IT 11 062
Examensarbete 30 hp
Augusti 2011

Approximating the Shuffle of Context-free Languages to Find Bugs in Concurrent Recursive Programs

Jari Stenman

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Approximating the Shuffle of Context-free Languages to Find Bugs in Concurrent Recursive Programs

Jari Stenman

Concurrent programming in traditional imperative languages is hard. The huge number of possible thread interleavings makes it difficult to find and correct bugs. We consider the reachability problem for concurrent recursive programs, which, in general, is undecidable. These programs have a natural model in systems of pushdown automata, which recognize context-free languages. We underapproximate the shuffle of context-free languages corresponding to single threads. The shuffle of two languages is the language you get by taking every pair of words in their cross-product and interleaving them in a way that preserves the order within the original words. We intersect this language with the language of erroneous runs, and get a context-free language. If this language is nonempty, the concurrent program contains an error. We implement a prototype tool using this technique, and use it to find errors in some example program, including a Windows NT Bluetooth driver. We believe that our approach complements context-bounded model checking, which finds errors only up to a certain number of context switches.

Handledare: Mohamed Faouzi Atig
Ämnesgranskare: Parosh Abdulla
Examinator: Anders Jansson
IT 11 062
Tryckt av: Reprocentralen ITC

Contents

List of Figures

1 Introduction

It is widely acknowledged that concurrent programming in traditional sequential programming languages is hard. Programmers' disposition towards sequential thinking, coupled with the huge amount of potential interleaving runs in a concurrent program, lead to errors that are very difficult to find and fix. At the same time, concurrent programming is becoming more and more important, since the number of cores in our devices is increasing.

We are seeing extensive research to counter the difficulties associated with concurrent programming. This research is done in many different areas, including programming language design, compiler design, testing and verification. Our focus is on the verification part; to ensure high-quality concurrent software, we need efficient bug-finding and verification tools.

We are interested in checking safety properties for concurrent recursive programs. This reduces to the control-point reachability problem, which is undecidable even for boolean programs (i.e. where all variables have a boolean domain) [11]. Therefore, we cannot formally verify the absence of errors, but we can still have useful procedures that can detect some errors. One approach to this problem is called *context-bounded model checking* [9]. Context-bounded model checking explores all possible interleaving runs up to a constant number k context-switches. The technique is sound and precise up to the bound, meaning that all errors in the first k context-switches are detected, and all reported errors are real errors. The intuition is that the majority of errors manifest within a small number of context-switches [7].

We are trying to complement context-bounded model checking with a technique that detects errors regardless of the number of context-switches. Instead of bounding the number of context-switches, we bound the granularity of the approximation of interleaving runs. This technique is based on grammars. The idea is that we take a program consisting of several threads, and generate a formal grammar which produces an approximation of all possible interleaving runs of the threads. We can then check if this grammar includes properties that we want to avoid.

The rest of this report is structured as follows. In section 2, we give the necessary theoretical background. In sections 3 and 4, we describe how the technique works. Section 5 contains a short description of the implementation. In section 6, we demonstrate the technique on 3 example programs with varying characteristics. We discuss the results in section 7. Finally, section 8 discusses

some related work.

2 Background

To get started, we need to define some basic notions of automata and formal language theory.

Definition 1 (Alphabets, words and languages)**.** *An **alphabet** is a finite set of symbols. A **word** over an alphabet is a finite sequence of symbols from that alphabet. The **length** $|w|$ of a word w is the number of symbols in it. We denote the empty word (which has length 0) by ϵ. The **concatenation** of two words w_1 and w_2 is the word w_1w_2. We have, for all words w, that $w\epsilon = \epsilon w = w$. The n**-th power** of a word w is the word $\underbrace{w...w}_{n \text{ times}}$. A **language** is a set of words over the same alphabet. If Σ is an alphabet, the language denoted by Σ^* is the set of all words over Σ.*

For example, let $\Sigma = \{0, 1\}$ be an alphabet. Then $\Sigma^* = \{\epsilon, 0, 1, 00, 11, 01, 10, ...\}$. Note that for any alphabet Σ, we have $\epsilon \in \Sigma^*$. When we take the n-th power of a word $w = a_1...a_k$, we write a_1^n when $k = 1$ and $(a_1...a_k)^n$ when $k > 1$. The parantheses show which part is to be repeated; they are not symbols. For example, $ab^2 = abb$, but $(ab)^2 = abab$. For any word w, $w^0 = \epsilon$.

We are now going to introduce the formal constructions that we are going to use. First, we define a simple model of computation called *nondeterministic finite automata*. Then, we will define two equivalent models of computation that are more powerful; *pushdown automata* and *context-free grammars*.

2.1 Nondeterministic finite automata

Nondeterministic finite automata are a class of very simple abstract machines. They are made up by a states, one of them marked initial and some of them marked final, and labelled transitions between these states. An automaton begins in a unique initial state and moves to other states according to its transitions. When it takes a transition, it reads the symbol that labels that transition. When the automaton ends up in some final state, it may either *accept* the sequence of symbols it read to get to that state, or it may continue reading symbols. Any nondeterministic finite automaton has a corresponding language; the set of words it accepts. It happens that the set of languages *recognized* by these automata forms a very important subset of formal languages.

Definition 2 (Nondeterministic finite automata)**.** *A **nondeterministic finite automaton** (NFA) is a tuple $\mathcal{R} = (Q, \Sigma, \Delta, q_0, F)$, where Q is a finite set of states, Σ is a finite input alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a set of transition rules, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of final states.*

Definition 3. *A configuration of an NFA $\mathcal{R}$ is a tuple (q, w), where $q \in Q$ is a state and $w \in \Sigma^*$ represents the remaining input.*

In order to define the formal semantics of nondeterministic finite automata, we introduce a transition relation $\vdash_{\mathcal{R}}$ on configurations of $\mathcal{R}$.

Definition 4. *Let $\mathcal{R} = (Q, \Sigma, \Delta, q_0, F)$ be an NFA and let (q_1, aw) and (q_2, w) be configurations of $\mathcal{R}$. Then $(q_1, aw) \vdash_{\mathcal{R}} (q_2, w)$ if $(q_1, a, q_2) \in \Delta$.*

We can now formally define what it means for an automaton to accept a word.

Definition 5. *Let $\vdash^*_{\mathcal{R}}$ denote the reflexive transitive closure of $\vdash_{\mathcal{R}}$. We say that an NFA $\mathcal{R} = (Q, \Sigma, \Delta, q_0, F)$ **accepts** a word $w \in \Sigma^*$ if $(q_0, w) \vdash^*_{\mathcal{R}} (f, \epsilon)$, where $f \in F$. The **language** of $\mathcal{R}$, denoted $L(\mathcal{R})$, is the set of words accepted by $\mathcal{R}$, i.e. $\{w \mid w \in \Sigma^*,$ and there exists $f \in F$ s.t. $(q_0, w) \vdash^*_{\mathcal{R}} (f, \epsilon)\}$. We say that $\mathcal{R}$ **recognizes** $L(\mathcal{R})$.*

The set of languages recognized by nondeterministic finite automata can be shown to be equal to the set of languages produced by something called regular grammars [13]. Therefore, we call these languages regular.

Definition 6. *A language is **regular** if it is recognized by some NFA.*

Remark 2.1. *Most definitions of NFA include ϵ-transitions, i.e. transitions that read ϵ. For practical reasons, we don't allow these trasnitions. Regardless of definition, the resulting automata recognize the same languages. In fact, they are both equivalent to deterministic finite automata (DFA), in the sense that you can construct a DFA that recognizes the same language as any NFA, and vice versa [13].*

We can present NFAs in a more intuitive way with graphs. Figure 1 shows the graphs representation of the NFA $(\{q_0, q_1\}, \{a, b\}, \{(q_0, a, q_1), (q_1, b, q_0)\}, q_0, \{q_0\})$. The initial state is marked with an incoming arrow, and the final states are marked with a double border. The NFA recognizes the language $\{\epsilon, ab, abab, ...\}$.

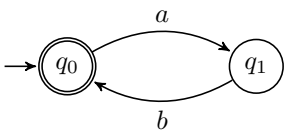

Figure 1: NFA recognizing $\{(ab)^n \mid n \in \mathbb{N}\}$

2.2 Pushdown automata

Nondeterministic finite automata are widely used and have many practical applications. However, NFA generally fail to recognize languages with recursive structure. The canonical example is the language $\{a^n b^n \mid n \in \mathbb{N}\}$ of strings with a finite number a's followed by the same number of b's. We need pushdown automata, a class of more powerful abstract machines, to recognize these languages.

Pushdown automata are similar to NFAs in many aspects. The main difference is the addition of a *stack*. The contents of the stack influence the decisions of the automaton. The automaton can change the contents of the stack by popping from and/or pushing to the stack while perfoming transitions. Unlike our NFA, the automaton can also make ϵ-transitions, i.e. transitions which don't read an input symbol.

There are two different, but equivalent, modes of acceptance for pushdown automata; accepting by final state and accepting by empty stack. Our pushdown automata are going to accept by empty stack. This decision affects the time and space complexity of converting pushdown automata to context-free grammars.

Definition 7 (Pushdown automata)**.** *A **pushdown automaton** (PDA) is a tuple $\mathcal{P} = (P, \Gamma, \Sigma, \Delta, p_0, \gamma_0)$, where P is a finite set of states, Γ is a finite stack alhabet, Σ is a finite input alphabet, $p_0 \in P$ is an initial initial state and γ_0 is an initial stack symbol. The set of rules Δ is a finite set of transition rules, each of the form $\langle p, \gamma \rangle \xhookrightarrow{a} \langle p', w \rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, $w \in \Gamma^{\leq 2}$, $a \in \Sigma \cup \{\epsilon\}$.*

Definition 8 (Configurations)**.** *A **configuration** of a PDA $\mathcal{P}$ is a triple (p, w, α), where $p \in P$ is a state, $w \in \Sigma^*$ represents the remaining input and $\alpha \in \Gamma^*$ represents the current stack contents.*

To define the semantics of pushdown automata, we first introduce a transition relation $\vdash_{\mathcal{P}}$ on configurations of $\mathcal{P}$.

