Non-Monotonic Refinement of Control Abstraction for Concurrent Programs

Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko

Technische Universität München

Abstract. Verification based on abstraction refinement is a successful technique for checking program properties. Conventional abstraction refinement schemes increase precision of the abstraction monotonically, and therefore cannot recover from overly precise refinement decisions. This problem is exacerbated in the context of multi-threaded programs, where keeping track of all control locations in concurrent threads is the inevitably discovered abstraction and is prohibitively expensive. In contrast to the conventional (partition refinement-based) approaches, non-monotonic abstraction refinement schemes rely on re-partitioning and have promising potential for avoiding excess of precision. In this paper, we propose a non-monotonic refinement scheme for the control abstraction (of concurrent programs). Our approach employs a constraint solver to discover re-partitioning at each refinement step. An experimental evaluation of our non-monotonic control abstraction refinement on a collection of multi-threaded verification benchmarks indicates its effectiveness in practice.

1 Introduction

Automatic abstraction [10] is one of the essential components for the construction of software verification tools. The success of verification tools based on abstract domains equipped with widening operators, e.g., ASTREE [5], Clousot [13], and Dagger [17], and software model checkers based on predicate abstraction, e.g., SLAM/SDV [3], Blast [21], Magic [6], F-Soft [23], Terminator [9], and ARMC [30], demonstrates the effectiveness of abstraction in practice. Finding the right abstraction is a difficult task, since a too coarse abstraction may lead to inconclusive verification results and, on the other hand, excess of precision may impose a prohibitive efficiency penalty. In practice, the desired level of details tracked during the abstraction process is determined through a trial-and-error like process that adjusts abstraction at each failed verification attempt. The existing refinement methods can automatically tune precision of various abstraction techniques, including infinite abstract domains equipped with widening operators, e.g., [11, 17], and finitary predicate abstraction domains, e.g., [1, 4, 7, 21, 22].

One of the most important properties of the iterative abstraction discovery approaches is called progress of refinement. This property ensures that the verification effort does not get stuck in a loop trying to eliminate the same reason

for imprecision over and over again. The majority of existing approaches achieve the progress of refinement by adjusting abstraction *monotonically*. That is, they compute a proper refinement of abstraction at each adjusting step. For example, when using predicate abstraction such monotonic refinement can be easily achieved by adding appropriate predicates to the abstract domain [1,7,21,22,24].

While monotonic refinement approaches are well-studied and widely applied, the monotonicity property can lead to overly precise abstraction and hence unnecessarily slow down verification. In fact, monotonicity is just one possible way to achieve refinement progress and alternative approaches have started to emerge. As an example, consider 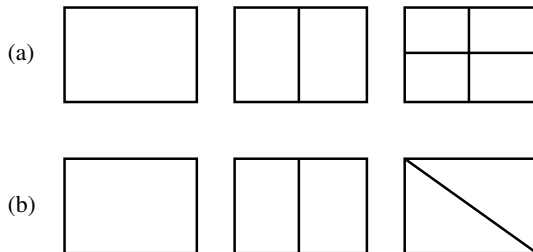two sequences of abstraction adjustments shown in Figure 1. The sequence (a) uses a monotonic scheme that creates a properly

Fig. 1. Abstraction sets of program states using equivalence classes. Boxes denote equivalence classes.

refined partition at each adjustment step. Assume that after making the first adjustment the verifier recognizes that a re-partitioning following the sequence (b) yields an abstraction that is sufficiently precise to prove the property. While not admissible in monotonic refinement scheme, (b) can be achieved using a non-monotonic refinement scheme, which would lead to more efficient verification that considers two instead of four equivalence classes. The potential of such non-monotonic refinement schemes has been identified in recent verification efforts. In a seminal paper [29], a non-monotonic scheme is used to discover a localization abstraction which is improved based on proofs of unsatisfiability. A second related work describes a method for computing an optimal localization abstraction from a collection of broken traces [19]. These approaches led to effective verification methods able to recover from overly precise abstraction decisions.

In this paper, we apply a non-monotonic abstraction refinement scheme for the control-flow abstraction of multi-threaded programs. Abstraction of control-flow deals with program counter variables that range over finite sets of control locations of program threads, and is a crucial building block for achieving scalable reasoning about concurrent programs [8,15]. In our experience, monotonic abstraction results in too fine grained partitioning of control locations into equivalence classes and hence is too expensive.

