A Tool for Intersecting Context-Free Grammars and Its Applications

Graeme Gange[1], Jorge A. Navas[2], Peter Schachte[1], Harald Søndergaard[1], and Peter J. Stuckey[1]

[1] The University of Melbourne, Vic. 3010, Australia
[2] NASA Ames Research Center, Moffett Field, CA 94035, USA

Abstract. This paper describes a tool for intersecting context-free grammars. Since this problem is undecidable the tool follows a refinement-based approach and implements a novel refinement which is complete for regularly separable grammars. We show its effectiveness for safety verification of recursive multi-threaded programs.

1 Introduction

Checking emptiness of intersections of context-free grammars is a well-known undecidable problem. However, the fact that this problem is equivalent to safety verification of recursive multi-threaded programs has kept motivating the design of semi-decision procedures that can still be effective in practice.

In this paper, we describe COVENANT, a tool for checking whether the languages of an arbitrary number of context-free grammars are disjoint and show its role as a component in the analysis of recursive multi-threaded programs. The tool takes a *grammatical* approach [7], in the sense that it is formalized in terms of context-free grammars rather than pushdown automata [1, 2, 10]. It implements a *counter-example guided abstraction refinement (CEGAR)* of regular over-approximations and integrates a *complete* refinement procedure that guarantees termination if the context-free grammars are *regularly separable*.[3] We show its application to safety verification of recursive multi-threaded programs.

To the best of our knowledge, our tool is the only publicly available implementation tackling the problem of intersecting *unbounded* context-free grammars.

2 Approach

The tool discussed in this paper follows the so-called *counter-example guided abstraction refinement (CEGAR)* of regular over-approximations. Without loss of generality, in this presentation we consider the intersection of just two context-free grammars G_1 and G_2. The scheme is based on an initial abstraction which is repeatedly refined until either the languages are proven disjoint, an intersection witness has been found, or resources have been exhausted:

[3] Two context-free grammars G_1 and G_2 are *regularly separable* if there exist two regular languages L_1 and L_2 such that $\mathcal{L}(G_1) \subseteq L_1$, $\mathcal{L}(G_2) \subseteq L_2$ and $L_1 \cap L_2 = \emptyset$.

1. *Abstraction:* compute *regular approximations* R_1 and R_2 such that $\mathcal{L}(G_1) \subseteq \mathcal{L}(R_1)$ and $\mathcal{L}(G_2) \subseteq \mathcal{L}(R_2)$.
2. *Verification:* using a decision procedure for regular languages (and writing $\nmid$ for disjointness), if $\mathcal{L}(R_1) \nmid \mathcal{L}(R_2)$ then $\mathcal{L}(G_1) \nmid \mathcal{L}(G_2)$, so answer "the languages are disjoint." If $w \in \mathcal{L}(R_1) \cap \mathcal{L}(R_2)$, $w \in \mathcal{L}(G_1)$, and $w \in \mathcal{L}(G_2)$ then $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \emptyset$, so answer "the languages are not disjoint" and provide w as a witness. Otherwise, go to step 3.
3. *Refinement:* produce new regular approximations R'_1 and R'_2 such that for each R'_i, $i \in \{1, 2\}$, we have $\mathcal{L}(G_i) \subseteq \mathcal{L}(R'_i) \subseteq \mathcal{L}(R_i)$, and $\mathcal{L}(R'_i) \subset \mathcal{L}(R_i)$ for some i. Update the approximations $R_1 \leftarrow R'_1, R_2 \leftarrow R'_2$, and go to step 2.

Abstraction. Note that a regular approximation always exists for any grammar G since we can use Σ^*, where Σ is the alphabet of G. However, the precision of the initial abstraction often has a significant impact on the convergence of the refinement loop, so non-trivial initial abstractions such as the i^{th}-prefix abstraction [1, 2] and the *downward closure* with a *cycle-breaking heuristic* [7] are more suitable candidates.

Verification. This step assumes a decision procedure that returns "no" if $\mathcal{L}(A_1) \nmid \mathcal{L}(A_2)$ or returns a witness w if $w \in \mathcal{L}(A_1) \cap \mathcal{L}(A_2) \neq \emptyset$, where A_1 and A_2 are finite-state automata recognizing regular languages R_1 and R_2, respectively (that is, $\mathcal{L}(A_1) = R_1$ and $\mathcal{L}(A_2) = R_2$). This can be solved using, for instance, the classical product construction. Note that a different approach would make use of the fact that the class of context-free languages is closed under intersection with regular languages. However, one advantage of our approach is that we are able to leverage the latest advances made in string solving [4, 6, 11].