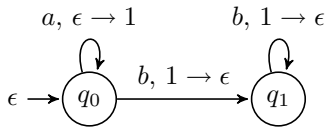

Figure 2: PDA recognizing $\{a^n b^n \mid n \in \mathbb{N}\}$

Definition 9 (Transitions)**.** *Given a PDA $\mathcal{P}$, we have $(p, aw, \sigma\alpha) \vdash (p', w, \gamma\alpha)$ if $\langle p, \sigma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma\rangle \in \Delta$ for some $p, p' \in P$, $\sigma, \gamma \in \Gamma \cup \{\epsilon\}$ and $a \in \Sigma \cup \{\epsilon\}$.*

Definition 10 (Acceptance)**.** *Let $\vdash_{\mathcal{P}}^*$ denote the reflexive transitive closure of $\vdash_{\mathcal{P}}$. We say that $\mathcal{P}$ **accepts** a word $w \in \Sigma^*$ if $(p_0, w, \gamma_0) \vdash_{\mathcal{P}}^* (p, \epsilon, \epsilon)$, for some $p \in P$. The **language** of $\mathcal{P}$, denoted $L(\mathcal{P})$ is the set of all words accepted by $\mathcal{P}$, i.e. $\{w \mid w \in \Sigma^*$ and there exists a $p \in F$ s.t. $(p_0, w, \gamma_0) \vdash_{\mathcal{P}}^* (p, \epsilon, \epsilon)\}$.*

Like NFAs, PDAs can be nicely represented with graphs. Figure 2 shows the graphs representation of the PDA $(\{q_0, q_1\}, \{a, b\}, \{1\}, \{\langle q_1, \epsilon\rangle \overset{a}{\hookrightarrow} \langle q_1, 1\rangle\langle q_1, 1\rangle \overset{b}{\hookrightarrow} \langle q_2, \epsilon\rangle\langle q_2, 1\rangle \overset{b}{\hookrightarrow} \langle q_2, \epsilon\rangle\}, q_0, \epsilon)$. The initial state is again marked with an incoming arrow. This arrow is labelled with the initial stack symbol. In this case, the initial stack will be empty. A transition from a state p to a state q is labelled with a string σ, $\gamma \to w$, where $\sigma \in \Sigma, \gamma \in \Gamma, w \in \Gamma^{\leq 2}$, denoting that the PDA contains the transition rule $\langle p, \gamma\rangle \overset{\sigma}{\hookrightarrow} \langle q, w\rangle$. This particular PDA recognizes the language $\{\epsilon, ab, aabb, aaabbb, ...\}$.

2.3 Pushdown automata from recursive programs

Pushdown automata serve as a natural model for sequential recursive programs (for example, programs written in Java) with finite variable domains [12]. The states of the pushdown automaton correspond to valuations of the global variables, and the stack contains the current values of the local variables and the program pointer.

Assume that we have a program represented by a control flow graph, consisting of a set of nodes *Loc*, which represent the program locations, a set of statements *Stmnt*, and a set of transitions $Trans \subseteq Loc \times Stmnt \times Loc$ which represent all the possible actions of the program. When there are several functions in the program, the graph consists of several disjoint subgraphs, each corresponding to the behaviour of one particular function. We assume that *Stmnt* include statements for function calls.

```
boolean i;

function main(){
  f()
  if(i=true){
     return
  }
  else{
     return
  }
}

function f(){
  i := true
}
```

Figure 3: Example program

For example, consider program in Figure 2.3, written in C-like pseudo-code. This program does not do anything, but we will use it as an example, since it contains an assignment, a function call and a conditional statement. The control flow graph of this program is illustrated in Figure 2.3. Assume that the control flow graph contains "empty" locations that return transitions can go to.

We will now translate the control flow graph to a pushdown automaton. The input alphabet of the PDA is the set $Stmnt$ of program statements, and it has as states the set of valuations of global variables. For this particular example, we have that the set of states is $\{i_{true}, i_{false}\}$. In general, if the global variables have domains $D_1, ..., D_n$, the set of states will be $D_1 \times ... \times D_n$.

If there are no local variables, the rules of the PDA will be of form

$$\langle g_1, n_1 \rangle \xhookrightarrow{a} \langle g_2, ns \rangle$$

where g_1, g_2 are valuations of global variables, $a \in Stmnt$ is a statement and $n_1 \in Loc, ns \in Loc^{\leq 2}$ are locations in the control flow graph. If there are local variables, the rules will instead be of form

$$\langle g_1, (l_1, n_1) \rangle \xhookrightarrow{a} \langle g_2, (l_2, ns) \rangle$$

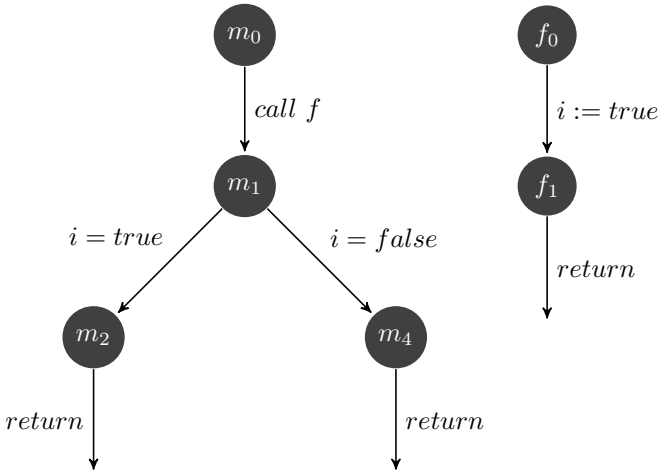

Figure 4: Example control flow graph

where l_1 and l_2 are valuations of local variables.

The translation works as follows. For each transition $(n_i, a, n_j) \in Trans$:

- If a modifies a global variable, add the rule

$$\langle g_1, (l, n_i)\rangle \overset{a}{\hookrightarrow} \langle g_2, (l, n_j)\rangle$$

  where g_2 is the effect of a on g_1, for every valuation of local variables.

- If a modifies a local variable, add the rule

$$\langle g, (l_1, n_i)\rangle \overset{a}{\hookrightarrow} \langle g, (l_2, n_j)\rangle$$

  where l_2 is the effect of a on l_1, for every valuation of global variables.

- If a is a conditional, add the rule

$$\langle g, (l, n_i)\rangle \overset{a}{\hookrightarrow} \langle g, (l, n_j)\rangle$$

  for every valuation of global and local variables which is consistent with the semantics of a.

$$\langle i_{false}, m_0\rangle \overset{call\ f}{\hookrightarrow} \langle i_{false}, f_0 m_0\rangle \quad \langle i_{true}, m_0\rangle \overset{call\ f}{\hookrightarrow} \langle i_{true}, f_0 m_0\rangle$$
$$\langle i_{false}, f_0\rangle \overset{i:=true}{\hookrightarrow} \langle i_{true}, f_1\rangle \quad \langle i_{true}, f_0\rangle \overset{i:=true}{\hookrightarrow} \langle i_{true}, f_1\rangle$$
$$\langle i_{false}, f_1\rangle \overset{return}{\hookrightarrow} \langle i_{false}, \epsilon\rangle \quad \langle i_{true}, f_1\rangle \overset{return}{\hookrightarrow} \langle i_{true}, \epsilon\rangle$$
$$\langle i_{false}, m_2\rangle \overset{return}{\hookrightarrow} \langle i_{false}, \epsilon\rangle \quad \langle i_{true}, m_2\rangle \overset{return}{\hookrightarrow} \langle i_{true}, \epsilon\rangle$$
$$\langle i_{false}, m_3\rangle \overset{return}{\hookrightarrow} \langle i_{false}, \epsilon\rangle \quad \langle i_{true}, m_3\rangle \overset{return}{\hookrightarrow} \langle i_{true}, \epsilon\rangle$$
$$\langle i_{false}, m_1\rangle \overset{i=false}{\hookrightarrow} \langle i_{false}, m_3\rangle \quad \langle i_{true}, m_1\rangle \overset{i=true}{\hookrightarrow} \langle i_{true}, m_2\rangle$$

Figure 5: Rules of resulting PDA

- If a is a function call, add the rule

  $$\langle g, (l, n_i)\rangle \overset{a}{\hookrightarrow} \langle g, (l, f_0 n_j)\rangle$$

  where f_0 is the entry point of the called function, for each valuation of global and local variables.

- If a is a return statement, add the rule

  $$\langle g, (l, n_i)\rangle \overset{a}{\hookrightarrow} \langle g, (l, \epsilon)\rangle$$

  for each valuation of global and local variables.

In the case we don't have local variables, the translation is similar, but simpler. Figure 5 shows the transition rules of the PDA corresponding to our example program. This automaton has two states, one where i is true, and one where i is false. The initial stack content is m_0, the entry point of the program. The automaton performs the actions of the program, using the stack for keeping track of the current location and previous return locations. This particular program has only one possible run; *call f, i:=true, return, i:=true, return.* This is the only word that the PDA accepts.

2.4 Context-free grammars

Definition 11 (Context-free grammars)**.** *A **context-free grammar** (CFG) is tuple $\mathcal{G} = (V, \Sigma, P, S)$, where V and Σ are disjoint finite sets of variable symbols and terminal symbols, respectively, P is a set of production rules, and $S \in V$ is the start variable. A **production rule** is of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$. A is called the **head** of the rule.*

We define the notion of production by defining a relation $\Rightarrow_{\mathcal{G}}$ on words $w \in (V \cup \Sigma)^*$.

