Language-Theoretic Abstraction Refinement

Zhenyue Long[1,2,3,⋆], Georgel Calin[4], Rupak Majumdar[1], and Roland Meyer[4]

[1] Max Planck Institute for Software Systems, Germany
[2] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[3] Graduate University, Chinese Academy of Sciences
[4] Department of Computer Science, University of Kaiserslautern

Abstract. We give a language-theoretic counterexample-guided abstraction refinement (CEGAR) algorithm for the safety verification of recursive multi-threaded programs. First, we reduce safety verification to the (undecidable) language emptiness problem for the intersection of context-free languages. Initially, our CEGAR procedure overapproximates the intersection by a context-free language. If the overapproximation is empty, we declare the system safe. Otherwise, we compute a bounded language from the overapproximation and check emptiness for the intersection of the context free languages and the bounded language (which is decidable). If the intersection is non-empty, we report a bug. If empty, we refine the overapproximation by removing the bounded language and try again. The key idea of the CEGAR loop is the language-theoretic view: different strategies to get regular overapproximations and bounded approximations of the intersection give different implementations. We give concrete algorithms to approximate context-free languages using regular languages and to generate bounded languages representing a family of counterexamples. We have implemented our algorithms and provide an experimental comparison on various choices for the regular overapproximation and the bounded underapproximation.

1 Introduction

Counterexample-guided abstraction refinement (CEGAR) has become a widely applied paradigm for automated verification of systems [2, 5, 14]. While CEGAR has had a lot of successes in the analysis of single-threaded programs (most notably, device drivers), its application to *recursive multi-threaded* programs has been relatively unexplored.

We present a uniform language-theoretic view of abstraction refinement for the safety verification of recursive multi-threaded programs. First, with known encodings [3, 8], we reduce the safety verification problem to checking if the intersection of a set of context-free languages (CFLs) is empty. This is a well-known undecidable problem (and equivalent to safety verification of recursive

⋆ This work is supported by the National Natural Science Foundation of China (Grant No.60833001) and by a fellowship at the Max-Planck Institute for Software Systems.

multi-threaded programs, which is also undecidable [26]). Then we give a purely language-theoretic abstraction refinement algorithm for checking emptiness of such an intersection.

To illustrate the idea of our CEGAR loop, consider CFLs L_1 and L_2 for which we would like to check whether $L_1 \cap L_2 = \emptyset$. In our algorithm, we have to specify the following steps: (a) how to abstract the intersection of two CFLs and check that the abstraction is empty? (b) in case the abstraction is not empty, how to refine it by eliminating spurios counterexamples?

Abstraction and Checking. To abstract the context-free intersection $L_1 \cap L_2$ we rely on *regular* overapproximations of the component languages. Once we have approximated one of the languages, say L_1, by a regular language R_1 such that $L_1 \subseteq R_1$, we can check emptiness of $R_1 \cap L_2$. Because of the overapproximation, $R_1 \cap L_2 = \emptyset$ entails $L_1 \cap L_2 = \emptyset$.

Counterexample Analysis and Refinement. Suppose $A = R_1 \cap L_2 \neq \emptyset$. In the analysis and refinement step, we check if the counterexample to emptiness is genuine, and try to eliminate any string in the intersection A that is not in $L_1 \cap L_2$. A naive approach takes an arbitrary word in A (a potential counterexample) and checks if it is in $L_1 \cap L_2$. We generalize this heuristic to produce candidate counterexamples B such that we can effectively check if $L_1 \cap L_2 \cap B = \emptyset$. The advantage is that we consider (and rule out) the potentially infinite set B in one step. Refinement simply removes B from A.

Our abstraction refinement algorithm runs these two steps in a loop with one of the following outcomes. Either the abstract intersection is eventually found to be empty (the system is safe), or the counterexample analysis finds a bug (the system is unsafe), or (because the problem is undecidable) the algorithm loops forever. Unlike other abstraction-refinement algorithms [2, 5, 14], it is not the case that the abstraction always overapproximates the original program. The proof of soundness shows that refinement steps never remove buggy behaviors from the abstraction.

We give concrete constructions for the abstraction and counterexample analysis steps. To find a regular overapproximation to a context-free language, we adapt the construction of a downward closure of a CFL [28] and combine it with a graph-theoretic heuristic from [7]. We show that our construction produces regular approximations that lie between the original CFL and its downward closure (and is tighter w.r.t. set inclusion than the construction in [7]).

For the counterexample analysis, we use *bounded languages* [10] to represent an infinite set of counterexamples. Bounded languages are regular sets of the form $w_1^* w_2^* \dots w_k^*$, for words $w_1, \dots, w_k$. Given a bounded language B, checking $L_1 \cap L_2 \cap B = \emptyset$ is known to be decidable and NP-complete[8, 10]. What is an appropriate bounded language? We experimented with two algorithms. First, using a construction from [9, 20], we constructed, from a CFL L, a bounded language B such that $L \cap B$ has the same Parikh image as L. Unfortunately, this construction did not scale well in the implementation. Instead, we relied on a simple heuristic based on pumping derivation trees.

We have implemented our algorithm, and we have tried our implementation for the safety verification of several recursive multi-threaded programs. Our first class of examples models variants of a bluetooth driver [18] (also studied in [22, 25, 27]). Our second class contains example programs written in Erlang [1]. Erlang programs communicate via message passing, and are naturally written in a functional, recursive style. (In our experiments, we assume a rendezvous communication rather than asynchronous communication.) Most of the correctness properties we considered could be proved using the regular approximation. When there was a bug, the bounded language based procedure could find it.