Refinement. At this point, the regular solver has found some witness w such that $w \in (\mathcal{L}(A_1) \cap \mathcal{L}(A_2))$, but $w \notin (\mathcal{L}(G_1) \cap \mathcal{L}(G_2))$. There are three cases: (1) $w \notin \mathcal{L}(G_1) \wedge w \in \mathcal{L}(G_2)$, (2) $w \notin \mathcal{L}(G_1) \wedge w \notin \mathcal{L}(G_2)$, and (3) $w \in \mathcal{L}(G_1) \wedge w \notin \mathcal{L}(G_2)$. For (1) and (3) we should refine A_1 and A_2, respectively. For (2) we could choose to refine either A_1 or A_2, or both. COVENANT aggressively refines both.

We say a language L is a *safe generalization* of a witness w with respect to a context-free grammar G if (a) $L \supseteq \{w\}$ and (b) $L \nmid \mathcal{L}(G)$. If $w \notin \mathcal{L}(G_i)$ then a straightforward refinement is to produce a new abstraction that recognizes $\mathcal{L}(A_i) \setminus \{w\}$ in place of A_i. However, that refinement process will rarely converge, as it excludes only finitely many examples. Instead, we would like to produce safe generalizations of w containing an infinite number of words, to hasten convergence. For this purpose, our tool implements the concept of *star*-generalization [5]. Informally, a *star-generalization* of a word w is a language that applies the Kleene $*$ operator (*i.e.*, unbounded repetition) to any number of non-overlapping, but possibly nested, subsequences of w, while ensuring the resulting augmented language remains disjoint with the language of G.

3 Covenant

The tool is publicly available at `https://bitbucket.org/jorgenavas/covenant`.

3.1 Design and Implementation Choices

The tool is implemented in C++ and parameterized by the initial approximation, the regular solver, and the refinement procedure.

Abstraction. One advantage of having a grammatical view is that COVENANT can easily leverage the advances made in areas such as speech processing where precise abstraction of context-free grammars into regular grammars is an active topic of research. COVENANT implements the method described by Nederhof [8] for approximating context-free grammars with strongly regular languages.

We say a grammar is *strongly regular* if all productions are of the form: $A \rightarrow B\ w \mid w$ or $A \rightarrow w\ B \mid w$, where $w \in \Sigma^*$ and A, B are nonterminals. The abstraction relies on the following observation: a grammar with productions of the form $A \rightarrow \alpha A \beta$ with both α, β non-empty might not be represented as a strongly regular grammar because α and β might be related through an "unbounded" communication not expressible by regular languages. The abstraction consists of conservatively breaking those unbounded communications in such a way that the grammar becomes regular while preserving the original grammar structure as far as possible. Nederhof [8] also proposed a transformation from strongly regular grammars to finite automata, also implemented in COVENANT.

Regular Solver. COVENANT currently implements only the naive product construction for intersecting finite automata but other regular solvers can easily be plugged in. In fact, an initial implementation of COVENANT was tested using REVENANT [4], an efficient regular solver based on bounded model checking with interpolation, though the released version does not incorporate it.

Refinement. COVENANT implements both the greedy and maximum star-epsilon generalizations of [5]. We explain these by example—[5] has details. Consider witness $w \equiv \mathtt{aab}$ and context-free language $L = \{\mathtt{a}^i\mathtt{b}^{i+1} \mid i \geq 0\}$. The greedy algorithm starts by checking whether $W_1 \equiv \mathtt{a}^*\mathtt{ab} \nmid L$. As the query succeeds, W_1 is a safe generalization of w *wrt*. L so we could stop here. However, we can continue and next ask whether $W_2 \equiv \mathtt{a}^*\mathtt{a}^*\mathtt{b} \nmid L$; but $\mathtt{b} \in L$ so W_2 is discarded. Next, we try whether $W_3 \equiv \mathtt{a}^*\mathtt{ab}^* \nmid L$; but $\mathtt{abb} \in L$ and so W_3 too fails to be a safe generalization. Finally, we query $W_4 \equiv \mathtt{a}^*(\mathtt{ab})^* \nmid L$ and $W_5 \equiv (\mathtt{a}^*(\mathtt{ab})^*)^* \nmid L$ which both succeed. Thus, starting from $\mathtt{aab}$, we can produce the safe generalization $(\mathtt{a}^*(\mathtt{ab})^*)^*$. In fact, COVENANTgenerates three safe generalizations: $W_1 \subseteq W_4 \subseteq W_5$. Although this greedy method is reasonably cheap ($O(|w|^2)$), if resources are scarce it can stop any time, returning the weaker W_1 or W_4.

