A Complete Refinement Procedure for Regular Separability of Context-Free Languages

Graeme Gange[a], Jorge A. Navas[b], Peter Schachte[a], Harald Søndergaard[a], Peter J. Stuckey[a]

[a] *Department of Computing and Information Systems, The University of Melbourne, Vic. 3010, Australia*
[b] *NASA Ames Research Center, Moffett Field, CA 94035, USA*

Abstract

Often, when analyzing the behaviour of systems modelled as context-free languages, we wish to know if two languages overlap. To this end, we present an effective semi-decision procedure for regular separability of context-free languages, based on counter-example guided abstraction refinement. We propose two refinement methods, one inexpensive but incomplete, and the other complete but more expensive. We provide an experimental evaluation of this procedure, and demonstrate its practicality on a range of verification and language-theoretic instances.

Keywords: abstraction refinement, context-free languages, regular approximation, separability

1. Introduction

We address the problem of checking whether two given context-free languages L_1 and L_2 are disjoint. This is a fundamental language-theoretical problem. It is of interest in many practical tasks that call for some kind of automated reasoning about programs. This can be because program behaviour is modelled using context-free languages, as in software verification approaches that try to capture a program's control flow as a (pushdown-system) path language. Or it can be because we wish to reason about string-manipulating programs, as is the case in software vulnerability detection problems, where various types of injection attack have to be modelled.

The problem of context-free disjointness is of course undecidable, but semi-decision procedures exist for non-disjointness. For example, one can systematically generate strings w over the intersection $\Sigma_1 \cap \Sigma_2$, where Σ_1 is the alphabet of L_1 and Σ_2 is that of L_2. If some w belongs to both L_1 and L_2, answer "yes, the languages overlap." It follows that no semi-decision procedure exists for disjointness. However, semi-decision procedures exist for the stronger requirement of being separable by a regular language. For example, one can systematically generate (representations of) regular languages over $\Sigma_1 \cup \Sigma_2$, and, if some such language R is found to satisfy $L_1 \subseteq R \wedge L_2 \subseteq \overline{R}$, answer "yes, the languages are disjoint".

A radically different approach, which we will follow here, uses so-called *counter-example guided abstraction refinement (CEGAR)* [6] of regular over-approximations. The scheme is based on repeated approximation refinement, like so:

1. *Abstraction:* Compute *regular approximations* R_1 and R_2 such that $L_1 \subseteq R_1$ and $L_2 \subseteq R_2$. (Here R_1 and R_2 are regular languages, represented using regular expressions, say.)
2. *Verification:* Check whether the intersection of R_1 and R_2 is empty using a decision procedure for regular expressions. If $R_1 \cap R_2 = \emptyset$ then $L_1 \cap L_2 = \emptyset$, so answer "the languages are disjoint." If $w \in (R_1 \cap R_2)$, $w \in L_1$, and $w \in L_2$ then $L_1 \cap L_2 \neq \emptyset$, so answer "the languages overlap" and provide w as a witness. Otherwise, go to step 3.

Email addresses: gkgange@unimelb.edu.au (Graeme Gange), jorge.a.navaslaserna@nasa.gov (Jorge A. Navas), schachte@unimelb.edu.au (Peter Schachte), harald@unimelb.edu.au (Harald Søndergaard), pstuckey@unimelb.edu.au (Peter J. Stuckey)

3. *Refinement:* Produce new regular approximations R_1' and R_2' such that $L_1 \subseteq R_1' \subseteq R_1$, $L_2 \subseteq R_2' \subseteq R_2$, and $R_i' \subset R_i$ for some $i \in \{1, 2\}$. Update the approximations $R_1 \leftarrow R_1', R_2 \leftarrow R_2'$, and go to step 2.

For the abstraction step, note that regular approximations exist, trivially. For the verification step, we could also take advantage of the fact that the class of context-free languages is closed under intersection with regular languages; however, this does not eliminate the need for a refinement procedure. For the refinement step, note that there is no indication of *how* the tightening of approximations should be done; indeed that is the focus of this paper. The step is clearly well-defined since, if $L \subset R$, where R is regular, there is always a regular language $R' \subset R$ such that $L \subseteq R'$.

For a given language L there may well be an infinite chain $R_1 \supset R_2 \supset \cdots \supset L$ of regular approximations. This is a source of possible non-termination of the CEGAR scheme. An interesting question therefore is: Are there refinement techniques that can guarantee termination at least when L_1 and L_2 are *regularly separable* context-free languages, that is, when there exists a regular language R such that $L_1 \subseteq R$ and $L_2 \subseteq \overline{R}$?

In this paper we answer this question in the affirmative. We propose a refinement procedure which can ensure termination of the CEGAR-based loop assuming the context-free languages involved are regularly separable. In this sense we provide a refinement procedure which is *complete* for regularly separable context-free languages. Of course the question of regular separability of context-free languages is itself undecidable [22]. The method we propose will also successfully terminate whenever the given languages overlap.

Contribution. The paper rests on regular approximation ideas by Nederhof [20] and we utilise the efficient pre^* algorithm [8] for intersecting (the language of) a context-free grammar with (that of) a finite-state automaton. We propose a novel refinement procedure for a CEGAR inspired method to determine whether context-free languages are disjoint, and we prove the procedure complete for determining regular separability. In the context of regular approximation, where languages must be over-approximated using *regular* languages, separability is equivalent to regular separability, so the completeness means that the refinement procedure is optimal. On the practical side, the method has important applications in software verification and security analysis. We demonstrate its feasibility through an experimental evaluation.

Outline. Section 2 introduces concepts, notation and terminology used in the paper. It also recapitulates relevant results about regular separability and language representations. Section 3 proposes a new procedure for regular approximation of context-free languages and shows that the procedure is complete, in the sense that it proves the separability for any pair of regularly separable context-free languages. Section 4 provides an example. In Section 5 we place our method in context, comparing with previously proposed refinement techniques. In Section 6 we evaluate the method empirically, comparing an implementation with the most closely related tool. Section 7 discusses more broadly related work, and Section 8 concludes. An appendix contains a description of the test cases used in the experimental evaluation.

2. Preliminaries

In this section we recall the notion of regular separability and introduce a concept of "star-contraction" for regular expressions.

2.1. Regular and Context-Free Languages

We first recall some basic formal-language concepts. These are assumed to be well understood—the only purpose here is to fix our terminology and notation. Given an alphabet Σ, Σ^* denotes the set of all finite strings of symbols from Σ. The string y is a *substring* of the finite string w iff $w = xyz$ for some (possibly empty) strings x and z.

The *regular expressions* over an alphabet $\Sigma = \{a_1, \ldots, a_n\}$ are $\emptyset$, ε, $a_1, \ldots a_n$, together with expressions of form $e_1|e_2$, $e_1 \cdot e_2$, and e^*, where e, e_1 and e_2 are regular expressions. Here $|$ denotes union, $\cdot$ denotes language concatenation, and * is Kleene star. As is common, we will often omit $\cdot$, so that juxtaposition of e_1 and e_2 denotes concatenation of the corresponding languages. Given a finite set $E = \{e_1, \ldots, e_k\}$ of

regular expressions, we let $||E$ stand for the regular expression $e_1|\cdots|e_k$ (in particular, $||\emptyset = \emptyset$). We let Reg_Σ denote the set of regular expressions over alphabet Σ.

A (non-deterministic) *finite-state automaton* is a quintuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ where Q is the set of states, Σ is the alphabet, δ is the transition relation, q_0 is the start state, and F is the set of accept states. The presence of (q, x, q') in $\delta \subseteq Q \times \Sigma \times Q$ indicates that, on reading symbol x while in state q, the automaton may proceed to state q'. If δ is a total function, that is, if $|\{q' \mid (q, x, q') \in \delta\}| = 1$ for all $q \in Q$ and $x \in \Sigma$ then the automaton is *deterministic*.

A language which can be expressed as a regular expression (or equivalently, has a finite-state automaton that recognises it) is *regular*. The language recognised by automaton A is written as $\mathcal{L}(A)$. Similarly, $\mathcal{L}(e)$ is the language denoted by regular expression e. We shall sometimes need to reason about the underlying languages denoted by *sets* of regular expressions/automata. Let E_1 and E_2 be sets of regular expressions. We use $E_1 \cap_{\mathcal{L}} E_2$ to denote the set of *languages* L for which both E_1 and E_2 contain some expression denoting L. That is,

$$E_1 \cap_{\mathcal{L}} E_2 = \{\mathcal{L}(e_1) \mid e_1 \in E_1, \exists e_2 \in E_2, \mathcal{L}(e_1) = \mathcal{L}(e_2)\}.$$

A *context-free grammar*, or CFG, is a quadruple $G = \langle V, \Sigma, P, S\rangle$, where V is the set of variables (non-terminals), S is the start symbol, and P is the set of productions (or rules). Each production is of form $X \to \alpha$ with $X \in V$ and $\alpha \in (V \cup \Sigma)^*$. If $X \to \alpha$ is a production in P then, for all $\beta, \gamma \in (V \cup \Sigma)^*$, we say that $\beta X \gamma$ *yields* $\beta\alpha\gamma$, written $\beta X\gamma \Rightarrow \beta\alpha\gamma$. The language *generated* by G is $\mathcal{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$, where $\Rightarrow^*$ is the reflexive transitive closure of $\Rightarrow$. A set of strings is a context-free language (CFL) iff it is generated by some CFG.

In algorithms, we usually represent regular languages using finite-state automata, and CFLs using CFGs. When there is little risk of confusion, we ignore the distinction between a language and its representation. Hence we may, for example, apply set operations to (the transition relation of) a finite-state automaton.

2.2. Regular Separability

As our approach uses regular approximations to context-free languages, we cannot hope to prove separation for arbitrary disjoint pairs of context-free languages. Instead we focus on pairs of of *regularly separable* languages.

Definition 1 (Regularly separable)**.** Two context-free languages L_1 and L_2 are *regularly separable* iff there exists a regular language R such that $L_1 \subseteq R$ and $L_2 \subseteq \overline{R}$ where $\overline{R}$ is the complement of R.

It will be useful to have a slightly different view of separability:

Definition 2 (Separating pair)**.** Given a pair (L_1, L_2) of context-free languages, a pair (R_1, R_2) of regular languages form a *separating pair* for (L_1, L_2) iff $L_1 \subseteq R_1$, $L_2 \subseteq R_2$, and $R_1 \cap R_2 = \emptyset$.

Lemma 1. *Context-free languages L_1 and L_2 are regularly separable iff there exists some separating pair (R_1, R_2) for (L_1, L_2).*

Proof. If (R_1, R_2) is a separating pair then $L_1 \subseteq R_1$, and $L_2 \subseteq R_2 \subseteq \overline{R_1}$, so L_1 and L_2 are regularly separable. Conversely, if the regular language R separates L_1 and L_2 then we have $L_1 \subseteq R$, $L_2 \subseteq \overline{R}$, and $R \cap \overline{R} = \emptyset$. So $(R, \overline{R})$ is a separating pair. □

To see that there are disjoint context-free languages that are not regularly separable, consider $L = \{\mathsf{a}^n\mathsf{b}^n \mid n \geq 0\}$. Both L and $\overline{L}$ are non-regular context-free languages, and therefore not regularly separable. The problem of checking whether a pair of context-free languages is regularly separable is undecidable [13].

2.3. Star-Contraction

Definition 3 (Union-free regular language)**.** A regular expression is *union-free* iff it does not use the union operation. A regular language is *union-free regular* if it can be written as a union-free regular expression. We use Reg'_Σ to denote the set of union-free regular expressions.

Union-free regular languages are also known as *star-dot regular* languages [4].

Definition 4 (Union-free decomposition)**.** A union-free decomposition [19] of a regular language R is a finite set of union-free regular languages $R_1, \ldots, R_n$ such that $R = R_1 \cup \ldots \cup R_n$.

Theorem 1 (Nagy [19])**.** *Every regular language R admits some finite union-free decomposition.*

Theorem 1 is not surprising; it utilises the well-known equivalence $(r_1|r_2)^* = (r_1^* r_2^*)^*$.