Related Work. Analysis techniques for recursive multi-threaded programs can be categorized in four main classes. First, *context-bounded* reachability techniques [19, 23], and their generalizations using reachability modulo bounded languages [8, 9], provide a systematic way to underapproximate the reachable state space, and have proved useful in finding bugs. Second, for specific structural restrictions on the communication, one can get decidability results [4, 16, 17, 22, 25]. Third, there are some techniques to abstract the behaviors of these programs. For example, [3] explores language-based approximations by abstracting queue contents with their Parikh images. Finally, although abstraction-refinement has been studied for *non-recursive* multi-threaded programs before [6, 11–13], its systematic study for recursive multi-threaded programs has been little investigated.

Our constructions for regular overapproximations for context-free languages are inspired by language-theoretic constructions for the downward closure of context-free languages [28], together with approximation techniques originating in speech processing [7, 21]. We use bounded languages to represent families of counterexamples using ideas from [8, 9].

2 From Safety Verification to Language Emptiness

We recall the reduction from the safety verification of recursive multi-threaded programs to the emptiness problem for the intersection of context free languages.

2.1 Preliminaries

We assume familiarity with language theory (see, e.g. [15]) and only briefly recall the basic notions on regular and context free languages that we shall need in our development. A *context free grammar* (CFG) is a tuple $\langle N, \Sigma, \mathcal{P}, S \rangle$ where N is a finite and non-empty set of *non-terminals*, Σ is an alphabet of *terminals*, $\mathcal{P} \subseteq N \times (N \cup \Sigma)^*$ is the set of *production rules*, and $S \in N$ is the *start non-terminal*. We typically write $X \to w$ to denote a production $(X, w) \in \mathcal{P}$ and use $X \to w_1 | \ldots | w_k$ for $\{(X, w_i) \mid 1 \leq i \leq k\} \subseteq \mathcal{P}$.

Given two strings $u, v \in (N \cup \Sigma)^*$, we write $u \Rightarrow v$ for the fact that v *is derived from* u by an application of a production rule. Technically, $u \Rightarrow v$ if there are a non-terminal $X \in N$, a production rule $X \to w$ in $\mathcal{P}$, and strings $y, z \in (N \cup \Sigma)^*$ so that $u = yXz$ and $v = ywz$. The reflexive and transitive closure of $\Rightarrow$ is written $\Rightarrow^*$.

The *language* $L(G)$ *of a grammar* G is the set of terminal words that can be derived from the start symbol: $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$. A language $L \subseteq \Sigma^*$ is *context-free* if there is a context-free grammar G such that $L = L(G)$. We denote the class of all context-free languages by CFL. A context-free grammar is *regular* if all its productions are in $N \times (\Sigma^* N \cup \{\epsilon\})$. A language is *regular* if $L = L(G)$ for some regular grammar. The class of *all regular languages* is REG.

Example 1. Consider the grammars $G_1 = \langle \{S_1\}, \{a, b\}, \{S_1 \to abS_1b \mid \epsilon\}, S_1\rangle$ and $G_2 = \langle \{S_2\}, \{a, b\}, \{S_2 \to aS_2b \mid baS_2b \mid \epsilon\}, S_2\rangle$ that define the languages $L(G_1) = \{(ab)^n b^n \mid n \in \mathbb{N}\}$ and $L(G_2) = \{(a + ba)^n b^n \mid n \in \mathbb{N}\}$. Both languages are context free, and it can be shown that they are not regular.

The following results are well-known (see, e.g., [15]).

Theorem 1. *Let L, L' be context-free languages and R a regular language.*

1. *It is undecidable whether $L \cap L' = \emptyset$.*
2. *It is decidable whether $L \cap R = \emptyset$.*

2.2 From Programs to Context Free Languages

We encode the behaviour of recursive multi-threaded programs by an intersection of context-free languages. We consider both asynchronous communication via a shared memory and synchronous communication via rendezvous-style message exchange. The shared memory is modelled by a finite set of shared variables that range over finite data domains. This covers programs with finite data abstractions. For rendevous-style communication, we use actions $m!$ to let a thread broadcast a message m taken from a finite set M. To receive the message, all other threads have to execute a corresponding receive action $m?$ in lock step.

Encoding Programs. We illustrate the encoding of shared memory concurrency by means of an example, rendevous synchronisation immediately translates into language intersection.

Algorithm 1. Shared Memory "Toy" Program

```
                    initially $x = 0$ and $y = 0$
T1:                          ||          T2:
  $x := \neg y$              ||            $y := \neg x$
  if $(*)$ then              ||            if $(*)$ then
    call T1                  ||              call T2
  $x := \neg y$              ||            $y := \neg x$

                  assert "$x = 0 \vee y = 0$"
```

Table 1. Context-free grammars G_{T1} encoding routine $\mathsf{T1}$ (left) and G_X encoding variable x (right). G_{T2} and G_Y are similar. $\mathsf{A_X}$ is described in the text below.