Definition 12 (Production)**.** *Let $u, x, v, w \in (V \cup \Sigma)^*$. We define $\Rightarrow_{\mathcal{G}}$ by*

$$uxv \Rightarrow_{\mathcal{G}} uwv \iff x \to w \in P.$$

*Let $\Rightarrow_{\mathcal{G}}^*$ denote the reflexive transitive closure of $\Rightarrow_{\mathcal{G}}$. We say that $\mathcal{G}$ **produces** a word $w \in \Sigma^*$ if $S \Rightarrow_{\mathcal{G}}^* w$. The **language** of $\mathcal{G}$, denoted $L(\mathcal{G})$, is the set $\{w \mid w \in \Sigma^*, S \Rightarrow_{\mathcal{G}}^* w\}$ of all words that $\mathcal{G}$ produces. We also say that $\mathcal{G}$ produces that language.*

Definition 13. *A language L is called **context-free** if it is produced by some context-free grammar.*

We will now state a fundamental theorem that relates pushdown automata and context-free grammars. It says that pushdown automata and context-free grammars are equivalent, in the sense that they accept and produce exactly the same languages.

Theorem 2.2. *For any language K over Σ, there exists a PDA $\mathcal{P}$ s.t. $L(\mathcal{P}) = K$ if and only if there exists a CFG $\mathcal{G}$ s.t. $L(\mathcal{G}) = K$.*

Proof. There is a constructive proof in most introductory textbooks, e.g. [13, 4]. □

2.5 Context-free grammars from pushdown automata

Assume that we have a pushdown automaton $\mathcal{P} = (P, \Gamma, \Sigma, \Delta, p_0, \gamma_0)$. We can convert $\mathcal{P}$ to an equivalent context-free grammar $\mathcal{G}$ via the following construction. The non-terminals of $\mathcal{G}$ are of form (q, γ, q'). Intuitively, a non-terminal (q, γ, q') produces everything that $\mathcal{P}$ accepts while going from state q with γ on top of the stack to q' with an empty stack.

- For any transition rule $\langle p, \gamma \rangle \xhookrightarrow{a} \langle q, \gamma_1 \gamma_2 \rangle$, where $\gamma_1, \gamma_2 \in \Gamma$, add production rules $\{(p, \gamma, p') \to a(q, \gamma_1, q')(q', \gamma_2, p') \mid p', q' \in P, \mathit{Reach}(p, q'), \mathit{Reach}(q', p')\}$, where P is the set of states in $\mathcal{P}$ and $\mathit{Reach}(p, q)$ means state q is reachable from state p (if q is not reachable from p, $\mathcal{P}$ cannot accept anything between those states).
- For any transition rule $\langle p, \gamma \rangle \xhookrightarrow{a} \langle q, \gamma' \rangle$, where $\gamma' \in \Gamma$, add production rules $\{(p, \gamma, p') \to a(q, \gamma', p') \mid p' \in P, \}$.

- For any transition rule $\langle p, \gamma \rangle \xhookrightarrow{a} \langle q, \epsilon \rangle$, add a production rule $(p, \gamma, q) \to a$.
- Add production rules $\{S \to (p_0, \gamma_0, p) \,|\, p \in P, Reach(p_0, p)\}$, where p_0 is the initial state and γ_0 is the initial stack symbol of $\mathcal{P}$, and let S be the start variable.

2.6 Converting to 2NF

For practical reasons, we only consider grammars which are in Binary Normal Form (2NF). A grammar is in 2NF if the right hand side of every production contains at most 2 symbols. Any context-free grammar can be converted to an equivalent context-free grammar in 2NF by expanding productions that contain more than 2 symbols in their right hand side. For example, the production $S \to ABC$ can be expanded to $S \to AS'$ and $S' \to BC$.

2.7 Minimizing Context-free grammars

The cost of performing most operations on context-free grammars grows with the number of rules in the grammar. Therefore, it makes sense to always keep our grammars as small as possible. We do this by removing useless variables and their associated rules. It is also useful to have the grammars in a form where there are no ϵ-producing rules, except for $S \to \epsilon$, where S is the start variable, in the case the language contains ϵ.

There are two kinds of useless variables: variables that can't generate anything, and variables that are not reachable from the start variable.

2.7.1 Removal of non-generating variables

The procedure for removing non-generating variables is essentially a fixpoint computation of the set Gen of generating variables. Let $\mathcal{G} = (V, T, P, S)$ be the grammar in question. We define

$$Gen_0 = \{x \,|\, x \in V \text{ and there exists } x \to a \in P \text{ s.t. } a \in T\}$$

and

$$Gen_{n+1} = \{x \mid x \in V \text{ and there exists } x \to a_1 ... a_k \in P \text{ s.t. for each } a_i, \text{ either } a_i \in T \text{ or } a_i \in Gen_n\}. \quad n \in \mathbb{N}$$

Then Gen is defined as

$$Gen = \bigcup_{i \in \mathbb{N}} Gen_i$$

Since $\mathcal{G}$ is finite, we have $Gen_{n+1} = Gen_n$, i.e. $Gen = Gen_n$, for some n. Naturally, Gen can be computed by iteratively computing $Gen_1, Gen_2, ..., Gen_n$. We can now remove all rules which mention variables that are not in Gen.

Proposition 2.3. *The computation of Gen takes $O(|P|^3)$ time.*

Proof. Consider an arbitrary context-free grammar $\mathcal{G} = (V, T, P, S)$ in 2NF. In the worst case, all production rules have different variables in their head, all generating, but we only discover one at a time. Assume that the grammar contains no superfluous variables, i.e. $|V| = |P|$.

When we compute Gen_{n+1} for some n, we go through $|P|$ production rules. For each rule, we check if the body of that rule contains only generating variables, which is $O(|P|)$, since $|V| = |P|$. So the computation of Gen_{n+1} is $O(|P|^2)$. When we compute $Gen_{|P|}$, we will have done $O(|P|^2)$ operations $|P|$ times. □

2.7.2 Removal of non-reachable variables

The other variables that are useless are the one which are not reachable from the start variable, and the computation is similar to the one for non-generating variables. Again, assume we have the grammar $\mathcal{G} = (V, T, P, S)$. Define

$$Reach_0 = \{S\}$$

and

$$Reach_{n+1} = \{x \mid x \in V \text{ and there exists } y \to a_1 ... a_k \in P \text{ s.t. } x = a_i \text{ for some } i \text{ and } y \in Reach_n\}$$

Then, again, *Reach* is defined as

$$Reach = \bigcup_{i \in \mathbb{N}} Reach_i$$

After computing *Reach*, we can remove all rules which mention variables that are not in this set.

Proposition 2.4. *The computation of Reach takes $O(|P|^3)$ time.*

Proof. Analogous to the proof of Proposition 2.3. □

2.7.3 Removal of ϵ-productions

We say that a context-free grammar is ϵ-reduced if ϵ does not occur in any production except for the production $S \to \epsilon$, where S is the start variable.

Definition 14. *A variable A is **nullable** if $A \Rightarrow_{\mathcal{G}}^{*} \epsilon$.*

Definition 15. *Let $\mathcal{G} = (V, \Sigma, P, S)$ be a CFG and let N be the set of nullable variables in V. We say that $\mathcal{G}$ is ϵ-**reduced** if $N \subseteq \{S\}$.*

Any context-free grammar can be converted to an equivalent ϵ-reduced context-free grammar. The procedure for finding nullable variables is similar to the one for finding non-generating variables.

Let

$$Null_0 = \{x \mid x \in V, x \to \epsilon \in P\}$$

Now define

$$Null_{n+1} = \{x \mid x \in V, x \to a_1 a_2 \in P, a_1 \in E_n, a_2 \in E_n \cup \{\epsilon\}\}$$

The set of nullable variables E is given by

$$Null = \bigcup_{n \in \mathbb{N}} Null_n$$

We can now remove the nullable variables by expanding the rules in which they occur. Assume that our grammar is in 2NF. We expand the rules in the following way.

For a production rule $r = X \to w, X \in V, w \in (V \cup \Sigma)^{\leq 2}$, let $Expand(r)$ be the set of production rules that we can obtain by removing all different

occurences of variables $v \in Null$. For example, $Expand(A \to BC) = \{A \to B, A \to C, A \to BC, A \to \epsilon\}$ if both B and C are nullable. We can then remove ϵ-productions and self rules. For a production rule r, let $Remove$ be defined by

$$Remove(r) = \begin{cases} \emptyset & \text{if } r = X \to \epsilon \text{ for some } X \in V. \\ \emptyset & \text{if } r = X \to X \text{ for some } X \in V. \\ r & \text{otherwise.} \end{cases}$$

Then the new set of production rules without ϵ-productions is the set

$$P' = \bigcup_{r' \in E} Remove(r') \text{ where } E = \bigcup_{r \in P} Expand(r)$$

Proposition 2.5. *The computation of $Null$ is $O(|P|^3)$.*

Proof. The proof is similar to the proof of Proposition 2.3. □

2.8 Deciding emptiness

The emptiness problem for context-free grammars is defined as follow.

Definition 16 (The emptiness problem for CFGs)**.** *Given a CFG $\mathcal{G}$, is it true that $L(\mathcal{G}) = \emptyset$?*

Theorem 2.6. *The emptiness problem for CFGs is decidable in polynomial time.*

Proof. Given a CFG $\mathcal{G}$, let $\mathcal{G}' = (V', \Sigma', P', S')$ be the CFG that we obtain by removing non-generating and non-reachable variables (in that order). Then it is easy to see that $L(\mathcal{G}') = \emptyset \iff P' = \emptyset$. Since the removal of non-generating and non-reachable variables doesn't change the language of the grammar, $L(\mathcal{G}) = \emptyset \iff P' = \emptyset$. Removing non-generating and non-reachable variables can both be done in polynomial time (Propositions 2.3 and 2.4). □

2.9 Intersecting context-free grammars with finite automata

We begin with a well-known but important observation.