The maximum star-epsilon version is similar to the greedy one except it will compute the union of all possible safe generalizations without committing to any successful partial generalization. That is, the greedy version started by checking W_1 and since W_1 succeeded the rest of queries were relative to W_1. The non-greedy version will not commit to W_1 but also try other possibilities. For instance, $\mathtt{aa}^*\mathtt{b}$, $(\mathtt{aa})^*\mathtt{b}$, $(\mathtt{aab})^*$, and $\mathtt{a}(\mathtt{ab})^*$ are also safe generalizations from which we can keep generalizing. Although more expensive, it is worth mentioning that this version ensures termination of the CEGAR loop whenever the context-free grammars are regularly separable.

For the implementation, two operations are important: (a) intersection between a context-free grammar and a finite automaton for checking safe generalizations, and (b) automata difference for refining the current abstraction by discarding a safe generalization W of w. For (a), COVENANT uses a modified version of the efficient pre^* algorithm [3] and for (b) it intersects the current abstraction with the complement of the determinization of W. Although determinization of automata can have an exponential size blowup, this behavior is rare; we have not seen it during experiments. Based on our experience, it is also useful to minimize after (b) has been performed, to keep the abstraction small.

3.2 Preprocessing and Output

We require that the input grammars are in the following normal form: $A \rightarrow BC \mid B \mid a \mid \varepsilon$, where A, B, C are nonterminals and $a \in \Sigma^+$, where Σ is the alphabet of the grammar. Any context-free grammar can be converted to this form by a linear increase in terms of the size of the original grammar. COVENANT performs this normalization as a preprocessing step but it does not require any further (more expensive) normalizations, such as Chomsky Normal Form.

If COVENANT proves that the language of the grammars are not disjoint it will return a witness. The user can set the option `--solutions n` to ask the solver for n solutions. The option `--dot` will output the automata resulting from the initial abstraction, each of the safe generalizations, and the final abstractions when emptiness was proven, all in the `dot` language of the `Graphviz` package.

4 Safety Verification of Multithread Programs

Bouajjani *et al.* [1] pioneered safety verification of recursive multi-threaded programs by reduction to checking the intersection of context free languages for emptiness. For lack of space, we refer to [1, 2, 7] for details of the encoding.

We have tested COVENANT and compared with LCEGAR [7] using two classes of programs: textbook Erlang programs and several variants of a real Bluetooth driver. A detailed description of the programs as well as the safety properties can be found in [2, 7, 9]. We ran LCEGAR with the setting provided by the authors and tried with the two available initial abstractions: *pseudo-downward closure* (PDC) and *cycle breaking* (CB). Table 1 shows the results. The symbol ∞ means a timeout expired after 2 hours. We ran COVENANT with the greedy refinement.

5 Related Work

To the best of our knowledge, COVENANT is the first publicly available implementation for intersecting context-free grammars ensuring termination for regularly separable grammars. Several CEGAR approaches have been proposed before. Here, we do not consider the effect of initial approximations, as they do not affect the expressiveness of the refinement loop and are easily interchangeable.

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

(a) Verification of multi-thread Erlang programs

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

(b) Verification of multi-thread Bluetooth drivers

Table 1: Comparison of COVENANT with LCEGAR; times in seconds. All experiments ran on a single core of a 2.4GHz Core i5-M520 with 8GiB of memory.

The first CEGAR approach was proposed in [1] based on the concept of *refinable finite-chain abstraction* which consists of computing the series $(\alpha_i)_{i \geq 1}$ overapproximating the language of a CFG G such that $\mathcal{L}(\alpha_1(G)) \supset \mathcal{L}(\alpha_2(G)) \supset \cdots \supseteq \mathcal{L}(G)$. Several refinable abstractions were described in [1] although no experimental data was provided. Instead, we compare here with the i^{th}*-prefix abstraction*[4] implemented in [2]. In this, $\alpha_i(G)$ is the set of words of G of length less than i, together with the set of prefixes of length i of G. We argue that the refinement implemented in COVENANT is more expressive as it is not hard to find regularly separable languages that cannot be proven so by the i^{th}-prefix abstraction. For instance, with $R_1 \equiv \mathtt{a}^*\mathtt{b}$ and $R_2 \equiv \mathtt{a}^*\mathtt{c}$, we have $R_1 \nmid R_2$, while for every length i, the string $\mathtt{a}^i$ forms a prefix to words in both R_1 and R_2. Therefore the intersection of the two abstractions will always be non-empty.