The following concept is central to this paper's ideas. For a given union-free language, it is convenient to consider particular sets of sub-languages:

Definition 5 (Star-contraction)**.** The star-contraction $\kappa(e)$ of a union-free regular expression e is the set of languages obtained by replacing some subset of $*$-enclosed subterms in e with ε. The star-contraction is defined as:

$$\begin{array}{rcll} \kappa(\emptyset) & = & \emptyset & \\ \kappa(\varepsilon) & = & \{\varepsilon\} & \\ \kappa(x) & = & \{x\} & \text{for } x \in \Sigma \\ \kappa(e^*) & = & \{(\|\, E)^* \mid E \subseteq \kappa(e)\} & \\ \kappa(e_1 \cdot e_2) & = & \{r_1 \cdot r_2 \mid r_1 \in \kappa(e_1) \wedge r_2 \in \kappa(e_2)\} & \end{array}$$

Example 1. Consider the union-free regular expression $\mathtt{ab}^*\mathtt{c}^*$. First note that $\kappa(\mathtt{a}) = \{\mathtt{a}\}$, $\kappa(\mathtt{b}^*) = \{\varepsilon, \mathtt{b}^*\}$, and $\kappa(\mathtt{c}^*) = \{\varepsilon, \mathtt{c}^*\}$. Hence the star-contraction $\kappa(\mathtt{ab}^*\mathtt{c}^*) = \{\mathtt{a}, \mathtt{ab}^*, \mathtt{ac}^*, \mathtt{ab}^*\mathtt{c}^*\}$.

In examples with nested $*$ operators, the star-contraction allows distinct portions of an inner $*$-expression's subterms to occur when the outer $*$ operation is processed:

Example 2. Continuing from the previous example, consider the union-free regular expression $e = (\mathtt{ab}^*\mathtt{c}^*)^*$. We saw that the contraction of the parenthesised expression is $\{\mathtt{a}, \mathtt{ab}^*, \mathtt{ac}^*, \mathtt{ab}^*\mathtt{c}^*\}$. The star-contraction of e (after elimination of equivalent languages) is then:

$$\kappa(e) = \{\varepsilon, \mathtt{a}^*, (\mathtt{ab}^*)^*, (\mathtt{ac}^*)^*, (\mathtt{ab}^*|\mathtt{ac}^*)^*, (\mathtt{ab}^*\mathtt{c}^*)^*\}.$$

Note how the elements $r \in \kappa(e)$ with $r \neq e$ make particular subsets of $\mathcal{L}(e)$ explicit. For example, $\mathtt{a}^*$ is the subset that makes no use of $\mathtt{b}$ or $\mathtt{c}$, whereas $(\mathtt{ab}^*|\mathtt{ac}^*)^*$ is the set of words in which $\mathtt{b}$ and $\mathtt{c}$ are not adjacent.

We later use $\kappa(R)$, that is, κ applied to a regular language, to denote $\kappa(e_1) \cup \ldots \cup \kappa(e_m)$, where $\{e_1, \ldots, e_m\}$ is some (arbitrary) union-free decomposition of R. Note that the star-decomposition of a regular language is not unique; different union-free decompositions give rise to different star-contractions. We assume, for a regular language R, that $\kappa(R)$ deterministically returns *some* valid star-contraction of R.

κ has a number of properties that we will use in the following:

Proposition 1. *For every regular language R,*

1. *$\kappa(R)$ is finite.*
2. *For every regular expression $e \in \kappa(R)$, $\mathcal{L}(e) \subseteq R$.*
3. *$\bigcup_{r \in \kappa(R)} \mathcal{L}(r) = R$.*

Proof. Let e be a union-free regular expression for R.

1. The syntax-directed nature of κ's definition ensures that, to find $\kappa(e)$, we only need to apply κ finitely often. It follows that $\kappa(e)$ is finite.
2. Consider some $r \in \kappa(e)$. We show, by structural induction, that $\mathcal{L}(r) \subseteq \mathcal{L}(e)$. For the base cases ($e = \emptyset$, $e = \varepsilon$, and $e = \mathtt{a}$), this is immediate. Consider the case $e = e_1 \cdot e_2$. By definition, $r \in \kappa(e_1 \cdot e_2)$ means r is of form $r_1 \cdot r_2$, with $r_1 \in \mathcal{L}(e_1)$ and $r_2 \in \mathcal{L}(e_2)$. By the induction hypothesis, $\mathcal{L}(r_1) \subseteq \mathcal{L}(e_1)$ and $\mathcal{L}(r_2) \subseteq \mathcal{L}(e_2)$. But $\circ$ is monotone in both its arguments, so $\mathcal{L}(r_1) \circ \mathcal{L}(r_2) \subseteq \mathcal{L}(e_1) \circ \mathcal{L}(e_2)$. That is, $\mathcal{L}(r) \subseteq \mathcal{L}(e)$. Finally consider the case $e = e_1^*$. By definition, $r \in \kappa(e_1^*)$ means r is of form $(r_1 \mid r_2 \mid \cdots \mid r_k)^*$, with each $r_i \in \kappa(e_1)$. By the induction hypothesis, $\mathcal{L}(r_i) \subseteq \mathcal{L}(e_1)$. But then $\mathcal{L}(r_1 \mid r_2 \mid \cdots \mid r_k) \subseteq \mathcal{L}(e_1)^*$. It follows that $\mathcal{L}(r_1 \mid r_2 \mid \cdots \mid r_k)^* \subseteq \mathcal{L}(e_1)^*$, as $*$ is monotone and $(L^*)^* = L^*$ for all languages L. That is, $\mathcal{L}(r) = \mathcal{L}((r_1 \mid r_2 \mid \cdots \mid r_k)^*) \subseteq \mathcal{L}(e_1^*) = \mathcal{L}(e)$.
3. This is similarly shown by structural induction. The details are left to the reader.

□

3. Refining Regular Abstractions

We now describe the main idea behind the refinement phase. We are interested in the intersection of a finite set of languages. Without loss of generality, we consider the intersection of just two context-free languages L_1 and L_2. We assume that these are provided in the form of context-free grammars.

We furthermore assume a decision procedure that returns "no" if $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) = \emptyset$ or returns a witness w if $w \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \neq \emptyset$, where A_1 and A_2 are finite-state automata recognising regular languages R_1 and R_2, respectively (that is, $\mathcal{L}(A_1) = R_1$ and $\mathcal{L}(A_2) = R_2$). Moreover, our refinement procedure will require the solving of constraints of the form $A = A_1 \setminus A_2$ where A, A_1 and A_2 are finite-state automata. The interpretation of the constraint is that $\mathcal{L}(A) = \mathcal{L}(A_1) \cap \overline{\mathcal{L}(A_2)}$.

Assume that at some point we have regular approximations $R_1 \supseteq L_1$ and $R_2 \supseteq L_2$, and we have found some witness w such that $w \in R_1 \cap R_2$, but $w \notin L_1 \cap L_2$. There are three cases to consider:

1. $w \notin L_1 \wedge w \in L_2$
2. $w \in L_1 \wedge w \notin L_2$
3. $w \notin L_1 \wedge w \notin L_2$

For cases (1) and (2) we should refine R_1 and R_2, respectively. For case (3) we could choose to refine either R_1 or R_2, or both. In our implementation, we always refine all the regular approximations.

If $w \notin L_i$ then a straightforward refinement is to produce a new abstraction $R_i \setminus \{w\}$ in place of R_i. However, this refinement process will rarely converge, as we can exclude only finitely many strings in finite time. Instead we seek a refinement procedure that is able to *generalize* a counterexample to an infinite set of words.

3.1. Star-Generalizations

The refinement procedure in this section operates by taking the regular expression w for the single counterexample w, and progressively augmenting it with $*$ operators while ensuring the counterexample and query remain disjoint.

Definition 6 (Star-generalization)**.** The star-generalizations of a word w is $\Xi(w)$ where $\Xi : \Sigma^* \to \mathcal{P}(Reg'_\Sigma)$ is defined

$$\begin{aligned} \Xi(\varepsilon) &= \{\varepsilon\} \\ \Xi(x_1 \dots x_n) &= \{x_1 \dots x_n, (x_1 \dots x_n)^*\} \cup \left\{ e_1 e_2, (e_1 e_2)^* \;\middle|\; \begin{array}{c} e_1 \in \Xi(x_1 \dots x_i), \\ e_2 \in \Xi(x_{i+1} \dots x_n), \\ i \in [1, n-1] \end{array} \right\} \quad (n \geq 0) \end{aligned}$$

Note that, for $x \in \Sigma$, we have $\Xi(x) = \{x, x^*\}$. Informally, a star-generalization of w is a union-free regular language which can be constructed by adding (nested) unbounded repetition of intervals in w.

We shall represent a star-generalization as a pair $\langle w, S \rangle$ consisting of a word $w = x_1 \dots x_n$ and a set S of ranges within w covered by $*$-operators. A range is represented by a pair (i, j), with $0 \leq i < j$. The range (i, j) identifies the substring $x_{i+1} \dots x_j$ of w.

Given that w has length n, the set S must satisfy the constraints

$$\begin{array}{l} \forall (i,j) \in S \;.\; i, j \in [0, n] \wedge i < j \\ \forall (i,j), (i',j') \in S \;.\; j \leq i' \vee j' \leq i \vee (i \leq i' \oplus j \leq j') \end{array} \tag{1}$$

That is, two ranges must either be disjoint, or else one contains the other ($\oplus$ is the "exclusive or" operation). This ensures the set of $*$-enclosed ranges correspond to properly nested expressions. We shall use $\mathcal{L}(\langle w, S \rangle)$ to denote the language that results from the generalization $\langle w, S \rangle$.

Definition 7 (Star-generalization with respect to a language)**.** The star-generalizations of w with respect to some language L (denoted $\Xi_L(w)$) is the set of star-generalizations of w which are contained in L. Formally,

$$\Xi_L(w) = \{r \mid r \in \Xi(w),\ \mathcal{L}(r) \subseteq L\}.$$

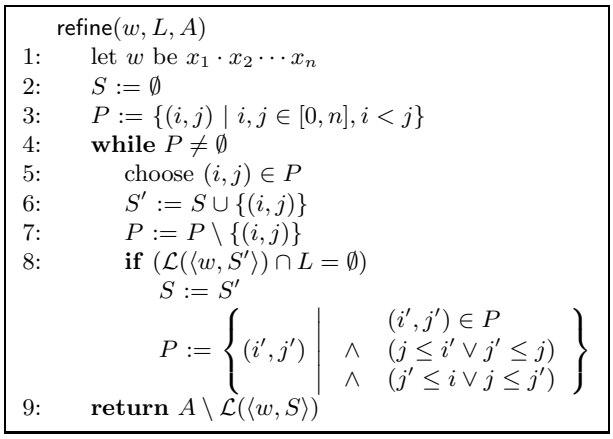


Figure 1: Refining a regular approximation greedily by removing a maximal star-generalization of some counterexample w.

Definition 8 (Maximal star-generalization)**.** A maximal star-generalization of w with respect to some language L is a star-generalization r of w such that there is no other star-generalization r' with $\mathcal{L}(r) \subset \mathcal{L}(r') \subseteq L$.

A maximal star-generalization is not necessarily unique. Consider generalizing `ab` with respect to (`a`*`b`|`ab`*)—both `a`*`b` and `ab`* are maximal generalizations, and they are incomparable.

A greedy procedure for constructing a star-generalization is given in Figure 1. The algorithm takes as input a witness $w \in R_1 \cap R_2$, context-free language L such that $w \notin L$, and L's regular approximation A (either R_1 or R_2). The procedure begins with a trivial star-generalization recognizing only w. P stores the set of (i, j) pairs where a $*$-operation may be introduced without causing the generalization to be malformed. At each step, we add one of the candidate operations to the generalization, then remove any pairs from P which are no longer feasible (because they violate the nestedness requirement). Following (1), this is the set of pairs (i', j') such that $i < i' < j < j' \vee i' < i < j' < j$.