$\mathsf{ST1} \to \mathsf{T1}.(r,x,1)$	$\mathsf{X}_{=0} \to (r,x,0).\mathsf{X}_{=0} \mid (w,x,1).\mathsf{X}_{=1}$
$\mathsf{T1} \to \mathsf{A_X}.\mathsf{IF}$	$\mid (w,x,0).\mathsf{X}_{=0} \mid \mathsf{L_X}.\mathsf{X}_{=0} \mid \epsilon$
$\mathsf{IF} \to \mathsf{T1}.\mathsf{FI} \mid \mathsf{FI}$	$\mathsf{X}_{=1} \to (r,x,1).\mathsf{X}_{=1} \mid (w,x,0).\mathsf{X}_{=0}$
$\mathsf{FI} \to \mathsf{A_X}$	$\mid (w,x,1).\mathsf{X}_{=1} \mid \mathsf{L_X}.\mathsf{X}_{=1} \mid \epsilon$

The program given by Algorithm 1 above consists of two threads that execute the procedures $\mathsf{T1}$ and $\mathsf{T2}$, respectively. They communicate through shared Boolean variables x, y that take values in $\{0,1\}$. The first thread assigns the negation of y to x. Based on a non-deterministic choice, it then calls itself recursively and finally repeats the assignment. Thread $\mathsf{T2}$ is symmetric, substituting y for x. We assume that each statement is executed atomically and concurrency is represented by interleaving. We are interested in the safety property that the assertion $x = 0 \vee y = 0$ holds upon termination.

We model the program's behaviour by an intersection of four context-free languages as defined by the grammars in Table 1. We explain the behaviour of G_{T1}. From its start location $\mathsf{ST1}$ (say the main routine) the thread calls procedure $\mathsf{T1}$. In $\mathsf{T1}$ it first assigns $\neg y$ to x. This is encoded by the non-terminal $\mathsf{A_X}$ that we discuss below in more detail. When the assignment has been executed, the thread proceeds with the *if* statement. If the non-determinism decides to call $\mathsf{T1}$ again, the thread proceeds with symbol $\mathsf{T1}$ and pushes the *endif* location onto the stack, indicated by $\mathsf{T1.FI}$. Otherwise, a second $\mathsf{A_X}$ statement terminates this $\mathsf{T1}$ call. When the overall execution terminates, the thread runs a final $(r,x,1)$ action which checks the assertion violation. We discuss it below.

Since we model the variables x and y by separate grammars, we split the assignment $x := \neg y$ into two actions, each modifying a single variable. First, a read $(r,y,0)$ or $(r,y,1)$ determines the value of y. A following write $(w,x,1)$ or $(w,x,0)$ finishes the assignment. This yields

$$\mathsf{A_X} \to \mathsf{L_{T1}}.(r,y,0).(w,x,1).\mathsf{L_{T1}} \mid \mathsf{L_{T1}}.(r,y,1).(w,x,0).\mathsf{L_{T1}}$$
$$\mathsf{L_{T1}} \to (r,x,0).\mathsf{L_{T1}} \mid (r,x,1).\mathsf{L_{T1}} \mid (w,y,0).\mathsf{L_{T1}} \mid (w,y,1).\mathsf{L_{T1}} \mid \epsilon.$$

Here, $\mathsf{L_{T1}}$ lets the first thread loop on the actions of the second. Similarly for G_X, writes and reads of y are synchronized via $\mathsf{L_X}$.

From Safety to Emptiness. The safety verification problem takes as input a multi-threaded program, a tuple of locations (one for each thread), and an assertion, and asks if the assertion holds when the threads are simultaneously at these locations. With the above encoding, there is an execution of the program reaching the locations specified in the safety verification problem so that the assertion fails iff there is a word common to the languages of all context-free grammars [26].

In the above example, the assertion is violated if both threads finished their execution and the variables had values $x = y = 1$. Note that the latter condition matches the execution of the reads $(r, x, 1)$ and $(r, y, 1)$ in the grammars G_{T1} and G_{T2}. Indeed, program safety is confirmed by

$$L(G_{\mathsf{T1}}) \cap L(G_{\mathsf{T2}}) \cap L(G_X) \cap L(G_Y) = \emptyset.$$

To check it note that $L(G_{\mathsf{T1}}) = \{L(\mathsf{A_X})^n.L(\mathsf{A_X})^n.(r, x, 1) \,|\, n \in \mathbb{N}\}$. The simple intuition to why the assertion holds is as follows. The first assignment changes the variables' valuation from $x = y = 0$ to either $x = 1, y = 0$ or $x = 0, y = 1$. This valuation is never altered throughout the rest of the execution.

3 The Abstraction-Refinement Procedure

We now give an abstraction-refinement algorithm to check whether the intersection $L_1 \cap L_2$ of two CFLs L_1 and L_2 is empty.

Algorithm 2. LCegar: test emptiness of context free language intersection

```
Input: Context-free languages $L_1$ and $L_2$
Output: "empty" or "non-empty"
 1: $A_1 := \mathsf{mkreg}(L_1) \cap L_2$, $A_2 := \mathsf{mkreg}(L_2) \cap L_1$, $B_1 := \emptyset$; $B_2 := \emptyset$
 2: loop
 3:   Invariant: $w \in B_1 \cup B_2 \;\Rightarrow\; w \notin L_1 \cap L_2$
 4:   if $A_1 = \emptyset$ or $A_2 = \emptyset$ then
 5:     return "empty"
 6:   else
 7:     $B_1 := \mathsf{gencx}(A_1)$ and $B_2 := \mathsf{gencx}(A_2)$
 8:     if $L_1 \cap L_2 \cap B_1 \neq \emptyset$ or $L_1 \cap L_2 \cap B_2 \neq \emptyset$ then
 9:       return "non-empty"
10:     else
11:       $A_1 := A_1 \setminus (B_1 \cup B_2)$, $A_2 := A_2 \setminus (B_1 \cup B_2)$
```

3.1 CEGAR Loop

The abstraction refinement procedure we present in Algorithm 2 takes as input two CFLs L_1 and L_2 and correctly returns "empty" or "non-empty" upon termination, reflecting whether the intersection $L_1 \cap L_2$ is empty or not. Termination, however, is not guaranteed due to the undecidability of the problem.