The main component of our scheme is a procedure that takes as input a set of program paths that were the root cause for failed verification attempts so far (including the current evidence for inadequacy of chosen abstraction) and returns a set of predicates that eliminates all these failures. The abstract domain

```
int t1 = 0, t2 = 0; // ticket variables
bool choosing1 = 0, choosing2 = 0; // boolean flags
int x; //variable to update in critical section


void thr1() {                          void thr2() {
    int tmp;                               int tmp;
1   choosing1 = 1;                     11  choosing2 = 1;
2   tmp = t2 + 1;                      12  tmp = t1 + 1;
3   t1 = tmp;                          13  t2 = tmp;
4   choosing1 = 0;                     14  choosing2 = 0;
5   while (choosing2 != 0);            15  while (choosing1 != 0);
6   while (t1 >= t2 && t2 != 0);       16  while (t2 >= t1 && t1 != 0);
    // begin: critical section             // begin: critical section
7   x = 0;                             17  x = 1;
8   assert(x <= 0);                    18  assert(x >= 1);
    // end: critical section               // end: critical section
9   t1 = 0;                            19  t2 = 0;
10 }                                   20 }
```

Fig. 2. An implementation of Lamport's Bakery algorithm.

is adjusted by *replacing* the previously used set of predicates by the output of the above procedure (and not adding predicates as in conventional CEGAR-based approaches). More specifically, our algorithm crucially relies on a repartitioning step encoded as a SAT problem.

We implemented the two schemes for abstraction refinement and observed on a set of multi-threaded examples that non-monotonic refinement enables an improvement in the verification time ranging from 18% to 52% when compared to a monotonic refinement scheme.

In summary, our contributions are a non-monotonic abstraction refinement algorithm for control-flow abstraction, its implementation and experimental evaluation.

2   Example

In this section, we illustrate our approach on Lamport's Bakery algorithm, which is a classic verification benchmark. We use the complete version of the algorithm [25] with a set of Boolean flags (`choosing1` and `choosing2`) where the reading and the incrementing of the ticket variable is done non-atomically (see lines 2 and 3). We are interested in verifying that the Bakery algorithm achieves mutual exclusion. This safety property is instrumented in Figure 2 using a global variable `x` in the critical section of the two threads. We want to prove that no interleaving of the threads leads to an assertion violation at either line 8 or 18.

To prove the program correct, our algorithm performs a combination of standard abstract reachability computation and non-monotonic abstraction refine-

ment. Abstract states represent sets of concrete program states. If the reachability computation finds an error state to be reachable, we analyze the reason for the failure and update the abstraction, if possible.

For our example, a reason for failure to prove safety is the following interleaving of statements from first and second threads

$$(\underbrace{1,2,3,4,5,}_{\text{thr1}} \underbrace{13,}_{\text{thr2}} \underbrace{6,7,}_{\text{thr1}} \underbrace{17,}_{\text{thr2}} \underbrace{8}_{\text{thr1}}),$$

where we identify program statements by the corresponding line numbers. This counterexample is in fact infeasible, and is discovered due to abstraction. Two different reasons make this counterexample infeasible:

- the first statement executed from the second thread cannot be 13,
- the statement 13 cannot be followed in the second thread by the statement 17.

The mismatches between the program locations that lead to the infeasibility of the counterexample are denoted using the following notation: $(11 \not\equiv 13)$, $(14 \not\equiv 17)$. An abstraction function that maps the concrete program locations 11 and 13 to different abstract program locations will be able to avoid this counterexample in subsequent reachability iterations. Similarly, this counterexample can be avoided if the concrete locations 14 and 17 map to different abstract locations $(14 \not\equiv 17)$.

Let us assume that the refinement procedure picks the first possibility. The resulting abstraction function can be represented using the following partition of program locations: $\{11\}, \{13\}, PC_2 \backslash \{11, 13\}, PC_1$. PC_1 and PC_2 represent the sets of all program locations from the first and, respectively, second thread.

A subsequent reachability computation finds another counterexample that can be shown infeasible with the following mismatch relation, $12 \not\equiv 16$. To avoid this counterexample, the partitioning of the locations is updated to $\{11, 12\}, \{13, 16\}, PC_2 \backslash \{11, 12, 13, 16\}, PC_1$. With a third mismatch relation $11 \not\equiv 12$, the program locations are again repartitioned to $\{11, 16\}, \{12, 13\}, PC_2 \backslash \{11, 12, 13, 16\}, PC_1$. Note that these repartitionings are possible only with a non-monotonic abstraction refinement scheme. The standard monotonic refinement would compute a more fine-grained partitioning that is unnecessarily precise and leads to expensive abstract reachability computations. Overall, the verification of the Bakery example using non-monotonic abstraction refinement concludes after 26 seconds and computes a 10-way partitioning of control locations.