It is worth pointing out that the refinement procedure is *anytime*: If for some reason it would seem necessary or advantageous, one can, without compromising correctness, interrupt the while loop having considered only a subset of the possible $*$-augmentations, thereby settling for a smaller generalization.

Example 3. Let $L = \{\mathtt{a}^i\mathtt{b}^{i+1} \mid i \geq 0\}$ be approximated (currently) by `a`*`b`*, and let the witness $w \notin L$ be `aab`. To refine the regular approximation, refine$(w, L,$ `a`*`b`$^*)$ begins with the trivial star-generalization $\langle w, \emptyset\rangle$. We greedily augment the counterexample with $*$-operations, in this case following lexicographic order, with the accumulated language here described with a regular expression:

$*$-augmentation (i, j)	S	$\mathcal{L}(\langle w, S'\rangle)$	result
	$\emptyset$	`aab`	
$(0, 1)$	$\emptyset$	`a`*`ab`	include, obtaining `a`*`ab`
$(1, 2)$	$\{(0, 1)\}$	`a`*`a`*`b`	exclude, as `b` $\in L$
$(2, 3)$	$\{(0, 1)\}$	`a`*`a(b)`*	exclude, as `abb` $\in L$
$(0, 2)$	$\{(0, 1)\}$	`(a`*`a)`*`b`	exclude, as `b` $\in L$
$(1, 3)$	$\{(0, 1)\}$	`a`*`(ab)`*	include, obtaining `a`*`(ab)`*
$(0, 3)$	$\{(0, 1), (1, 3)\}$	`(a`*`(ab)`*`)`*	include, obtaining `(a`*`(ab)`*`)`*
	$\{(0, 1), (1, 3), (0, 3)\}$		

The complement of the resulting language is `(a`*`ab)`*`b(a|b)`*. The language returned by refine$(w, L,$ `a`*`b`$^*)$ is (represented by) the intersection automaton of this and the initial approximation `a`*`b`*, yielding `bb`*`|aa`*`bbb`*. This is the new, improved, regular approximation of L. By construction, it does not contain `aab`, but more importantly, along with `aab`, an infinite number of other strings have been discarded from the previous approximation `a`*`b`*, for example, all the strings that start with `a` and fail to have consecutive `b`s.

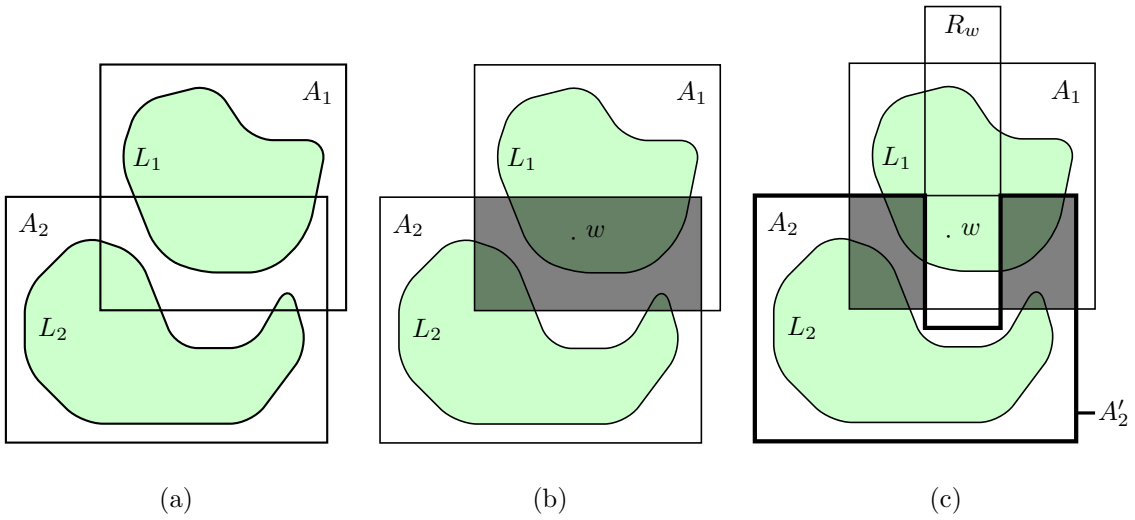


Figure 2: Refining a regular approximation

The overall flow of the refinement step is illustrated in Figure 2. Part (a) shows two context-free languages L_1 and L_2, together with their initial regular approximations A_1 and A_2, respectively. (We depict regular languages as rectilinear polygons, to suggest their more limited expressiveness.) In part (b), a counterexample $w \in A_1 \cap A_2$ (the shaded area) has been identified. Since $w \notin L_2$, we build a generalization R_w such that $\{w\} \subseteq R_w \subseteq \overline{L_2}$. This generalization is another regular language. Part (c) shows L_2's new approximation $A_2' = A_2 \setminus R_w$. The thick contours indicate the extent of this improved regular approximation.

Figure 1's refinement procedure has these properties:

1. It is *sound*: $L_i \subseteq (R_i \setminus \mathcal{L}(R)) \subseteq R_i$.
2. It *terminates*: the while loop will be executed at most $\frac{n(n+1)}{2}$ times, where $n = |w|$.
3. It is *progressive*: the same witness w cannot be produced again upon successive calls to refine.

The refinement algorithm involves a check (line 8) to see if a regular and a context-free language overlap. This problem is decidable in polynomial time. The algorithm pre^* described in [7, 8] has a time complexity of $O(|P| \times |Q|^3)$ and space complexity of $O(|P| \times |Q|^2)$, where $|P|$ is the number of productions in the context-free grammar and $|Q|$ is the number of states in the automaton.[1]

The following lemma and theorem is critical to the completeness result, showing the relationship between κ and Ξ. Note that here we are intersecting sets of languages, rather than the languages themselves.

Lemma 2. *Let the regular expression e be union-free and let κ and Ξ be the star-contraction and star-generalizations given in Definitions 5 and 6. Then for all $w \in \mathcal{L}(e)$, $\kappa(e) \cap_{\mathcal{L}} \Xi(w) \neq \emptyset$.*

Proof. Assume $e = x$, with $x \in \Sigma \cup \{\varepsilon\}$. Then $w = x$. From the definitions, we have $x \in \kappa(e)$, and $x \in \Xi(w)$.

Assume $e = e_1 \cdot e_2$, and that the induction hypothesis holds on e_1 and e_2. Consider some word $w \in \mathcal{L}(e)$. We can partition w into $w_1 \cdot w_2$, such that $w_1 \in \mathcal{L}(e_1)$, $w_2 \in \mathcal{L}(e_2)$. By the induction hypothesis, there are $r_{\kappa 1}, r_{\Xi 1}, r_{\kappa 2}, r_{\Xi 2}$ such that $\mathcal{L}(r_{\Xi 1}) = \mathcal{L}(r_{\kappa 1}) \in \kappa(e_1) \cap_{\mathcal{L}} \Xi(w_1)$, $\mathcal{L}(r_{\kappa 2}) = \mathcal{L}(r_{\Xi 2}) \in \kappa(e_2) \cap_{\mathcal{L}} \Xi(w_2)$. As $r_{\kappa 1} \in \kappa(e_1)$ and $r_{\kappa 2} \in \kappa(e_2)$, by Definition 5 we have $r_{\kappa 1} \cdot r_{\kappa 2} \in \kappa(e)$. By Definition 6, we also have $r_{\Xi 1} \cdot r_{\Xi 2} \in \Xi(w)$. Therefore $\mathcal{L}(r_{\kappa 1} \cdot r_{\kappa 2}) = \mathcal{L}(r_{\Xi 1} \cdot r_{\Xi 2}) \in \kappa(e) \cap \Xi(w)$.

Assume $e = (e')^*$, for some e' satisfying the induction hypothesis. Consider some word $w \in e$. We can partition w into $w_1 \ldots w_k$, with each $w_i \in e'$. By the induction hypothesis, each w_i admits some star-generalization r_i such that $\mathcal{L}(r_i) \in \kappa(e') \cap_{\mathcal{L}} \Xi(w_i)$. Consider the generalization r given by $r = (r_1^* \ldots r_k^*)^*$.

[1] We use this algorithm in the implementation of COVENANT.

By Definition 6, $r \in \Xi(w)$. Moreover, r is equivalent to $(r_1 \mid \cdots \mid r_k)^*$, which is in $\kappa(e)$. Therefore, $\mathcal{L}(r) \in \Xi(w) \cap_{\mathcal{L}} \kappa(e)$. □

Theorem 2. *Let R be a regular language, and κ and Ξ be the star-contraction and star-generalizations given in Definitions 5 and 6. Then for all $w \in R$, $\kappa(R) \cap_{\mathcal{L}} \Xi(w) \neq \emptyset$.*

Proof. As noted in Section 2.3, the star-contraction of a regular language R is computed based on some union-free decomposition $E = \{e_1, \ldots, e_n\}$ of R. Consider $w \in R$. Since $R = \mathcal{L}(\|\, E)$, there is some $e \in E$ such that $w \in \mathcal{L}(e)$. By Lemma 2, $\kappa(e) \cap_{\mathcal{L}} \Xi(w) \neq \emptyset$. As $\kappa(e) \subseteq \kappa(R)$, we have $\kappa(R) \cap_{\mathcal{L}} \Xi(w) \neq \emptyset$. □

3.2. Epsilon-Generalization

The notion of star-generalization, while useful for reasoning, does not integrate well into existing automaton algorithms. In this section, we introduce a slightly different form of generalization.

Let $\nu(w)$ denote the automaton recognizing the single word $w = x_1 \ldots x_n$. We have $\nu(x_1 \ldots x_n) = \langle Q, \Sigma, \delta, q_0, \{q_n\}\rangle$, with $Q = \{q_0, \ldots, q_n\}$ and $\delta = \{(q_{i-1}, x_i, q_i) \mid i \in [1, n]\}$.

Definition 9 (Epsilon-generalization)**.** An epsilon-generalization of w is any language obtained by augmenting the transition function of $\nu(w)$ with additional edges $E \cup P$, given by:

$$E \subseteq \{(q_i, \varepsilon, q_j) \mid i < j\} \tag{2}$$

$$P \subseteq \{(q_{j-1}, w_j, q_i) \mid i < j\} \tag{3}$$

That is, we may only introduce epsilon-transitions forwards; backwards transitions always consume the same input character as the original outgoing transition from the source state. We shall use $\mathsf{Gen}(w)$ to denote the set of epsilon-generalizations of w. Note that where $\Xi(w)$ is a set of languages, $\mathsf{Gen}(w)$ is a set of automata. Similarly, $\mathsf{Gen}_L(w)$ denotes the set of epsilon-generalizations with respect to some language L.

Lemma 3. *Let $A_1 = \langle Q_1, \Sigma, \delta_1, q_{10}, \{q_{1n}\}\rangle$ and $A_2 = \langle Q_2, \Sigma, \delta_2, q_{20}, \{q_{2n}\}\rangle$ be (nondeterministic) finite-state automata such that q_{1n} has no outgoing edges.*

Then the automaton $A_1 \circ A_2 = \langle (Q_1 \cup Q_2) \setminus \{q_{1n}\}, \Sigma, \delta_1 \cup \delta_2[q_{1n} \mapsto q_{20}], q_{10}, \{q_{2n}\}\rangle$ recognizes the language $\mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$.

Proof. Let $A = A_1 \circ A_2$. Assume $w \in \mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$. Then $w = w_1 \cdot w_2$, for some $w_1 \in \mathcal{L}(A_1), w_2 \in \mathcal{L}(A_2)$. So there is some path from q_{10} to q_{20} matching w_1 in $A_1 \circ A_2$, and a path from q_{20} to q_{2n} matching w_2. Therefore w is recognized by A.

Assume w is recognized by A. There are no transitions from states in Q_1 to states in Q_2 except q_{20}; therefore, any path from q_{10} to q_{2n} in A must pass through q_{20}. There are no transitions from states in Q_2 to states in Q_1 (as q_{1n} had no outgoing edges). Therefore, once reaching a state in Q_2, a path through A must remain in Q_2.

