Theoretical Computer Science 625 (2016) 1–24

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

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

ARTICLE INFO

Article history:
Received 17 December 2014
Received in revised form 14 December 2015
Accepted 12 January 2016
Available online 27 January 2016
Communicated by D. Sannella

Keywords:
Abstraction refinement
Context-free languages
Regular approximation
Separability

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 a class of semi-decision procedures for regular separability of context-free languages, based on counter-example guided abstraction refinement. We propose two effective instances of this approach, one that is complete but relatively expensive, and one that is inexpensive and sound, but for which we do not have a completeness proof. The complete method will prove disjointness whenever the input languages are regularly separable. Both methods will terminate whenever the input languages overlap. We provide an experimental evaluation of these procedures, and demonstrate their practicality on a range of verification and language-theoretic instances.

© 2016 Elsevier B.V. All rights reserved.

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 kinds 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 property 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 L_1 and L_2 are disjoint".

* Corresponding author.
E-mail addresses: gkgange@unimelb.edu.au (G. Gange), jorge.a.navaslaserna@nasa.gov (J.A. Navas), schachte@unimelb.edu.au (P. Schachte), harald@unimelb.edu.au (H. Søndergaard), pstuckey@unimelb.edu.au (P.J. Stuckey).

http://dx.doi.org/10.1016/j.tcs.2016.01.026
0304-3975/© 2016 Elsevier B.V. All rights reserved.

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, as follows:

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 or finite-state automata.)
2. *Verification:* Check whether the intersection of R_1 and R_2 is empty using a decision procedure for regular languages. 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.
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 $i \in \{1, 2\}$. Tighten the approximations by performing the assignments $R_1, R_2 \leftarrow R'_1, R'_2$; then 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 a 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 [16]. The method we propose will also successfully terminate whenever the given languages overlap.

The method has been implemented in the form of a tool called COVENANT [10]. This tool is publicly available at https://github.com/sav-tools/covenant and is, as far as we know, the only publicly available implementation tackling the problem of (soundly) proving separation of context-free grammars.

Contribution The paper rests on regular approximation ideas by Nederhof [23] and we utilise the efficient pre^* algorithm [9] for intersecting (the language of) a context-free grammar with (that of) a finite-state automaton. We propose various ways to systematically "inflate" a word w in the context of a language L, that is, to enlarge $\{w\}$ to a (preferably infinite) superset without overlapping L. Based on such inflation techniques, we propose a novel refinement procedure for a CEGAR-like 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 of COVENANT.

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 describes a CEGAR-based refinement procedure for separating context-free languages by *inflating* counterexamples into regular languages. Section 4 then describes a number of strategies for word inflation. Section 5 provides an example. In Section 6, we construct a proof that the procedure will terminate for any pair of regularly separable or intersecting languages. In Section 7 we place our method in context, comparing with previously proposed refinement techniques. In Section 8 we evaluate the method empirically, comparing COVENANT with the most closely related tool. Section 9 discusses more broadly related work, and Section 10 concludes. An appendix contains a description of the test cases used in the experimental evaluation.

2. Preliminaries

In this section we recall the some basics, including the notion of regular separability. Table 1 gives a glossary of notation used in the paper.

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. We shall be developing algorithms that, conceptually, manipulate formal languages, that is, sets of symbol strings. However, the algorithms in fact manipulate *representations* of languages, such as regular expressions or grammars, or language recognisers, such as finite-state automata. Hence we will be careful to distinguish objects such as automata or grammars, on the one hand, from their *denotations*. We shall use the function symbol $\mathcal{L}$ for the function that, applied to some object X, gives the language denoted/generated/recognised by X. In the case of regular

Table 1
Quick reference to notations used in this paper, and the sections where they are introduced.

Symbol types			
a, b, c	Literal symbols from alphabet Σ	C, L	Context-free languages
R, S	Regular languages	G	Context-free grammars
r	Regular expressions	$\mathcal{L}(x)$	Language denoted by x
e, f	Union-free regular expressions	$X^{\#}$	Approximation of a language X
E, F	Sets of regular expressions	$\overline{X}$	Complement of set X
M	Finite-state automata	$\mathcal{P}(Y)$	Powerset of Y
$\circ, \cdot, \odot$	Concatenation of regular languages, regular expressions and finite-state automata, respectively		

Notation	Description	§
Regular expressions		
Reg_Σ	The set of regular expressions over alphabet Σ	2.1
Exp_Σ	The set of union-free regular expressions over alphabet Σ	2.1
$X \cap_{\mathcal{L}} Y$	The set of *languages* denoted by regular expressions in both X and Y Equivalent to $\{\mathcal{L}(x) \mid x \in X\} \cap \{\mathcal{L}(y) \mid y \in Y\}$	2.1
Refinement; star- and epsilon-inflations		
$A(L)$	(Initial) regular approximation of L	3
$I(w), I(w, L)$	Inflation of w (wrt. language L)	3
$\mathbf{SI}(w)$	The set of star inflations of a word w: a set of regular expressions	4.1
$\mathbf{SI}_L(w)$	The elements of $\mathbf{SI}(w)$ which denote subsets of language L	4.1
$M(w)$	The DFA recognizing word w	4.2
$\mathbf{EI}(M)$	The set of epsilon-inflations of an automaton M: a set of finite-state automata	4.2
$\mathbf{EI}_L(M)$	The elements of $\mathbf{EI}(M)$ which denote subsets of language L	4.2
$\mathsf{Agg}(I)(w)$	Aggregate inflation of a word w over some class of inflations I The union of all languages denoted by elements of $I(w)$	4.3
Language dissection		
$\mathsf{decomp}(R)$	The canonical union-free decomposition of a regular language R	6.1
$\mathbf{D}(e)$	The dissection of a regular expression	6.2
$\mathbf{D}_{\mathcal{L}}(R)$	The dissection of a regular language: $\bigcup\{\mathbf{D}(e) \mid e \in \mathsf{decomp}(R)\}$	6.2

expressions we occasionally break this principle, as is common practice, relying on the context to help distinguish sense from reference. That is, we will allow ourselves to talk about both “the regular expression $\mathtt{a}^*$” and “the language $\mathtt{a}^*$” (rather than “the language $\mathcal{L}(\mathtt{a}^*)$”); the context makes it clear that, in the first use, $\mathtt{a}^*$ stands for itself, while in the second, it *denotes* something (a language).

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. We let ε denote the empty string (of length 0).

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 $r_1 \mid r_2$, $r_1 \cdot r_2$, and r^*, where r, r_1 and r_2 are regular expressions. Here $\mid$ denotes union, $\cdot$ denotes language concatenation, and * is the Kleene star operation. More precisely,[1]

$$\begin{aligned}
\mathcal{L}(x) &= \{x\} \qquad \text{for } x \in \Sigma \\
\mathcal{L}(\emptyset) &= \emptyset \\
\mathcal{L}(\varepsilon) &= \{\varepsilon\} \\
\mathcal{L}(r_1 \mid r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1 \cdot r_2) &= \mathcal{L}(r_1) \circ \mathcal{L}(r_2) \\
\mathcal{L}(r^*) &= \mathcal{L}(r)^*
\end{aligned}$$

where

$$\begin{aligned}
L_1 \circ L_2 &= \{w_1 \cdot w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\} \\
L^* &= \{w_1 \cdot w_2 \cdots w_k \mid k \geq 0, \text{each } w_i \in L\}
\end{aligned}$$

We shall also use $r?$ as a synonym for the regular expression $\varepsilon \mid r$, and r^+ as a synonym for $r \cdot r^*$. Our notation for sets and set operations (including complements and powersets) is standard, see Table 1. We define the *size* of a regular expression to be the number of alphabet and operator symbols occurring in the expression.

As is common, we will often omit $\cdot$, so that juxtaposition of r_1 and r_2 denotes concatenation of the corresponding languages. Given a finite set $E = \{r_1, \ldots, r_k\}$ of regular expressions, we let $\parallel E$ stand for the regular expression $r_1 | \cdots | r_k$ (in particular, $\parallel \emptyset = \emptyset$). We let Reg_Σ denote the set of regular expressions over alphabet Σ.

[1] Note that $\emptyset$ and ε are overloaded; both are regular expressions, but the former is also a language, while the latter is also a string.

A regular expression is *union-free* iff it does not use the union operation. We let Exp_Σ denote the set of union-free regular expressions over alphabet Σ. Union-free regular expressions will play a central role in Sections 6's proof of completeness.

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 the relation δ is a total function, that is, if for all $q \in Q, x \in \Sigma, |\{q' \mid (q, x, q') \in \delta\}| = 1$, 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*. It is *union-free* if it can be expressed as a union-free regular expression. Union-free regular languages are also known as *star-dot regular* languages [4].

The language recognised by automaton M is $\mathcal{L}(M)$. 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,

$$E_1 \cap_{\mathcal{L}} E_2 = \{\mathcal{L}(r_1) \mid r_1 \in E_1, r_2 \in E_2, \mathcal{L}(r_1) = \mathcal{L}(r_2)\}$$

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.

2.2. Regular separability

As our approach uses regular approximations, we cannot hope to prove separation for arbitrary disjoint pairs of context-free languages. Instead we focus on pairs 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 $R_1 \cap R_2 = \emptyset$.

Proposition 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 $L = \{\mathrm{a}^n\mathrm{b}^n \mid n \geq 0\}$. Both L and $\bar{L}$ 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 [16].

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, but without loss of generality, we consider the intersection of just two context-free languages L_1 and L_2. We assume these languages are provided as context-free grammars.

We furthermore assume a decision procedure that returns "no" if $\mathcal{L}(M_1) \cap \mathcal{L}(M_2) = \emptyset$ or returns a witness w if $w \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2) \neq \emptyset$, where M_1 and M_2 are finite-state automata recognising regular languages R_1 and R_2, respectively (that is, $\mathcal{L}(M_1) = R_1$ and $\mathcal{L}(M_2) = R_2$). Moreover, our refinement procedure will require the solving of constraints of the form $M = M_1 \setminus M_2$ where M, M_1 and M_2 are finite-state automata. The interpretation of the constraint is that $\mathcal{L}(M) = \mathcal{L}(M_1) \cap \overline{\mathcal{L}(M_2)}$.

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. $w \notin L_1 \wedge w \notin L_2$

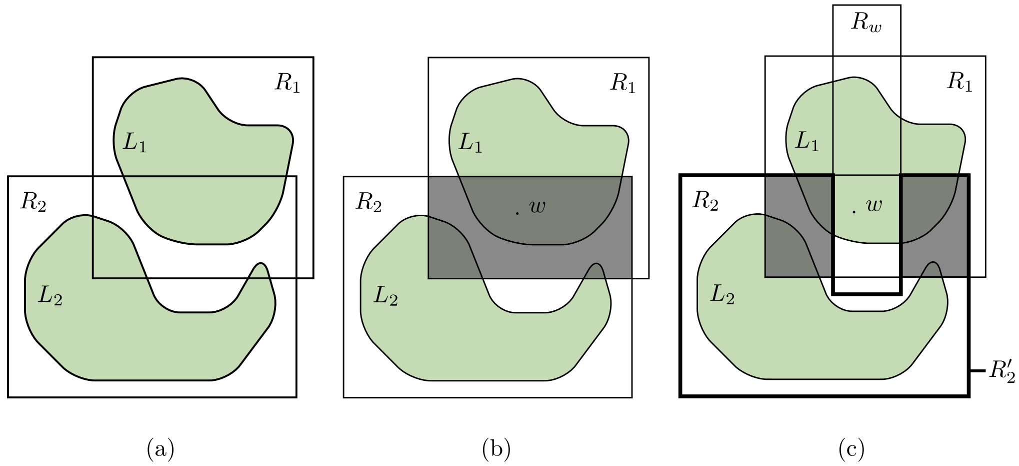

Fig. 1. Refining a regular approximation.

separate$(A, I)(L_1, L_2)$
1: $R_1 := A(L_1)$; $R_2 := A(L_2)$
2: **while**$(R_1 \cap R_2 \neq 0)$
3: $w :=$ choose$(R_1 \cap R_2)$
4: **if** $w \in L_1 \wedge w \in L_2$
5: **return** `non-empty`(w)
6: **if** $w \notin L_1$
7: $R_1 := R_1 \setminus I(w, \overline{L_1})$
8: **if** $w \notin L_2$
9: $R_2 := R_2 \setminus I(w, \overline{L_2})$
10: **return** `empty`(R_1, R_2)

Fig. 2. CEGAR-based refinement procedure for testing intersection of languages L_1 and L_2, given initial approximation strategy A and inflation strategy I. A call $I(w, L)$ *inflates* w into some language R_w such that $w \in R_w$, and $R_w \subseteq L$. A call choose(R) (where $R \neq \emptyset$) returns some word w from R.