The monotonic abstraction refinement concludes after 54 seconds. The mismatch $11 \not\equiv 13$ and the second mismatch $12 \not\equiv 16$ lead immediately to the partitioning $\{11\}, \{12\}, \{13\}, \{16\}, PC_2 \backslash \{11, 12, 13, 16\}, PC_1$. Through all the reachability iterations, the control locations are split into 14 partitions and this large number explains the increased verification time based on monotonic abstraction refinement.

3 Preliminaries

In this section we define programs and computations, and provide a brief description of predicate abstraction-based approach to program verification together with a standard counterexample-guided abstraction refinement procedure.

Programs and computations We assume an abstract representation of programs by transition systems [27]. A *program* $P = (\Sigma, s_\mathcal{I}, \mathcal{T}, s_\mathcal{E})$ is given by a set of program *states* Σ, an *initial* state $s_\mathcal{I} \in \Sigma$, a set of *transitions* $\mathcal{T}$, and an *error* state $s_\mathcal{E} \in \Sigma$. Each transition $\tau \in \mathcal{T}$ has a corresponding *transition relation* $\rho_\tau \subseteq \Sigma \times \Sigma$. The error state $s_\mathcal{E}$ is used to represent assertion statements commonly present in programming languages. Each failed assertion leads to $s_\mathcal{E}$.

A *computation* of P is a sequence of states $s_1, s_2, \ldots$ such that s_1 is the initial state, i.e., $s_1 = s_\mathcal{I}$, and there is a transition $\tau \in \mathcal{T}$ between each pair of consecutive states s and s', i.e., $(s, s') \in \rho_\tau$. A state s is *reachable* if it appears in some computation. The program is *safe* if the error state is *not* reachable in any computation.

A *path* is a sequence of transitions. Let $\circ$ be the *relational composition* function for binary relation over states, i.e., for $X, Y \subseteq \Sigma \times \Sigma$ we have $X \circ Y = \{(s, s') \mid \exists s'' \in \Sigma : (s, s'') \in X \wedge (s'', s') \in Y\}$. Then, a path relation ρ_π is a relational composition of transition relations along the path, i.e., for $\pi = \tau_1 \ldots \tau_n$ we have $\rho_\pi = \rho_{\tau_1} \circ \cdots \circ \rho_{\tau_n}$. A path is *feasible* if its path relation is not empty.

Predicate abstraction Our goal is to verify whether a given program is safe. To achieve this goal we need to consider all reachable program states and check if the error state appears among them. The set of all reachable states can be computed iteratively using the function $post : (\mathcal{T} \times 2^\Sigma) \to 2^\Sigma$ such that

$$post(\tau, S) = \{s' \mid \exists s \in S : (s, s') \in \rho_\tau\} \ .$$

Its least fixed point above $\{s_\mathcal{I}\}$ is the set of reachable states, i.e.,

$$s \text{ is reachable if and only if } s \in lfp(\lambda S. \bigcup_{\tau \in \mathcal{T}} post(\tau, S), \{s_\mathcal{I}\}) \ .$$

The exact computation of the set of reachable states is an undecidable problem, however for the verification purposes a sufficiently close *abstraction* is enough. The framework of abstract interpretation [10] provides a formal foundation for the approximate, yet sound abstraction of reachable states, where abstraction is defined as an *over-approximation*. Given an abstraction function $\alpha : 2^\Sigma \to 2^\Sigma$ such that

$$\forall S \subseteq \Sigma : S \subseteq \alpha(S) \ ,$$

we construct an *abstraction* $post^\#$ of $post$ as follows:

$$post^\#(\tau, S) = \alpha(post(\tau, S)) \ .$$

Our abstraction puts together and operates on sets of program states. We call such sets *abstract states* and let $\Sigma^\# = 2^\Sigma$ be the set of all abstract states.

The least fixed point of $post^\#$ above the abstraction of the initial state is an over-approximation of the reachable states, i.e.,

$$lfp(\lambda S. \bigcup_{\tau \in \mathcal{T}} post^\#(\tau, S), \alpha(\{s_\mathcal{I}\})) \supseteq lfp(\lambda S. \bigcup_{\tau \in \mathcal{T}} post(\tau, S), \{s_\mathcal{I}\}) \ .$$

If the error state is not included in the over-approximation then the program is safe, that is, we obtain a sound method for verifying program safety. For completeness of presentation, Appendix A contains an algorithm for abstract fixpoint checking together with the re-construction of counterexamples, which is required by our refinement scheme.