Hence we can divide the path through A into a prefix, following transitions exclusively in A_1 and reaching q_{20}, and a suffix from q_{20} to q_{2n} following transitions exclusively in A_2. Therefore, $w \in \mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$. □

Theorem 3. *For every word w and $e \in \Xi(w)$, there is some $A \in \mathsf{Gen}(w)$ such that $\mathcal{L}(e) = \mathcal{L}(A)$. That is, star-generalization of w may be expressed by an equivalent epsilon-generalization.*

Proof. Consider $e \in \Xi(w)$.

Assume $e = x$, $x \in \Sigma \cup \{\varepsilon\}$. $\nu(x) \in \mathsf{Gen}(w)$, and $\mathcal{L}(\nu(x)) = \mathcal{L}(e)$. Also, the final state of $\nu(x)$ has no outgoing transitions.

Assume that for all expressions e' of up to depth k, if $e' \in \Xi(w)$ there is some $A \in \mathsf{Gen}(w)$ such that $\mathcal{L}(A) = \mathcal{L}(e')$, and the accept state of A has no outgoing transitions. Consider some star-generalization $e \in \Xi(w)$ of depth $k + 1$.

Assume $e = e_1 \cdot e_2$ for $e_1 \in \Xi(w_1)$, $e_2 \in \Xi(w_2)$ and $w = w_1 w_2$. As e_1 and e_2 have depth at most k, the induction hypothesis provides automata $A_1 \in \mathsf{Gen}(w_1)$ and $A_2 \in \mathsf{Gen}(w_2)$. But then the automaton $A' = A_1 \circ A_2$ (with $\circ$ as defined in Lemma 3) is a valid epsilon-generalization. So $A \in \mathsf{Gen}(w)$. By Lemma 3,

$A' = A_1 \circ A_2$ recognizes $\mathcal{L}(A_1) \cdot \mathcal{L}(A_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$. Therefore, $\mathcal{L}(A) = \mathcal{L}(e)$. As the final state of A_2 had no outgoing transitions, and we have not added any outgoing transitions from q_n, the final state of A' has no outgoing transitions.

Assume $e = (e')^*$ for some generalization $e' \in \Xi(w)$. As e has depth $k+1$, e' is of depth k. By the induction hypothesis there is some automaton $A = \langle Q, \Sigma, \delta, q_0, \{q_n\}\rangle \in \mathsf{Gen}(w)$ such that $\mathcal{L}(A) = \mathcal{L}(e')$. We construct a new automaton A' with transition relation δ'

$$\delta' = \delta \cup \{q_0 \xrightarrow{\varepsilon} q_n\} \cup \{(q_{j-1} \xrightarrow{w_j} q_0) \mid j > 0 \wedge (q_j \xrightarrow{\varepsilon^*} q_n) \in \delta\}$$

(Here $q_j \xrightarrow{\varepsilon^*} q_n$ says that q_n is in the epsilon-closure of $\{q_j\}$.) The added transitions are of the form permitted by Definition 9. Since A is an epsilon-generalization of w, A' is also a valid epsilon-generalization. As the accept state of A had no outgoing transitions, and we have not added any transitions beginning at q_n, the accept state of A' also has no outgoing transitions. We now consider the language recognized by A'. Assume some word w is in $\mathcal{L}((e')^*)$. Then either $w = \varepsilon$, or $w = w_1 \dots w_m$ such that $w_i \in \mathcal{L}(e') \setminus \{\varepsilon\}$. If $w = \varepsilon$, then w is recognized by A' (by $q_0 \xrightarrow{\varepsilon} q_n$). Otherwise, each w_i is recognized by some path p_i from q_j to q_k in A, such that q_n is reachable from q_k by ε-transitions. Let $q_{k'}$ be the second-last state in p_i. By construction, there must be some alternate transition from $q_{k'}$ to q_0 in A'; then there must be some path, along which w_i is matched, from q_0 to q_0 in A'. Therefore, w is recognized by A'. Now assume there is some word $w \neq \varepsilon$ recognized by A'. We can partition w into sub-words $w_1 \dots w_m$ such that the path of each w_i with $i < m$ starts at q_0, makes its final transition via an introduced edge, and uses no other introduced edges. As the path corresponding to w_i finishes with an introduced edge, there must be some corresponding path from q_0 to q_n in A. So $w_i \in \mathcal{L}(e')$ for $i < m$. And as the path corresponding to w_m starts at q_0 and does not use any introduced edges, $w_m \in \mathcal{L}(e')$. Therefore, $w \in \mathcal{L}((e')^*) = \mathcal{L}(e)$. It follows that $\mathcal{L}(A') = \mathcal{L}((e')^*) = \mathcal{L}(e)$.

As we can construct epsilon-generalizations for trivial star generalizations (depth $k = 1$), and epsilon-generalizations of size k can be constructed from star generalizations of size $k-1$, every element of $\Xi(w)$ must correspond to an equivalent element of $\mathsf{Gen}(w)$. □

We can incrementally construct an epsilon-generalization with respect to some co-context-free language $\overline{L}$ by adapting algorithms for the intersection of context-free languages and finite automata, such as the pre^* algorithm described in [7].

Essentially, we maintain a table $\tau \subseteq Q \times \Gamma \times Q$ such that $(q_i, P, q_j) \in \tau$ iff the context-free production P can be generated by some sub-word matched along a path from q_i to q_j. Whenever a new transition is added to the generalization, we update τ with any newly feasible productions; if the start production S is ever generated on a path from q_0 to q_n, the generalization has ceased to be valid—in which case we revert the table and discard the most recent augmentation. The pre^* algorithm [8], exhibits $O(|\Gamma||Q|^3)$ worst-case time complexity. In the worst case, where every generalization step fails after the maximum number of steps, this gives the generalization procedure a worst-case complexity of $O(|\Gamma||Q|^5)$.

Example 4. Consider again the star-generalization of `aab` described in Example 3. If we are instead constructing an epsilon-generalization, we start with the automaton A, recognizing $\{w\}$, shown in Figure 3(a). We first try greedily adding forwards epsilon transitions:

ε-augmentation (i,j)	S	$\mathcal{L}(\langle w, S'\rangle)$	result
	$\emptyset$	`aab`	
$(0,1)$	$\emptyset$	`a?ab`	include, obtaining `a?ab`
$(1,2)$	$\{(0,1)\}$	`a?a?b`	exclude, as `b` $\in L$
$(2,3)$	$\{(0,1)\}$	`a?a?b`	exclude, as `b` $\in L$
$(2,3)$	$\{(0,1)\}$	`a?ab?`	include, obtaining `a?ab?`
$(0,2)$	$\{(0,1),(2,3)\}$	`(a?a)?b?`	exclude, as `b` $\in L$
$(1,3)$	$\{(0,1),(2,3)\}$	`a?(ab?)?`	include, obtaining `a?(ab?)?`
$(0,3)$	$\{(0,1),(2,3),(1,3)\}$	`(a?(ab?)?)?`	include, obtaining `(a?(ab?)?)?`
	$\{(0,1),(2,3),(1,3),(0,3)\}$		

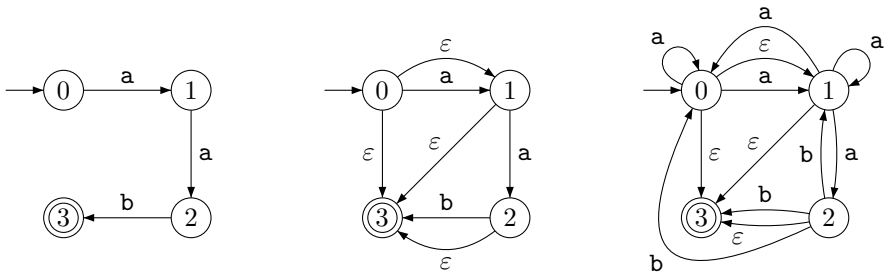


Figure 3: Construction of $\text{refine}_\varepsilon(w, L)$, where $w = \texttt{aab}$ and $L = \{\texttt{a}^i\texttt{b}^{i+1} \mid i \geq 0\}$: (a) Initial automaton for w; (b) the automaton after adding forward ε-transitions; these are the edges E from (2); (c) the final epsilon-generalization after adding non-ε edges; these are the edges P from (3).

This yields the automaton shown in Figure 3(b), corresponding to the language $\{\varepsilon, \texttt{a}, \texttt{aa}, \texttt{ab}, \texttt{aab}\}$. We then progressively introduce backwards transitions as follows:

ε-augm. (i, j)	S	$\mathcal{L}(\langle w, S'\rangle)$	result
	$\emptyset$	$(\texttt{a?(ab?)?)?}$	
$(1-1, \texttt{a}, 0)$	$\emptyset$	$\texttt{a}^*\texttt{(a?(ab?)?)?}$	include, obtaining $\texttt{a}^*\texttt{(a?(ab?)?)?}$
$(2-1, \texttt{a}, 1)$	$\{(0, \texttt{a}, 0)\}$	$\texttt{a}^*\texttt{(a}^*\texttt{(ab?)?)?}$	include, obtaining $\texttt{a}^*\texttt{(a}^*\texttt{(ab?)?)?}$
$(3-1, \texttt{b}, 2)$	$\{(0, \texttt{a}, 0), (1, \texttt{a}, 1)\}$	$\texttt{a}^*\texttt{(a}^*\texttt{(ab}^*\texttt{)?)?}$	exclude, as $\texttt{abb} \in L$
$(2-1, \texttt{a}, 0)$	$\{(0, \texttt{a}, 0), (1, \texttt{a}, 1)\}$	$\texttt{a}^*\texttt{(ab}^*\texttt{)?}$	include, obtaining $\texttt{a}^*\texttt{(ab}^*\texttt{)?}$
$(3-1, \texttt{b}, 1)$	$\{(0, \texttt{a}, 0), (1, \texttt{a}, 1), (1, \texttt{a}, 0)\}$	$\texttt{a}^*\texttt{(a}^*\texttt{ab)}^*\texttt{(ab)?}$	include, obtaining $\texttt{a}^*\texttt{(a}^*\texttt{ab)}^*\texttt{b?}$
$(3-1, \texttt{b}, 0)$	$\{(0, \texttt{a}, 0), (1, \texttt{a}, 1), (1, \texttt{a}, 0), (2, \texttt{b}, 1)\}$	$\texttt{a}^*\texttt{(a}^*\texttt{ab)}^*\texttt{(ab)?}$	include, obtaining $\texttt{a}^*\texttt{(a}^*\texttt{ab)}^*\texttt{(ab)?}$

Note that the listed regular expressions recognize the same language as, but do not necessarily directly correspond to, the state of the automaton. In several cases, the augmentations do not add any words to the language recognized by the automaton (for example, the ε-transition $(0, 3)$, and the backward transition $(2, \texttt{b}, 0)$). The process terminates with the automaton shown in Figure 3(c). The language recognized by this automaton is $(\texttt{a}^*\texttt{ab})^*\texttt{a}^*$, which is equivalent to the language obtained by star-generalization in Example 3.

3.3. Maximum Generalization

The procedure described in the previous section constructs *some* maximal element of $\Xi_L(w)$ (or $\mathsf{Gen}_L(w)$). It is, however, undirected; the generalization is chosen blindly from the set of possible maximal generalizations.

Even if the input languages are regularly separable, it is possible that the refinement step may choose an infinite sequence of generalizations which, though maximal, cannot separate the languages.

We can instead construct a generalization $\mathsf{gen}(L, w)$ which computes the *union* of all maximal star-generalizations of w with respect to $\overline{L}$. That is, it computes $\xi_{\overline{L}}(w) = \bigcup \Xi_{\overline{L}}(w)$ directly. We shall refer to this as the *maximum* star-generalization. A possible (though inefficient) method for computing it is given in Figure 4.