Proposition 2.7. *The intersection between a context-free language and a regular language is a context-free language [4].*

Assume that we have a context-free language, represented by a context-free grammar $\mathcal{G} = (V, T, P, S)$, and a regular language, represented by a nondeterministic finite automaton $\mathcal{R} = (Q, \Sigma, \Delta, q_0, F)$. We can construct a CFG $\mathcal{G}'$ that generates the intersection of these languages by creating variables $A_{n,m}$ for each variable A in $\mathcal{G}$ and pair of states n, m in $\mathcal{R}$. The idea is that $A_{n,m}$ produces everything that is both produced by $\mathcal{G}$ and accepted by $\mathcal{R}$ between n and m. The construction works as follows. Assume $\mathcal{G}$ is in 2NF.

1. Let $V_{var} = \{A_{n,m} \mid A \in V,\, n, m \in Q\}$ and $V_{ter} = \{t_{n,m} \mid (n, t, m) \in \Delta\}$.
2. Let $P_1 = \{v_{n,m} \to s_{n,m} \mid v \in V,\, n, m \in Q,\, v \to s \in P\}$.
3. Let $P_2 = \{v_{n,m} \to s_{n,i} s'_{i,m} \mid v \in V,\, n, m, i \in Q,\, v \to ss' \in P\}$.
4. Let $P_\epsilon = \{v_{n,n} \to \epsilon \mid n \in Q,\, v \to \epsilon \in P\}$.
5. Assume S' is disjunct from all the variables in V and V_{var}. Let $P_{start} = \{S' \to v_{q_0,m} \mid m \in F\}$.
6. Then, $\mathcal{G}' = (\{S'\} \cup V_{var} \cup V_{ter}, T \cap \Sigma, P_1 \cup P_2 \cup P_\epsilon \cup P_{start}, S')$

Proposition 2.8. *The computation of the intersection grammar $\mathcal{G}'$ is $O(|V||Q|^3)$.*

Proof. The costly parts are the computations of P_1 and P_2. The computation of P_1 is $O(|V||Q|^2)$ (the number of new rules we need to create) and the computation of P_2 is $O(|V||Q|^3)$. □

The intersection grammar is usually bloated and should be minimized by removing useless variables and ϵ-productions.

3 The shuffle operation

Now we introduce the notion of **shuffle**. The shuffle of two languages is the language you get by taking every pair of words and shuffling them together in a way that preserves the order within the original words.

Definition 17. *Let L_1 and L_2 be two languages with alphabets Σ_1 and Σ_2, respectively, and let $a_i, i \in \mathbb{N}$ denote symbols in $\Sigma_1 \cup \Sigma_2$. Then, the **shuffle** of*

L_1 and L_2 is the language

$$\{a_1...a_n \mid \{1...n\} \text{ can be partitioned into } \{S_1, S_2\} \text{ and there are monotonic functions } f : \{1...|S_1|\} \to S_1,\ g : \{1...|S_2|\} \to S_2 \text{ such that } a_{f(1)}...a_{f(|S_1|)} \in L_1 \text{ and } a_{g(1)}...a_{g(|S_2|)} \in L_2\}$$

We will denote the shuffle of L_1 and L_2 by $L_1 \sqcup\!\sqcup L_2$.

The shuffle is useful because it corresponds to the set of interleavings of programs. Assume that we have two languages L_1 and L_2, representing possible executions of programs P_1 and P_2, respectively. Then $L_1 \sqcup\!\sqcup L_2$ is the language that represents all possible interleaving executions of P_1 and P_2. For example,

$$\{12, 3\} \sqcup\!\sqcup \{a, bc\} = \{12a, 1a2, a12, 12bc, 1b2c, 1bc2, b12c, b1c2, bc12, 3a, a3, 3bc, b3c, bc3\}$$

In general, the idea is as follows.

1. Take the programs in question, and convert them to grammars.
2. Shuffle the languages of these grammars to get the language containing all interleaving executions of the programs.
3. Intersect this language with a language that describes bad executions.
4. Check the intersection for emptiness. If it is empty, there are no bad executions. Otherwise, the concurrent system contains some error.

One condition is critical for this method to work; the emptiness of the intersection must be decidable. This means that the intersection language must be at most context-free, since the emptiness problem for context-sensitive languages is undecidable. It is well known that context-free languages are not closed under intersection. However, we know that the intersection of a context-free and a regular language is context-free. So if the set of bad interleaving executions is a regular language [1] and the shuffle is context-free, emptiness is decidable. Unfortunately, since we are dealing with recursive programs, which give rise to context-free languages, the shuffle is generally not context-free.

Theorem 3.1. *The context-free languages are not closed under shuffle.*

[1] It turns out that most interesting properties can be described by regular grammars

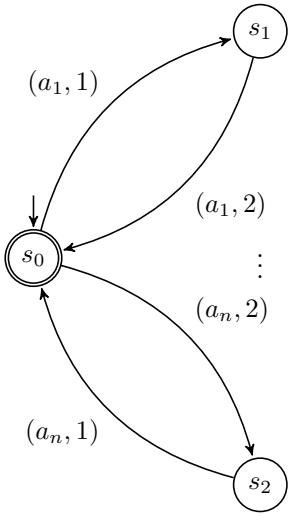

Figure 6: The NFA $\mathcal{R}$

Proof. The proof is by contradiction. Assume, contrarily, that the context-free languages are closed under shuffle. So given arbitrary context-free grammars $\mathcal{G}_1$ and $\mathcal{G}_2$, $L(\mathcal{G}_1) \sqcup\!\sqcup L(\mathcal{G}_2)$ is context-free. Assume that $\mathcal{G}_1$ and $\mathcal{G}_2$ have the same terminal alphabet Σ. Let $\mathcal{G}_1'$ and $\mathcal{G}_2'$ be grammars that are like $\mathcal{G}_1$ and $\mathcal{G}_2$ except that $\mathcal{G}_1'$ has the terminal alphabet $\Sigma_1 = \Sigma \times \{1\}$ and $\mathcal{G}_2'$ has the terminal alphabet $\Sigma_2 = \Sigma \times \{2\}$, with all production rules changed accordingly. In other words, Σ_1 and Σ_2 are tagged versions of Σ.

Now, consider the shuffle $\mathcal{G}_1' \sqcup\!\sqcup \mathcal{G}_2'$ intersected with the NFA $\mathcal{R}$, shown in Figure 6, where $a_1, ..., a_n \in \Sigma$. Call this intersection $\mathcal{G}'$. The NFA consists of $n + 1$ states and $2n$ transitions, and accepts all words that are composed of symbols taken alternatingly from some words $w_1' \in L(\mathcal{G}_1')$ and $w_2' \in L(\mathcal{G}_2')$, starting with w_1'. Now take the corresponding words $w_1 \in L(\mathcal{G}_1)$ and $w_2 \in L(\mathcal{G}_2)$. From the construction of $\mathcal{R}$, $w_1 = w_2$.

This means that we can substitute $(a, 1)$ with a and $(a, 2)$ with ϵ in the alphabet and production rules of $\mathcal{G}'$ to get a grammar $\mathcal{G}''$ that recognizes the intersection $L(\mathcal{G}_1) \cap L(\mathcal{G}_2)$. Since the intersection between a context-free and a regular language is context-free, $\mathcal{G}'$ and $\mathcal{G}''$ are context-free. But we know that the context-free languages are not closed under intersection [4].

□

Since context-free languages are not closed under shuffle, we cannot hope to

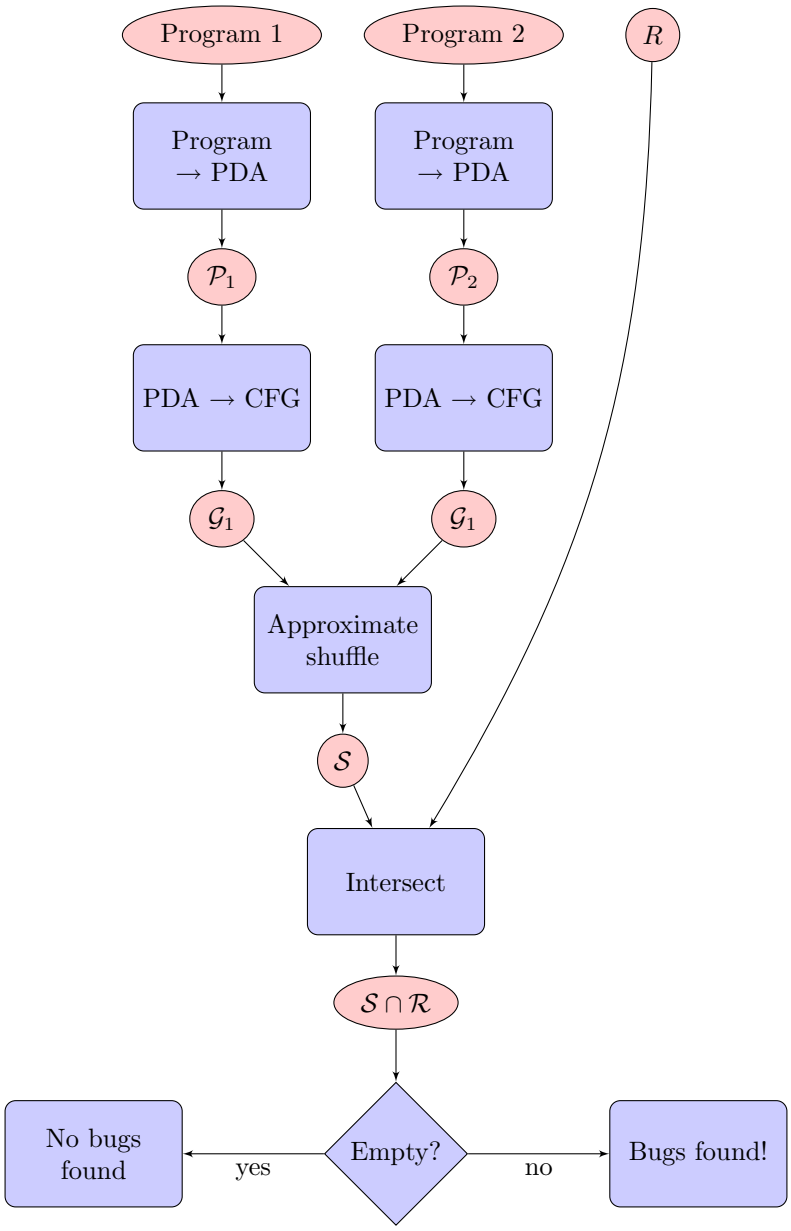

Figure 7: An overview

construct a context-free grammar that generates the whole shuffle language. Instead, given two CFG's $\mathcal{G}_1$ and $\mathcal{G}_2$, we construct a third CFG $\mathcal{S}$ which **underapproximates** the shuffle $L(\mathcal{G}_1) \sqcup\!\sqcup L(\mathcal{G}_1)$, meaning that $L(\mathcal{S}) \subseteq L(\mathcal{G}_1) \sqcup\!\sqcup L(\mathcal{G}_1)$. This means that the procedure we use is not complete, i.e. it can be used to find bugs but not to verify their absence. Figure 7 gives an overview of the process.

