abstraction: monotonicity in multi-threaded programs. Inf. Comput. (IaC) **252**, 30–47 (2017)
16. La Torre, S., Parthasarathy, M., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222 (2009)
17. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. Form. Methods Syst. Des. **35**(1), 73–97 (2009)

18. Liu, P., Wahl, T.: CUBA: interprocedural context-unbounded analysis of concurrent programs. In: Programming Languages Design and Implementation (PLDI), pp. 105–119 (2018)
19. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455 (2007)
20. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS, pp. 93–107 (2005)