The procedure maxgen uses S, a partial generalization, and P, the set of candidate $*$-augmentations. At each stage, an augmentation e is selected from P, and we recursively compute the set of valid further generalizations of $\langle w, S\rangle$ both including and excluding e, finally taking the union of the sub-languages.

The maximum epsilon-generalization with respect to L—denoted by $\mathsf{g}_L(w)$—may be constructed by an analogous procedure.

We now show that this generalization procedure is sufficiently powerful to prove separability for any pair of regularly separable languages.

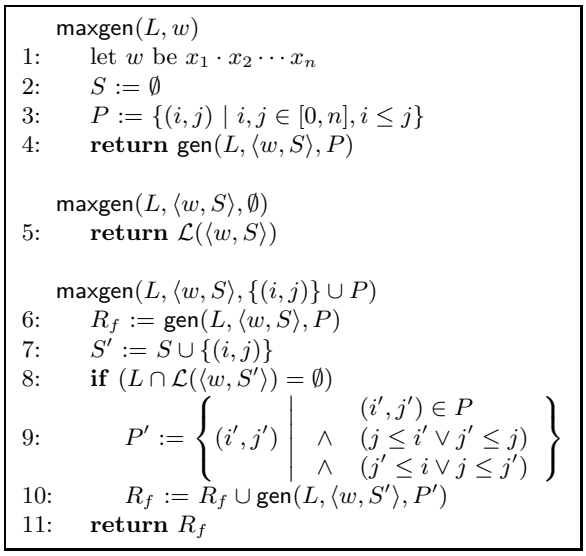


Figure 4: Computing $\xi_{\overline{L}}(w)$, the maximum star-generalization of w with respect to $\overline{L}$.

Lemma 4. *Consider a context-free language L, and regular language R such that $L \cap R = \emptyset$. Then for any word $w \in R$, there is some $e' \in \kappa(R)$ such that $\mathcal{L}(e') \subseteq \xi_{\overline{L}}(w)$.*

Proof. By Lemma 2, there is some $e \in \Xi(w)$ such that $\mathcal{L}(e) \in \Xi(w) \cap_{\mathcal{L}} \kappa(R)$. As $R \cap L = \emptyset$, we have $e \in \Xi_{\overline{L}}(w)$. Therefore $\mathcal{L}(e) \subseteq \bigcup\{\mathcal{L}(e') \mid e' \in \Xi_{\overline{L}}(w)\} = \xi_{\overline{L}}(w)$. □

Corollary 1. *Consider a context-free language L, and regular language R such that $L \cap R = \emptyset$. Then for any word $w \in R$, there is some $e' \in \kappa(R)$ such that $\mathcal{L}(e') \subseteq \mathtt{g}_{\overline{L}}(w)$.*

Proof. This follows from Theorem 3 and Lemma 4. □

Theorem 4. *Given a pair of regularly separable context-free languages (L, L') and initial regular approximations R_L and $R_{L'}$ with $L \subseteq R_L$ and $L' \subseteq R_{L'}$, the refinement process described in Section 3 will construct a separating pair $(S_L, S_{L'})$ in a finite number of steps when refining using the maximum star- or epsilon-generalization.*

Proof. Consider the (unknown) regular language S separating L and L'. Let K^i denote the elements of $\kappa(S) \cup \kappa(\overline{S})$ having non-empty intersections with the current approximation $R^i_L \cap R^i_{L'}$.

Assume that, at step i, there is a word $w \in R^i_L \cap R^i_{L'}$. w must be in exactly one of S and $\overline{S}$; we assume w is in S (the case of $\overline{S}$ is symmetric). As $w \in S$, there must be some $r \in \kappa(S)$ such that $\mathcal{L}(r) \subseteq \xi_{\overline{L'}}(w) \subseteq \mathtt{g}_{\overline{L'}}(w)$. Since $w \in R^i_L \cap R^i_{L'}$, and $w \in \mathcal{L}(r)$, we have $r \in K^i$.

As $R^{i+1}_{L'} = R^i_{L'} \setminus \xi_{\overline{L'}}(w)$ (or $R^i_{L'} \setminus \mathtt{g}_{\overline{L'}}(w)$), we have $\mathcal{L}(r) \cap R^{i+1}_{L'} = \emptyset$. Therefore, $K^{i+1} \subset K^i$. As $[K^1, K^2, \ldots]$ is a decreasing sequence, and K^1 is finite, the refinement process must terminate after finitely many steps. □

4. A Final Example

Consider the two context-free grammars $G_1 = \langle\{S_1, A_1, B_1\}, \Sigma, P_1, S_1\rangle$ and $G_2 = \langle\{S_2, A_2, B_2\}, \Sigma, P_2, S_2\rangle$ where $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and P_1 and P_2 are, respectively,

$$\begin{array}{lcl} S_1 & \to & A_1B_1 \\ A_1 & \to & \mathtt{aa} \mid \mathtt{bb} \mid \mathtt{a}S_1\mathtt{a} \mid \mathtt{b}S_1\mathtt{b} \\ B_1 & \to & \mathtt{ab}B_1 \mid \mathtt{ab} \end{array} \qquad \begin{array}{lcl} S_2 & \to & A_2B_2 \\ A_2 & \to & \mathtt{aa} \mid \mathtt{bb} \mid \mathtt{a}S_2\mathtt{a} \mid \mathtt{b}S_2\mathtt{b} \\ B_2 & \to & \mathtt{ba}B_2 \mid \mathtt{ba} \end{array}$$

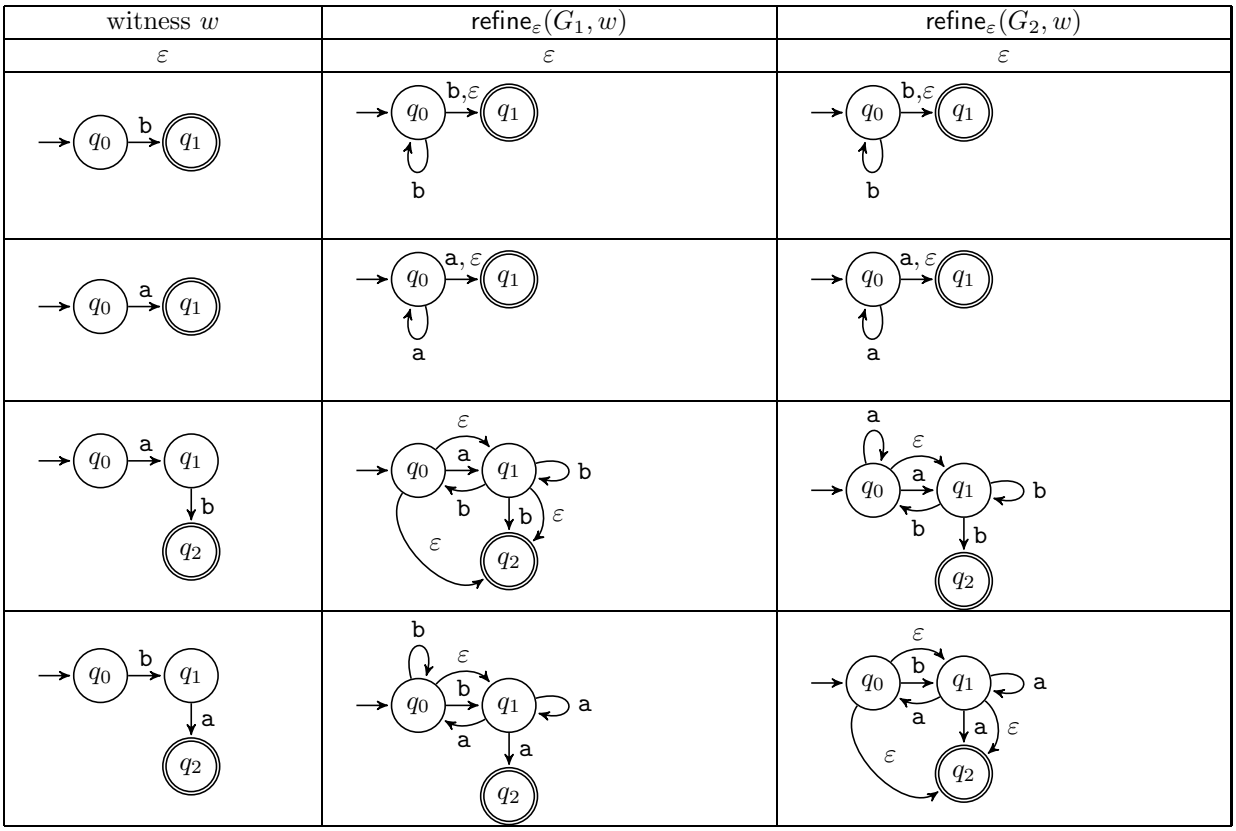


Figure 5: Relevant witnesses and generalizations obtained by greedy epsilon generalization.

Note that $\mathcal{L}(G_1) = \{ww^R(\mathtt{ab})^+ \mid w \in \Sigma^*\}$ and $\mathcal{L}(G_2) = \{ww^R(\mathtt{ba})^+ \mid w \in \Sigma^*\}$.

The first step of our method is to approximate G_1 and G_2 with finite-state automata A_1 and A_2. The only requirement is that $\mathcal{L}(G_1) \subseteq \mathcal{L}(A_1)$ and $\mathcal{L}(G_2) \subseteq \mathcal{L}(A_2)$. For simplicity, assume $A_1 = A_2 = \Sigma^*$. Next, we check if $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) \neq \emptyset$. In this case, the intersection is trivially not empty. Furthermore, our regular solver provides the witness $w = \varepsilon$. This cannot be generalized, so we eliminate ε from both approximations and try again, this time obtaining $w = \mathtt{b}$. In the third step we refine the regular approximations. We assume the use of greedy epsilon refinement, preferring backwards transitions. We first generalize the witness by calling $\mathsf{refine}_\varepsilon(G_1, w)$ and $\mathsf{refine}_\varepsilon(G_2, w)$ to produce new approximations $\mathcal{L}(A_1') = \mathcal{L}(A_1) \setminus \mathsf{refine}_\varepsilon(G_1, w)$ and $\mathcal{L}(A_2') = \mathcal{L}(A_2) \setminus \mathsf{refine}_\varepsilon(G_2, w)$, respectively. We show the automata obtained from $\mathsf{refine}_\varepsilon(G_1, b)$ and $\mathsf{refine}_\varepsilon(G_2, b)$ on the first row in Figure 5. In both cases, we obtain the language $\mathtt{b}^*$.

Since G_1 and G_2 are regularly separable, using maximal refinement would be guaranteed to eventually eventually halt proving that the languages are disjoint. While we have no such guarantee for greedy refinement, in this case it successfully proves separation after 5 refinement steps. Figure 5 depicts the rest of witnesses obtained as well as their generalizations produced by the procedure $\mathsf{refine}_\varepsilon$.

5. Previous Refinement Techniques

Several CEGAR-based approaches have been proposed for testing intersection of context-free languages. In this section, we attempt to characterise the expressiveness of existing refinement methods. For these comparisons we do not consider the effect of initial regular approximations, as they do not affect the expressiveness of the refinement method. For any fixed finite set of regularly-separable languages, there is always some approximation scheme which allows the languages to be trivially proven separate; however it is impossible to define such an approximation in general.

The idea of using CEGAR to check the intersection of CFGs was pioneered by Bouajjani *et al.* [3] for the context of verifying concurrent programs with recursive procedures. They rely on the concept of *refinable finite-chain abstraction* consisting of computing the series $(\alpha_i)_{i\geq 1}$ which over-approximates the language of a CFG L (that is, $L \subseteq \alpha_i(L)$) such that $\alpha_1(L) \supset \alpha_2(L) \supset \cdots \supseteq L$. The method is parameterized by the refinable abstraction. [3] describe several possible abstractions but no experimental evaluation is provided. Chaki *et al.* [5] extend [3] by, among other contributions, implementing and evaluating this method. The experimental evaluation of Chaki *et al.* uses both the i^{th}*-prefix* and i^{th}*-suffix* abstractions. Given language L, the i^{th}*-prefix* abstraction $\alpha_i(L)$ is the set of words of L of length less than i, together with the set of prefixes of length i of L. The i^{th}*-suffix abstraction* can be defined analogously. We next provide a theorem about the expressiveness of these abstractions.