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*, or "inflate", a counterexample to an infinite set of words.

The overall flow of the refinement step is illustrated in Fig. 1. Part (a) shows two context-free languages L_1 and L_2, together with their initial regular approximations R_1 and R_2, respectively. (We depict regular languages as rectilinear polygons, to suggest their more limited expressiveness.) In part (b), a counter-example $w \in R_1 \cap R_2$ (the darkly 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 $R'_2 = R_2 \setminus R_w$. The thick contours indicate the extent of this improved regular approximation.

After this, refinement is repeated. Fig. 2 gives the overall algorithm for performing an intersection test by repeated refinement. The procedure is parameterized, allowing for different ways of constructing the initial approximations and, via some inflation strategy, producing successively tighter approximations.

Fig. 2 does not make any assumptions about the choice of witness. Nevertheless, it will be useful to require choose to be somehow well-behaved.

Definition 3 *(Fair selection).* Let choose : $\mathcal{P}(\Sigma^*) \to \Sigma^*$ be a selection strategy and let $R_0 \supset R_1 \supset \cdots$ be a (possibly infinite) strictly decreasing sequence of regular languages, with the property that, for each i, $R_i = \emptyset \vee \text{choose}(R_i) \notin R_{i+1}$. The strategy choose is *fair* iff, for every word $w \in R_0$, we have choose$(R_i) = w$ for some i.

Requiring our witness selection to be *fair* prevents the algorithm from showing certain bias, such as continuously ignoring `b`, given the language `a`$^* \mid$ `b`. Since Σ^* is enumerable, fairness is easy to achieve.

4. Strategies for inflation

We now discuss techniques for inflation. It will be useful to consider different *classes* of inflation. An inflation of a word w is some formal object X denoting a language containing w. A *class* of inflations I is a mapping from words to *finite* sets of inflations.

Fig. 1 is intended to give the idea of inflating some word w as far as possible, while staying within a given co-context-free language (in Fig. 1, $\overline{L_2}$ is that language). Alternatively, think of the inflation as "pushing up" against some context-free language (here L_2) without encroaching on it.

Definition 4 *(Confined inflation).* Given some class of inflations I, the L-confined inflations of w (denoted $I_L(w)$) is the set of inflations of w which denote subsets of L. Formally,

$$I_L(w) = \{r \in I(w) \mid \mathcal{L}(r) \subseteq L\}$$

An inflation $X \in I(w)$ is *maximal* if there is no $X' \in I(w)$ such that $\mathcal{L}(X) \subset \mathcal{L}(X')$.

The first class of inflations we describe, "star-inflations", are easy to understand and will play a role in completeness proofs. The second class, "epsilon-inflations", are what COVENANT actually implements.

4.1. Star-inflation

The refinement procedure that we introduce in this section operates by taking the regular expression w for the single counterexample w, and progressively augmenting it with $*$ operators while ensuring the inflated counterexample remains within the boundary language L.

Definition 5 *(Star-inflation).* The set of star-inflations of a word w is $\mathbf{SI}(w)$, where $\mathbf{SI} : \Sigma^* \to \mathcal{P}(Exp_\Sigma)$, is defined

$$\begin{aligned}\mathbf{SI}(\varepsilon) &= \{\varepsilon\} \\ \mathbf{SI}(x_1 \ldots x_n) &= \{x_1 \ldots x_n, (x_1 \ldots x_n)^*\} \cup \left\{ e_1e_2, (e_1e_2)^* \;\middle|\; \begin{array}{c} e_1 \in \mathbf{SI}(x_1 \ldots x_i), \\ e_2 \in \mathbf{SI}(x_{i+1} \ldots x_n), \\ i \in [1, n-1] \end{array} \right\} \quad (n \geq 0)\end{aligned}$$

Note that, for $x \in \Sigma$, we have $\mathbf{SI}(x) = \{x, x^*\}$. Informally, a star-inflation 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-inflation as a pair $\langle w, S\rangle$ consisting of a word $w = x_1 \ldots x_n$ and a set S of ranges within w covered by $*$-operators. A range is represented by a pair (i, j), with $0 \leq i < j \leq n$. The range (i, j) identifies the substring $x_{i+1} \ldots x_j$ of w. The set S must satisfy

$$\forall (i, j), (i', j') \in S \;.\; j \leq i' \vee j' \leq i \vee (i \leq i' \oplus j \leq j') \tag{1}$$

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 also refer to this kind of range as a *star-augmentation*. We shall use $\mathcal{L}(\langle w, S\rangle)$ to denote the language that results from the inflation $\langle w, S\rangle$.

The set of L-confined star-inflations of w is denoted $\mathbf{SI}_L(w)$. $\mathbf{SI}_L(w)$ may contain several maximal elements. Consider inflating `ab` confined to (`a*b` | `ab*`)—both `a*b` and `ab*` are maximal inflations, and they are incomparable.

A greedy procedure for constructing a star-inflation is given in Fig. 3(a). The algorithm takes as input a witness $w \in R_1 \cap R_2$ and a co-context-free language L such that $w \in L$. It produces a regular over-approximation of $\{w\}$, confined within L. The procedure begins with a trivial star-inflation consisting of just w. The inflation is captured by the set S. The set P holds those (i, j) pairs where a $*$-operation may be introduced without causing the inflation to be malformed. At each step, we add one of the candidate operations to the inflation, then remove any pairs from P which are no longer feasible (because they violate the nestedness requirement). According to (1), this is the set of pairs (i', j') such that $i < i' < j < j' \vee i' < i < j' < j$. Note that $\mathcal{L}(\langle w, S'\rangle) \subseteq L$ (line 8) is decidable, since $\mathcal{L}(\langle w, S'\rangle)$ is regular and L is co-context-free.

It is worth pointing out that the refinement procedure is an *anytime* method: 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 star-augmentations, thereby settling for a smaller inflation.

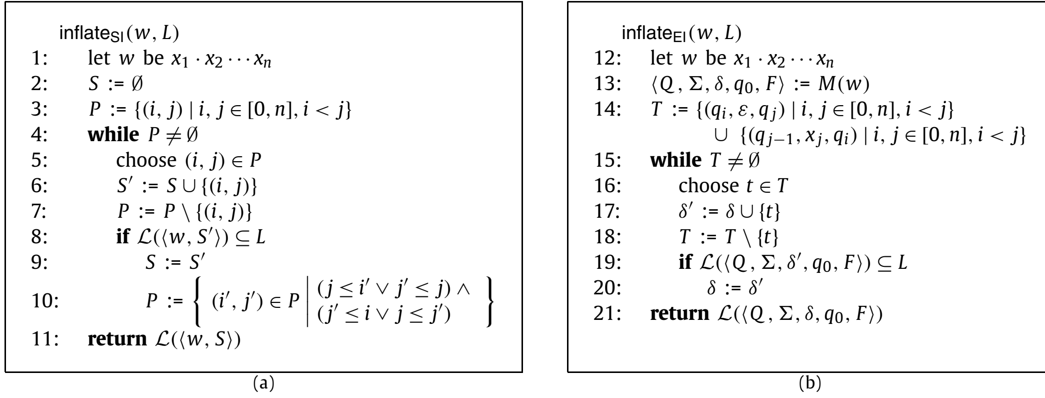

Fig. 3. Algorithm to greedily construct a maximal element of (a) $\mathbf{SI}_L(w)$, and (b) $\mathbf{EI}_L(w)$.

Example 1. Let $L = \{\mathtt{a}^i\mathtt{b}^{i+1} \mid i \geq 0\}$ be approximated (currently) by $\mathtt{a}^*\mathtt{b}^*$, and let the witness $w \in \bar{L}$ be $\mathtt{aab}$. To refine the regular approximation, $\mathsf{inflate}_{\mathsf{SI}}(w, \bar{L})$ begins with the trivial star-inflation $\langle w, \emptyset\rangle$. It greedily augments the counterexample with $*$-operations, in this case following lexicographic order, as follows (we describe the accumulated language using regular expressions):

star-inflation (i, j)	S	$\mathcal{L}(\langle w, S'\rangle)$	decision	result
	$\emptyset$	aab		
$(0, 1)$	$\emptyset$	a*ab	include, obtaining	a*ab
$(1, 2)$	$\{(0, 1)\}$	a*a*b	exclude, as $\mathtt{b} \in L$	
$(2, 3)$	$\{(0, 1)\}$	a*a(b)*	exclude, as $\mathtt{abb} \in L$	
$(0, 2)$	$\{(0, 1)\}$	(a*a)*b	exclude, as $\mathtt{b} \in L$	
$(1, 3)$	$\{(0, 1)\}$	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 resulting language is $(\mathtt{a}^*(\mathtt{ab})^*)^*$ and its complement is $(\mathtt{a}^*\mathtt{ab})^*\mathtt{b}(\mathtt{a} \mid \mathtt{b})^*$. The latter can now be intersected with the initial approximation $\mathtt{a}^*\mathtt{b}^*$, yielding $\mathtt{bb}^* \mid \mathtt{aa}^*\mathtt{bbb}^*$ as a new, improved, regular approximation of L. By construction, the new approximation does not contain aab, but more importantly, along with aab, an infinite number of other strings have been discarded from the previous approximation $\mathtt{a}^*\mathtt{b}^*$, for example, all the strings that start with a and fail to have two consecutive bs.

Refinement (Fig. 2) using Fig. 3(a)'s inflation procedure has these properties:

1. It is *sound*: $L_1 \subseteq (R_1 \setminus \mathsf{inflate}_{\mathsf{SI}}(w, \overline{L_1})) \subseteq R_1$ and $L_2 \subseteq (R_2 \setminus \mathsf{inflate}_{\mathsf{SI}}(w, \overline{L_2})) \subseteq R_2$.
2. It *terminates*: the while loop in $\mathsf{inflate}_{\mathsf{SI}}(w, L)$ 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 $\mathsf{inflate}_{\mathsf{SI}}$.

4.2. Epsilon-inflation

The notion of star-inflation, while useful for reasoning, does not integrate well into existing automaton algorithms—difference and intersection of regular expressions is somewhat inconvenient, and existing incremental algorithms for testing intersection with context-free languages are expressed in terms of automata. In this section, we introduce an alternative form of inflation that is better suited to the representation using automata.

Let $M(w)$ denote the unique ε-free automaton that accepts $w = x_1 \cdots x_n$ and rejects all other words. That is, $M(x_1 \cdots x_n) = \langle Q, \Sigma, \delta, q_0, \{q_n\}\rangle$, with $Q = \{q_0, \ldots, q_n\}$ and $\delta = \{(q_{i-1}, x_i, q_i) \mid i \in [1, n]\}$.

Definition 6 *(Epsilon-inflation).* An epsilon-inflation of w is any language obtained by augmenting the transition relation of $M(w)$ with additional edges $E \cup P$, given by:

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

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

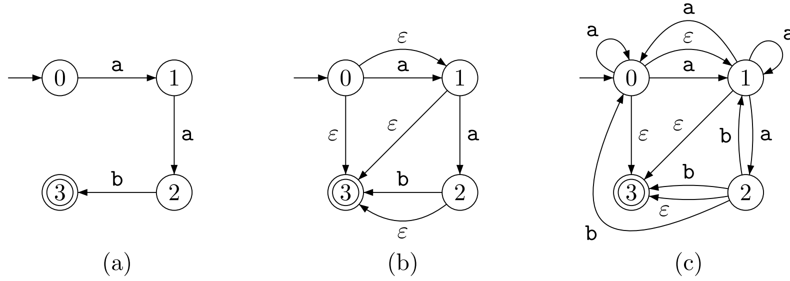

Fig. 4. Construction of $\mathbf{EI}_L(w)$, where $w = \mathtt{aab}$ and $L = (\mathtt{a} \mid \mathtt{b})^* \setminus \{\mathtt{a}^i\mathtt{b}^{i+1} \mid i \geq 0\}$: (a) Initial automaton for w; (b) the automaton after adding forward ε-transitions; these are the edges E from (2); (c) the final automaton after adding non-ε edges; these are the edges P from (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. The role of the symbol-consuming backwards transition (q_{j-1}, x_j, q_i) is to emulate the behaviour of a *non-transitive* ε-transition from (q_j, ε, q_i); that is, the automaton cannot traverse a chain of ε-transitions.