The LCEGAR method described in [7] is based on a similar refinement framework, but the approach differs radically. LCEGAR maintains a pair of context-free grammars A_1, A_2, over-approximating the intersection of the original languages. At each refinement step, an *elementary bounded language* B_i is generated from each grammar A_i.[5] The refinement ensures $B_i \cap A_i \neq \emptyset$, but B_i is not necessarily either an over- or under-approximation of A_i. After that, $I = B_i \cap L_1 \cap L_2$ is

[4] [2] also implemented the i^{th}*-suffix abstraction* which suffers from same limitations.

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

computed. 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 removing B_i.

Here a comparison between methods is more involved, and we refer to [5] for details. Suffice it to say that the refinements done by LCEGAR and COVENANT are incomparable. That is, there are grammars which are not regularly separable for which LCEGAR can terminate but COVENANT cannot, and there are also grammars which are regularly separable but LCEGAR cannot terminate.

Finally, the verification phase in COVENANT consists of intersecting finite automata, for which efficient solvers are available. Instead, LCEGAR intersects several context-free grammars and a bounded language which, although decidable, is NP-Complete. In our experience, LCEGAR makes a smaller number of refinements than COVENANT, but each refinement in COVENANT is considerably cheaper than in LCEGAR, resulting in the better performance.

6 Conclusions

The main contributions of this paper have been to describe a tool for intersecting context-free grammars, and to show it can be effective for safety verification of recursive multi-threaded programs.

Acknowledgement. Partially funded by the ARC through grant DP140102194.

References

1. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, pages 62–73, 2003.
2. S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *TACAS*, pages 334–349, 2006.
3. J. Esparza, P. Rossmanith, and S. Schwoon. A uniform framework for problems on context-free grammars. *Bulletin of the EATCS*, 72:169–177, 2000.
4. G. Gange, J. A. Navas, P. J. Stuckey, H. Søndergaard, and P. Schachte. Unbounded model-checking with interpolation for regular language constraints. In *TACAS*, pages 277–291, 2013.
5. G. Gange, J. A. Navas, P. J. Stuckey, H. Søndergaard, and P. Schachte. A complete refinement procedure for regular separability of context-free languages, 2014. `http://people.eng.unimelb.edu.au/gkgange/pubs/cfg_preprint.pdf`.
6. P. Hooimeijer and W. Weimer. StrSolve: Solving string constraints lazily. *ASE*, 19(4):531–559, 2012.
7. Z. Long, G. Calin, R. Majumdar, and R. Meyer. Language-theoretic abstraction refinement. In *FASE*, pages 362–376, 2012.
8. M.-J. Nederhof. Regular approximation of CFLs: A grammatical view. In *Advances in Probabilistic and Other Parsing Technologies*, volume 16, pages 221–241. 2000.
9. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI*, pages 14–24, 2004.
10. D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *TACAS*, pages 541–545, 2005.
11. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *ICSTVV*, pages 498–507, 2010.

Appendix: Tool Demonstration

Installation. The installation process relies on `cmake` which allows COVENANT being installed across different platforms[6]. The only requirements are:

- to install the `boost` library
- to install a reasonable modern `C++` compiler version. COVENANT has been tested for `clang++` 3.2 and `g++` 4.8.

Usage. The options for running COVENANT are:

```
covenant [OPTIONS] INPUT
```

where `INPUT` is a text file containing the context-free grammars to be intersected. We will describe the most relevant options as they are needed during this presentation. All options are visible by typing `covenant -h`.

"UNSAT" example. Consider the two contrived regularly-separable context-free languages $L_1 = \{w\mathsf{c}w^R \mid w \in \{\mathsf{a},\mathsf{b}\}^*\}$ and $L_2 = \{\mathsf{a}^n\mathsf{c}\mathsf{b}^n\}$. The content of `INPUT` is the following:

```
;; L1 = { wcw^{R}  | w \in {a,b}^{*} }
( S1  -> [ "c" ];
  S1  -> [ "a" S1 "a", "b" S1 "b" ];
)

;; L2 = { (a^n)c(b^n) | n>0 }
(
  S2  -> [ "a" A2 "b" ];
  A2  -> [ "c" ];
  A2  -> [ S2 ]
)
```

The nonterminal symbol appearing on the left-hand side in the first grammar production is considered the start symbol of the grammar. In the above example the start symbols are `S1` and `S2`, respectively. Terminal symbols must be between double quotes (*e.g.*, "a" and "b"). Each grammar production is of the form `A -> [ ...];` where A is a nonterminal and ... is a sequence of any nonterminal or terminal symbol separated by one or more blanks. We also allow `A -> [ ..., ...];` to express two grammar productions with the same left-hand side `A`. That is, `S1-> [ "a" S1 "a", "b" S1 "b" ];` is syntactic sugar for `S1 -> [ "a" S1 "a"]; S1 ->[ "b" S1 "b" ];`. Any text after `;;` is considered a comment.