Theorem 5. *There exist regularly separable languages that can be shown separate neither by the i^{th}-prefix, nor by the i^{th}-suffix, abstraction.*

Proof. Consider the languages $R_1 = \mathtt{a}^*\mathtt{ba}^*$ and $R_2 = \mathtt{a}^*\mathtt{ca}^*$. $R_1 \cap R_2$ is empty. However, for a given length i, the string $\mathtt{a}^i$ forms a prefix to words in both R_1 and R_2. It follows that the intersection of the two abstractions will always be non-empty, so the refinement method cannot prove the languages separate. A similar argument proves the case for suffix abstraction. □

The LCEGAR method described by Long *et al.* [15] is based on a similar refinement framework, but the approach differs radically. They maintain a pair of context-free grammars A_1, A_2 over-approximating the intersection of the original languages. At each refinement step, an *elementary bounded language* B_i is generated from each grammar A_i.[2] The refinement ensures $B_i \cap A_i \neq \emptyset$, but B_i is not necessarily either an over- or under-approximation of A_i. They then compute $I = B_i \cap L_1 \cap L_2$. If I is non-empty, $L_1 \cap L_2$ must also be non-empty. If I is empty, then the approximations can safely be refined by subtracting the B_i.

We now wish to characterise the set of languages for which LCEGAR can prove separation. Note that we do not consider the initial approximation; for any fixed pair of regularly-separable languages, there necessarily exists *some* approximation method which immediately proves separation without refinement.

Theorem 6. *There exist non-regularly-separable languages which can be proven separate by* LCEGAR.

Proof. Consider the languages L and $\overline{L}$, where $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$. These are not regularly separable. Still, LCEGAR will find that they do not overlap. Assume initial approximations $A_1 = L$ and $A_2 = \overline{L}$. At the first iteration, LCEGAR may choose bounded approximation $B = \mathtt{a}^*\mathtt{b}^*$. It will find $B \cap L \cap \overline{L} = \emptyset$, then update $A_1 = A_1 \setminus B = \emptyset$. As $A_1 = \emptyset$, the refinement process has successfully proven separation. □

Lemma 5. *For every bounded regular language $B = w_1^* \cdots w_k^*$, there is some word p that is not a substring of any word in B.*

Proof. For each word w_1, we pick some character c_1 which differs from the *last* character of w_1. We then construct $p = p_1 \cdots p_k$, such that:

$$p_i = \underbrace{c_i \ldots c_i}_{|w_i|}$$

Assume there is some word $t = up_1 \ldots p_k v \in B$. t must consist of some number of occurrences of w_1 through w_k, in order. Since p_1 differs from the last character of w_1, up_1 cannot consist only of occurrences of w_1; therefore, $p_2 \cdots p_k$ must be made up of occurrences of w_2 through w_k.

Similarly, since no occurrence of w_2 may end in p_2, so $p_3 \cdots p_k$ must consist only of w_3 through w_k. By induction, we find that p_k must be an occurrence of w_k. However, no occurrence of w_k may occur in p_k. Therefore, there can be no word $t \in B$ such that $t \in \Sigma^* p \Sigma^*$. □

Corollary 2. *For every finite set of bounded regular languages $\{B_1, \ldots, B_n\}$, we can construct some substring p that is not a substring of $B_1 \cup \cdots \cup B_n$.*

[2] An elementary bounded language is some language of the form $B = w_1^* \ldots w_k^*$, where each w_i is a (finite) word in Σ^*.

Proof. By Lemma 5, we can find $p_1, \ldots, p_n$ such that p_i is not a substring in B_i. Then $p = p_1 \cdots p_n$ cannot occur as a substring in $B_1 \cup \ldots \cup B_n$. □

Theorem 7. *There exist regularly separable languages for which the* LCEGAR *refinement method cannot prove separability.*

Proof. Consider an LCEGAR process with $L_1 = A_1 = (\mathtt{a}|\mathtt{b})^*\mathtt{a}$, and $L_2 = A_2 = (\mathtt{a}|\mathtt{b})^*\mathtt{b}$. These languages are disjoint, and regularly separable. After some finite number of steps, the approximations have been refined with bounded languages $\{B_1, \ldots, B_n\}$. By Corollary 2, there is some substring p such that $\Sigma^* p \Sigma^* \subseteq (\overline{B_1} \cap \ldots \cap \overline{B_n})$. The updated approximation A_1' is non-empty, as it contains $p\mathtt{a}$. Similarly, the approximation A_2' is non-empty, as it contains $p\mathtt{b}$.

Since after any finite sequence of refinement steps neither A_1' nor A_2' is empty, the refinement process will never prove separation of L_1 and L_2. □

From Theorems 4, 6 and 7, we conclude that the classes of languages which can be proven separate by LCEGAR and COVENANT are incomparable.

6. Experimental Evaluation

We have implemented the CEGAR method proposed in this paper in a prototype tool called COVENANT.[3] The tool is implemented in C++ and parameterized by the initial approximation and the refinement procedure. COVENANT implements the method described in [20] for approximating CFGs with strongly regular languages as well as the coarsest abstraction Σ^* for comparison purposes. For refinement, the tool implements both the greedy and maximum star-epsilon generalizations (described in Sections 3.1 and 3.3, respectively). COVENANT currently implements only the classical product construction for solving the intersection of regular languages but other regular solvers (for example, [9, 12]) can be easily integrated.[4]

To assess the effectiveness of our tool, we have conducted two experiments. First, we used COVENANT for proving safety properties in recursive multi-threaded programs. Second, we crafted pairs of challenging context free grammars and intersected them using COVENANT. The motivation for this second experiment was to exercise features of COVENANT that were not required during the first experiment. All experiments were run on a single core of a 2.4GHz Core i5-M520 with 7.8Gb memory.

6.1. Safety Verification of Recursive Multi-Threaded Programs

Pioneering work by Bouajjani *et al.* [3] has showing that the safety verification problem of recursive multi-threaded programs can be reduced to testing the intersection of context-free languages for emptiness. Since then, several encodings have been described [3, 5, 15]. As a result, we can use COVENANT to prove certain safety properties in recursive multi-threaded programs assuming the programs have been translated accordingly. We briefly exemplify the translation of a concurrent program to context-free grammars, using the approach of [15].

For simplicity, we assume a concurrency model in which communication is based on shared memory. Shared memory is modelled via a set of global variables. We assume that each statement is executed atomically. We will consider only *Boolean programs*. Any program P can be translated into a Boolean program $\mathcal{B}(P)$ using techniques such as predicate abstraction [10]. A key property is that $\mathcal{B}(P)$ is an over-approximation of P preserving the control flow of P, but the only type available in $\mathcal{B}(P)$ is Boolean. Therefore, if $\mathcal{B}(P)$ is correct then P must be correct but, of course, if $\mathcal{B}(P)$ is unsafe, P may still be safe.

Each (possibly recursive) procedure in $\mathcal{B}(P)$ is modelled as a context-free grammar as well as each shared variable specifying the possible values that the variable can take. In addition, extra production rules are added to specify the synchronization points.

[3] Publicly available at `https://bitbucket.org/jorgenavas/covenant` together with all the benchmarks used in this section.

[4] In fact, an initial implementation of COVENANT was tested using REVENANT [9], an efficient regular solver based on bounded model checking with interpolation, though the released version does not incorporate it.

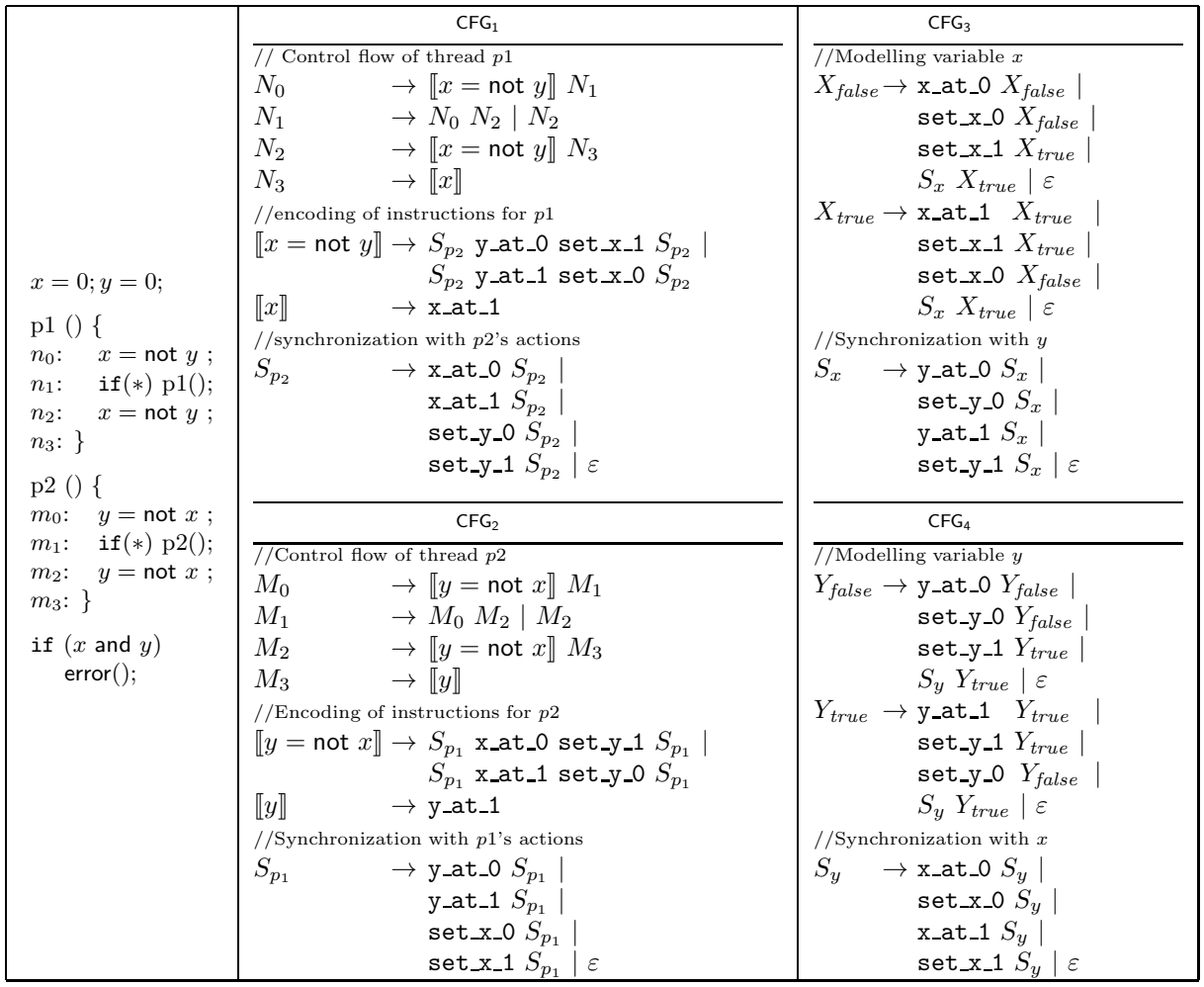


Figure 6: A concurrent Boolean program (SharedMem) and its translation to CFGs.

The left hand column of Figure 6 shows a small program SharedMem [15]. It consists of two symmetric, recursive procedures $p1$ and $p2$ which are executed by two different threads. The communication between the threads is done through the global variables x and y which are initially set to 0. Note that the program is already Boolean since x and y can only take values 0 and 1. We would like to prove that after $p1$ and $p2$ terminate, x and y cannot be true simultaneously.