These non-transitive ε-transitions allow us to express more languages without introducing additional states; we cannot reconstruct an automaton for $\mathtt{a}^*\mathtt{b}^*$ from $M(\mathtt{ab})$ using conventional ε-transitions, because we cannot keep the $\mathtt{a}^*$ and $\mathtt{b}^*$ cycles separate.[2]

We let $\mathbf{EI}(w)$ denote the set of epsilon-inflations of w. Note that where $\mathbf{SI}(w)$ is a set of (union-free) regular expressions, $\mathbf{EI}(w)$ is a set of automata. Similarly, $\mathbf{EI}_L(w)$ denotes the set of L-confined epsilon-inflations. Note that, to simplify concatenation, we do not allow backwards transitions from the accept state q_n.

We can incrementally construct an epsilon-inflation confined within some co-context-free language L by adapting algorithms for the intersection of context-free languages and finite automata, such as the *pre** algorithm described by Esparza, Rossmanith and Schwoon [8,9]. See Fig. 3(b). In the implementation of COVENANT, essentially, we maintain a table $\tau \subseteq Q \times P \times Q$ 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 inflation, we update τ with any newly feasible productions; if $\bar{L}$'s start production p_0 is ever generated on a path from q_0 to q_n, the inflation has ceased to be valid—in which case we revert the table and discard the most recent augmentation. The *pre** algorithm has $O(|P||Q|^3)$ 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 $O(|P||Q|^5)$.

Example 2. In Example 1 we were seeking a regular approximation of the language $L = \{\mathtt{a}^i\mathtt{b}^{i+1} \mid i \geq 0\}$. Recall the star-inflation of aab described in that example. If we are instead constructing an epsilon-inflation, we start with the automaton $M(w)$, recognizing $\{w\}$, shown in Fig. 4(a). We then add, in some order, forwards and backwards transitions as defined by (2) and (3). Here let us first add the forwards (epsilon) transitions greedily:

(q_i, ε, q_j)	S	$\mathcal{L}(\langle w, S' \rangle)$	decision	result
	$\emptyset$	aab		
$(0, \varepsilon, 1)$	$\emptyset$	a?ab	include, obtaining	a?ab
$(1, \varepsilon, 2)$	$\{(0,1)\}$	a?a?b	exclude, as $\mathtt{b} \in L$	
$(2, \varepsilon, 3)$	$\{(0,1)\}$	a?ab?	include, obtaining	a?ab?
$(0, \varepsilon, 2)$	$\{(0,1),(2,3)\}$	(a?a)?b?	exclude, as $\mathtt{b} \in L$	
$(1, \varepsilon, 3)$	$\{(0,1),(2,3)\}$	a?(ab?)?	include, obtaining	a?(ab?)?
$(0, \varepsilon, 3)$	$\{(0,1),(2,3),(1,3)\}$	(a?(ab?)?)?	include, obtaining	(a?(ab?)?)?
	$\{(0,1),(2,3),(1,3),(0,3)\}$			

This yields the automaton shown in Fig. 4(b), corresponding to the language $\{\varepsilon, \mathtt{a}, \mathtt{aa}, \mathtt{ab}, \mathtt{aab}\}$. We then progressively introduce backwards transitions as follows:

[2] Using conventional ε-transitions yields a sound inflation strategy which we would expect to display similar asymptotic behaviour, but makes establishing a correspondence between star and epsilon-inflations much more involved (this correspondence will be used in Section 6's completeness proof).

(q_{j-1}, x_j, q_i)	S	$\mathcal{L}(\langle w, S'\rangle)$	decision	result
	$\emptyset$	(a?(ab?)?)?		
(0, a, 0)	$\emptyset$	a*(a?(ab?)?)?	include, obtaining	a*(a?(ab?)?)?
(1, a, 1)	{(0, a, 0)}	a*(a*(ab?)?)?	include, obtaining	a*(a*(ab?)?)?
(2, b, 2)	{(0, a, 0), (1, a, 1)}	a*(a*(ab*)?)?	exclude, as abb $\in L$	
(1, a, 0)	{(0, a, 0), (1, a, 1)}	a*(ab*)?	include, obtaining	a*(ab*)?
(2, b, 1)	{(0, a, 0), (1, a, 1), (1, a, 0)}	a*(a*ab)*(ab)?	include, obtaining	a*(a*ab)*b?
(2, b, 0)	{(0, a, 0), (1, a, 1), (1, a, 0), (2, b, 1)}	a*(a*ab)*(ab)?	include, obtaining	a*(a*ab)*(ab)?
	{(0, a, 0), (1, a, 1), (1, a, 0), (2, b, 1), (2, b, 0)}			

Note that the listed regular expressions recognize the same language as, but do not necessarily directly correspond to, the current 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, b, 0)). The process terminates with the automaton shown in Fig. 4(c). The language recognized by this automaton is (a*ab)*a*, which is also the language obtained with the star-inflation in Example 1.

4.3. Aggregate inflation

The procedures described in the two previous sub-sections construct *some* maximal element of $\mathbf{SI}_L(w)$ (or $\mathbf{EI}_L(w)$). They are, however, undirected; the inflation is chosen blindly from the set of possible maximal inflations.

Even if the input languages are regularly separable, it is possible that the refinement step may choose an infinite sequence of inflations which, though maximal, cannot separate the languages.[3]

We can instead construct a language which covers all possible inflations of w confined within L.

Definition 7 *(Aggregate inflation).* The *aggregate inflation* $\mathsf{Agg}(I)(w)$ over some class of inflations I is the union of all languages denoted by members of I. $\mathsf{Agg}(I)(w)$ is defined as follows:

$$\mathsf{Agg}(I)(w) = \bigcup \{\mathcal{L}(e) \mid e \in I(w)\}$$

We let $\mathsf{inflate}_{\mathsf{Agg(I)}}$ denote the function which takes language L and word w, and returns $\mathsf{Agg}(I_L)(w)$.

Of particular interest will be the aggregate star- and epsilon-inflations confined within some co-context-free language L (that is, $\mathsf{Agg}(\mathbf{SI}_L)(w)$ and $\mathsf{Agg}(\mathbf{EI}_L)(w)$). A possible (though inefficient) method for computing $\mathsf{Agg}(\mathbf{SI}_L)(w)$ is given in Fig. 5(a).

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

An analogous procedure for computing $\mathsf{Agg}(\mathbf{EI}_L)(w)$ is given in Fig. 5(b).

5. A worked example

In this section we work through an example of refinement to prove separability of two context-free languages. We use the greedy epsilon-inflation approach.

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,

$$\begin{array}{lll} S_1 & \to & T_1 U_1 \\ T_1 & \to & \mathtt{aa} \mid \mathtt{bb} \mid \mathtt{a}T_1\mathtt{a} \mid \mathtt{b}T_1\mathtt{b} \\ U_1 & \to & \mathtt{ba}U_1 \mid \mathtt{ba} \end{array} \qquad \begin{array}{lll} S_2 & \to & T_2 U_2 \\ T_2 & \to & \mathtt{aa} \mid \mathtt{bb} \mid \mathtt{a}T_2\mathtt{a} \mid \mathtt{b}T_2\mathtt{b} \\ U_2 & \to & \mathtt{ab}U_2 \mid \mathtt{ab} \end{array}$$

Note that $\mathcal{L}(G_1) = \{ww^R(\mathtt{ba})^+ \mid w \in \Sigma^+\}$ and $\mathcal{L}(G_2) = \{ww^R(\mathtt{ab})^+ \mid w \in \Sigma^+\}$. Let us call them L_1 and L_2, respectively.

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 $L_1 \subseteq \mathcal{L}(A_1)$ and $L_2 \subseteq \mathcal{L}(A_2)$. As a crude starting point, we assume that $\mathcal{L}(A_1) = \mathcal{L}(A_2) = \Sigma^*$.

Next, we check if $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) \neq \emptyset$. In this case, the intersection is trivially non-empty. Furthermore, our regular solver provides the witness $w = \varepsilon$. This cannot be generalized, so we eliminate ε from both approximations, leaving us with $(\mathtt{a} \mid \mathtt{b})^+$ as the approximation of both L_1 and L_2.

[3] Actually the existence of such a sequence remains an open question—we return to this at the end of Section 6.

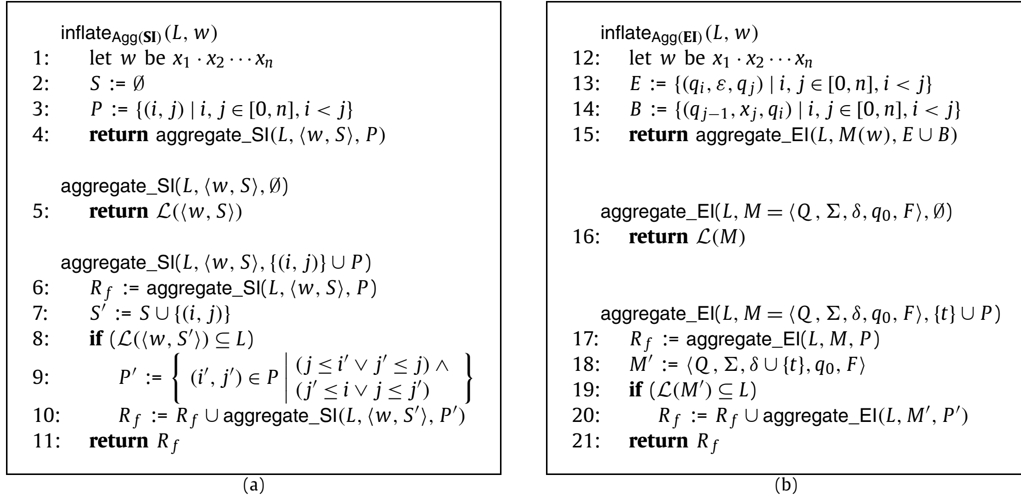

Fig. 5. Computing the aggregate star-inflation ($\mathsf{Agg}(\mathbf{SI}_L)(w)$) and epsilon-inflation ($\mathsf{Agg}(\mathbf{EI}_L)(w)$) of w, confined to language L.

Next, our regular solver provides $w = \mathtt{b}$ as a witness. We assume the use of greedy epsilon-inflation, this time adding backwards transitions before forwards (ε) transitions. We first generalize the witness by calling $\mathsf{inflate}_{\mathsf{EI}}(w, L_1)$ and $\mathsf{inflate}_{\mathsf{EI}}(w, L_2)$ to produce new approximations $\mathcal{L}(A_1') = \mathcal{L}(A_1) \setminus \mathsf{inflate}_{\mathsf{EI}}(w, L_1)$ and $\mathcal{L}(A_2') = \mathcal{L}(A_2) \setminus \mathsf{inflate}_{\mathsf{EI}}(w, L_2)$, respectively. We show the automata obtained from $\mathsf{inflate}_{\mathsf{EI}}(\mathtt{b}, G_1)$ and $\mathsf{inflate}_{\mathsf{EI}}(\mathtt{b}, G_2)$ in row (ii) of Fig. 6. In both cases, we obtain the language $\mathtt{b}^*$. For both L_1 and L_2 we then obtain the improved regular approximation $(\mathtt{a} \mid \mathtt{b})^*\mathtt{a}(\mathtt{a} \mid \mathtt{b})^*$.

Fig. 7 shows the successively tighter approximations generated. After row (ii) we have the same approximation for both L_1 and L_2, given by the automaton shown in row (II) of Fig. 7.

The next witness produced is $w = \mathtt{a}$. Again, for both L_1 and L_2, the witness is generalized identically, to $\mathtt{a}^*$ shown in row (iii) of Fig. 6. Subtracting this language from the current approximations yields the improved approximation for both L_1 and L_2, represented by the automaton in row (III) of Fig. 7. The language is $(\mathtt{a}^+\mathtt{b} \mid \mathtt{b}^+\mathtt{a})(\mathtt{a} \mid \mathtt{b})^*$.

The next witness produced is $w = \mathtt{ab}$—see row (iv) of Fig. 6. This time ε-inflation produces different results for the two context-free languages. The inflated witness for G_1 is the language $(\mathtt{a} \mid \mathtt{b})^*\mathtt{b}$, and for G_2 it is $\varepsilon \mid \mathtt{a}(\mathtt{b}^+\mathtt{a})^*\mathtt{b}^*$. Their complements are $(\mathtt{b}^*\mathtt{a})^*$ and $(\mathtt{b} \mid \mathtt{a}(\mathtt{ba})^*\mathtt{a})(\mathtt{a} \mid \mathtt{b})^*$, respectively. Intersecting these with the current approximation $(\mathtt{a}^+\mathtt{b} \mid \mathtt{b}^+\mathtt{a})(\mathtt{a} \mid \mathtt{b})^*$ (for both G_1 and G_2) produces the languages whose DFAs are shown in row (IV) of Fig. 7. Note that the approximation to L_1 has been tightened to $(\mathtt{a}^*\mathtt{b})^+\mathtt{a}^+$.