The key idea in our approach is to *abstract and refine the intersection* $L_1 \cap L_2$. The algorithm takes two parameters that define this approximation. For the initial abstraction, $\mathsf{mkreg}\colon \text{CFL} \to \text{REG}$ computes regular overapproximations of the context-free languages of interest, i.e., we require $L \subseteq \mathsf{mkreg}(L)$ for any CFL L. Function $\mathsf{gencx}\colon \text{CFL} \to \text{REG}$ serves in the refinement step. It isolates a regular language $\mathsf{gencx}(L)$ from the current overapproximation L of $L_1 \cap L_2$. Intuitively, $\mathsf{gencx}(L)$ contains candidate words that may lie in the intersection.

To check whether the candidates are spurious, the problem $L_1 \cap L_2 \cap \mathsf{gencx}(L) = \emptyset$ is required to be decidable for all CFLs L, L_1, L_2.

We now explain the algorithm in detail. Initially (Line 1), it overapproximates $L_1 \cap L_2$ by $A_1 = \mathsf{mkreg}(L_1) \cap L_2$ and $A_2 = L_1 \cap \mathsf{mkreg}(L_2)$ using the regular overapproximations $\mathsf{mkreg}(L_i)$. It should be noted that the A_i are CFLs. If either approximation is empty, the algorithm stops and returns "empty". Otherwise, it computes (regular) approximations $B_i = \mathsf{gencx}(A_i)$ from the (non-empty) overapproximations (Line 7). If the intersection $L_1 \cap L_2$ can be proved non-empty with words in B_1 or B_2 (Line 8), the algorithm returns "non-empty". Otherwise, we refine the approximations A_i by removing the B_i and run the loop again. By the assumptions on mkreg and gencx, each step of the algorithm is effective.

The choice of mkreg determines the granularity of the initial abstraction, and thus how quickly one can prove emptiness (in case the intersection is empty). The choice of gencx influences how fast counterexamples are found (and whether they are found at all).

Theorem 2 (Soundness). *If on input L_1, L_2 Algorithm 2 outputs "empty" then $L_1 \cap L_2 = \emptyset$, and if it outputs "non-empty" then $L_1 \cap L_2 \neq \emptyset$.*

Proof. The key to the proof is the invariant on Line 3. It states that only words outside the intersection $L_1 \cap L_2$ are removed from the A_i. Therefore $A_i \supseteq L_1 \cap L_2$ always holds. For the first iteration, the invariant is trivial. For an arbitrary iteration, $B_1 \cup B_2$ is removed from (e.g.) A_1 only when $L_1 \cap L_2 \cap B_i = \emptyset$ for $i = 1, 2$. Thus any strings removed from A_1 do not belong to $L_1 \cap L_2$ and the invariant holds.

If, e.g., $A_1 = \emptyset$ we conclude that $L_1 \cap L_2 = \emptyset$ since A_1 is throughout the program an overapproximation of the intersection. Thus the "empty" output is sound. On the other hand, any strings in $L_1 \cap L_2 \cap B_i$ are also in $L_1 \cap L_2$, thus the check on Line 8 ensures the "non-empty" output is sound. $\square$

Example 2. Consider languages $L(G_1)$ and $L(G_2)$ from Example 1. Suppose we approximate them by regular sets $(ab)^*b^*$ and $(a + ba)^*b^*$, respectively. The intersections $(ab)^*b^* \cap L(G_2)$ and $L(G_1) \cap (a + ba)^*b^*$ are both non-empty. Suppose gencx returns $(ab)^*b^*$ for the first intersection. We can compute that $L(G_1) \cap L(G_2) \cap (ab)^*b^* = \emptyset$. In the next iteration, the new approximation $A_1 = \{(ab)^n b^n | n \in \mathbb{N}\} \setminus (ab)^*b^* = \emptyset$, and we conclude that $L(G_1) \cap L(G_2) = \emptyset$.

Since checking emptiness of the intersection is undecidable, the algorithm can run forever. But what happens if $L_1 \cap L_2 \neq \emptyset$ holds? In this case the algorithm is guaranteed to terminate and return "non-empty" if gencx satisfies the following additional enumeration property.

Definition 1 (Enumerator). *A sequence $(L_i)_{i \in \mathbb{N}}$ of languages* **enumerates** *a language L if for every $w \in L$ there is an index $i \in \mathbb{N}$ so that $w \in L_i$.*

Let $(L_{1,i})_{i \in \mathbb{N}}$, $(L_{2,i})_{i \in \mathbb{N}}$ be the sequences of languages generated by $\mathsf{gencx}(A_1)$ and $\mathsf{gencx}(A_2)$ when running Algorithm 1 on the input languages L_1 and L_2.

Proposition 1 (Semidecider). *If $(L_{1,i} \cup L_{2,i})_{i\in\mathbb{N}}$ enumerates $L_1 \cap L_2 \neq \emptyset$ then Algorithm 2 terminates with output "non-empty" on input L_1, L_2.*

3.2 The Regular Approximator mkreg

We now describe our implementation of mkreg, the regular overapproximation of a context-free language.