The rest of Figure 6 describes the corresponding translation to context-free grammars. The four resulting grammars, which we explain shortly, are

$$\begin{aligned}
\mathsf{CFG}_1 &: \ \langle \{N_0, N_1, N_2, N_3, [\![x = \mathsf{not}\ y]\!], [\![x]\!], S_{p_2}\}, \Sigma, P_1, N_0 \rangle \\
\mathsf{CFG}_2 &: \ \langle \{M_0, M_1, M_2, M_3, [\![y = \mathsf{not}\ x]\!], [\![y]\!], S_{p_1}\}, \Sigma, P_2, M_0 \rangle \\
\mathsf{CFG}_3 &: \ \langle \{X_{false}, X_{true}, S_x\}, \Sigma, P_3, X_{false} \rangle \\
\mathsf{CFG}_4 &: \ \langle \{Y_{false}, Y_{true}, S_y\}, \Sigma, P_4, Y_{false} \rangle
\end{aligned}$$

where $\Sigma = \{\texttt{x\_at\_0}, \texttt{x\_at\_1}, \texttt{y\_at\_0}, \texttt{y\_at\_1}, \texttt{set\_x\_0}, \texttt{set\_x\_1}, \texttt{set\_y\_0}, \texttt{set\_y\_1}\}$ and P_1, P_2, P_3, and P_4 are the respective sets of productions, as shown in Figure 6.

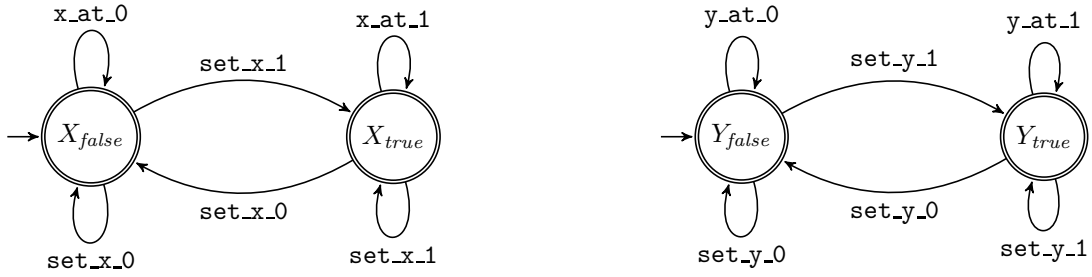
Procedures $p1$ and $p2$ are translated into CFG_1 and CFG_2, respectively. First, we need to encode the control flow of the procedures. For instance, "$p1$ reaches location n_0 and it executes the statement $x =$ not y" is translated into the grammar production $N_0 \rightarrow [\![x =$ not $y]\!]\ N_1$, where N_1 represents the next program

location n_1. We use the notation $[\![s]\!] \in V$ to refer to the corresponding translation of statement s. A function call such as "$p1$ calls itself recursively after location n_1 is executed" is translated through the production $N_1 \to N_0 N_2$ where N_0 is the entry location of the callee function and N_2 is the continuation of the caller after the callee returns.

The non-terminal symbol $[\![x = \mathsf{not}\ y]\!]$ models the execution of negating y and storing its result in x. We create a terminal symbol for each possible action on x (and analogously for y): `x_at_0` (the value of x is 0), `x_at_1` (the value of x is 1), `set_x_0` (x is updated to 0), and `set_x_1` (x is updated to 1). For instance, the grammar production $[\![x = \mathsf{not}\ y]\!] \to S_{p_2}$ `y_at_0 set_x_1` S_{p_2} represents that if we read 0 as the value of y then it must be followed by writing 1 to x. The rest of the logical operations are encoded similarly.

Note that whenever a global variable is read or written we need to consider the synchronization between threads. To this end we define the non-terminal symbol S_{p_2} (S_{p_1}) which loops zero or more times with all possible actions of $p2$ ($p1$): value of x is 0 (value of y is 0), value of x is 1 (value of y is 1), y is updated to 0 (x is updated to 0), and y is updated to 1 (x is updated to 1).

Next, we need to model which are the possible values that x and y can take. For this we use CFG_3 and CFG_4, respectively. Ignoring synchronization, the set of values that x and y can take are indeed expressed by regular automata:



Finally, we need to synchronize x and y by allowing them to loop zero or more times while new values from the other variable can be generated. We use non-terminal symbols (and their productions) S_x and S_y for that.

Once we have obtained the CFGs described in Figure 6 we are ready to ask reachability questions. For this example, we would like to prove that when threads start at n_0 and m_0, respectively, error cannot be reachable simultaneously by both threads. This question can be answered by checking if the intersection of the above CFGs is empty. If the intersection is not empty then COVENANT will return a witness $w \in \Sigma^*$ containing the sequence of reads and writes to x and y. Otherwise, COVENANT will return either "yes" (that is, the program is safe) if the languages of the CFGs are regularly separable or run until resources are exhausted.

We have tested COVENANT with the programs used in [15] and compared with LCEGAR [15]. There are two classes of programs: textbook Erlang programs and several variants of a real Bluetooth driver. The Bluetooth variants labelled W/ Heuri are encoded with an unsound heuristic that permits context switches only at basic block boundaries. We refer readers to Appendix A for a detailed description of the programs as well as the safety properties.

Table 1(a) and Table 1(b) show the times in seconds for both solvers when proving the Erlang programs and the Bluetooth drivers. The symbol ∞ indicates that the solver failed to terminate after 2 hours. We ran LCEGAR using the settings suggested by the authors and tried with the two available initial abstractions: *pseudo-downward closure* (PDC) and *cycle breaking* (CB). For our implementation, we use as the initial abstraction the one that is described by Nederhof [20]. We also tried Σ^* as an initial approximation, but in this case, COVENANT did not converge for any of the programs in a reasonable amount of time.

It is somewhat surprising that all properties were successfully proven by LCEGAR using the initial regular approximation, including Bluetooth instances. The same is true for COVENANT, except for Version 1 which required 12 refinements using the greedy strategy. Nevertheless, these programs show cases in which COVENANT can significantly outperform LCEGAR. Since almost no refinements were required by any of the

Program		COVENANT	LCEGAR	
			PDC	CB
SharedMem	safe	**0.01**	14.37	24.75
Mutex	safe	**0.04**	6.12	0.14
RA	safe	**0.01**	∞	0.39
Modified RA	safe	**0.03**	∞	27.90
TNA	unsafe	**0.01**	0.02	0.25
Banking	unsafe	**0.01**	∞	3.36

(a) Verification of multi-thread Erlang programs

Program		COVENANT	LCEGAR	
			PDC	CB
Version 1	unsafe	**0.84**	19.74	21.04
Version 2	unsafe	**0.25**	5560.00	4852.00
Version 2 w/ Heuri	unsafe	**0.11**	44.68	38.89
Version 3 (1A2S)	unsafe	**0.12**	217.74	217.27
Version 3 (1A2S) w/ Heuri	unsafe	**0.05**	6.68	11.37
Version 3 (2A1S)	safe	**0.27**	4185.00	3981.00

(b) Verification of multi-thread Bluetooth drivers

		COVENANT				LCEGAR	
		Σ^*		[20]		PDC	CB
		Greedy	Gen	Greedy	Gen		
$C_1 \cap C_7$	sat	**0.01** (8)	7.88 (11)	**0.01** (5)	6.20 (8)	∞	–
$C_7 \cap C_1$						0.13 (0)	0.32 (0)
$C_1 \cap C_8$	sat	**0.01** (8)	8.36 (13)	**0.01** (7)	2.22 (9)	20.28 (0)	–
$C_8 \cap C_1$						∞	∞
$C_2 \cap C_3$	sat	**0.01** (10)	9.10 (13)	**0.01** (2)	0.02 (2)	0.03 (0)	**0.01** (0)
$C_3 \cap C_2$						0.03 (0)	**0.01** (0)
$C_2 \cap C_4$	unsat	0.02 (15)	∞	**0.01** (3)	0.80 (3)	**0.01** (1)	**0.01** (0)
$C_4 \cap C_2$						∞	**0.01** (0)
$C_3 \cap C_4$	unsat	**0.01** (11)	∞	**0.01** (2)	0.04 (2)	**0.01** (0)	**0.01** (0)
$C_4 \cap C_3$						**0.01** (0)	**0.01** (0)
$C_5 \cap C_6$	unsat	**0.01** (6)	∞	**0.01** (5)	∞	∞	**0.01** (0)
$C_6 \cap C_5$						∞	**0.01** (0)
$C_5 \cap C_7$	sat	0.04 (14)	∞	**0.02** (11)	∞	∞	–
$C_7 \cap C_5$						0.33 (0)	∞
$C_5 \cap C_8$	sat	**0.01** (7)	2.81 (9)	**0.01** (5)	3.54 (5)	∞	–
$C_8 \cap C_5$						0.04 (0)	∞
$C_6 \cap C_7$	sat	0.04 (14)	∞	**0.02** (11)	∞	∞	–
$C_7 \cap C_6$						0.10 (0)	∞
$C_6 \cap C_8$	sat	**0.01** (8)	2.86 (9)	**0.01** (5)	3.46 (5)	1.21 (0)	–
$C_8 \cap C_6$						∞	∞
$C_7 \cap C_8$	sat	**0.01** (4)	**0.01** (4)	**0.01** (3)	**0.01** (3)	0.70 (0)	–
$C_8 \cap C_7$						∞	–

(c) Interesting/challenging grammars (∞ indicates time-out at 60 sec and "-" a raised exception.)

Table 1: Comparison of COVENANT with LCEGAR, on several classes of context free grammars; times in seconds. (Equal) best times are in bold.

tools, it also suggests that the approximation of all CFGs at once and the use of a regular solver is often a more efficient choice than relying on computing intersection of CFLs and regular languages as LCEGAR does.

6.2. Some Other Interesting Context-Free Languages

The verification instances [15] are in fact all solved with no use of refinement by LCEGAR (and by COVENANT, except for one instance). To explore more interesting cases that exercise the refinement procedures, we have added experiments involving the following languages ($\Sigma = \{\mathtt{a}, \mathtt{b}\}^*$; note that C_5 is $\mathcal{L}(G_1)$ from Section 4 and C_6 is $\mathcal{L}(G_2)$):

$$
\begin{array}{ll}
C_1: & \{ww^R \mid w \in \Sigma^*\} \\
C_2: & \{wcw^R \mid w \in \Sigma^*\} \\
C_3: & \{\mathtt{a}^n\mathtt{c}\mathtt{a}^n \mid n > 0\} \\
C_4: & \{\mathtt{a}^n\mathtt{c}\mathtt{b}^n \mid n > 0\} \\
C_5: & \{ww^R(\mathtt{ab})^+ \mid w \in \Sigma^*\} \\
C_6: & \{ww^R(\mathtt{ba})^+ \mid w \in \Sigma^*\} \\
C_7: & \{w \in \Sigma^* \mid w \text{ has equal numbers of } \mathtt{a}\text{s and } \mathtt{b}\text{s}\} \\
C_8: & \{ww' \mid |w| = |w'|, w \neq w'\}
\end{array}
$$

Table 1(c) shows the pairs of languages whose disjointness can be proven or a counterexample can be found requiring at least one refinement for COVENANT. We ignore pairs of languages which are disjoint but not regularly separable.

We ran COVENANT using two initial abstractions: Σ^* and the more precise one described by Nederhof [20]. For each one, we used our greedy (Greedy) refinement described in Section 3.1 and the complete refinement (Gen) from Section 3.3. We compared again with LCEGAR using its two abstractions PDC and CB. For a given $C_i \cap C_j$ LCEGAR checks first whether $\mathcal{L}(C_i) \cap \mathcal{L}(\alpha(C_j)) = \emptyset$ and then, only if it is not empty a refinement is triggered. Therefore, LCEGAR fixes a priori that the first input grammar will not be abstracted while the second will (that is, the order in which the grammars are given to LCEGAR matters). That is why we test both $C_i \cap C_j$ and $C_j \cap C_i$. In COVENANT the order is irrelevant. We use the format T(R) to indicate that the tool needed R refinements to prove disjointness or to find a counterexample, in T seconds. We set a timeout (∞) of 60 seconds.