At this point we can verify that $L_2 \cap (\mathtt{a}^*\mathtt{b})^+\mathtt{a}^+ = \emptyset$, that is, $(\mathtt{a}^*\mathtt{b})^+\mathtt{a}^+$ is a separating language. Alternatively we can continue the refinement process until the approximations themselves are disjoint, that is, $A_1 \cap A_2 = \emptyset$.

If we do continue, and the next witness is $w = \mathtt{ba}$, then we get the ε-inflations shown in row (v) of Fig. 6. For G_1 the language is $\varepsilon \mid \mathtt{b}(\mathtt{a}^+\mathtt{b})^*\mathtt{a}^*$, whose complement is $(\mathtt{a} \mid \mathtt{b}(\mathtt{ab})^*\mathtt{b})(\mathtt{a} \mid \mathtt{b})^*$. For G_2 the language is $(\mathtt{a} \mid \mathtt{b})^*\mathtt{a}$, whose complement is $(\mathtt{a}^*\mathtt{b})^*$. Intersecting these with the current approximations for L_1 and L_2 (row (IV) of Fig. 7) produces the DFAs shown in row (V). It is easy to see that no word is accepted by both: one requires an input word to end in $\mathtt{a}$, while the other insists that it ends in $\mathtt{b}$. In this case we have proven separation without ever intersecting a regular language with a context-free language.

6. Separating power of the refinement

We now discuss some formal properties of the refinement-based separation process. These are then used to establish the completeness of the refinement-based separation procedure defined in Section 3, assuming one of the "aggregate" variants of inflation is used.

The proof of completeness over regularly-separable languages will proceed in several stages. First, we recapitulate the concept of union-free regular decomposition. We show that every regular language has a *canonical* finite union-free decomposition.

Next, in Section 6.2, we introduce the notion of the *dissection* of a union-free regular language into a finite set of sub-languages; this dissection is needed to establish a well-founded ordering over refinement steps.

We then show that any word in a language R can be star- or epsilon- inflated to cover exactly some element of its dissection.

While each word *can* be inflated to some member of the dissection, we cannot know *which* of the feasible inflations is needed. To ensure completeness, we refine using the union of all feasible inflations. We then prove that each refinement step eliminates at least one element of the dissection, establishing termination of the refinement process.

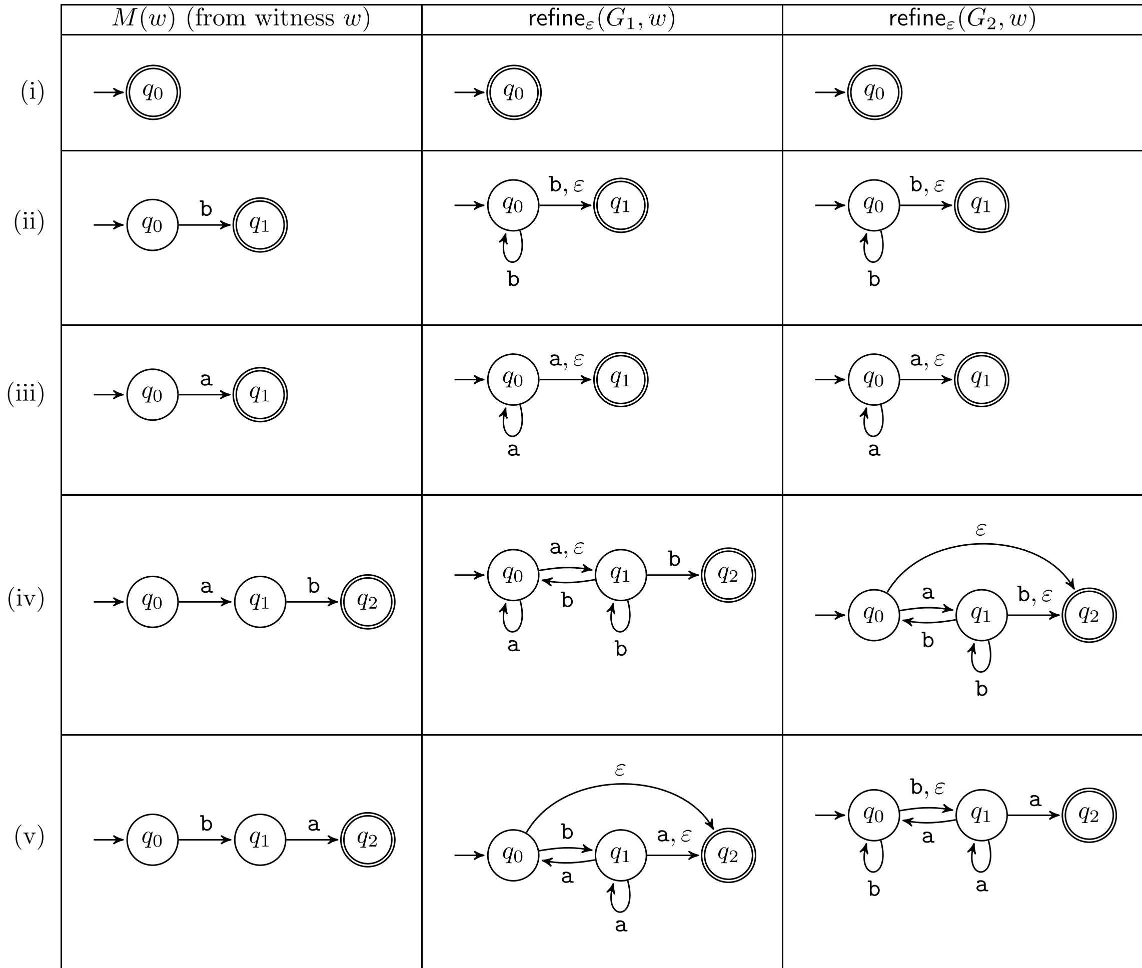

Fig. 6. Witnesses and inflations obtained by greedy epsilon-inflation.

6.1. Union-free regular languages and decomposability

In Section 2.1 we defined union-free regular expressions. It is well-known that a regular language can be expressed as a finite union of union-free expressions.

Definition 8 *(Union-free decomposition).* A union-free decomposition of a regular language R is a finite collection of union-free regular languages $R_1, \ldots, R_n$ such that $R = R_1 \cup \cdots \cup R_n$.

Proposition 2. *(See Nagy [22].) Every regular language R admits a finite union-free decomposition.*

Proposition 2 is not surprising; it utilises the well-known equivalence $(r_1 \mid r_2)^* = (r_1^* r_2^*)^*$.

A regular language potentially admits many union-free decompositions, and a union-free language may be represented by many union-free regular expressions. However, it will be convenient to associate a regular language with a *particular* decomposition.

Let $\Sigma_{exp} = \Sigma \cup \{^*, (,)\}$, so that Σ_{exp}^* is a superset of the well-formed union-free regular expressions. We equip Σ_{exp} with some linear order $<$ (it will make no difference exactly how the symbols are ordered). For non-empty words xs and yt, with $x, y \in \Sigma_{exp}$ and $s, t \in \Sigma_{exp}^*$, we define $xs <_{lex} yt$ iff $x < y \vee (x = y \wedge s <_{lex} t)$. We then obtain a linear ordering $<_{exp}$ on Σ_{exp}^* by defining $s <_{exp} t$ iff $|s| < |t| \vee (|s| = |t| \wedge s <_{lex} t)$. That is, we order the elements of Σ_{exp}^* first by increasing

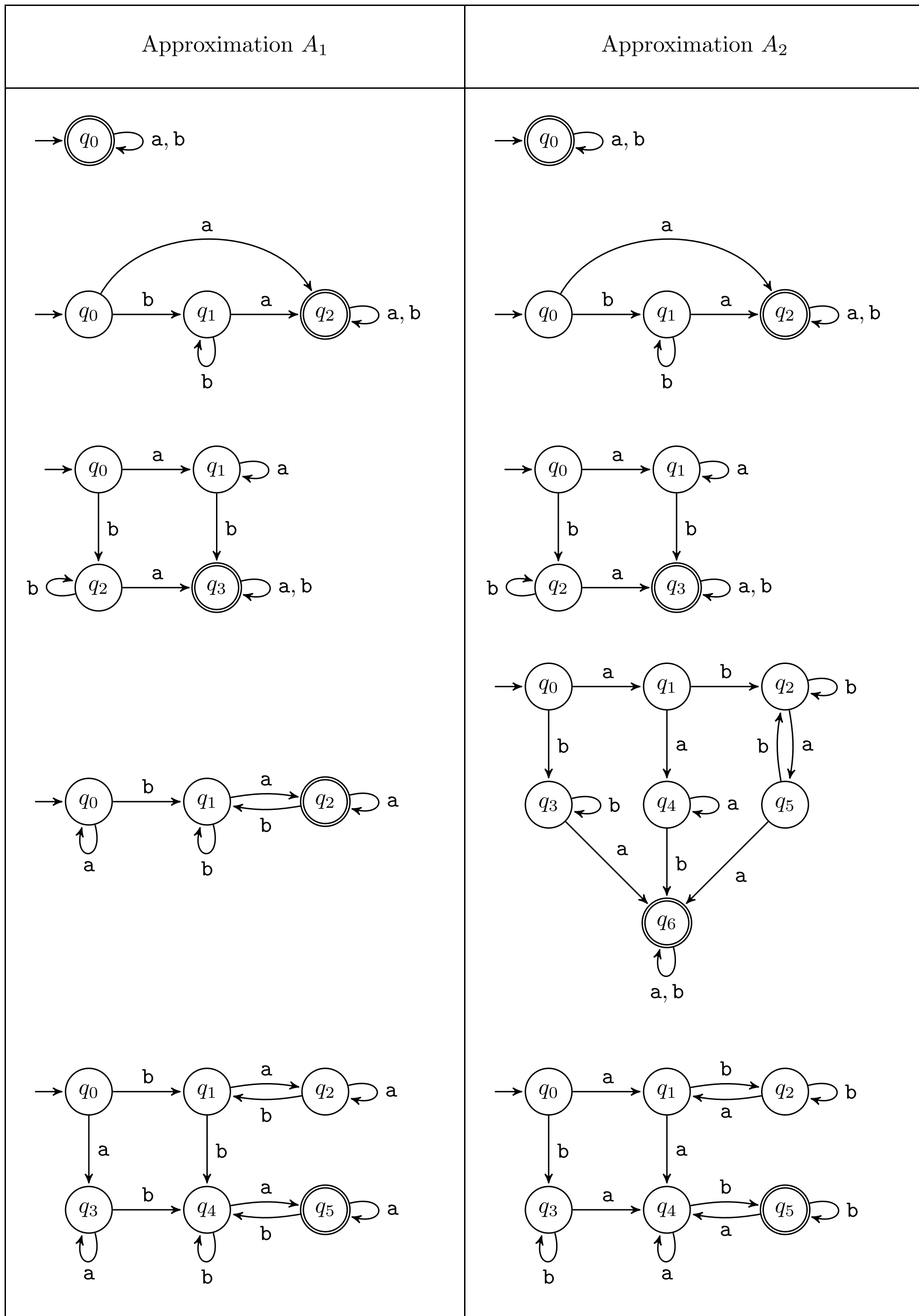

Fig. 7. The successive approximations of L_1 and L_2.

length, then lexicographically.[4] Note that $<_{exp}$ gives a well-founded linear ordering of the set of well-formed union-free expressions. That is, every union-free regular language has a unique smallest union-free expression under $<_{exp}$.

---

[4] The complexity of the definition is necessary. For example, if we were to use simply lexicographic ordering of Σ^*_{exp}, we would obtain infinite descending chains consisting of increasingly longer, but equivalent expressions, such as a*b, a*a*b, a*a*a*b, ...