Step I: Intuition: Downward Closures. Given strings $u, v \in \Sigma^*$, we define $u \preceq v$ if u is a (not necessarily contiguous) substring of v. For example, $abd \preceq aabccd$. For a language L, we define the downward closure of L, denoted by $L{\downarrow}$, as $L{\downarrow} = \{u \in \Sigma^* \mid \exists v \in L.u \preceq v\}$. It is known that $L{\downarrow}$ is effectively regular for a CFL L [28], and clearly $L \subseteq L{\downarrow}$. Thus, the downward closure $-{\downarrow}$ is an immediate candidate for mkreg.

Step II: A Modification to the Downward Closure. Unfortunately, the downward closure is usually too coarse and leads to many spurious counterexamples. The problem occurs already when the original language is regular; for example, the downward closure of $(ab)^*$ is $(a+b)^*$. Further, since our downward closure construction is doubly exponential in the worst case, the obtained approximation can get too big to apply further operations in non-trivial examples. We encountered both these shortcomings in our experiments.

As a first modification, we "tightened" the downward closure algorithm to provide a regular approximation that lies (w.r.t. set inclusion) between the CFL and its downward closure. We observe that for $t \in \Sigma$, the downward closure $L(t){\downarrow} = \{t, \epsilon\}$ can drop the t. In the inductive construction of the downward closure, this introduces too many new (sub)words. We show how to avoid dropping some letters in the downward closure computation.

Our modification, called pseudo-downward closure (PDC), constructs a finer regular overapproximation $L{\Downarrow}$ of a CFL L. The idea is to preserve contiguous subwords. We proceed by iterating over all the grammar non-terminals. More precisely, we set $L(t){\Downarrow} = \{t\}$ for a letter $t \in \Sigma$. To ease the definition, let $-{\Downarrow}$ distribute over concatenation ($L(u.v){\Downarrow} = L(u){\Downarrow}.L(v){\Downarrow}$) and set

$$L(X){\Downarrow} = \begin{cases} \Sigma(X)^* & \text{if } X \Rightarrow^* uXvXw \\ (\bigcup L(u){\Downarrow})^*.(\bigcup_{X\Rightarrow M, X \not\lhd M} L(M){\Downarrow}).(\bigcup L(v){\Downarrow})^* & \text{if } X \Rightarrow^* uXv \\ \bigcup_{X\Rightarrow M, X \not\lhd M} L(M){\Downarrow} & \text{otherwise.} \end{cases}$$

$\Sigma(X) \subseteq \Sigma$ denotes the set of all letters that appear in some word derived from X. The unions $\bigcup L(u){\Downarrow}$ and $\bigcup L(v){\Downarrow}$ range over the *shortest words* so that X reproduces itself via a derivation $X \Rightarrow^* uXv$. $X \not\lhd M$ means there is no derivation $M \Rightarrow^* uXv$. Finally, we require the first case to have precedence over the second.

Proposition 2 below states that $-{\Downarrow}$ is also a candidate for mkreg, and a better approximation than $-{\downarrow}$. Its proof is a induction on the structure of the grammar.

Proposition 2. *Given a CFG $G = (N, \Sigma, \mathcal{P}, S)$, $L(G) \subseteq L(G){\Downarrow} \subseteq L(G){\downarrow}$.*

Step III: Refinement by Cycle Breaking. We can further tighten our PDC construction as follows. Consider the grammar G defined by

$$A \to aAbAc \mid t \tag{1}$$

where lower-case letters denote terminals. Each word in the language $L(G)$ is either t, or starts with an a, has a b in the middle, and ends with a c. However, $L(G)\Downarrow = (a+b+c+t)^*$, and the PDC construction loses the order of the letters in the original language.

We augment the PDC construction with the following insight [7]. Given a CFG G in Chomsky normal form, we can construct a regular over-approximation of $L(G)$ by replacing each rule $A \to BC$ with a rule $A \to R_B C$, where R_B generates a regular over-approximation of $L(B)$. After the replacement, the grammar will be right regular and the new language will over-approximate $L(G)$. If there is no production $B \Rightarrow^* uAv$, we can inductively compute an over-approximation R_B of B. In the case when $B \Rightarrow^* uAv$ we use the PDC construction to compute an overapproximation of B.

The intuition behind the construction is similar to the *cycle-breaking* heuristic to approximate CFLs by regular languages used in speech processing [7]. Our construction below computes a regular approximation that is guaranteed to be as tight as the construction in [7].

To clarify the intuition, we construct a directed graph from the CFG as follows. The nodes of the graph are the non-terminals of the grammar. For each rule $A \to BC$ in the grammar, we have an edge in the graph from A to B labeled l (for left) and an edge from A to C labeled r (for right). A simple cycle in this graph is called mono-chromatic if it only contains edges marked l or r, and duo-chromatic otherwise. Our construction "breaks" every duo-chromatic cycle in the graph as follows. It picks an l-edge (A, B) in the cycle, and replaces the rule $A \to BC$ from which the (A, B) edge was constructed with $A \to R_B C$. The language of the new nonterminal R_B is $L(B)\Downarrow$. We denote this approximation by $L(G)\downdownarrows$. To give an example, if we turn the grammar in Equation (1) into Chomsky normal form, we obtain $L(A)\downdownarrows = t + a(a+b+c+t)^*b(a+b+c+t)^*c$.