The abstraction function α can be constructed automatically from a given set of basic building blocks, called *predicates*, where a predicate represents a set of program states. Given a set of predicates $\mathcal{P} = \{P_1, \ldots, P_n\}$, where $P_i \subseteq \Sigma$, and a *theorem prover* that can decide validity of subset inclusion between sets of states represented in a logical language, we use an abstraction function $\alpha^\mathcal{P} : 2^\Sigma \to 2^\Sigma$ which returns the strongest conjunction of the predicates implied by S as follows.

$$\alpha^\mathcal{P}(S) = \cap\{P \in \mathcal{P} \mid S \subseteq P\}$$

Abstraction refinement In order to verify program safety using predicate abstraction, we need to supply a set of predicates. Predicates can be provided manually, collected from the program text by applying heuristics, or derived in a goal-oriented way by using the counterexample-guided abstraction refinement approach [7]. The crux of this approach to predicate discovery lies in leveraging *spurious counterexamples*, which are program paths that expose the coarseness of the abstraction function determined by the currently used set of predicates.

A path $\pi = \tau_1 \ldots \tau_n$ is a spurious counterexample if the abstract reachability computation along the path leads to the error states, i.e.,

$$s_\mathcal{E} \in post^\#(\tau_n, post^\#(\tau_{n-1}, \ldots post^\#(\tau_1, \alpha^\mathcal{P}(\{s_\mathcal{I}\})))) \ ,$$

but the actual, not abstracted path does not lead to the error state, i.e., $(s_\mathcal{I}, s_\mathcal{E}) \notin \rho_\pi$. Conventional techniques for analyzing spurious counterexamples use automated reasoning approaches , e.g., proofs [22] and interpolation [21], to extract a set of new predicates that *excludes* the spurious counterexample.

We define an auxiliary predicate *SafeInd* that takes as input a sequence of predicates of length $n + 1$ and a sequence of program transitions of length n, where $n \geq 1$, as follows.

$$SafeInd(\varphi_0 \ldots \varphi_n, \tau_1 \ldots \tau_n) = \ s_\mathcal{I} \in \varphi_0 \wedge s_\mathcal{E} \notin \varphi_n \ \wedge$$
$$\forall i \in 1..n : \bigwedge_{i \in 1..n} post(\tau_i, \varphi_{i-1}) \subseteq \varphi_i$$

Given a spurious counterexample $\tau_1 \ldots \tau_n$, we say that the sequence of predicates $\varphi_0 \ldots \varphi_n$ excludes the counterexample if $SafeInd(\varphi_0 \ldots \varphi_n, \tau_1 \ldots \tau_n)$ holds. We will use *SafeInd* in our non-monotonic refinement scheme.

```
        function NONMONREFINE
        input
           Paths : spurious counterexamples so far
           π : current spurious counterexample
        begin
1       choose 𝒫 such that
2            ∀τ_1 . . . τ_n ∈ {π} ∪ Paths
3                 ∃φ_0, . . . , φ_n ⊆ 𝒫 :
4                      SafeInd(φ_0 . . . φ_n, τ_1 . . . τ_n)
5       return 𝒫
        end
        procedure NONMONCEGAR
        input
           P : program
        vars
           𝒫 : abstraction predicates
           Paths : spurious counterexamples so far
        begin
1         𝒫 := ∅
2         Paths := ∅
3         repeat
4            match FINDCOUNTEREXAMPLE(P, 𝒫) with
5             | Some π ->
6               if ρ_π = ∅ then
7                   𝒫 := NONMONREFINE(Paths, π)
8                   Paths := {π} ∪ Paths
9               else
10                  return "Counterexample π to program safety"
11            | None ->
12               return "Program is safe"
        end.
```

Fig. 3. Predicate abstraction-based algorithm for checking program safety that is based on the non-monotonic abstraction refinement scheme.

4 Non-monotonic refinement scheme

In this section we present an abstraction refinement scheme that adjusts abstraction in a non-monotonic way. By not committing to a monotonic evolution of the abstraction, we can obtain a greater choice of possible refinement steps and hence can reach more favorable efficiency/precision trade-offs.

See Figure 3 for an algorithm NONMONCEGAR that implements a safety verification procedure based on counterexample guided abstraction refinement using the non-monotonic refinement NONMONREFINE. The algorithm NON-MONCEGAR crucially differs from a conventional CEGAR algorithm by keeping the history of discovered counterexamples, as stored in the variable Paths. Every time an infeasible counterexample π is found, see lines 4–6, the set of pre-

viously discovered counterexamples π together with the current one is passed to NonMonRefine. The function NonMonRefine in our scheme chooses a set of predicates $\mathcal{P}$ that excludes all counterexamples discovered so far. In Section 5 we present an instantiation of NonMonRefine for control-flow abstraction that uses an encoding into SAT to implement lines 1–4 of NonMonRefine.

NonMonCEGAR overwrites the set of abstraction predicates $\mathcal{P}$ using the result of calling NonMonRefine on the current set of counterexamples. At this step, non-monotonicity takes place. Note however that the progress of refinement is guaranteed, as formalized by the theorem below.