Given two finite non-empty sequences eE and fF of union-free regular expressions, each sequence sorted in increasing order using $<_{exp}$, we define[5]

$$eE \prec_{lex} fF \text{ iff } e <_{exp} f \vee (e = f \wedge E \prec_{lex} F)$$

Finally, for sets E and F of union-free regular languages we define

$$E \prec F \text{ iff } |E| < |F| \vee (|E| = |F| \wedge E' \prec_{lex} F')$$

where E' is the result of sorting E into increasing order, according to $<_{exp}$, and similarly for F'. That is, we order finite sequences of union-free expressions first by cardinality, then lexicographically (having first sorted the sequences). Again, the ordering $\prec$ provides a well-founded linear ordering, so each regular language R has a unique least union-free decomposition. We let $\mathsf{decomp}(R)$ denote this canonical union-free decomposition.

6.2. Dissection

The following concept is central to the completeness proof. For a given regular language, given by a union-free regular expression, it will be convenient to isolate certain sub-languages. The dissection of the union-free regular expression e is the set of regular expressions obtained by replacing subsets of $*$-enclosed subterms of e with ε.

Definition 9 *(Dissection).* The function $\mathbf{D} : Exp_\Sigma \to \mathcal{P}(Reg_\Sigma)$ is defined as follows:

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

Example 3. Consider the union-free regular expression $\mathtt{ab^*c^*}$. First note that $\mathbf{D}(\mathtt{a}) = \{\mathtt{a}\}$, $\mathbf{D}(\mathtt{b}^*) = \{\varepsilon, \mathtt{b}^*\}$, and $\mathbf{D}(\mathtt{c}^*) = \{\varepsilon, \mathtt{c}^*\}$. Hence the dissection $\mathbf{D}(\mathtt{ab^*c^*}) = \{\mathtt{a}, \mathtt{ab}^*, \mathtt{ac}^*, \mathtt{ab^*c^*}\}$.

For expressions with nested $*$ operators, the dissection allows distinct portions of an inner $*$-expression's subterms to occur when the outer $*$ operation is processed:

Example 4. Continuing from the previous example, consider the union-free regular expression $e = (\mathtt{ab^*c^*})^*$. We saw that the dissection of the parenthesised expression is $\{\mathtt{a}, \mathtt{ab}^*, \mathtt{ac}^*, \mathtt{ab^*c^*}\}$. The dissection of e (after elimination of equivalent languages[6]) is then:

$$\mathbf{D}(e) = \{\varepsilon, \mathtt{a}^*, (\mathtt{ab}^*)^*, (\mathtt{ac}^*)^*, (\mathtt{ab}^* \mid \mathtt{ac}^*)^*, (\mathtt{ab^*c^*})^*\}$$

Note how the elements $r \in \mathbf{D}(e)$ 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 $(\mathtt{ab}^* \mid \mathtt{ac}^*)^*$ is the set of words in which $\mathtt{b}$ and $\mathtt{c}$ are not adjacent.

We will later refer to $\mathbf{D}_\mathcal{L}(R)$, the dissection of a regular *language* R. This is the union of dissections of its canonical decomposition:

$$\mathbf{D}_\mathcal{L}(R) = \bigcup\{\mathbf{D}(e) \mid e \in \mathsf{decomp}(R)\}$$

$\mathbf{D}_\mathcal{L}$ has a number of properties that we will use in the following:

Proposition 3. *For every regular language R,*

1. *$\mathbf{D}_\mathcal{L}(R)$ is a finite set of regular expressions.*
2. *For every regular expression $r \in \mathbf{D}_\mathcal{L}(R)$, $\mathcal{L}(r) \subseteq R$.*
3. *$\bigcup_{r \in \mathbf{D}_\mathcal{L}(R)} \mathcal{L}(r) = R$.*

[5] We could add as a "base case" that $E \prec_{lex} F$ holds when E is the empty sequence and F is not, but it will make no difference, as $\prec_{lex}$ will only be applied to sequences of identical lengths.

[6] The elimination is not needed for this section's proofs; we use it merely to keep examples simpler.

Proof.

1. Let e be a union-free regular expression, with $e \in \mathsf{decomp}(R)$. The syntax-directed nature of $\mathbf{D}$'s definition ensures that, to find $\mathbf{D}(e)$, we only need to apply $\mathbf{D}$ finitely often. It follows that $\mathbf{D}(e)$ is finite, hence $\mathbf{D}_{\mathcal{L}}(R)$ is finite.
2. Let e be a union-free regular expression, with $e \in \mathsf{decomp}(R)$. Consider some $r \in \mathbf{D}(e)$. We show, by structural induction on e, 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, $r \in \mathbf{D}(e_1 \cdot e_2)$ means r is of form $r_1 \cdot r_2$, with $r_1 \in \mathbf{D}(e_1)$ and $r_2 \in \mathbf{D}(e_2)$. 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 $\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \circ \mathcal{L}(r_2) \subseteq \mathcal{L}(e_1) \circ \mathcal{L}(e_2) = \mathcal{L}(e_1 \cdot e_2)$. That is, $\mathcal{L}(r) \subseteq \mathcal{L}(e)$. Finally consider the case $e = f^*$. By definition, $r \in \mathbf{D}(f^*)$ means r is of form $(r_1 \mid r_2 \mid \cdots \mid r_k)^*$, with each $r_i \in \mathbf{D}(f)$. By the induction hypothesis, $\mathcal{L}(r_i) \subseteq \mathcal{L}(f)$. But then $\mathcal{L}(r_1 \mid r_2 \mid \cdots \mid r_k) \subseteq \mathcal{L}(f)^*$. It follows that $\mathcal{L}((r_1 \mid r_2 \mid \cdots \mid r_k)^*) \subseteq \mathcal{L}(f)^*$, as $*$ is monotone and $(L^*)^* = L^*$ for all languages L. That is, $\mathcal{L}(r) = \mathcal{L}((r_1 \mid r_2 \mid \cdots \mid r_k)^*) \subseteq \mathcal{L}(f^*) = \mathcal{L}(e)$.
3. We first show that $\bigcup_{r \in \mathbf{D}_{\mathcal{L}}(R)} \mathcal{L}(r) \subseteq R$. Let $w \in \bigcup_{r \in \mathbf{D}_{\mathcal{L}}(R)} \mathcal{L}(r)$. Then there is some $r \in \mathbf{D}_{\mathcal{L}}(R)$ such that $w \in \mathcal{L}(r)$, and by (2) above, $\mathcal{L}(r) \subseteq R$, so $w \in R$.
   Conversely, let $w \in R$. By the definition of decomposition, there is some $e \in \mathsf{decomp}(R)$ for which $w \in \mathcal{L}(e)$. We want to show that $w \in \mathcal{L}(r)$ for some $r \in \mathbf{D}(e)$, from which the assertion will follow. To do this we proceed by structural induction on e. If $e = x$ for $x \in \Sigma$ then $r = x$ and $w = x$, so indeed $w \in \mathcal{L}(r)$. If $e = \varepsilon$ then $r = \varepsilon$ and $w = \varepsilon$, so indeed $w \in \mathcal{L}(r)$.
   Now consider the case $e = e_1 \cdot e_2$. A word $w \in \mathcal{L}(e)$ can be written as $w = w_1 \cdot w_2$ with $w_1 \in \mathcal{L}(e_1)$ and $w_2 \in \mathcal{L}(e_2)$. By the induction hypothesis, $w_1 \in \mathcal{L}(r_1)$ and $w_2 \in \mathcal{L}(r_2)$ for some $r_1 \in \mathbf{D}(e_1)$ and $r_2 \in \mathbf{D}(e_2)$. By definition of dissection, $r = r_1 \cdot r_2 \in \mathbf{D}(e_1 \cdot e_2) = \mathbf{D}(e)$. So indeed there is some $r \in \mathbf{D}(e)$ for which $w \in \mathcal{L}(r)$.
   Finally consider the case $e = f^*$. A word $w \in \mathcal{L}(e)$ can be written as $w = w_1 w_2 \cdots w_k$ $(k \geq 0)$ with $w_i \in \mathcal{L}(f)$ for all $i \in \{1..k\}$. By the induction hypothesis, $w_i \in \mathcal{L}(r_i)$ for some $r_i \in \mathbf{D}(f)$. Hence $\{r_1, r_2, \ldots, r_k\} \subseteq \mathbf{D}(f)$. By definition of dissection, it follows that $(r_1 \mid r_2 \mid \cdots \mid r_k)^* \in \mathbf{D}(f^*) = \mathbf{D}(e)$. So indeed there is some $r \in \mathbf{D}(e)$ for which $w \in \mathcal{L}(r)$, namely, $w = w_1 w_2 \cdots w_k \in (r_1 \mid r_2 \mid \cdots \mid r_k)^*$. $\square$

6.3. Completeness results

We now show that the inflation procedures $\mathsf{Agg}(\mathbf{SI})$ and $\mathsf{Agg}(\mathbf{EI})$ are sufficiently powerful to prove separability for any pair of regularly separable languages.

The following lemma establishes an important relationship between $\mathbf{D}$ and $\mathbf{SI}$. Namely, given a regular language R, the star-inflation of any word $w \in R$ remains contained in some regular expression resulting from R's dissection. Note that the lemma is concerned with intersecting sets of languages, rather than the languages themselves.

Lemma 1. *Let e be a union-free regular expression. If $w \in \mathcal{L}(e)$ then $\mathbf{D}(e) \cap_{\mathcal{L}} \mathbf{SI}(w) \neq \emptyset$.*

Proof. The proof proceeds by structural induction on e.

Assume $e = x$, with $x \in \Sigma \cup \{\varepsilon\}$. Then $w = x$. From definitions, $x \in \mathbf{D}(e)$, and $x \in \mathbf{SI}(w)$.

Assume $e = e_1 \cdot e_2$, and that the induction hypothesis holds on e_1 and e_2. Consider a 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, $\mathbf{D}(e_1) \cap_{\mathcal{L}} \mathbf{SI}(w_1)$ and $\mathbf{D}(e_2) \cap_{\mathcal{L}} \mathbf{SI}(w_2)$ are non-empty sets of regular expressions. In other words, there is some $f_1 \in \mathbf{SI}(w_1)$ and some $r_1 \in \mathbf{D}(e_1)$ such that $\mathcal{L}(f_1) = \mathcal{L}(r_1) \in \mathbf{D}(e_1) \cap_{\mathcal{L}} \mathbf{SI}(w_1)$; and similarly there is some $f_2 \in \mathbf{SI}(w_2)$ and some $r_2 \in \mathbf{D}(e_2)$ such that $\mathcal{L}(f_2) = \mathcal{L}(r_2) \in \mathbf{D}(e_2) \cap_{\mathcal{L}} \mathbf{SI}(w_2)$. By Definition 5, $f_1 \cdot f_2 \in \mathbf{SI}(w)$. And by Definition 9, $r_1 \cdot r_2 \in \mathbf{D}(e)$. Hence $\mathcal{L}(f_1 \cdot f_2) = \mathcal{L}(r_1 \cdot r_2) \in \mathbf{SI}(w) \cap_{\mathcal{L}} \mathbf{D}(e)$.

Assume $e = (e')^*$, for some e' satisfying the induction hypothesis. Consider some word $w \in \mathcal{L}(e)$. We can partition w into $w_1 \cdots w_k$, with each $w_i \in e'$. By the induction hypothesis, each w_i admits some star-inflation f_i such that $\mathcal{L}(f_i) \in \mathbf{D}(e') \cap_{\mathcal{L}} \mathbf{SI}(w_i)$. Consider the inflation f given by $f = (f_1^* \ldots f_k^*)^*$. By Definition 5, $f \in \mathbf{SI}(w)$. Moreover, f is equivalent to $(f_1 \mid \cdots \mid f_k)^*$, which is in $\mathbf{D}(e)$. Hence $\mathcal{L}(f) \in \mathbf{SI}(w) \cap_{\mathcal{L}} \mathbf{D}(e)$. $\square$

The following theorem then establishes that we can inflate any word $w \in R$ into some member of $\mathbf{D}_{\mathcal{L}}(R)$.

Proposition 4. *Let R be a regular language. If $w \in R$ then $\mathbf{D}_{\mathcal{L}}(R) \cap_{\mathcal{L}} \mathbf{SI}(w) \neq \emptyset$.*

Proof. As noted in Section 6.2, the dissection of a regular language R is computed based on the canonical 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 1, $\mathbf{D}(e) \cap_{\mathcal{L}} \mathbf{SI}(w) \neq \emptyset$. From the definition of $\mathbf{D}_{\mathcal{L}}$, we have $\mathbf{D}(e) \subseteq \mathbf{D}_{\mathcal{L}}(R)$, as $e \in E = \mathsf{decomp}(R)$. As $\mathbf{D}(e) \subseteq \mathbf{D}_{\mathcal{L}}(R)$ and $\mathbf{D}(e) \cap_{\mathcal{L}} \mathbf{SI}(w) \neq \emptyset$, we have $\mathbf{D}_{\mathcal{L}}(R) \cap_{\mathcal{L}} \mathbf{SI}(w) \neq \emptyset$. $\square$

We now wish to establish that every star-inflation admits a corresponding epsilon-inflation. First we describe how to "concatenate" automata.