Proposition 3. *Given a CFG G, we have $L(G) \subseteq L(G)\downdownarrows \subseteq L(G)\Downarrow$.*

The proof is again a direct application of structural induction.

3.3 The Counterexample Generator gencx

We now give an algorithm to generate families of counterexamples in case A_1 and A_2 are non-empty. A naive idea is to lexicographically enumerate words in A_1 and A_2 and to check if one of them is in $L_1 \cap L_2$. Since the length of a path leading to the unsafe location is finite, this approach guarantees termination (the sequence is clearly an enumerator for $L_1 \cap L_2$). However, enumeration-based approaches do not scale, even if the language is finite. For example, depending on the choice of the w_i's, a language of the following type has upto 2^k words:

$$(w_1 + \epsilon) \cdot (w_2 + \epsilon) \cdots (w_k + \epsilon) \tag{2}$$

Instead, we use *elementary bounded languages* (EBLs) [10] to represent families of counterexamples. EBLs are regular languages of the form $w_1^* w_2^* \cdots w_k^*$ for some fixed words $w_1, \cdots, w_k \in \Sigma^*$. For CFLs L_1 and L_2 and an EBL B, checking if $L_1 \cap L_2 \cap B = \emptyset$ is NP-complete [8]. In case of Language (2), the bounded language $w_1^* \ldots w_k^*$ contains all the words in (2) (and more). In general, an EBL captures an infinite number of potential counterexamples.

Which EBL should one choose? We first implemented an algorithm from [9] which computes, given a CFL L, an EBL B such that for every word $w \in L$, there is a permutation of w in $L \cap B$. (That is, the commutative images of $L \cap B$ and L coincide. Recall that the commutative image of a word w is an integer vector that represents the number of times each alphabet letter occurs in w.) Intuitively, the EBL B captures "many" behaviors of L. Unfortunately, our experiments indicated that this construction does not scale well. Therefore, our implementation makes use of a simple heuristic. The idea is to pump derivation trees as follows.

Starting with a CFG $G = \langle N, \Sigma, \mathcal{P}, S \rangle$ for L, we first construct an initialized *partial derivation tree* up to a fixed depth. A partial derivation tree is a tree whose nodes are labeled with symbols from $N \cup \Sigma$ with the following property. Each leaf is labeled with a symbol from $N \cup \Sigma$. Each internal node is labeled with a non-terminal from N, and if an internal node is labeled with $T \in N$ and has k children labeled with $A_1, \ldots, A_k \in N \cup \Sigma$, then $T \to A_1 \ldots A_k$ is a production in $\mathcal{P}$. The partial derivation tree is *initialized* if its root is labeled with S. A partial derivation tree corresponds to a (partial) derivation of the grammar G, and conversely, each (partial) derivation of the grammar defines a partial derivation tree. The *yield* of a partial derivation tree is the word of symbols at the leaf nodes. More formally, for a leaf l labeled by $\sigma \in N \cup \Sigma$ we set $\mathit{yield}(l) = \sigma$. For an internal node n with k children $n_1, \ldots, n_k$, we define $\mathit{yield}(n) = \mathit{yield}(n_1) \cdot \mathit{yield}(n_2) \cdot \ldots \cdot \mathit{yield}(n_k)$.

A partial derivation tree t is *pumpable* if its root is labeled by a non-terminal A and some (not necessarily immediate) child of t again carries label A. For the sake of clarity, we denote the (not necessarily unique) descendant node labeled by A by t_A. Words x, z are then called *pump-words* for the pumpable tree t with descendant t_A if the derivation corresponding to t can be written as $A \Rightarrow^* xAz \Rightarrow^* xyz$ for some $y \in (N \cup \Sigma)^*$. Here, the non-terminal A in the derivation labels node t_A.

The EBL is computed in two steps. We first construct an initialized partial derivation tree up to some fixed depth, and then traverse this tree with a depth-first search. The corresponding procedure traverse in Algorithm 3 is called with the root of the initialized partial derivation tree, an empty list of words, and an empty initial word ε. It returns a (possibly empty) word w and a list of words $[w_1, \ldots, w_k]$. From these words, we construct the required elementary bounded language $h(w_1)^* \ldots h(w_k)^* h(w)^*$ using the following homomorphism h. Since the words are defined over $N \cup \Sigma$ but we look for a language over Σ, homomorphism h replaces non-terminals $A \in N$ by words $w_A \in \Sigma^*$ that can be derived from A. Terminals are left unchanged, $h(\sigma) = \sigma$ for $\sigma \in \Sigma$.

Algorithm 3. traverse

```
Input: partial derivation tree $t$, word list $l$, current word $w$
Output: pair of word $w \in (N \cup \Sigma)^*$ and list of words $l'$
 1: match $t$ with
 2: | leaf labeled with $\sigma \in N \cup \Sigma$ : return $\langle w \cdot \sigma, l \rangle$
 3: | internal node labeled with $A$ :
 4:     if $t$ is pumpable with descendant $t_A$ and pump words $x, z$
 5:     then let $\langle w_t, l_t \rangle = \mathsf{traverse}(t_A, [\,], \varepsilon)$
 6:       return $\langle \varepsilon, l@[w, x]@l_t@[w_t, z] \rangle$
 7:     otherwise for the children $t_1, \ldots, t_k$ of $t$,
 8:       let $\langle w_1, l_1 \rangle = \mathsf{traverse}(t_1, l, w)$
 9:       ...
10:       let $\langle w_k, l_k \rangle = \mathsf{traverse}(t_k, l_{k-1}, w_{k-1})$
11:       return $\langle w_k, l_k \rangle$
```

The procedure recursively traverses the partial derivation tree, collecting the yield of the tree in the word w. However, when it sees a "cycle" in the derivation tree (i.e., a subtree that can be pumped), it pumps its pump-words. Pumping moves the partial yield w constructed so far to the list of words (Line 6). Therefore, a word w that is returned actually gives the partial yield of the nodes visited after the last pumping situation. This explains why $h(w)^*$ is added to the end of the EBL and why w_t is placed behind l_t in Line 6. As a special case, if the partial derivation tree consists of a single node containing only the start symbol, the algorithm returns w^* for some word w in the language.