4 Shuffle grammars

We begin with a simple example of how one might underapproximate the shuffle of two context-free languages.

4.1 Shuffle up to 1

Assume that $\mathcal{G}_1 = (V_1, \Sigma_1, P_1, S_1)$ and $\mathcal{G}_2 = (V_2, \Sigma_2, P_2, S_2)$ are context-free grammars in 2NF. We will construct a context-free grammar $\mathcal{S}$ that approximates the shuffle $L(\mathcal{G}_1) \sqcup\!\sqcup L(\mathcal{G}_2)$. The important part of $\mathcal{S}$ is the set of production rules. For each pair of productions rules $(p_1, p_2) \in P_1 \times P_2$, we create a set of rules that approximate the shuffle of p_1 and p_2. Assume that $S \in (\Sigma_1 \cup V_1)$ and $T \in (\Sigma_2 \cup V_2)$. Then, the variable (S, T) stands for the shuffle of S and T, i.e. producing (approximately) the shuffle of what S produces in $\mathcal{G}_1$ and what T produces in $\mathcal{G}_2$. Each pair of rules (p_1, p_2) yields a set of rules, depending on what p_1 and p_2 look like, which we add to $\mathcal{S}$:

- If $p_1 = S \to A$ and $p_2 = T \to X$, then add the rules

$$(S, T) \to AX$$
$$(S, T) \to XA$$

- If $p_1 = S \to AB$ and $p_2 = T \to X$, then add the rules

$$(S, T) \to A(B, X)$$
$$(S, T) \to (A, X)B$$

- If $p_1 = S \to A$ and $p_2 = T \to XY$, then add the rules

$$\begin{aligned}(S,T) &\to X(A,Y)\\ (S,T) &\to (A,X)Y\end{aligned}$$

- If $p_1 = S \to AB$ and $p_2 = T \to XY$, then add the rules

$$\begin{aligned}(S,T) &\to A(B,T)\\ (S,T) &\to X(S,Y)\\ (S,T) &\to (A,X)(B,Y)\\ (S,T) &\to (A,T)B\\ (S,T) &\to (S,X)Y\end{aligned}$$

The union of all these sets makes up the set of production rules in $\mathcal{S}$. The set of terminals in $\mathcal{S}$ is the union of the terminals in $\mathcal{G}_1$ and $\mathcal{G}_2$, and the set of variables is $\{(S,T) \mid S \in (\Sigma_1 \cup V_1), T \in (\Sigma_2 \cup V_2)\}$. Naturally, the start variable of $\mathcal{S}$ is (S_1, S_2). We call $\mathcal{S}$ *shuffle up to* 1 of $\mathcal{G}_1$ and $\mathcal{G}_2$, and denote it by $\mathcal{G}_1 \sqcup\!\sqcup_1 \mathcal{G}_2$. In Chapter 6 We will discuss some examples of how we use the shuffle up to 1 to find bugs in concurrent systems.

4.2 Shuffle up to k

It is clear that $\mathcal{G}_1 \sqcup\!\sqcup_1 \mathcal{G}_2$, being a context-free grammar, does not produce all of $\mathcal{G}_1 \sqcup\!\sqcup \mathcal{G}_2$ for arbitrary context-free grammars $\mathcal{G}_1$ and $\mathcal{G}_2$. In order to improve our approximation, we must see where $\sqcup\!\sqcup_1$ fails.

Consider the example shown in Figure 8. We have two grammars $\mathcal{G}_1$ and $\mathcal{G}_2$, with start symbols A and B, respectively, and the top level rules of $\mathcal{G}_1 \sqcup\!\sqcup_1 \mathcal{G}_2$. Variables are in upper case and terminals are in lower case.

Now consider the word $w = a_1b_1a_2a_3b_2b_3a_4b_4$. Clearly, $w \in L(\mathcal{G}_1) \sqcup\!\sqcup L(\mathcal{G}_2)$, since it is a shuffle of the words $a_1a_2a_3a_4 \in L(\mathcal{G}_1)$ and $b_1b_2b_3b_4 \in L(\mathcal{G}_2)$. However, $w \notin L(\mathcal{G}_1 \sqcup\!\sqcup_1 \mathcal{G}_2)$. Why is this? Let's look at the top level rules of $\mathcal{G}_1 \sqcup\!\sqcup_1 \mathcal{G}_2$, which can be directly applied on the start symbol (A, B). The first and second rules cannot produce w, since the first rule produces a string that start with a_1a_2 and the second rule produces one that start with b_1b_2. The

$\mathcal{G}_1$

$$\begin{array}{rcl} A & \longrightarrow & X_1 X_2 \\ X_1 & \longrightarrow & a_1 a_2 \\ X_2 & \longrightarrow & a_3 a_4 \end{array}$$

$\mathcal{G}_2$

$$\begin{array}{rcl} B & \longrightarrow & Y_1 Y_2 \\ Y_1 & \longrightarrow & b_1 b_2 \\ Y_2 & \longrightarrow & b_3 b_4 \end{array}$$

$\mathcal{G}_1 \sqcup\!\sqcup_1 \mathcal{G}_2)$

$$\begin{array}{rcl} (A,B) & \longrightarrow & X_1(X_2,B) \\ (A,B) & \longrightarrow & Y_1(A,Y_2) \\ (A,B) & \longrightarrow & (X_1,Y_1)(X_2,Y_2) \\ (A,B) & \longrightarrow & (X_1,B)X_2 \\ (A,B) & \longrightarrow & (A,Y_1)Y_2 \\ & \vdots & \end{array}$$

Figure 8: Example shuffle

fourth and fifth rules cannot produce w either, since they cannot produce a string that ends with a_4b_4. This leaves the third rule. The third rule produces the concatenation of two strings, the first produced by (X_1, Y_1) and the second by (X_2, Y_2). Both of these string will be of length 4. But the first 4 symbols of w includes a_3, which can only be produced by X_2 in $\mathcal{G}_1$. Thus, none of the rules can produce w, so $w \notin \mathcal{G}_1 \sqcup\!\sqcup_1 \mathcal{G}_2$.

Recall the definition of the *shuffle up to* 1. Using this notion of shuffling, the head of a production rule in the resulting grammar was a tuple (w_1, w_2) s.t. $|w| \leq 1, |w_2| \leq 1$. As we could see, the problem with this approximation is that we cannot "break down" the words that w_1 and w_2 produce. In order to do that, we generalize this notion by defining the *shuffle up to k*. Intuitively, when we shuffle up to k, the heads of the production rules are tuples of sequences of symbols, each of length up to k.

Definition 18. *The* ***shuffle up to*** k *of two context-free grammars* $\mathcal{G}_1 = (V_1, \Sigma_1, P_1, S_1)$ *and* $\mathcal{G}_2 = (V_2, \Sigma_2, P_2, S_2)$ *is the grammar* $Shuffle_k(\mathcal{G}_1, \mathcal{G}_2) = (V_1^{\leq k} \times V_2^{\leq k}, \Sigma_1 \cup \Sigma_2, P_{shuffle}, (S_1, S_2))$*, where* $P_{shuffle}$ *is the set of production rules* $\{(w_1, w_2) \to (u_1, v_1)(u_2, v_2) \mid w_1, u_1, u_2 \in (V_1 \cup \Sigma_1)^{\leq k}, w_2, v_1, v_2 \in (V_2 \cup \Sigma_2)^{\leq k}$ *such that* $w_1 \Rightarrow_{\mathcal{G}_1} u_1u_2, w_1, \Rightarrow_{\mathcal{G}_2} v_1v_2\}$.

5 Implementation

We implemented a prototype tool using this technique in Haskell. The implementation is relatively simple, using straight-forward representations of grammars and automata. It currently supports the shuffle up to 1.