If we type `covenant INPUT --gen=max-gen`, where `max-gen` refers to the maximum star-epsilon generalization, we obtain:

[6] COVENANT has been tested only for x86_64.

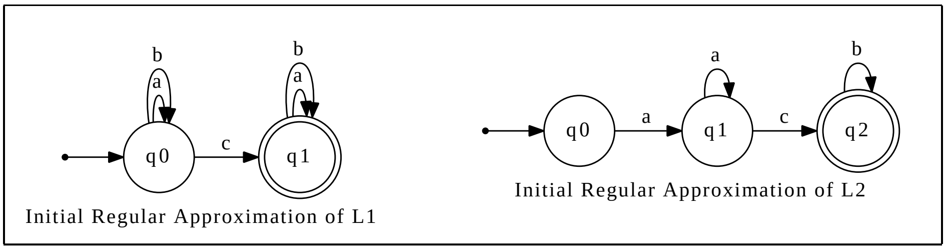

b
a
q0
c
q1
b
a
Initial Regular Approximation of L1
a
b
q0
a
q1
c
q2
Initial Regular Approximation of L2

Fig. 1: `abstractions.dot`: initial regular abstractions

```
Finished after 2 cegar iterations.
======
UNSAT
======
```

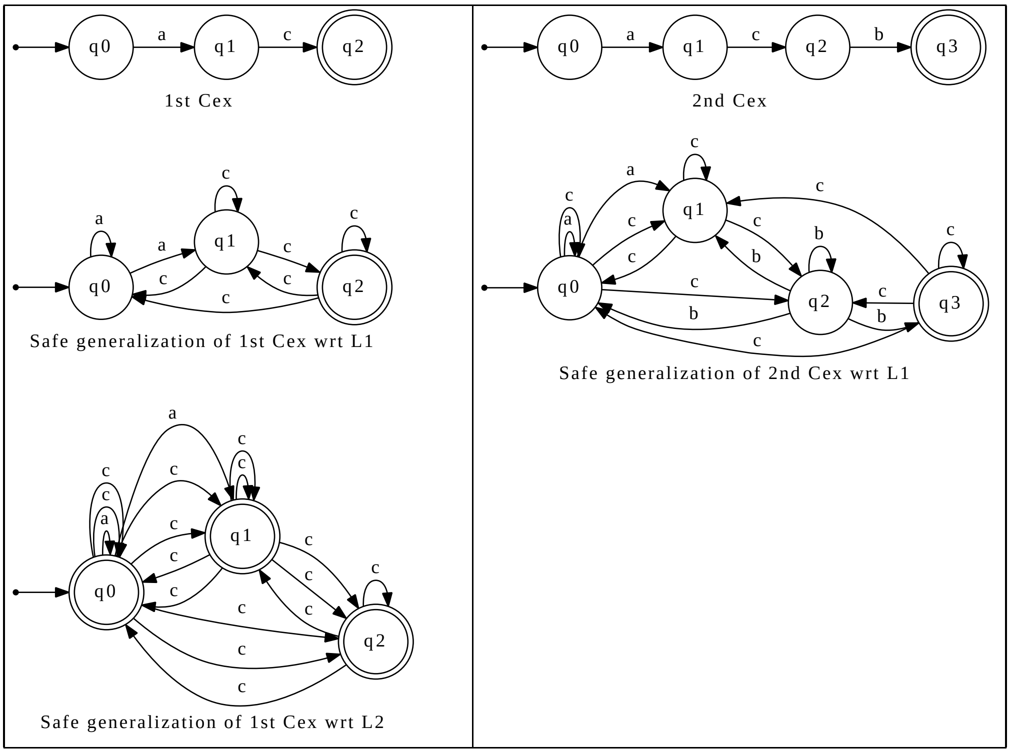
This means that COVENANT proved that the two languages L_1 and L_2 are disjoint. Similar result and number of iterations are obtained if we run with option `--gen=greedy` (for the greedy generalization).