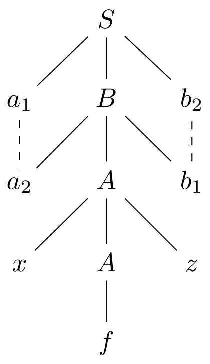

Fig. 1. Traverse Sample

As clarification, applied to the partial derivation tree depicted in Figure 1, the algorithm returns $\langle b_1 \cdot b_2, [a_1 \cdot a_2, x, f, z] \rangle$ with x and z being pump words for the given derivation tree and $a_1 a_2$ as well as $b_1 b_2$ being partial yields w.

4 Experiments

We have implemented the language-theoretic CEGAR algorithm and tested our implementation on the shared memory example in Section 2, a set of Erlang examples from [1], and the bluetooth driver examples from [22, 25, 27]. We manually convert programs to the input format of our tool (a description of context free grammars). We compare the results of the PDC and cycle-breaking algorithms for mkreg and we use the simplified EBL generation algorithm for gencx. To check emptiness of $L_1 \cap L_2 \cap B$ (for CFLs L_1 and L_2 and EBL B), we use the algorithm of [8] and reduce the check to a satisfiability problem in Presburger arithmetic. We use Yices to check satisfiability of this formula.

All experiments were carried on an Intel Core2 Q9400 PC with 8GB memory, running 64-bit Linux (Ubuntu 11.10 x86_64). Tables 2 and 3 describe our results.

4.1 Recursive Multi-threaded Programs

We applied our algorithm to a set of recursive multi-threaded programs: the shared memory program in Section 2 and some selected Erlang programs. We chose these examples since they provide a good test suite for checking both our method's bug finding and proving capabilities. For each test we used both the *pseudo downward closure* (PDC) and *cycle breaking* (CB) overapproximations.

Toy Example (from Section 2). For the shared memory program in Section 2 we produced 4 CFLs, for T1, T2, x and y, with a total of 118 production rules. It took our implementation about 16 seconds to report the system's safety by adopting the PDC approach and about 26 seconds by the CB approach. Note that this example does not fall into the subclass considered in [4].

Peterson Mutual Exclusion Protocol (from [29], made recursive). In this example, two processes try to acquire a lock. The one which receives it can enter the critical section to perform operations on shared variables, then frees the lock. The code is written in functional style with tail-recursive calls in each component. The checked property is that at any time, at most one process is in the critical section. The model comprised 4 CFLs with 242 production rules. LCegar reported the system's safety in 6.8 seconds by adopting the PDC closure, and in 0.2 seconds by using the CB closure.

Resource Allocator (from [1], pp. 81, 111). A resource allocator (RA) manages a number of resources and handles “alloc” and “free” messages sent from clients. When it receives an “alloc” message, the RA checks if there is a free resources in the system. If yes, it replies to the client with the resource id, and also marks that the resource has been allocated; otherwise it replies that no free resources are available. When it receives a “free” message with a resource id from a client, the RA checks if the resource is actually held by the client. If yes, the resource is freed, otherwise an error is reported to the client. The safety property checked was that the server would not allocate more resources to clients than the current free resources in the system. We produced 2 CFLs for the server and resource with a total of 100 production rules from the original program. Though simple, LCegar did not terminate by adopting the PDC over-approximation. It took LCegar 0.5 seconds to report the system's safety by using the CB approach.

In the modified version of the resource allocator (MoRA), the situation that a client can exit normally or abnormally is handled. When allocating a resource id to a client, MoRA makes a link between server and client, and traps the exit signal of the client. Once the client leaves without freeing the resource, the MoRA will detect the event, free the resource on server side, and unlink the client. The property to verify was identical to the one for the RA. This more complicated example needed 5 CFLs with 239 production rules. It took LCegar 31 seconds to report safety by using the CB closure and PDC did not terminate.

Table 2. Experimental results for recursive multi-threaded (Erlang) programs

	SharedMem		Mutex		RA		Modified RA		TNA	
	PDC	CB	PDC	CB	PDC	CB	PDC	CB	PDC	CB
CFL	4		4		2		5		3	
Terminal	8		16		20		22		17	
Non-Ter	22		27		22		32		23	
Rule	118		242		100		239		93	
Time	15.7s	26.0s	6.8s	0.2s	N/A	0.5s	N/A	31.2s	0.8s	0.3s

Table 3. Experimental results for Bluetooth drivers

		Version 1		Version 2		Version 3 (2A1S)		Version 3 (1A2S)	
		PDC	CB	PDC	CB	PDC	CB	PDC	CB
w/o Heuri	CFL	7		9		9		8	
	Terminal	17		26		25		22	
	Non-Ter	29		47		47		39	
	Rule	362		839		846		585	
	Time	19s	18s	109m57s	96m48s	81m7s	77m21s	3m50s	3m55s
w/ Heuri	CFL	7		9		x		8	
	Terminal	17		26		x		22	
	Non-Ter	29		47		x		39	
	Rule	285		591		x		408	
	Time	unknown		56s	50s	x	x	7s	7s