We use the following representation for NFAs, PDAs and CFGs. Here, `Set` is a type from `Data.Set`, which implements pure set operations based on balanced binary trees.

```
data NFA a b = Nfa
               { states' :: Set a
               , transitions' :: Set (a, b, a)
               , initial' :: a
               , final' :: Set a
               } deriving Show

data PDA a b c = Pda
                 { pdaname :: String
                 , states :: Set a
                 , transitions :: Set (a, b, c, a, [b])
                 , stackEps :: b
                 , inputEps :: c
                 , initial :: (a, b)
                 } deriving (Show, Eq)

data CFG a = Cfg
             { cfgname :: String
             , start :: Symbol a
             , rules :: Set (Symbol a, [Symbol a])
             } deriving (Show, Eq)
```

We have the following core funtions. These functions implement the algorithms described in Section 2.

```
-- Constructors
makeNFA :: (Ord b, Ord a) =>
           [a] -> [(a, b, a)] -> a -> [a] -> NFA a b
makePDA :: (Ord b, Ord c, Ord a, Enum a) =>
           String -> [a] -> b -> c -> [(a, b, c, a, [b])]
           -> (a, b) -> PDA a b c
makeCFG :: Ord a =>
           String -> Symbol a -> [(Symbol a, [Symbol a])]
           -> CFG a

-- Conversion from PDA to CFG
toStringCFG :: (Show b, Show c, Show a, Eq c, Eq b, Enum a) =>
           PDA a b c -> CFG String

-- Simplification
removeNonGenerating :: Ord a => CFG a -> CFG a
removeNonReachable :: Ord a => CFG a -> CFG a
removeUseless :: Ord a => CFG a -> CFG a
removeEpsilon :: Ord a => CFG a -> CFG a
to2NF :: CFG String -> CFG String
normalizeCFG :: CFG String -> CFG String

-- Shuffling
shuffleGrammars :: CFG String -> CFG String -> CFG String

-- Intersection with NFA
intersectNFA :: (Show a, Ord a) =>
                CFG String -> NFA a String -> CFG String
```

6 Examples

This section contains some examples of the type of programs our prototype is able to handle. We use different modelling techniques for each program, which affects the performance. The first program is a simple example that demonstates two techniques; the modelling of counters and communication via shared vari-

```
function s()
{
  if(i<3)
  {
    i := i + 1
    s()
  }

  return
}
```

Figure 9: Simple counter: Simple recursive counter program

ables. Note that this program does not contain any bugs. The second example demonstrates modelling counters and message passing. This program contains an error [10], which we detect. The third example demonstrates communication via message passing and a more efficient way to synchronize the messages. This program also contains an error [6], which we detect.

6.1 Simple counter

Assume we have a program that has a counter variable i with an infinite range. The pushdown automaton describing this program would then have an infinite number of states. To solve this problem, we simulate the counter with another pushdown automaton. Consider the example program in Figure 9 to demonstrate this technique.

Assume that `i` is a global counter variable with initial value 0. It is straightforward to see that the function `s` calls itself recursively 3 times, increasing the counter to 3. The control flow graph in Figure 10 describes this program.

To model this transition system with pushdown automata, we need to have two processes, one storing the counter value and the other storing the call stack. The easiest way to construct this kind of system is to do a straightforward translation of the control flow graph into a PDA that keeps track of the call stack. Then, synchronize this PDA with another PDA which contains the counter value. These processes communicate via message passing. For example, the counter PDA could only take the transition $i < 3$ if the value of i was actually less that than 3. This transition would be synchronized with the call stack

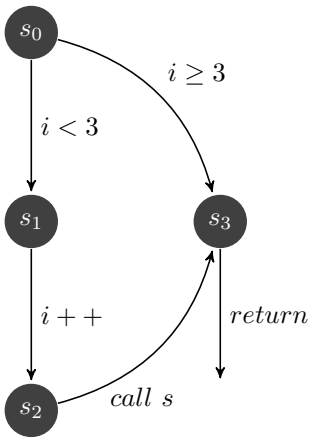

Figure 10: Simple counter: Control flow graph

PDA's $i < 3$-transition, ensuring that the system of 2 PDAs behaves as the original program. This is communication via *message passing*. The two other examples will use message passing.

For this example, we will use communication via *shared varianbles*. The idea is that we have a shared variable which contains the current state. We construct two new control flow graphs: one which contains the statements related to the counter variable i, and on which contains the other statements (except $return$). Additionally, these systems will contain statements that guess the movement of the other system. Figure 11 describes these systems. The guess statements are marked in grey. A guess statement (n, m) means that the PDA in question guesses that the other PDA will update the shared variable from n to m.

Figure 12 shows the transitions of the corresponding counter PDA. The value of the counter is the number of 1s on the stack. The symbol $\perp$ represents the empty stack, in which case the counter is 0. Note that the marked transitions do not conform to the definition of a PDA; they are a convenient shorthand for several transitions with states inbetween. We have also not included the statements from the original control flow graph. We do not need to make these statements visible in order to capture the behaviour of the program. This greatly reduces the size of the resulting shuffle grammar.

We convert the two PDA to CFGs and shuffle them to obtain a grammar which produces the interleaving executions. This shuffle grammar will still produce words that are not valid runs of the concurrent system, since we have not

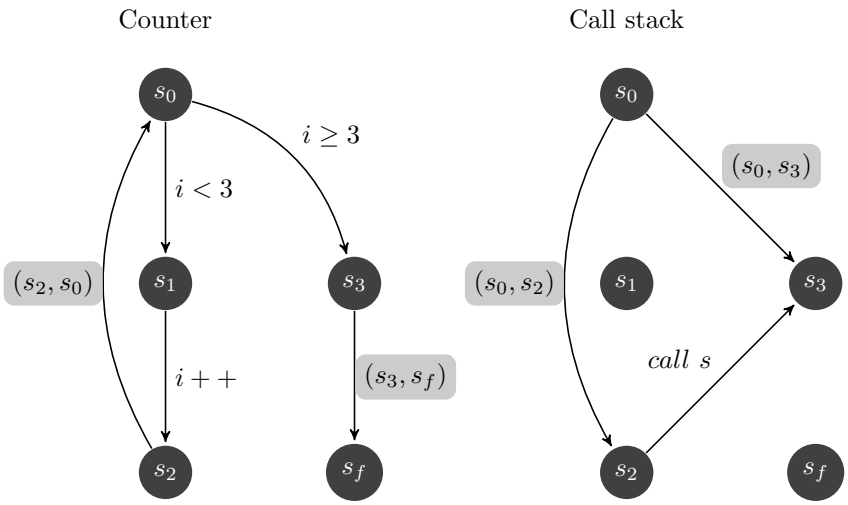

Figure 11: Simple counter: New control flow graphs

$$\langle s_0, \bot\rangle \xhookrightarrow{\epsilon} \langle s_1, \bot\rangle \qquad \langle s_0, 1\bot\rangle \xhookrightarrow{\epsilon} \langle s_1, 1\bot\rangle$$
$$\langle s_0, 11\bot\rangle \xhookrightarrow{\epsilon} \langle s_1, 11\bot\rangle \qquad \langle s_0, 111\rangle \xhookrightarrow{\epsilon} \langle s_3, 111\rangle$$
$$\langle s_1, \bot\rangle \xhookrightarrow{\epsilon} \langle s_2, 1\bot\rangle \qquad \langle s_1, 1\rangle \xhookrightarrow{\epsilon} \langle s_2, 11\rangle$$
$$\langle s_2, 1\rangle \xhookrightarrow{(s_2,s_0)} \langle s_0, 1\rangle \qquad \langle s_3, 1\rangle \xhookrightarrow{(s_3,s_f)} \langle s_f, \epsilon\rangle$$
$$\langle s_f, 1\rangle \xhookrightarrow{\epsilon} \langle s_f, \epsilon\rangle \qquad \langle s_f, \bot\rangle \xhookrightarrow{\epsilon} \langle s_f, \epsilon\rangle$$

Figure 12: Simple counter: transitions

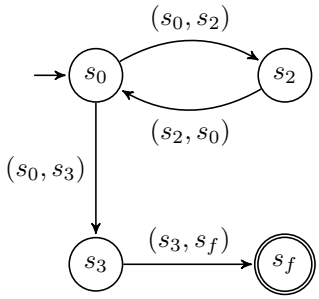

Figure 13: Simple counter: NFA characterizing valid runs

synchronized them. The synchronization is done on the grammar level. To get only the valid runs, we intersect the shuffle grammar with the NFA shown in Figure 13. The resulting grammar produces the only valid run of the system:

$$(s_0, s_2)(s_2, s_0)(s_0, s_2)(s_2, s_0)(s_0, s_2)(s_2, s_0)(s_0, s_3)(s_3, s_f)$$

6.2 Bluetooth driver

In this example, we look at a version of a Windows NT Bluetooth driver described in [10, 2, 14]. The driver has two types of threads; adders and stoppers. It keeps count of the number of threads that are executing the driver via a counter variable `pendingIO`, which is initialized to 1. The stoppers' task is to stop the driver. They set the field `stopFlag` to true, decrease `pendingIO`, wait for an event `stopEvent` which signals that no other threads are working in the driver and finally stop the driver by setting `stopped` to true. The adders perform the I/O operations. They try to increment `pendingIO`. If this is succesful, they assert that the driver is not stopped by checking that `stopped` is false, perform some I/O operations and then decrement `pendingIO`. When `pendingIO` reaches 0, the `stopEvent` event is set to true.

This system is faulty. In [10], a bug is reported which involves one adder thread, one stopper thread and two context switches. The error occurs when the adder thread runs until it calls `inc`. Before it checks for `stopFlag`, the stopper thread runs until the end, and stops the driver. Then, the adder thread continues running and reaches the error label. We are going to try to find this bug. To do this, we first model this system with pushdown automata.