Theorem 1 (Progress of refinement in NonMonCEGAR). *The algorithm* NonMonCEGAR *never discovers the same counterexample twice, i.e., for given values of $\mathcal{P}$ and* Paths *we have that if $\pi \in$* Paths *then $\pi \notin$* FindCounterexample$(P, \mathcal{P})$.

5   Non-monotonic refinement for control-flow abstraction

In this section we present an application of the non-monotonic abstraction refinement scheme to control-flow abstraction for concurrent programs. Our algorithm for the verification of multi-threaded programs [20] relies on the abstraction of control-flow, i.e., over-approximation of set of control locations in which threads can be residing. This abstraction plays a crucial role for enabling scalable reasoning in the multi-threaded setting. Our experiments with a conventional monotonic abstraction refinement procedure for dealing with control-flow abstraction were not satisfactory. The refinement process was creating as many individual abstract values as there are control locations, which subverted the application of abstraction by effectively making the abstraction function to be an identity function. In this section, we only present the non-monotonic control abstraction refinement and refer to [20] for its client algorithm.

We assume a multi-threaded program that consists of N threads whose control locations are given by the set $\mathcal{L}$. For each thread $i \in 1..N$ we use a variable pc_i and its primed version pc_i' to refer to the corresponding program counter value. We consider counterexamples given by sequences of transitions whose transition relations are of the form

$$pc_i = \ell \wedge pc_i' = \ell' \wedge \bigwedge_{j \in 1..N \setminus \{i\}} pc_j = pc_j' \ ,$$

where ℓ and ℓ' are control locations. This transition relation corresponds to a step of the thread i, whereas each other thread $j \in 1..N \setminus \{i\}$ idles and hence does not change its control location. We assume a function $from : \mathcal{T} \to \mathcal{L}$ that given a transition τ returns its start location, which ℓ for the transition relation above.

For example, the counterexample $\pi = \tau_1 \tau_2 \tau_3 \tau_4 \tau_5 \tau_{13} \tau_6 \tau_7 \tau_{17} \tau_8$ presented in Section 2 involves the following transition relations:

$$\rho_i = \begin{cases} pc_1 = i \wedge pc_1 = i+1 \wedge pc_2 = pc_2' \ , & \text{for } i \in \{1, \ldots, 8\} \ , \\ pc_1 = pc_1' \wedge pc_2 = i \wedge pc_2' = i+1 \ , & \text{for } i \in \{13, 17\} \ . \end{cases}$$

function NonMonControlRefine
input
    Paths : spurious counterexamples so far
vars
    Φ, Ψ : auxiliary constraints
    m : number of partitions
    B : auxiliary propositional variables
    $bits$: encodes equivalence classes of control locations as bit strings
begin
1    $m := 2$
2   **repeat**
3        $B := \emptyset$
4   **for** each $\ell \in \mathcal{L}$ **do**
5            $b_1 \ldots b_{\lceil \log_2(m) \rceil} :=$ fresh propositional variables
6            $B := \{b_1, \ldots, b_{\lceil \log_2(m) \rceil}\} \cup B$
7            $bits(\ell) := b_1 \ldots b_{\lceil \log_2(m) \rceil}$
8        $\Phi :=$ true
9   **for** each $\pi = \tau_1 \ldots \tau_n \in$ Paths **do**
10         $\Psi :=$ false
11   **for** each $k \in 0..n$ and $i \in 1..N$ and $j \in 1..n-k+1$ and $\ell \in \mathcal{L}$ such that

$$SafeInd(\underbrace{\text{true} \ldots \text{true}}_{k \text{ times}} \underbrace{pc_i = \ell \ldots pc_i = \ell}_{j \text{ times}} \underbrace{\text{false} \ldots \text{false}}_{n-k-j+1 \text{ times}}, \pi)$$

12   **do**
13            $\ell' :=$ **if** $k+j = n+1$ **then** $s_{\mathcal{E}}(pc_i)$ **else** $from(\tau_{k+j})$
14            $\Psi := \Psi \lor bits(\ell) \neq bits(\ell')$
15        $\Phi := \Psi \land \Phi$
16        $m := m + 1$
17   **until** exists $\sigma : B \rightarrow \{\text{true}, \text{false}\}$ such that $\models \sigma(\Phi)$
18   **for** each $\ell \in \mathcal{L}$ **do**
19        $f_{\equiv}(\ell) := \{\ell' \in \mathcal{L} \mid \sigma(bits(\ell)) = \sigma(bits(\ell'))\}$
20   **return** $f_{\equiv}$
end.