Table 1(c) indicates that, generally, the more precise the initial abstraction, the fewer refinements are necessary. This claim was also made in [15] although we were not able to fully confirm it because LCEGAR raised an exception with many of the instances while using CB (denoted by the symbol –). Interestingly, Greedy performs quite well, terminating for all instances. This suggests that Greedy might be a good practical choice in cases where Gen spends too much time computing the generalization of the witnesses.

Regarding LCEGAR either a timeout is reached or the tool can either prove disjointness or find a witness without any refinement except for one instance ($C_2 \cap C_4$). It is worth noticing that even if both tools would start with the same initial abstraction LCEGAR might not refine at all while COVENANT might do. The reason is that LCEGAR does not abstract all the CFLs which forced us try with both pair orderings. On the other hand, this gives some unpredictability to LCEGAR because depending on the ordering, the tool can behave very differently (for example, $(C_2 \cap C_4)$ versus $(C_4 \cap C_2)$).

7. Related Work

7.1. Intersections between Context-Free and Regular Languages

It is a well known result that testing the intersection of a context-free language and a regular language is decidable in polynomial time. This has been applied to the static detection of SQL injection and cross site scripting attacks [16, 23, 24]. These methods construct a set of regular *attack patterns* $\{R_1, \ldots, R_n\}$ representing common dangerous inputs. One then computes a context-free approximation G for each user-supplied input to a vulnerable system (usually a database), and tests whether $\mathcal{L}(G) \cap \mathcal{L}(R_i)$ is non-empty.

7.2. Intersections between Context-Free and Visibly Pushdown Languages

In the context of HTML/XML validation (for example, [17, 18]), the main idea is to check whether a CFG derived from a string variable is well-formed with respect to a Document Type Definition (DTD). For well-formedness, it is important to guarantee certain tag matching, or balance conditions, and because of this, regular languages cannot be used. *Visibly pushdown languages*, or VPLs [1], constitute a suitable intermediate class of languages between regular and deterministic context-free languages. For this reason, VPLs have attracted much attention over the last decade. *Nested word automata* act as recognisers for VPLs.

While VPLs are more expressive than regular languages, they more or less maintain the tractability and robustness of that class. In particular, the class of visibly pushdown languages is closed under intersection, union, complement, concatenation and Kleene star. The subset problem, while undecidable for deterministic context-free languages, is EXPTIME complete for VPLs. Importantly, the intersection of a CFL and a VPL is decidable.

The method proposed in this paper is *incomparable* with methods (such as Minamide and Tozawa's [17]) that rely on the intersection between a CFL and a VPL. There are language pairs (for example $\{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}$ and $\{\mathtt{a}^n\mathtt{b}^{n+1} \mid n \in \mathbb{N}\}$) that are visibly pushdown but at the same time are not regularly separable. On the other hand, our method can reason about nondeterministic context-free languages while VPL-based methods cannot. Of the languages C_1–C_8 in Section 6, only C_4 is a VPL.

7.3. Intersections between Context-Free Languages

Axelsson *et al.* [2] describe a method to check the intersection of bounded CFGs by unrolling the nonterminals of each CFG up to some fixed depth k, using an incremental SAT solver. Each unrolled CFG is symbolically encoded in propositional logic such that the formula is unsatisfiable iff the intersection of the bounded CFGs is empty. If satisfiable then the intersection of the unbounded CFGs is not empty. The main difference with our method is that this method cannot prove emptiness if the formula is unsatisfiable.

7.4. Interpolant Automata

Our refinement procedure can be seen as a generator of *interpolant automata*. To the best of our knowledge, this term was coined by Heizmann *et al.* [11] in the context of computing interpolants for interprocedural verification. This context is completely different from the present paper's. An interpolant automaton as used by Heizmann *et al.* [11] is a nested word automaton generated through an inductive sequence of interpolants from nested words (extracted from an error trace). Thus, the method is not applicable to context-free grammars. Similar to what we do, Heizmann *et al.* [11] generalize the interpolant automaton representing a single counterexample to multiple counterexamples by adding backward transitions. However, they add those transitions only between automaton states that represent the same program location.

8. Conclusions and Future Work

We have presented a CEGAR-based semi-decision procedure for regular separability of context-free languages. We have described two refinement strategies; an inexpensive greedy approach, and a more expensive exhaustive strategy. We have implemented these approaches in a prototype solver, COVENANT. The method outperforms existing methods on a range of verification and language-theoretic instances. The greedy approach often requires more refinement steps, but tends to quickly find witnesses in cases with non-empty intersections; the exhaustive method performs substantially more expensive refinement steps, but can prove separation of some instances not solved by other methods.

The maximum ε-generalization algorithm can become extremely expensive for large witnesses. It could be worthwhile considering whether one can find a cheaper generalization which still ensures completeness. Similarly, it may be possible to develop a specialized intersection algorithm for computing ε-generalizations, rather than relying on the standard regular/context-free intersection algorithm. It would be interesting also to explore algorithms for approximation by visibly pushdown languages. Finally, there is considerable work

to be done on the practical side, to hone the methods in applications in software verification, including the context of cross site scripting attacks.

Acknowledgments

We wish to thank Georgel Calin for providing the test programs and the implementation of LCEGAR. We also thank Pierre Ganty for fruitful discussions about this topic. We acknowledge support of the Australian Research Council through Discovery Project Grant DP140102194.

References

[1] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on the Theory of Computing*, pages 202–211. ACM Publ., 2004.

[2] Roland Axelsson, Keijo Heljanko, and Martin Lange. Analyzing context-free grammars using an incremental SAT solver. In *Automata, Languages and Programming: Proceedings of the 35th International Colloquium*, volume 5126 of *Lecture Notes in Computer Science*, pages 410–422. Springer, 2008.

[3] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73. ACM Publ., 2003.

[4] Janusz A. Brzozowski and Rina S. Cohen. On decompositions of regular events. *Journal of the ACM*, 16(1):132–144, 1969.

[5] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2006.

[6] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[7] Javier Esparza and Peter Rossmanith. An automata approach to some problems on context-free grammars. In C. Freksa, M. Jantzen, and R. Valk, editors, *Foundations of Computer Science: Potential, Theory, Cognition*, volume 1337 of *Lecture Notes in Computer Science*, pages 143–152. Springer, 1997.

[8] Javier Esparza, Peter Rossmanith, and Stefan Schwoon. A uniform framework for problems on context-free grammars. *Bulletin of the EATCS*, 72:169–177, 2000.

[9] Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, and Peter Schachte. Unbounded model-checking with interpolation for regular language constraints. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 277–291. Springer, 2013.

[10] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[11] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 471–482. ACM Publ., 2010.

[12] Pieter Hooimeijer and Westley Weimer. StrSolve: Solving string constraints lazily. *Automated Software Engineering*, 19(4):531–559, 2012.

[13] H. B. Hunt, III. On the decidability of grammar problems. *Journal of the ACM*, 29(2):429–447, 1982.

[14] Nicholas Kidd. Bluetooth protocol. `http://pages.cs.wisc.edu/~kidd/bluetooth`.

[15] Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. Language-theoretic abstraction refinement. In J. de Lara and A. Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 362–376, 2012.

[16] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, pages 432–441. ACM Press, 2005.

[17] Yasuhiko Minamide and Akihiko Tozawa. XML validation for context-free grammars. In N. Kobayashi, editor, *Programming Languages and Systems (APLAS'06)*, volume 4279 of *Lecture Notes in Computer Science*, pages 357–373. Springer, 2006.

[18] Anders Møller and Mathias Schwarz. HTML validation of context-free languages. In M. Hofmann, editor, *Foundations of Software Science and Computational Structures (FOSSACS'11)*, volume 6604 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 2011.

[19] Benedek Nagy. A normal form for regular expressions. In *Supplemental Papers for the Eighth International Conference on Developments in Language Technology*, CDMTCS Research Report Series, pages 53–62. CDMTCS, 2004.

[20] Mark-Jan Nederhof. Regular approximation of CFLs: A grammatical view. In H. Bunt and A. Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, volume 16 of *Text, Speech and Language Technology*, pages 221–241. Springer, 2000.

[21] Shaz Qadeer and Dinghao Wu. KISS: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 14–24. ACM Publ., 2004.

[22] Thomas G. Szymanski and John H. Williams. Non-canonical parsing. In *Conference Record of the 14th Annual Symposium on Switching and Automata Theory*, pages 122–129. IEEE Comp. Soc., 1973.

[23] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, pages 32–41. ACM Publ., 2007.

[24] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 171–180. IEEE Comp. Soc., 2008.

Appendix A. Recursive Multithreaded Programs

Detailed descriptions of the programs used in Section 6's experimental evaluation, as well as their safety properties, can be found in [15, 5, 21]. This appendix is intended as a self-contained short description.

There are two classes of programs: Erlang programs extracted from textbook algorithms and several variants of a real Bluetooth driver implementation. Table A.2 shows the sizes of the programs after each context-free grammar has been normalized, so that all productions are of form $A \to B\ C$, $A \to B$, $A \to \mathsf{a}$, or $A \to \varepsilon$.

- $\#CFGs$: the number of context-free grammars
- $|\Sigma|$ number of terminal symbols
- $|N|$: total number of nonterminal symbols
- $|P|$: total number of grammar productions

Program	$\#CFGs$	$\|\Sigma\|$	$\|N\|$	$\|P\|$	Program	$\#CFGs$	$\|\Sigma\|$	$\|N\|$	$\|P\|$
SharedMem	4	8	138	234	Version 1	7	17	471	804
Mutex	4	22	297	512	Version 2	9	26	1055	1847
RA	2	20	127	205	Version 2 w/ Heuri	9	26	807	1351
Modified RA	5	22	323	530	Version 3 (1A2S)	9	22	746	1292
TNA	3	17	134	204	Version 3 (1A2S) w/ Heuri	8	22	569	938
Banking	3	13	144	244	Version 3 (2A1S)	9	25	1053	1052

Table A.2: Sizes of the programs shown in Table 1(a-b)

Erlang programs. SharedMem is the shared memory program shown in detail in Figure 6. Mutex is an implementation of the Peterson mutual exclusion protocol where two processes try to acquire a lock. The checked property is that at most one process can be in the critical section at any one time. RA is a resource allocator manager that handles "allocate" and "free" requests. We check that the manager cannot allocate more resources to clients than there are currently free resources in the system. Modified RA adds some new functionality to the logic of the resource allocator manager. We check the same property used in RA. TNA is a telephone number analyzer that serves "lookup" and "add number" requests. The property to check is that certain programming errors cannot happen. Finally, Banking is a toy banking application where users can check a balance as well as deposit and withdraw money. We check that deposits and withdrawals of money are done atomically.

Bluetooth driver [14]. This is a simplified implementation of a Windows NT Bluetooth driver and several variants discussed originally by Qadeer and Wu [21]. The driver keeps track of how many threads are executing in the driver. The driver increments (decrements) atomically a counter whenever a thread enters (exits) the driver. Any thread can try to stop the driver at any time, and after that, new threads are not supposed to enter the driver. When the driver checks that no threads are currently executing the driver, a flag is set to true to establish that the driver has been stopped. Other threads must assert this flag is false before they start their work in the driver. There are two dispatch functions that can be executed by the operative system: one that performs I/O in the driver and another to stop the driver. Assuming threads can asynchronously execute both dispatch functions, we check the following race condition: no thread can enter in the driver after the driver has been stopped. Version 1 and Version 2 [21] are two buggy versions of the driver implementation. Version 2 w/ Heuri is an alternative encoding of Version 2 [15] to limit context switches only at basic block boundaries. This makes the verification task easier but it is, in general, unsound as it does not cover all possible behaviours of the driver. Version 3 (2A1S) [5] is a safe version after blocking the counterexample found in Version 2 where one stopper and two adder processes are considered, Version 3 (1A2S) is a buggy version with one adder and two stopper processes, and finally, Version 3 (1A2S) w/ Heuri is an alternative encoding with the unsound heuristics used in Version 2 w/ Heuri.