Definition 10. Let $M_1 = \langle Q_1, \Sigma, \delta_1, q_{10}, \{q_{1n}\}\rangle$ and $M_2 = \langle Q_2, \Sigma, \delta_2, q_{20}, \{q_{2n}\}\rangle$ be (nondeterministic) finite-state automata such that q_{1n} has no outgoing edges. The concatenation automaton

$$M_1 \odot M_2 = \langle (Q_1 \cup Q_2) \setminus \{q_{1n}\}, \Sigma, \delta_1 \cup \delta_2[q_{1n} \mapsto q_{20}], q_{10}, \{q_{2n}\}\rangle.$$

The formulation of $\odot$ given here differs slightly from standard constructions (e.g., [25]), which forbid ε-transitions or make no use of overlapping states in the concatenation.[7] We include the following (straight-forward) proposition for completeness.

Proposition 5. *Let $M_1 = \langle Q_1, \Sigma, \delta_1, q_{10}, \{q_{1n}\}\rangle$ and $M_2 = \langle Q_2, \Sigma, \delta_2, q_{20}, \{q_{2n}\}\rangle$ be (nondeterministic) finite-state automata such that q_{1n} has no outgoing edges. The automaton $M_1 \odot M_2$ recognizes the language $\mathcal{L}(M_1) \circ \mathcal{L}(M_2)$.*

Proof. Let $M = M_1 \circ M_2$. Assume $w \in \mathcal{L}(M_1) \circ \mathcal{L}(M_2)$. Then $w = w_1 \cdot w_2$, for some $w_1 \in \mathcal{L}(M_1)$, $w_2 \in \mathcal{L}(M_2)$. So there is some path from q_{10} to q_{20} matching w_1 in $M_1 \circ M_2$, and a path from q_{20} to q_{2n} matching w_2. Therefore w is accepted by M.

Assume w is accepted by M. 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 M 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 M must remain in Q_2.

Hence we can divide the path through M into a prefix, following transitions exclusively in M_1 and reaching q_{20}, and a suffix from q_{20} to q_{2n} following transitions exclusively in M_2. Therefore, $w \in \mathcal{L}(M_1) \circ \mathcal{L}(M_2)$. □

The next theorem says that for every star-inflation of a word there is an equivalent epsilon-inflation.

Theorem 1. *For every word w and $e \in \mathbf{SI}(w)$, there is some $M \in \mathbf{EI}(w)$ such that $\mathcal{L}(e) = \mathcal{L}(M)$.*

Proof. Consider $e \in \mathbf{SI}(w)$. The proof proceeds by induction on the size k of the expression e.

For $k \leq 1$ we have $e = x$, $x \in \Sigma \cup \{\varepsilon\}$. Here, by definition, $M(x) \in \mathbf{EI}(w)$, and $\mathcal{L}(M(x)) = \mathcal{L}(e)$. Also note that the accept state of $M(x)$ has no outgoing transitions.

For the induction step, assume that for all expressions e' of size at most k, if $e' \in \mathbf{SI}(w)$ there is some $M \in \mathbf{EI}(w)$ such that $\mathcal{L}(M) = \mathcal{L}(e')$, and the accept state of M has no outgoing transitions. Consider some star-inflation $e \in \mathbf{SI}(w)$ of size $k+1$.

Assume $e = e_1 \cdot e_2$, so that $e_1 \cdot e_2 \in \mathbf{SI}(w)$. Then, by Definition 5, there is some partition of w into words w_1 and w_2 (that is, $w = w_1 w_2$) such that $e_1 \in \mathbf{SI}(w_1)$ and $e_2 \in \mathbf{SI}(w_2)$. As e_1 and e_2 have size at most k, the induction hypothesis provides automata $M_1 \in \mathbf{EI}(w_1)$ and $M_2 \in \mathbf{EI}(w_2)$. But then the concatenation automaton $M = M_1 \odot M_2$ is a valid epsilon-inflation. So $M \in \mathbf{EI}(w)$. By Proposition 5, $M = M_1 \odot M_2$ recognizes $\mathcal{L}(M_1) \circ \mathcal{L}(M_2) = \mathcal{L}(e_1) \circ \mathcal{L}(e_2)$. Therefore, $\mathcal{L}(M) = \mathcal{L}(e)$. As the accept state of M_2 had no outgoing transitions, and we have not added any outgoing transitions from it, the accept state of M has no outgoing transitions.

Assume $e = f^*$ for some inflation $f \in \mathbf{SI}(w)$. As e has size $k+1$, f is of size k. By the induction hypothesis there is some automaton $M = \langle Q, \Sigma, \delta, q_0, \{q_n\}\rangle \in \mathbf{EI}(w)$, with $Q = \{x_0, x_1, \ldots, x_n\}$, such that $\mathcal{L}(M) = \mathcal{L}(f)$. We construct a new automaton M' with the following transition relation δ' (assuming $w = x_1 \cdots x_n$):

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

(Here $q_j \xrightarrow{\varepsilon^*} q_n$ says that q_n is in the epsilon-closure of $\{q_j\}$.) Note that the added transitions are of the form permitted by Definition 6. Since M is an epsilon-inflation of w, M' is also a valid epsilon-inflation. As the accept state of M had no outgoing transitions, and we have not added any transitions beginning at q_n, the accept state of M' also has no outgoing transitions. Now consider the language recognized by M'. Assume some word u is in $\mathcal{L}(f^*)$. Then either $u = \varepsilon$, or $u = u_1 \cdots u_m$ such that $u_i \in \mathcal{L}(f) \setminus \{\varepsilon\}$. If $u = \varepsilon$, then u is accepted by M' (since $q_0 \xrightarrow{\varepsilon} q_n$). Otherwise, each u_i is accepted along some path p_i from q_0 to q_k in M, 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 M'; then there must be some path, along which u_i is matched, from q_0 to q_0 in M'. Therefore, u is accepted by M'. Now assume there is some word $u \neq \varepsilon$ recognized by M'. We can partition u into sub-words $u_1 \ldots u_m$ such that the path of each u_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 u_i finishes with an introduced edge, there must be some corresponding path from q_0 to q_n in M. So $u_i \in \mathcal{L}(f)$ for $i < m$. And as the path corresponding to u_m starts at q_0 and does not use any introduced edges, $u_m \in \mathcal{L}(f)$. Therefore, $u \in \mathcal{L}(f^*) = \mathcal{L}(e)$. It follows that $\mathcal{L}(M') = \mathcal{L}(f^*) = \mathcal{L}(e)$.

[7] As ε-transitions are transitive, we run into trouble constructing $M_1 \odot M_2$ when there is an ε-transition out of the accept state of M_1, and into the start state of M_2.

Since we can construct epsilon-inflations for trivial star inflations (size $k = 1$), and epsilon-inflations for a star-inflation of size k can be constructed from the epsilon-inflations of its proper subterms, every element of $\mathbf{SI}(w)$ must correspond to an equivalent element of $\mathbf{EI}(w)$. $\square$

Corollary 1. *For any word w and language L, $\mathsf{Agg}(\mathbf{SI}_L)(w) \subseteq \mathsf{Agg}(\mathbf{EI}_L)(w)$.*

Proof. Let $w' \in \mathsf{Agg}(\mathbf{SI}_L)(w)$. From Definition 7, there is some $e \in \mathbf{SI}_L(w)$ such that $w' \in \mathcal{L}(e)$. From Definition 4, we have $e \in \mathbf{SI}(w)$, and $\mathcal{L}(e) \subseteq L$. By Theorem 1, there is some $M \in \mathbf{EI}(w)$ with $\mathcal{L}(M) = \mathcal{L}(e) \subseteq L$. As $\mathcal{L}(M) \subseteq L$, $M \in \mathbf{EI}_L(w)$ (by Definition 4). As $M \in \mathbf{EI}_L(w)$, we have $\mathcal{L}(M) \subseteq \mathsf{Agg}(\mathbf{EI}_L)(w)$. Since $w' \in \mathcal{L}(e) = \mathcal{L}(M)$, and $\mathcal{L}(M) \subseteq \mathsf{Agg}(\mathbf{EI}_L)(w)$, we have $w' \in \mathsf{Agg}(\mathbf{EI}_L)(w)$.

Thus, any word in $\mathsf{Agg}(\mathbf{SI}_L)(w)$ is also in $\mathsf{Agg}(\mathbf{EI}_L)(w)$, so $\mathsf{Agg}(\mathbf{SI}_L)(w) \subseteq \mathsf{Agg}(\mathbf{EI}_L)(w)$. $\square$

In preparation for the completeness proof, we establish a critical relation between the dissection of a regular language and the inflation of single words from the language (confined within some co-context-free language).

Lemma 2. *Consider a co-context-free language L, and regular language R with $R \subseteq L$. For each word $w \in R$, there is some $r \in \mathbf{D}_{\mathcal{L}}(R)$ such that $\mathcal{L}(r) \subseteq \mathsf{Agg}(\mathbf{SI}_L)(w)$.*

Proof. By Proposition 4, there is some $e \in \mathbf{SI}(w)$ such that $\mathcal{L}(e) \in \mathbf{SI}(w) \cap_{\mathcal{L}} \mathbf{D}_{\mathcal{L}}(R)$. Hence there must be some element $r \in \mathbf{D}_{\mathcal{L}}(R)$ with $\mathcal{L}(r) = \mathcal{L}(e)$. As $r \in \mathbf{D}_{\mathcal{L}}(R)$, we have $\mathcal{L}(r) = \mathcal{L}(e) \subseteq R$.

Recall that $\mathbf{SI}_L(w)$ contains the members of $\mathbf{SI}(w)$ that denote subsets of L. Since $\mathcal{L}(e) \subseteq R \subseteq L$, e denotes a subset of L and so must be in $\mathbf{SI}_L(w)$.

By Definition 7, $\mathsf{Agg}(\mathbf{SI}_L)(w)$ is the union of all languages denoted by members of $\mathbf{SI}_L(w)$. The union-free expression e is in $\mathbf{SI}_L(w)$, so $\mathcal{L}(r) = \mathcal{L}(e) \subseteq \mathsf{Agg}(\mathbf{SI}_L)(w)$. $\square$

We are now ready to establish two main results. First, with the use of the aggregate star- or epsilon-inflation, our CEGAR-based approach will terminate for all pairs of regularly separable context-free languages. Second, under an assumption of fair witness selection, it will also terminate for all pairs of overlapping context-free languages, irrespective of the inflation strategy used.

Theorem 2. *Given a pair of regularly separable context-free languages (L, L') and initial regular approximation strategy A, the algorithms $\mathsf{separate}(A, \mathsf{inflate}_{\mathsf{Agg}(\mathbf{SI})})(L_1, L_2)$ and $\mathsf{separate}(A, \mathsf{inflate}_{\mathsf{Agg}(\mathbf{EI})})(L_1, L_2)$ will construct a separating pair $(S_L, S_{L'})$ in a finite number of steps.*

Proof. Consider some (unknown) regular language S separating L and L'. Let i be the current iteration of the refinement procedure and let D^i denote the elements of $\mathbf{D}_{\mathcal{L}}(S) \cup \mathbf{D}_{\mathcal{L}}(\bar{S})$ that denote languages having non-empty intersections with the current approximation $R^i_L \cap R^i_{L'}$. $\mathbf{D}_{\mathcal{L}}(S)$ and $\mathbf{D}_{\mathcal{L}}(\bar{S})$ are both finite by construction, so every D^i is finite.

Now consider some step i at which $R^i_L \cap R^i_{L'}$ is non-empty. Consider some word $w \in R^i_L \cap R^i_{L'}$. This word must be in exactly one of S and $\bar{S}$; we can assume, without loss of generality, that w is in S. Then, by Lemma 2 there must be some $r \in \mathbf{D}_{\mathcal{L}}(S)$ such that $\mathcal{L}(r) \subseteq \mathsf{Agg}(\mathbf{SI}_{L'})(w)$. By Corollary 1, $\mathsf{Agg}(\mathbf{SI}_{L'})(w) \subseteq \mathsf{Agg}(\mathbf{EI}_{L'})(w)$, so $\mathcal{L}(r) \subseteq \mathsf{Agg}(\mathbf{EI}_{L'})(w)$. As $w \in R^i_L \cap R^i_{L'}$ and $w \in \mathcal{L}(r)$, $\mathcal{L}(r) \cap R^i_L \cap R^i_{L'} \neq \emptyset$. It follows that $r \in D^i$.