Telephone Number Analyzer (from [1], p. 109). The telephone number analyzer (TNA) running on the server side handles "lookup" or "add_number" requests from the telephone ends. When it receives a request, it performs the corresponding action in a try-catch block. The example tries to show that the try-catch block can guard against the inadvertent programming errors. However, it also mentions that a malicious program can crash the server by sending an incorrect process id. The reason is that TNA does not check if the id in the message content is the same as its sender's. We modeled the program with 3 CFLs and a total of 93 production rules to spot the same bug in 0.8s (PDC) and 0.3s (CB).

4.2 Bluetooth Drivers

We considered the bluetooth driver [18] and its variations studied before through various methods [4, 24, 27]. Once LCegar terminates, it reports safety or a path leading to the buggy location of the system. Note that in contrast to bounded context-switch approaches, LCegar looks for error traces without an a priori context-switching bound, and thus can prove correctness of the protocol as well. However, our language-theoretic computations take much more time than bounded context-switching.

Table 3 shows the experimental results for the bluetooth drivers. For each program, we use the two kinds of overapproximations discussed.

The 2nd version and the erroneous 3rd version (1A2S) prove to be the most difficult to handle for our full fledged method. For this reason we considered an unsound heuristic restricting context switches at basic block boundaries. This heuristic sped up the tool and found the bugs. However, in the first version, our method reported "unknown" since the heuristic method is an underapproximation, and therefore, cannot report the system's correctness when no bug is found. If the method does not find a bug, we therefore run the original (sound) version of the algorithm. For example, we used the non-optimized algorithm for the 3rd version of the driver with 2 adders and 1 stopper (2A1S) which is safe. In this case, it took LCegar 77 minutes and 21 seconds (CB) to verify correctness.

4.3 Conclusion

While the run times of our implementation are somewhat disappointing, we believe our implementation demonstrates the potential of language-based techniques in the verification of recursive multi-threaded programs.

Also, although the run times could probably be improved by providing better symbolic implementations of the language-theoretic operations used, considering more succinct encodings of programs is a must.

References

1. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice Hall (1996)
2. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL 2002: Principles of Programming Languages, pp. 1–3. ACM (2002)
3. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. International Journal on Foundations of Computer Science 14(4), 551–582 (2003)
4. Chaki, S., Clarke, E., Kidd, N., Reps, T., Touili, T.: Verifying Concurrent Message-Passing C Programs with Recursive Calls. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
6. Clarke, E.M., Talupur, M., Veith, H.: Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008)
7. Eğecioğlu, Ö.: Strongly Regular Grammars and Regular Approximation of Context-Free Languages. In: Diekert, V., Nowotka, D. (eds.) DLT 2009. LNCS, vol. 5583, pp. 207–220. Springer, Heidelberg (2009)
8. Esparza, J., Ganty, P.: Complexity of pattern-based verification for multithreaded programs. In: POPL 2011: Principles of Programming Languages, pp. 499–510. ACM (2011)
9. Ganty, P., Majumdar, R., Monmege, B.: Bounded Underapproximations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 600–614. Springer, Heidelberg (2010)

10. Ginsburg, S., Spanier, E.H.: Bounded Algol-like languages. Transactions of the American Mathematical Society 113(2), 333–368 (1964)
11. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL 2011: Principles of Programming Languages, pp. 331–344. ACM (2011)
12. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: PLDI 2004: Programming Language Design and Implementation, pp. 1–13. ACM (2004)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-Modular Abstraction Refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
14. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002: Principles of Programming Languages, pp. 58–70. ACM (2002)
15. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
16. Kahlon, V.: Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In: LICS 2009: Logic in Computer Science, pp. 27–36. IEEE Computer Society (2009)
17. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: POPL 2003: Principles of Programming Languages, pp. 303–314. ACM (2007)
18. Kidd, N.: Bluetooth protocol, `http://pages.cs.wisc.edu/~kidd/bluetooth/`
19. Lal, A., Reps, T.: Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
20. Latteux, M., Leguy, J.: Une propriété de la famille GRE. In: FCT 1979, pp. 255–261. Akademie-Verlag (1979)
21. Mohri, M., Nederhof, M.-J.: Regular approximation of context-free grammars through transformation. In: Robustness in Language and Speech Technology, vol. 9, pp. 251–261. Kluwer Academic Publishers (2000)
22. Patin, G., Sighireanu, M., Touili, T.: Spade: Verification of Multithreaded Dynamic and Recursive Programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 254–257. Springer, Heidelberg (2007)
23. Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
24. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI 2004: Programming Language Design and Implementation, pp. 14–24. ACM (2004)
25. Ben Rajeb, N., Nasraoui, B., Robbana, R., Touili, T.: Verifying multithreaded recursive programs with integer variables. Electr. Notes Theor. Comput. Sci. 239, 143–154 (2009)
26. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM TOPLAS 22(2), 416–430 (2000)
27. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic Context-Bounded Analysis of Multithreaded Java Programs. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)
28. van Leeuwen, J.: Effective constructions in well-partially-ordered free monoids. Discrete Mathematics 21(3), 237–252 (1978)
29. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 3(12), 115–116 (1981)