Fig. 4. Function NonMonControlRefine implements an instantiation of the non-monotonic refinement scheme to control-flow abstraction. The application $\sigma(bits(\ell))$ computes a bit string by replacing propositional variables from $bits(\ell)$ by their values as determined in σ.

The transitions have the following starting locations:

$$from(\tau_i) = i \ , \ \text{for } i \in \{1, \ldots, 8, 13, 17\}.$$

Our goal is to compute an equivalence relation $\equiv$ on $\mathcal{L}$ that leads to absence of abstract counterexamples. We represent the equivalence relation by a characteristic function $f_\equiv : \mathcal{L} \to 2^{\mathcal{L}}$ from control locations to equivalence classes. The equivalence classes of this relation are used as predicates defining control abstraction for a thread $i \in 1..N$, i.e.,

$$\alpha(S) = \cup\{f_\equiv(\ell) \mid (S \cap (pc_1 = \ell \cup \cdots \cup pc_N = \ell)) \neq \emptyset\} \ .$$

For example, Section 2 first discovers an equivalence relation that consists of three equivalence classes $\{11\}$, $\{13\}$, and $PC_2 \backslash \{11, 13\}$. This equivalence relation yields a control-flow abstraction that, for example, yields the following result:

$$\alpha(\{pc_2 = 11, pc_2 = 16\}) = \{11\} \cup PC_2 \setminus \{11, 13\} \ .$$

The following observation underlines our algorithm for non-monotonic refinement of control-flow abstraction. Each spurious counterexample, say $\tau_1 \ldots \tau_n$ can be eliminated by keeping track of a certain predicate $pc_i = \ell$, i.e., if $\rho_{\tau_1 \ldots \tau_n} = \emptyset$ then there exists $i \in 1..N$ and $\ell \in \mathcal{L}$ such that for $k \in 1..n$ and $j \in 1..n - k + 1$ holds

$$SafeInd(\underbrace{\mathsf{true} \ldots \mathsf{true}}_{k \text{ times}} \underbrace{pc_i = \ell \ldots pc_i = \ell}_{j \text{ times}} \underbrace{\mathsf{false} \ldots \mathsf{false}}_{n-k-j+1 \text{ times}}, \tau_1 \ldots \tau_n) \ .$$

For our counterexample π shown above, one refinement possibility is given below:

$$SafeInd(\underbrace{pc_2 = 11 \ldots pc_2 = 11}_{6 \text{ times}} \underbrace{\mathsf{false} \ldots \mathsf{false}}_{5 \text{ times}}, \pi) \ .$$

Figure 4 shows an algorithm NONMONCONTROLREFINE that computes a characteristic function for adjusting the control-flow abstraction. The algorithm finds an equivalence relation with the minimal number of equivalence classes, which decreases the size of the abstract state space and improves efficiency of the abstract reachability computation. Our implementation relies on a propositional encoding that describes constraints on the characteristic functions. These constraints can be solved efficiency using a state-of-the-art SAT solver.

We illustrate NONMONCONTROLREFINE using the counterexample π above, which is taken from Section 2, and assume that the input set Paths contains only the path π. Line 1 in Figure 4 initializes the number of equivalence classes m to 2, which serves as the first candidate. The **repeat** loop (lines 2–17) attempts to find a control abstraction with at most m equivalence classes. If no such abstraction exists then m is incremented and the attempt is repeated. This iteration terminates after at most $|\mathcal{L}|$-many steps, where $|\mathcal{L}|$ is the size of $\mathcal{L}$.

At the first attempt, we start by creating propositional variables that keep track of equivalence classes for control locations, see lines 4–7. For our example, we assume $bits(11) = (b_1 b_2)$ and $bits(13) = (b_3 b_4)$, and hence B contains $\{b_1, b_2, b_3, b_4\}$.

Since Paths contains only one counterexample, namely π, the **for** loop (lines 9–15) is executed only once. This path has two root causes of infeasibility, which leads to two iterations of the inner **for** loop in lines 11–14. At the first one we obtain $pc_2 = 11$, $k = 0$, $j = 6$, and $\ell' = from(\tau_{13}) = 13$. Then, line 14 computes $\Psi = $ false $\vee\ bits(11) \neq bits(13)$. This constraint encodes the condition that the control locations 11 and 13 need to be distinguished by the control abstraction, formally, $11 \not\equiv 13$.