Now, $R^{i+1}_{L'} = R^i_{L'} \setminus I$, where $I \in \{\mathsf{inflate}_{\mathsf{Agg}(\mathbf{SI})}(\overline{L'}, w), \mathsf{inflate}_{\mathsf{Agg}(\mathbf{EI})}(\overline{L'}, w)\}$. By the definition of $\mathsf{inflate}_{\mathsf{Agg}(I)}$, $I \in \{\mathsf{Agg}(\mathbf{SI}_{\overline{L'}})(w), \mathsf{Agg}(\mathbf{EI}_{\overline{L'}})(w)\}$. We have $\mathcal{L}(r) \subseteq \mathsf{Agg}(\mathbf{SI}_{\overline{L'}})(w) \subseteq \mathsf{Agg}(\mathbf{EI}_{\overline{L'}})(w)$, so $R^{i+1}_{L'} \cap \mathcal{L}(r) = \emptyset$. Thus $\mathcal{L}(r) \cap R^i_L \cap R^i_{L'} = \emptyset$, so $r \notin D^{i+1}$.

We conclude that $D^{i+1} \subset D^i$. As $D^1, D^2, \ldots$ is a strictly decreasing sequence, and every D^i is finite, the refinement process terminates after finitely many steps. $\square$

Theorem 3. *Let L_1 and L_2 be context-free languages with non-empty intersection L. Given any initial regular approximation scheme A, inflation scheme I and* fair *witness selector* choose, $\mathsf{separate}(A, I)(L_1, L_2)$ *will eventually terminate with some witness $w \in L$.*

Proof. The proof is by contradiction. Assume $\mathsf{separate}(A, I)(L_1, L_2)$ does not terminate. Then the sequence of values of $R_1 \cap R_2$ form an infinite descending chain $S_0, S_1, \ldots$. Let w' be some word in $L = L_1 \cap L_2$. $L_1 \subseteq R_1$ and $L_2 \subseteq R_2$, so $w' \in R_1 \cap R_2 = S_0$. The refinement steps at Fig. 2's lines 7 and 9 remove only words that are in either $\overline{L_1}$ or $\overline{L_2}$; hence $w' \in S_i$ for all i.

As $w' \in L_1 \cap L_2$, the algorithm would terminate if $\mathsf{choose}(S_i) = w'$. Hence the assumption of non-termination means that $\mathsf{choose}(S_i) \neq w'$ for each i. But choose is fair, so no such descending sequence can exist (by the definition of fairness).

It follows that $\mathsf{separate}(A, I)(L_1, L_2)$ must terminate. As L is non-empty, separate cannot prove emptiness, so it must return some word $w \in L_1 \cap L_2$ (not necessarily w'). $\square$

While these theorems establish termination of separate under aggregate star and epsilon-inflations, we have neither established nor disproven corresponding results for the greedy use of $\mathsf{inflate}_{\mathsf{SI}}$ and $\mathsf{inflate}_{\mathsf{EI}}$. The following questions remain open:

Question 1. *Is there some pair* (L_1, L_2) *of regularly separable languages such that one of* $\mathsf{separate}(A, \mathsf{inflate}_{\mathsf{SI}})$ *and* $\mathsf{separate}(A, \mathsf{inflate}_{\mathsf{EI}})(L_1, L_2)$ *does not terminate?*

Question 2. *Is there some deterministic, polynomial-time inflation strategy* I *such that, for all pairs* (L_1, L_2) *of regularly separable context-free languages* $\mathsf{separate}(A, I)(L_1, L_2)$ *terminates?*

7. 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 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. Bouajjani et al. rely on a 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. Bouajjani et al. [3] describe several possible abstractions but no experimental evaluation is provided. Chaki et al. [5] extend [3] by, among other contributions, implementing and evaluating the method. The experimental evaluation of Chaki et al. uses both the *ith-prefix* and *ith-suffix* abstractions. Given language L, the *ith-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 *ith-suffix abstraction* can be defined analogously.

We next provide a theorem about the expressiveness of these abstractions.

Theorem 4. *There exist regularly separable languages that can be shown separate neither by the ith-prefix, nor by the ith-suffix, abstraction.*

Proof. Consider the languages $R_1 = \mathtt{a}^*\mathtt{b}\mathtt{a}^*$ and $R_2 = \mathtt{a}^*\mathtt{c}\mathtt{a}^*$. $R_1 \cap R_2$ 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. [18] is based on a similar refinement framework, but the approach differs radically. They maintain a pair of context-free grammars $G_1^{\#}, G_2^{\#}$ over-approximating the intersection of the original languages. At each refinement step, an *elementary bounded language* B_i is generated from each grammar $G_i^{\#}$. An elementary bounded language is a language of the form $B = w_1^* \cdots w_k^*$, where each w_i is a (finite) word in Σ^*. The refinement ensures $B_i \cap G_i^{\#} \neq \emptyset$, but B_i is not necessarily either an over- or under-approximation of $G_i^{\#}$. They then check whether $I = B_i \cap L_1 \cap L_2$ is empty. This problem is decidable [12], albeit NP-complete [7]. 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 5. *There exist non-regularly-separable languages which can be proven separate by* LCEGAR.

Proof. Consider the languages L and $\bar{L}$, where $L = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}$. These are not regularly separable. Nevertheless, LCEGAR may establish that they do not overlap. Assume initial approximations $G_1^{\#} = L$ and $G_2^{\#} = \bar{L}$. At the first iteration, LCEGAR may choose bounded approximation $B = \mathtt{a}^*\mathtt{b}^*$. It will find $B \cap L \cap \bar{L} = \emptyset$, then update $G_1^{\#} = G_1^{\#} \setminus B = \emptyset$. As $G_1^{\#} = \emptyset$, the refinement process has successfully proven separation. □

Lemma 3. *Let* $B = w_1^* \cdots w_k^*$ *be a bounded regular language over alphabet* Σ. *There is a word* $p \in \Sigma^*$ *which is not a substring of any word in* B.

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

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

Assume there is some word $t = up_1 \ldots p_k v \in B$. The word 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, 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, 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 $\{B_1, \ldots, B_n\}$ (over alphabet Σ) of bounded regular languages, there is a word $p \in \Sigma^*$ which is not a substring of any word in $B_1 \cup \cdots \cup B_n$.*

Proof. By Lemma 3, 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 any word from $B_1 \cup \cdots \cup B_n$. □

Theorem 6. *There exist regularly separable languages for which the* LCEGAR *refinement method cannot prove separability.*

Proof. Consider an LCEGAR process with $L_1 = G_1^\# = (\mathtt{a} \mid \mathtt{b})^*\mathtt{a}$, and $L_2 = G_2^\# = (\mathtt{a} \mid \mathtt{b})^*\mathtt{b}$. 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 $\Sigma^* p \Sigma^* \subseteq (\overline{B_1} \cap \cdots \cap \overline{B_n})$. 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 $p\mathtt{b}$.

It follows that, after any finite sequence of refinement steps, neither A_1' nor A_2' will be empty. Hence the refinement process will never prove the separation of L_1 and L_2. □

From Theorems 2, 5 and 6, we conclude that the classes of languages which can be proven separate by LCEGAR and COVENANT are incomparable.

8. Experimental evaluation

We have implemented the method proposed in this paper in a prototype tool called COVENANT.[8] The tool is implemented in C++ and parameterized by the initial approximation and the refinement procedure. For initial CFL approximation COVENANT uses the method described by Nederhof [23], as well as the coarsest abstraction Σ^* for comparison purposes. For refinement, the tool implements both the greedy epsilon-inflation $\mathsf{inflate}_{\mathbf{EI}}$ and aggregate epsilon-inflation $\mathsf{inflate}_{\mathsf{Agg}\mathbf{EI}}$ (described in Sections 4.2 and 4.3, respectively). COVENANT currently implements only the classical product construction for solving the intersection of regular languages but other regular solvers (for example, [11,15]) can be easily integrated.[9]

To assess the effectiveness of our tool, we have conducted two experiments. First, we used COVENANT to prove 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 2.4 GHz Core i5-M520 with 7.8 Gb memory.

8.1. Safety verification of recursive multi-threaded programs

Pioneering work by Bouajjani et al. [3] has shown 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 [3,5,18]. 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, following the approach of Long et al. [18].

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 [13]. A key

[8] Publicly available at https://github.com/sav-tools/covenant together with all the benchmarks used in this section.

[9] In fact, an initial implementation of COVENANT was tested using REVENANT [11], an efficient regular solver based on bounded model checking with interpolation, though the released version does not incorporate it.

```
x = 0; y = 0;

p1() {
n0:   x = not y ;
n1:   if(*) p1();
n2:   x = not y ;
n3: }

p2() {
m0:   y = not x ;
m1:   if(*) p2();
m2:   y = not x ;
m3: }

if (x and y)
   error();
```

Common

//sequences of read operations

$R \rightarrow$ x_is_0 R | x_is_1 R | y_is_0 R | y_is_1 R | ε

CFG$_1$

// Control flow of thread $p1$

$N_0 \rightarrow [\![x = \text{not } y]\!]\ N_1$

$N_1 \rightarrow N_0\ N_2 \mid N_2$

$N_2 \rightarrow [\![x = \text{not } y]\!]\ N_3$

$N_3 \rightarrow [\![x]\!]$

//encoding of instructions for $p1$

$[\![x = \text{not } y]\!] \rightarrow S_{p_2}$ y_is_0 set_x_1 S_{p_2}
 | S_{p_2} y_is_1 set_x_0 S_{p_2}

$[\![x]\!] \rightarrow R$ x_is_1 R

//synchronization with $p2$'s actions

$S_{p_2} \rightarrow$ x_is_0 S_{p_2}
 | x_is_1 S_{p_2}
 | set_y_0 S_{p_2}
 | set_y_1 S_{p_2} | ε

CFG$_2$

//Control flow of thread $p2$

$M_0 \rightarrow [\![y = \text{not } x]\!]\ M_1$

$M_1 \rightarrow M_0\ M_2 \mid M_2$

$M_2 \rightarrow [\![y = \text{not } x]\!]\ M_3$

$M_3 \rightarrow [\![y]\!]$

//Encoding of instructions for $p2$

$[\![y = \text{not } x]\!] \rightarrow S_{p_1}$ x_is_0 set_y_1 S_{p_1}
 | S_{p_1} x_is_1 set_y_0 S_{p_1}

$[\![y]\!] \rightarrow R$ y_is_1 R

//Synchronization with $p1$'s actions

$S_{p_1} \rightarrow$ y_is_0 S_{p_1}
 | y_is_1 S_{p_1}
 | set_x_0 S_{p_1}
 | set_x_1 S_{p_1} | ε

CFG$_3$

//Modelling variable x

$X_0 \rightarrow$ x_is_0 X_0
 | set_x_0 X_0
 | set_x_1 X_1
 | $S_y\ X_0$ | ε

$X_1 \rightarrow$ x_is_1 X_1
 | set_x_1 X_1
 | set_x_0 X_0
 | $S_y\ X_1$ | ε

//Synchronization with y

$S_y \rightarrow$ y_is_0 S_y
 | set_y_0 S_y
 | y_is_1 S_y
 | set_y_1 S_y | ε

CFG$_4$

//Modelling variable y

$Y_0 \rightarrow$ y_is_0 Y_0
 | set_y_0 Y_0
 | set_y_1 Y_1
 | $S_x\ Y_0$ | ε

$Y_1 \rightarrow$ y_is_1 Y_1
 | set_y_1 Y_1
 | set_y_0 Y_0
 | $S_x\ Y_1$ | ε

//Synchronization with x

$S_x \rightarrow$ x_is_0 S_x
 | set_x_0 S_x
 | x_is_1 S_x
 | set_x_1 S_x | ε

Fig. 8. A concurrent Boolean program (SharedMem) and its translation to CFGs.

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.

The top part of the left-most column of Fig. 8 shows a small program SharedMem [18]. 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 1 simultaneously.