We translate the program to a set of 3 pushdown automata; one for the

```
boolean stopFlag, stopped, stopEvent
int pendingIO

function adder(){
  int status
  status := inc()
  if(status = 0){
    if(stopped){
      error
    }
    else{
      // perform I/O
    }
  }
  dec()
  return
}

function stopper(){
  stopFlag := true
  dec()
  while (!stopEvent){
    // wait
  }
  stopped := true
}
```

Figure 14: Bluetooth driver: adder() and stopper()

```
function inc(){
  if(stopFlag){
    return -1
  }
  atomic{
    pendingIO++
  }
  return 0
}

function dec(){
  int i
  atomic{
    pendingIO--
    i := pendingIO
  }
  if(i=0){
    stopEvent := true
  }
  return
}
```

Figure 15: Bluetooth driver: inc() and dec()

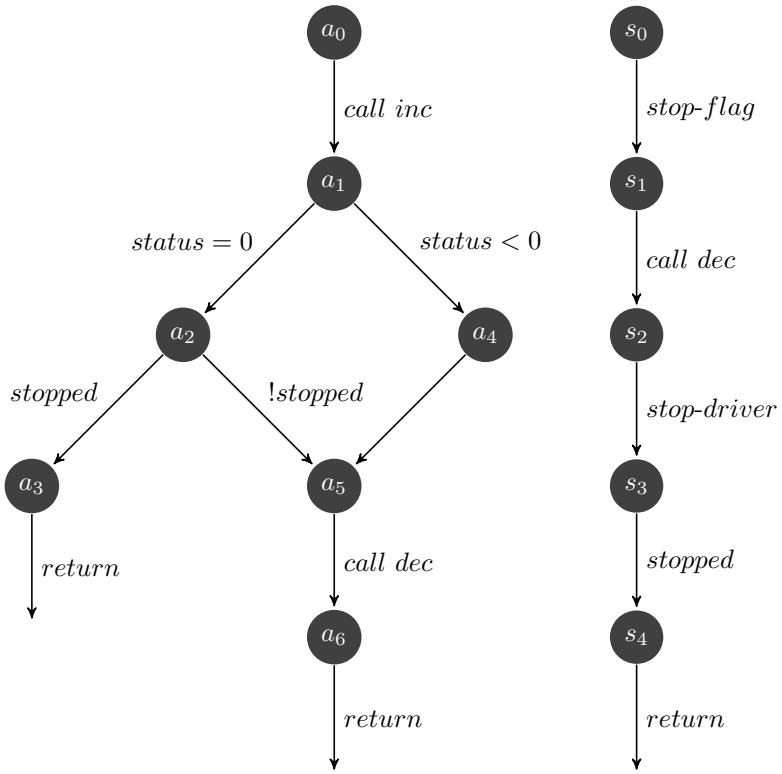

Figure 16: Bluetooth driver: Control flow graphs for adder and stopper

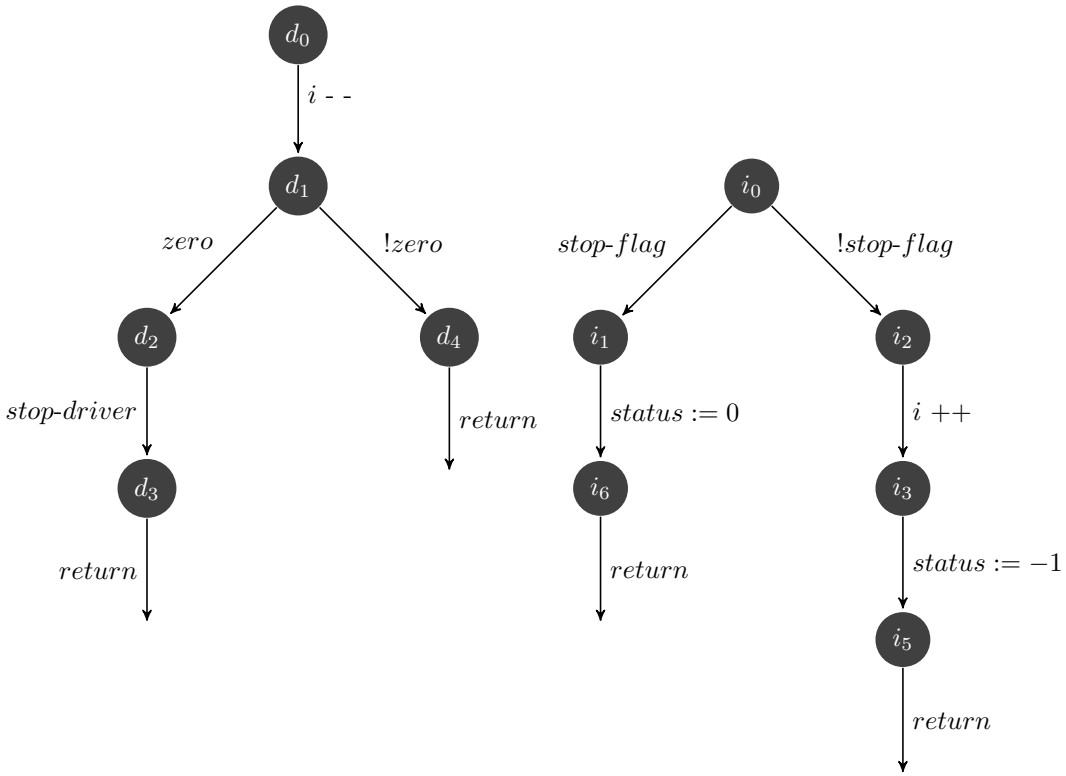

Figure 17: Bluetooth driver: Control flow graphs for dec and inc

$$\langle s_1, \bot\rangle \overset{i++}{\hookrightarrow} \langle s_1, 1\bot\rangle \quad \langle s_1, 1\rangle \overset{i++}{\hookrightarrow} \langle s_1, 11\rangle$$
$$\langle s_1, 1\rangle \overset{i--}{\hookrightarrow} \langle s_1, \epsilon\rangle \quad \langle s_1, \bot\rangle \overset{zero}{\hookrightarrow} \langle s_1, \bot\rangle$$
$$\langle s_1, 1\rangle \overset{!zero}{\hookrightarrow} \langle s_1, 1\rangle \quad \langle s_1, 1\rangle \overset{\epsilon}{\hookrightarrow} \langle s_2, \epsilon\rangle$$
$$\langle s_1, \bot\rangle \overset{\epsilon}{\hookrightarrow} \langle s_2, \epsilon\rangle \quad \langle s_2, 1\rangle \overset{\epsilon}{\hookrightarrow} \langle s_2, \epsilon\rangle$$
$$\langle s_2, \bot\rangle \overset{\epsilon}{\hookrightarrow} \langle s_2, \epsilon\rangle \quad \langle s_0, \bot\rangle \overset{\epsilon}{\hookrightarrow} \langle s_1, 1\bot\rangle$$

Figure 18: Bluetooth driver: Counter transitions

control flow of the adder, one for the control flow of the stopper, and one which simulates the counter. These automata communicate via message passing, i.e. the recognized languages of these automata contain messages that we must synchronize. We do this by intersecting with a set of NFA that filter out interleaved runs which do respect the semantics of the program.

Figure 18 shows the transitions for the counter automaton. Its stack alphabet is $\{1, \bot\}$, where $\bot$ represents an empty stack. It is also the initial stack symbol. The state s_0 is the initial state. To initialize, the automaton pushes 1 on the stack. Since our pushdown automata accept by empty stack, the counter automaton has ϵ-transitions to a final state, in which it pops everything of the stack.

Figure 19 shows the transitions for the adder. It has the initial state s_0, which corresponds to $status < 0$, and initial stack symbol add_0. The state s_1 corresponds to $status = 0$. We have combined some transitions of the control flow graph. For example, if the automaton is in state s_1, and the top of the stack is add_1, it take a *stopped*-transition to add_6, which corresponds to both a_3 and a_6 in the control flow graph. The transitions of the stopper automaton are shown in Figure 20. It has s_0 as initial state, and $stop_0$ as initial stack symbol.

We are now going to try to find a run of this system in which the `error` label occurs. First, we translate the pushdown automata to context-free grammars. This gives us three grammars, bt_{add}, bt_{stop}, and bt_{ctr}, which we then shuffle together. First, we shuffle the adder and stopper to get $\mathcal{G}_1 = bt_{add} \sqcup\!\sqcup_1 bt_{stop}$. When we perform this shuffle, we tag the terminal symbols with the name of the grammar producing them. For example, a terminal a in bt_{add} would be renamed $< a, bt_{add} >$. We then filter out non-valid runs by performing a sequence of NFA intersections. We could intersect with a single NFA that characterizes all valid runs, but it turns out that due to the high complexity of the intersection (it is cubic in the number of NFA states; see Proposition 2.8), it is better to

$\langle s_0, add_0\rangle \overset{\epsilon}{\hookrightarrow} \langle s_0, inc_0 add_1\rangle$
$\langle s_0, add_1\rangle \overset{\epsilon}{\hookrightarrow} \langle s_0, dec_0 add_6\rangle$
$\langle s_0, add_6\rangle \overset{\epsilon}{\hookrightarrow} \langle s_2, \rangle$
$\langle s_1, add_1\rangle \overset{!stopped}{\hookrightarrow} \langle s_1, dec_0 add_6\rangle$
$\langle s_1, add_1\rangle \overset{stopped}{\hookrightarrow} \langle s_1, add_3\rangle$
$\langle s_1, add_3\rangle \overset{error}{\hookrightarrow} \langle s_2, \epsilon\rangle$
$\langle s_0, inc_0\rangle \overset{stop-flag}{\hookrightarrow} \langle s_0, inc_1\rangle$
$\langle s_0, inc_1\rangle \overset{\epsilon}{\hookrightarrow} \langle s_1, \epsilon\rangle$
$\langle s_0, inc_0\rangle \overset{!stop-flag}{\hookrightarrow} \langle s_0, inc_2\rangle$
$\langle s_0, inc_2\rangle \overset{i++}{\hookrightarrow} \langle s_0, \epsilon\rangle$
$\langle s_0, dec_0\rangle \overset{i--}{\hookrightarrow} \langle s_0, dec_1\rangle$
$\langle s_0, dec_1\rangle \overset{zero}{\hookrightarrow} \langle s_0, dec_2\rangle$
$\langle s_0, dec_2\rangle \overset{stop-driver}{\hookrightarrow} \langle s_0, \epsilon\rangle$
$\langle s_0, dec_1\rangle \overset{non-zero}{\hookrightarrow} \langle s_0, dec_3\rangle$
$\langle s_0, dec_3\rangle \overset{\epsilon}{\hookrightarrow} \langle s_0, \epsilon\rangle$
$\langle s_1, dec_0\rangle \overset{i--}{\hookrightarrow} \langle s_1, dec_1\rangle$
$\langle s_1, dec_1\rangle \overset{zero}{\hookrightarrow} \langle s_1, dec_2\rangle$
$\langle s_1, dec_2\rangle \overset{stop-driver}{\hookrightarrow} \langle s_1, \epsilon\rangle$
$\langle s_1, dec_1\rangle \overset{non-zero}{\hookrightarrow} \langle s_1, dec_3\rangle$
$\langle s_1, dec_3\rangle \overset{\epsilon}{\hookrightarrow} \langle s_1, \epsilon\rangle$