The next iteration of the inner **for** loop discovers that for $k = 6$ and $j = 3$ we have

$$SafeInd(\underbrace{\text{true} \ldots \text{true}}_{6 \text{ times}} \underbrace{pc_2 = 14 \ldots pc_2 = 14}_{3 \text{ times}} \text{false false}, \pi) \ ,$$

and $\ell' = from(\tau_{17}) = 17$. We finish the execution of the inner **for** loop and obtain the final constraint

$$\Phi = \ bits(11) \neq bits(13) \vee bits(14) \neq bits(17) \ .$$

The first disjunct in Φ requires that at least one bit of $bits(11)$ is different from the corresponding bit in $bits(13)$. This condition translates to $(b_1 \neq b_3 \vee b_2 \neq b_4)$, which is equivalent to $(b_1 \wedge \neg b_3) \vee (\neg b_1 \wedge b_3) \vee (b_2 \wedge \neg b_4) \vee (\neg b_2 \wedge b_4)$.

The constraint Φ is satisfiable. We consider a solution σ such that $\sigma(bits(11)) = (0\,0)$, $\sigma(bits(13)) = (0\,1)$, $\sigma(bits(14)) = (0\,0)$, and $\sigma(bits(17)) = (0\,0)$. This solution leads to the characteristic function $f_\equiv$ that maps 11, 14, and 17 to the same equivalence class. This equivalence class is different from $f_\equiv(13)$.

At each refinement iteration more and more conjuncts are added to the constraint Φ in line 15. As an additional optimization, we first try to find same number of partitions among program counters as the number of partitions found in the last iteration. If this fails, then we grow the number of partitions one by one. In the worst case, the partition size may grow upto the number of program locations. However, in our experiments the number of control partitions was much lower indicating the benefit of control abstraction.

6 Experiments

We implemented the algorithm NONMONCONTROLREFINE in our tool for the verification of multi-threaded programs written in the C language. Since our tool uses both data abstraction and control abstraction, it may be possible that some spurious counterexample can be ruled out by both data abstraction refinement and control abstraction refinement. In this situation, we use an heuristic that prefers data refinement over control refinement. Our tool uses a standard (i.e. monotonic) abstraction refinement scheme for dealing with data variables, and relies on NONMONCONTROLREFINE for the discovery of adequate control abstraction. Constraints generated by NONMONCONTROLREFINE are resolved using the Z3 solver [12]. Next, we will report our experience with applying NON-MONCONTROLREFINE to the verification of multi-threaded programs.

Program	Monotonic refinement			Non-monotonic refinement		
	Time		$\mid\equiv\mid$	Time		$\mid\equiv\mid$
BAKERY-ATOMIC [27]	6.6s	5.7+0.9	8	4.8s	4.1+0.7	7
BAKERY [25]	54s	48.4+5.6	14	26s	23.1+2.9	10
BLUETOOTH [31]	19.5s	16.4+3.1	7	16.4s	11.3+5.1	5
MOZILLA-ORDER-FIXED [26]	2.7s	2.1+0.6	5	1.6s	0.9+0.7	3
TIME-VARYING-MUTEX [14]	9.6s	8.7+0.9	10	7.1s	6.3+0.8	7

Table 1. Comparison between monotonic and non-monotonic refinement of control abstraction. For each configuration we present (i) the total verification time, its decomposition into time spent on (ii) the abstract reachability computation and (iii) abstraction refinement, together with (iv) the number of equivalence classes $\mid\equiv\mid$ that determine the control abstraction.

We evaluated the non-monotonic refinement scheme in direct comparison with monotonic one and present a summary in Table 1. Our examples include two versions of the Bakery algorithm for mutual exclusion. BAKERY [25] is shown in Figure 2, while BAKERY-ATOMIC is its simplified version that increments the ticket variable atomically [27]. BLUETOOTH models the stopping procedure of a Windows NT Bluetooth driver [31], where a worker thread asserts that a boolean flag stopped is not set to false by a second stopper thread. MOZILLA-ORDER-FIXED is the fixed version of a vulnerability from the MOZILLA CVS repository, which was discussed in abbreviated form in [26, Figure 2]. The property to verify is that two operations performed by different threads are executed in the correct order. Lastly, TIME-VARYING-MUTEX illustrates a synchronization idiom found in the Frangipani file system [14], where it is verified if a thread has exclusive access over a disk block.

At a high level, our approach can be viewed as an optimization step with a trade-off. While the non-monotonic refinement keeps the number of equivalence classes $\mid\equiv\mid$ smaller, it has to solve a growing set of constraints which may impact on the refinement time. On our set of examples, we observed that the increase in refinement time is acceptable and the coarser abstraction that is discovered leads to a smaller time for abstract reachability computation. Consequently, the time for non-monotonic verification compares favorably to that for verification via monotonic refinement. We found overall time savings ranging from 18% for BLUETOOTH to 52% for BAKERY.