The rest of Fig. 8 describes the corresponding translation to context-free grammars. The four resulting grammars, which we explain shortly, are

$$\text{CFG}_1: \langle \{N_0, N_1, N_2, N_3, [\![x = \text{not } y]\!], [\![x]\!], S_{p_2}\}, \Sigma, P_1, N_0 \rangle$$

$$\text{CFG}_2: \langle \{M_0, M_1, M_2, M_3, [\![y = \text{not } x]\!], [\![y]\!], S_{p_1}\}, \Sigma, P_2, M_0 \rangle$$

$$\text{CFG}_3: \langle \{X_0, X_1, S_x\}, \Sigma, P_3, X_0 \rangle$$

$$\text{CFG}_4: \langle \{Y_0, Y_1, S_y\}, \Sigma, P_4, Y_0 \rangle$$

where $\Sigma = \{\texttt{x_is_0}, \texttt{x_is_1}, \texttt{y_is_0}, \texttt{y_is_1}, \texttt{set_x_0}, \texttt{set_x_1}, \texttt{set_y_0}, \texttt{set_y_1}\}$ and P_1, P_2, P_3, and P_4 are the respective sets of productions, as shown in Fig. 8.

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 $N_0 \rightarrow [\![x = \text{not } y]\!]\ N_1$, 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 $N_1 \rightarrow N_0 N_2$ 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 $[\![x = \text{not } y]\!]$ 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_is_0 (the value of x is 0), x_is_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 $[\![x = \text{not } y]\!] \rightarrow S_{p_2}$ y_is_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 finite-state 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 Fig. 8 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. Indeed COVENANT finds that the intersection is empty for this case. If the intersection was not empty then COVENANT would return a witness $w \in \Sigma^*$ containing a sequence of reads and writes that would lead to the error state being reached. In general, in the absence of a witness, 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 [18] and compared with LCEGAR [18]. 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 2(a) and Table 2(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 used the greedy epsilon inflation ($\mathsf{inflate}_{\mathsf{EI}}$) refinement described in Section 4.2, and as the initial abstraction the one that is described by Nederhof [23]. 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 tools, it also suggests that the approximation of all CFGs at once and the use of a regular solver may be a more efficient choice than relying on computing intersection of CFLs and regular languages as LCEGAR does.

8.2. Some other interesting context-free languages

The verification instances [18] 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 5 and C_6 is $\mathcal{L}(G_2)$):

$C_1 : \{ww^R \mid w \in \Sigma^*\}$

$C_2 : \{wcw^R \mid w \in \Sigma^*\}$

$C_3 : \{\mathtt{a}^n\mathtt{c}\mathtt{a}^n \mid n > 0\}$

$C_4 : \{\mathtt{a}^n\mathtt{c}\mathtt{b}^n \mid n > 0\}$

$C_5 : \{ww^R(\mathtt{ab})^+ \mid w \in \Sigma^*\}$

$C_6 : \{ww^R(\mathtt{ba})^+ \mid w \in \Sigma^*\}$

$C_7 : \{w \in \Sigma^* \mid w \text{ has equal numbers of } \mathtt{a}\text{s and } \mathtt{b}\text{s}\}$

$C_8 : \{ww' \mid |w| = |w'|, w \neq w'\}$

Table 2(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 [23]. From each initial abstraction, we tested disjointness using both the greedy and aggregate epsilon inflations ($\mathsf{inflate}_{\mathsf{EI}}$ and $\mathsf{inflate}_{\mathsf{Agg}(\mathbf{EI})}$). We compared again with LCEGAR using its two abstractions PDC and CB. The implementation of LCEGAR differs slightly from

Table 2
Comparison of COVENANT with LCEGAR, on several classes of context-free grammars; times in seconds. (Equal) best times are in bold.

Program		COVENANT	LCEGAR	
		$\text{inflate}_{\text{EI}}$, [23]	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	
		$\text{inflate}_{\text{EI}}$, [23]	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^*		[23]		PDC	CB
		$\text{inflate}_{\text{EI}}$	$\text{inflate}_{\text{Agg(EI)}}$	$\text{inflate}_{\text{EI}}$	$\text{inflate}_{\text{Agg(EI)}}$		
$C_1 \cap C_7$	sat	**0.01** (8)	2.81 (8)	**0.01** (5)	4.50 (7)	∞	–
$C_7 \cap C_1$						0.13 (0)	0.32 (0)
$C_1 \cap C_8$	sat	**0.01** (8)	0.80 (10)	**0.01** (7)	0.65 (9)	20.28 (0)	–
$C_8 \cap C_1$						∞	∞
$C_2 \cap C_3$	sat	**0.01** (10)	0.02 (6)	**0.01** (2)	**0.01** (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)	**0.01** (2)	**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)	0.02 (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)	0.13 (6)	**0.01** (5)	1.25 (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)	0.12 (6)	**0.01** (5)	1.35 (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.)

the published algorithm, in that it maintains only a single context-free approximation of $L_1 \cap L_2$ with initial approximation $G^\# = L_1 \cap \alpha(L_2)$.[10] The choice of *main* language can have a substantial performance impact. As such, for each pair we evaluate LCEGAR on 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 2(c) indicates that, generally, the more precise the initial abstraction, the fewer refinements are necessary. This claim was also made in [18] 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, $\text{inflate}_{\text{EI}}$ performs quite well, terminating for all

[10] Note that Theorems 5 and 6 still hold on this modified algorithm.

instances. This suggests that $\mathsf{inflate}_{\mathsf{EI}}$ might be a good practical choice in cases where the aggregate inflation spends too much time computing the inflations of witnesses.

In all but one instance ($C_2 \cap C_4$) LCEGAR either terminates without refinement or reaches the timeout. It is worth noticing that even if both tools use the same initial abstraction LCEGAR can prove more languages separate without refinement, as it maintains an exact representation of one language. This does however make LCEGAR somewhat volatile, as its behaviour may vary wildly due to language ordering (for example, $C_2 \cap C_4$ versus $C_4 \cap C_2$).

9. Related work

9.1. 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 [19,26,27]. 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.

9.2. Intersections between context-free and visibly pushdown languages

In the context of HTML/XML validation (for example, [20,21]), 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 [20]) that rely on the intersection between a CFL and a VPL. There are language pairs (for example $\{\mathrm{a}^n\mathrm{b}^n \mid n \in \mathbb{N}\}$ and $\{\mathrm{a}^n\mathrm{b}^{n+1} \mid n \in \mathbb{N}\}$) 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. Neither of the languages L_1 and L_2 that were separated in Section 5 is a VPL and of the languages C_1–C_8 in Section 8, only C_4 is a VPL.

9.3. Intersections between context-free languages

Axelsson et al. [2] describe a method to check the intersection of bounded CFGs by unrolling the non-terminals 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.

9.4. 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. [14] in the context of computing interpolants for interprocedural verification. An interpolant automaton as used by Heizmann et al. [14] 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. [14] 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.

10. 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.

Table 3
Sizes of the programs shown in Table 2(a–b).

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

The aggregate ε-inflation algorithm can become extremely expensive for large witnesses. It could be worthwhile considering whether one can find a cheaper inflation scheme which still ensures completeness. Similarly, it may be possible to develop a specialized intersection algorithm for computing ε-inflations, 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.

Acknowledgements

We wish to thank Pierre Ganty for fruitful discussions, and the anonymous reviewers for their detailed and very constructive feedback; the paper has been greatly improved as a result of these interactions. We also wish to thank Georgel Calin for providing the test programs and the implementation of LCEGAR. Finally we acknowledge support of the Australian Research Council through Linkage Project grant LP140100437.

Appendix A. Recursive multithreaded programs

Detailed descriptions of the programs used in Section 8's experimental evaluation, as well as their safety properties, can be found in [18,5,24]. 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 3 shows the sizes of the programs after each context-free grammar has been normalized, so that all productions are of form $A \to B\ C$, $A \to B$, $A \to \mathtt{a}$, 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

Erlang programs SharedMem is the shared memory program shown in detail in Fig. 8. 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 [17] This is a simplified implementation of a Windows NT Bluetooth driver and several variants discussed originally by Qadeer and Wu [24]. 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 that this flag is false before they start their work in the driver. There are two dispatch functions that can be executed by the operating 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 the driver after the driver has been stopped. Version 1 and Version 2 [24] are two buggy versions of the driver implementation. Version 2 w/Heuri is an alternative encoding of Version 2 [18] 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.

References

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

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

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

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

[5] S. Chaki, E. Clarke, N. Kidd, T. Reps, T. Touili, Verifying concurrent message-passing C programs with recursive calls, in: H. Hermanns, J. Palsberg (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Comput. Sci., vol. 3920, Springer, 2006, pp. 334–349.

[6] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, Helmut Veith, Counterexample-guided abstraction refinement, in: E.A. Emerson, A.P. Sistla (Eds.), Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 1855, Springer, 2000, pp. 154–169.

[7] Javier Esparza, Pierre Ganty, Complexity of pattern-based verification for multithreaded programs, in: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Publ., 2011, pp. 499–510.

[8] Javier Esparza, Peter Rossmanith, An automata approach to some problems on context-free grammars, in: C. Freksa, M. Jantzen, R. Valk (Eds.), Foundations of Computer Science: Potential, Theory, Cognition, in: Lecture Notes in Comput. Sci., vol. 1337, Springer, 1997, pp. 143–152.

[9] Javier Esparza, Peter Rossmanith, Stefan Schwoon, A uniform framework for problems on context-free grammars, Bull. Eur. Assoc. Theor. Comput. Sci. EATCS 72 (2000) 169–177.

[10] Graeme Gange, Jorge Navas, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, A tool for intersecting context-free grammars and its applications, in: K. Havelund, G. Holzmann, R. Joshi (Eds.), NASA Formal Methods: Proceedings of the Seventh International Symposium, in: Lecture Notes in Comput. Sci., vol. 9058, Springer, 2015, pp. 422–428.

[11] Graeme Gange, Jorge A. Navas, Peter J. Stuckey, Harald Søndergaard, Peter Schachte, Unbounded model-checking with interpolation for regular language constraints, in: N. Piterman, S. Smolka (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Comput. Sci., vol. 7795, Springer, 2013, pp. 277–291.

[12] Seymour Ginsburg, Edwin H. Spanier, Bounded Algol-like languages, Trans. Amer. Math. Soc. 113 (1964) 333–368.

[13] Susanne Graf, Hassen Saïdi, Construction of abstract state graphs with PVS, in: O. Grumberg (Ed.), Computer Aided Verification, in: Lecture Notes in Comput. Sci., vol. 1254, Springer, 1997, pp. 72–83.

[14] Matthias Heizmann, Jochen Hoenicke, Andreas Podelski, Nested interpolants, in: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Publ., 2010, pp. 471–482.

[15] Pieter Hooimeijer, Westley Weimer, StrSolve: solving string constraints lazily, Autom. Softw. Eng. 19 (4) (2012) 531–559.

[16] H.B. Hunt III, On the decidability of grammar problems, J. ACM 29 (2) (1982) 429–447.

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

[18] Zhenyue Long, Georgel Calin, Rupak Majumdar, Roland Meyer, Language-theoretic abstraction refinement, in: J. de Lara, A. Zisman (Eds.), Fundamental Approaches to Software Engineering, in: Lecture Notes in Comput. Sci., vol. 7212, 2012, pp. 362–376.

[19] Yasuhiko Minamide, Static approximation of dynamically generated web pages, in: Proceedings of the 14th International Conference on World Wide Web, ACM Publ., 2005, pp. 432–441.

[20] Yasuhiko Minamide, Akihiko Tozawa, XML validation for context-free grammars, in: N. Kobayashi (Ed.), Programming Languages and Systems, in: Lecture Notes in Comput. Sci., vol. 4279, Springer, 2006, pp. 357–373.

[21] Anders Møller, Mathias Schwarz, HTML validation of context-free languages, in: M. Hofmann (Ed.), Foundations of Software Science and Computational Structures, in: Lecture Notes in Comput. Sci., vol. 6604, Springer, 2011, pp. 426–440.

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

[23] Mark-Jan Nederhof, Regular approximation of CFLs: a grammatical view, in: H. Bunt, A. Nijholt (Eds.), Advances in Probabilistic and Other Parsing Technologies, in: Text Speech Lang. Technol., vol. 16, Springer, 2000, pp. 221–241.

[24] Shaz Qadeer, Dinghao Wu, KISS: keep it simple and sequential, in: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, ACM Publ., 2004, pp. 14–24.

[25] Michael Sipser, Introduction to the Theory of Computation. Thomson Course Technology, third edition, 2012.

[26] Gary Wassermann, 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, ACM Publ., 2007, pp. 32–41.

[27] Gary Wassermann, Zhendong Su, Static detection of cross-site scripting vulnerabilities, in: Proceedings of the 30th International Conference on Software Engineering, IEEE Comp. Soc., 2008, pp. 171–180.