We can also view the `dot` files produced by COVENANT by adding option `--dot`. Figure 1 depicts the initial regular approximations of the context-free grammars by means of finite-state automata. Note that in both cases the structure of the original context-free grammar is preserved as much as possible. For instance, for L_2, the abstraction comes from the need to “forget” the relation between the number of `a`s and `b`s while still remembering that a symbolic `c` must separate the `a`s and `b`s. Each spurious witness found during the refinement loop is also shown as a finite automata as well as their safe generalizations in Figure 2. Note that after the second witness $w \equiv \mathtt{acb}$ is found, COVENANT only needs to compute a safe generalization of w with respect to L_1, since w is already recognized by L_2. Figure 3 shows the final regular approximations which prove that L_1 and L_2 are disjoint.

“SAT” example. Consider now the languages $L_1 = \{w \mid \#\mathtt{a}'s = \#\mathtt{b}'s, w \in \{\mathtt{a},\mathtt{b}\}^*\}$ and $L_2 = \{ww^R \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$ with the corresponding context-free grammars in COVENANT format:

```
;; L1 = { w  | w \in {a,b}^{*}, #a's=#b's }
( S1 -> [];
  S1 -> [ "a" S1 "b" S1];
  S1 -> [ "b" S1 "a" S1]
)

;; L2 = { ww^{R} | w \in {a,b}^{*} }
( S2 -> ["a" "a", "b" "b", "a" S2 "a", "b" S2 "b"])
```



1st Cex

2nd Cex

Safe generalization of 1st Cex wrt L1

Safe generalization of 2nd Cex wrt L1

Safe generalization of 1st Cex wrt L2

Fig. 2: `refinements.dot`: witnesses and safe generalizations

By typing `covenant INPUT`, we obtain:

```
Found a solution after 5 iterations:
a b b a
======
SAT
======
```

That is, COVENANT found a word `abba` that is recognized by both languages. Moreover, we can ask COVENANT for more solutions, say, five more solutions, by typing `covenant INPUT --solutions 5`:

```
Found a solution after 5 iterations:
a b b a
Found a solution after 6 iterations:
b a a b
Found a solution after 20 iterations:
a a b b b b a a
```

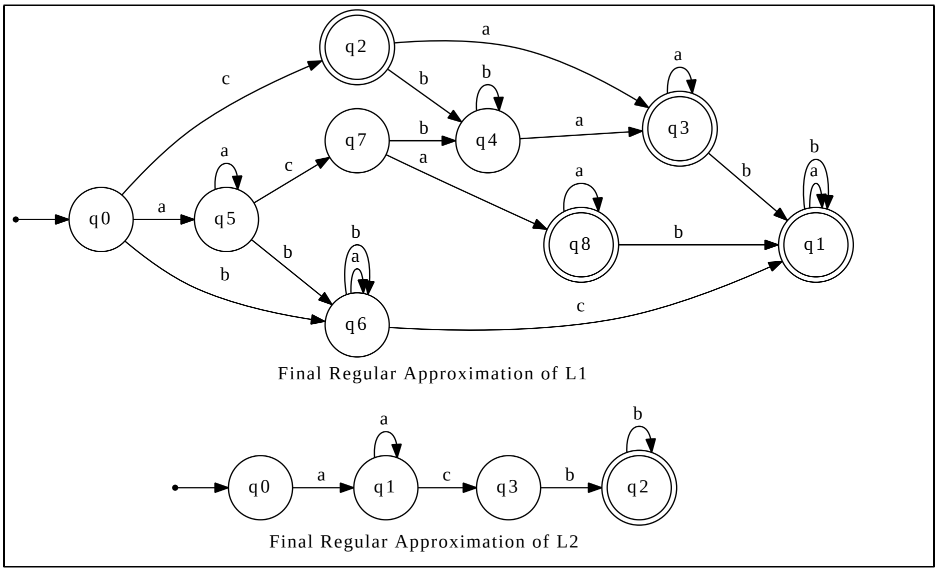

q0
q1
q2
q3
q4
q5
q6
q7
q8
a
b
c
Final Regular Approximation of L1
q0
q1
q3
q2
a
c
b
Final Regular Approximation of L2

Fig. 3: `proofs.dot`: final regular abstractions that prove emptiness

```
Found a solution after 22 iterations:
a b a b b a b a
Found a solution after 23 iterations:
a b b a a b b a
======
SAT
======
```