7   Related work

Our paper builds upon counterexample-based model checking [2, 6, 7, 22], which mostly employs monotonic refinement techniques that consider a single counterexample at a time and are based on weakest preconditions [2] and interpolation [21]. Our non-monotonic scheme eliminates all previously discovered spurious counterexample, which is in contrast to the elimination of all spurious counterexamples of a given length [16].

Previous non-monotonic abstraction refinement approaches focus on data refinement, see e.g. [19,28]. The collection of broken traces in [19] is closely related to our history of counterexamples. While [19] identifies which data variables to keep track by analysing broken traces, our approach first employs a constraint-based reduction, which may be viewed as a generalization. The non-monotonic abstraction refinement using interpolants [28] avoids explicit construction of abstract state transformer that is usually required for program verification. Instead, an interpolation procedure simultaneously adjusts precision for all previously discovered spurious counterexamples. In contrast to [28], our non-monotonic control abstraction imposes additional constraints on the form of the obtained abstraction using constraints.

Monotonicity plays a crucial role for widening operators in abstraction interpretation framework [10] and its automatic refinement [11,17,18,32]. Refinement techniques for widening achieve monotonicity by considering results of abstract reachability tree computation from the previous iterations, see e.g. [17,18]. We are not aware of non-monotonic refinement in this domain.

References

1. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS*, 2002.
2. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, 2001.
3. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.
4. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, 2003.
6. S. Chaki, E. M. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.
7. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
8. A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. *Formal Methods in System Design*, 34(2):104–125, 2009.
9. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
10. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
11. P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS*, 2007.
12. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
13. P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code for .NET. In *OOP-SLA*, 2008.
14. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *ESOP*, pages 262–277, 2002.
15. C. Flanagan and S. Qadeer. Thread-modular model checking. In *SPIN*, 2003.

16. M. Glusman, G. Kamhi, S. Mador-Haim, R. Fraer, and M. Y. Vardi. Multiple-counterexample guided iterative abstraction refinement: An industrial evaluation. In *TACAS*, 2003.
17. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS*, 2008.
18. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, 2006.
19. A. Gupta and E. M. Clarke. Reconsidering CEGAR: Learning good abstractions without refinement. In *ICCD*, pages 591–598, 2005.
20. A. Gupta, C. Popeea, and A. Rybalchenko. In preparation. http://www.model.in.tum.de/~popeea/research/environment.pdf, 2009.
21. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
23. F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *CAV*, 2005.
24. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, 2007.
25. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.
26. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.
27. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, 1995.
28. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.
29. K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. In *TACAS*, pages 2–17, 2003.
30. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. In *PADL*, 2007.
31. S. Qadeer and D. Wu. KISS: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.
32. F. Ranzato, O. Rossi-Doria, and F. Tapparo. A forward-backward abstraction refinement algorithm. In *VMCAI*, 2008.

A Abstract fixpoint checking

In this appendix we briefly revisit abstract fixpoint checking together with the re-construction of counterexample paths. See Figure 5 for the algorithm FIND-COUNTEREXAMPLE. The algorithm takes as input a program and a set of predicates defining the abstraction function. The computation of abstract reachable states is implemented using a queue of abstract states whose successors are yet to be computed. In order to be able to re-construct a counterexample path in case the error state of the program is reached (see line 8), the auxiliary relation Parent keeps track of how each abstract state is reached. The counterexample re-construction is performed in lines 9–13 via a backward traversal.

function FINDCOUNTEREXAMPLE
input
    P : program
    $\mathcal{P}$: abstraction predicates
vars
    Reach : reached abstract states
    Parent : parent relation
    Queue : queue of abstract states
    n, n' : abstract states
    τ : program transition
begin

1   Parent := $\emptyset$
2   Reach := $\{\alpha^{\mathcal{P}}(\{s_{\mathcal{I}}\})\}$
3   add $\alpha^{\mathcal{P}}(\{s_{\mathcal{I}}\})$ to Queue
4   **while** Queue is not empty **do**
5   n := take from Queue
6   **for** each $\tau \in \mathcal{T}$ **do**
7   n' := $post^{\#}(n, \tau)$
8   **if** $s_{\mathcal{E}} \in n'$ **then**
9            $\pi := \tau$
10   **while** exists n and τ such that $(n, \tau, n') \in$ Parent **do**
11   n' := n
12          $\pi := \tau' \cdot \pi$
13   **done**
14   **return Some** π
15   **else if** $\neg(\exists m \in$ Reach $: n' \subseteq m)$ **then**
16   add n' to Queue
17   Reach := $\{n'\} \cup$ Reach
18   Parent := $\{(n, \tau, n')\} \cup$ Parent
19   **done**
20   **return None**
    end

Fig. 5. Abstract fixpoint checking algorithm combined with a counterexample construction step.