Figure 19: Bluetooth driver: Adder transitions

$\langle s_0, stop_0\rangle \overset{stop-flag}{\hookrightarrow} \langle s_0, stop_1\rangle$
$\langle s_0, stop_1\rangle \overset{\epsilon}{\hookrightarrow} \langle s_0, dec_0 stop_2\rangle$
$\langle s_0, stop_2\rangle \overset{stop-driver}{\hookrightarrow} \langle s_0, stop_3\rangle$
$\langle s_0, stop_3\rangle \overset{stopped}{\hookrightarrow} \langle s_0, \epsilon\rangle$
$\langle s_0, dec_0\rangle \overset{i--}{\hookrightarrow} \langle s_0, dec_1\rangle$
$\langle s_0, dec_1\rangle \overset{zero}{\hookrightarrow} \langle s_0, dec_2\rangle$
$\langle s_0, dec_2\rangle \overset{stop-driver}{\hookrightarrow} \langle s_0, \epsilon\rangle$
$\langle s_0, dec_1\rangle \overset{non-zero}{\hookrightarrow} \langle s_0, dec_3\rangle$
$\langle s_0, dec_3\rangle \overset{\epsilon}{\hookrightarrow} \langle s_0, \epsilon\rangle$

Figure 20: Bluetooth driver: Stopper transitions

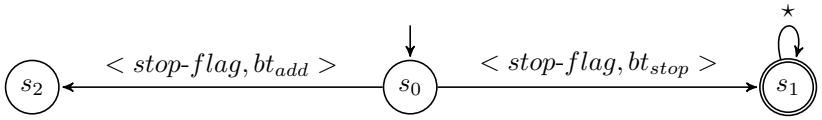

Figure 21: Bluetooth driver: Ordering of *stop-flag*

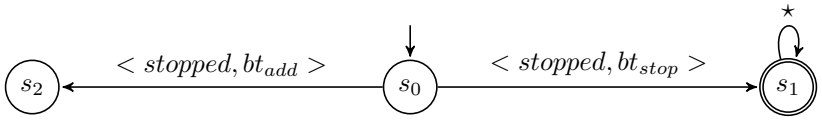

Figure 22: Bluetooth driver: Ordering of *stopped*

intersect with a larger number of small NFAs. We intersect $\mathcal{G}_1$ with the NFA shown in Figures 21, 22 and 23. In these NFA, the star transition represents a set of transitions; nameley, transitions labelled by all terminals that are not explicitly shown in the figure. This intersection yields a grammar $\mathcal{G}_2$, which produces semantically valid runs w.r.t. the interaction between the adder and stopper. To synchronize the counter statements, we shuffle $\mathcal{G}_2$ and bt_{ctr}, to produce $\mathcal{G}_3 = \mathcal{G}_2 \sqcup\!\sqcup_1 bt_{ctr}$. We intersect this grammar with the NFA shown in Figures 24 and 25. The result is a grammar that produces only valid runs of the concurrent system. We finally intersect this grammar with the NFA describing bad runs, shown in Figure 26. The resulting grammar is non-empty, which means that we have found a bug in the system. We can generate the shortest word in the language of this grammar:

$< stop{-}flag, \mathcal{G}_2 >< i{-}{-}, bt_{ctr} >< i{-}{-}, \mathcal{G}_2 >< zero, bt_{ctr} >< zero, \mathcal{G}_2 >< stop{-}driver, \mathcal{G}_2 >< stop{-}driver, \mathcal{G}_2 >< stopped, \mathcal{G}_2 >< stop{-}flag, \mathcal{G}_2 >< stopped, \mathcal{G}_2 >< error, \mathcal{G}_2 >$

This is the same bug reported in [10, 2]. We also tried to find a bug reported in [2] for a second version of the driver [14], requiring 3 processes and 4 context switches. Unfortunately, after intersecting with the NFA ensuring correct semantics, the language was empty. This means that the shuffle up to 1 is not enough to detect this bug.

6.3 Mozilla bug

This example consists of a real-world bug [6] in the Mozilla Application Suite, caused by an incorrect use of locks. We show a simplified version of the code in question in Figure 27.

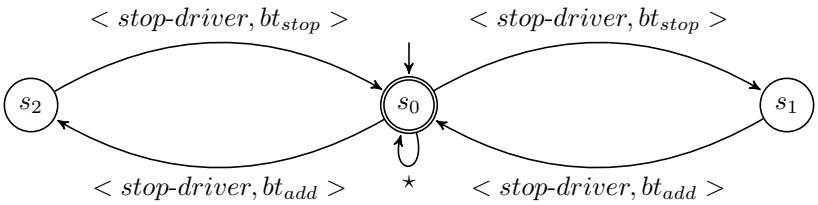

Figure 23: Bluetooth driver: Synchronization of *stop-driver*

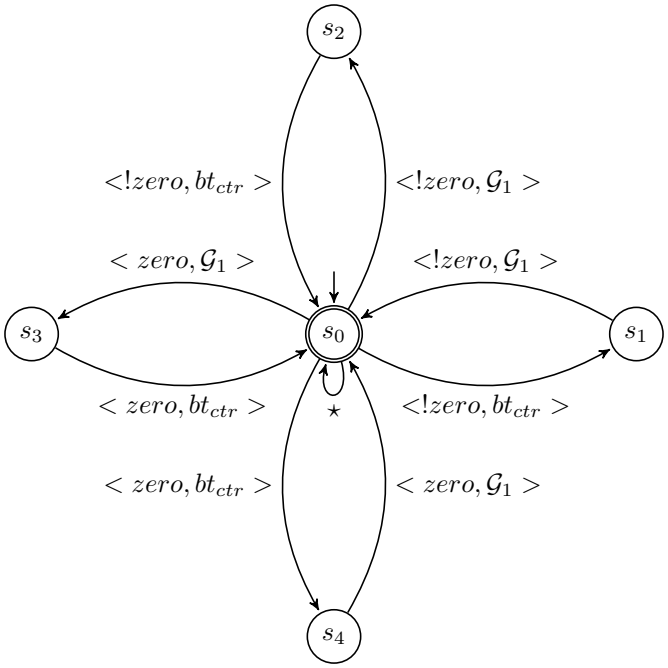

Figure 24: Bluetooth driver: Synchronization of conditionals

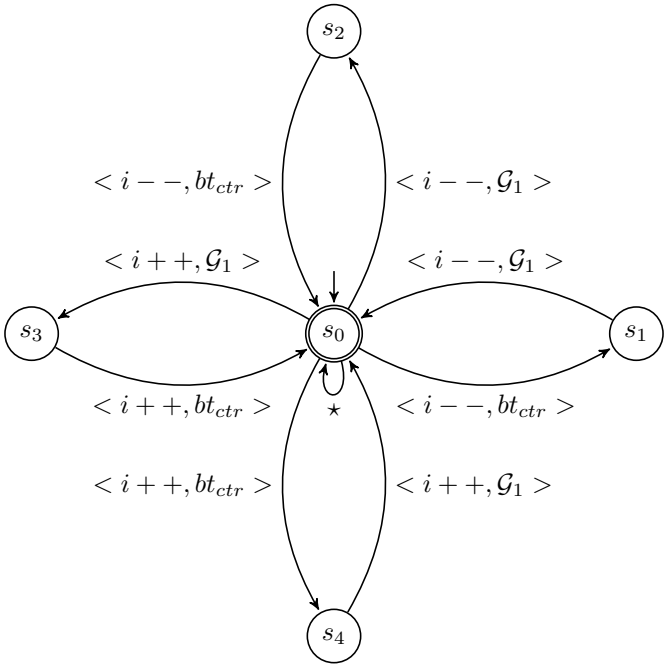

Figure 25: Bluetooth driver: Synchronization of counter updates

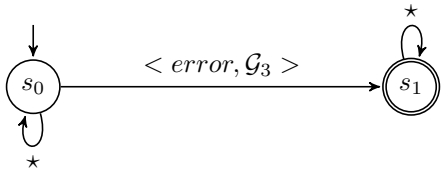

Figure 26: Bluetooth driver: Error traces

```
function thread1(){
  lock(l)
    gScript:=aspt
  unlock(l)
  // ...
  lock(l)
    if(gScript==null){
      error
    }
    else {
      // do stuff
    }
  unlock(l)
}

function thread2(){
  lock(l)
  gScript:=null
  unlock(l)
}
```

Figure 27: Mozilla: Simplified code from Mozilla Application Suite

$$\langle s_0, f_0\rangle \overset{lock}{\hookrightarrow} \langle s_0, f_1\rangle \qquad \langle s_0, f_1\rangle \overset{g:=aspt}{\hookrightarrow} \langle s_0, f_2\rangle$$
$$\langle s_0, f_2\rangle \overset{unlock}{\hookrightarrow} \langle s_0, f_3\rangle \qquad \langle s_0, f_3\rangle \overset{\epsilon}{\hookrightarrow} \langle s_0, f_4\rangle$$
$$\langle s_0, f_4\rangle \overset{lock}{\hookrightarrow} \langle s_0, f_5\rangle \qquad \langle s_0, f_5\rangle \overset{null}{\hookrightarrow} \langle s_0, f_6\rangle$$
$$\langle s_0, f_6\rangle \overset{error}{\hookrightarrow} \langle s_0, f_7\rangle \qquad \langle s_0, f_5\rangle \overset{!null}{\hookrightarrow} \langle s_0, f_7\rangle$$
$$\langle s_0, f_7\rangle \overset{unlock}{\hookrightarrow} \langle s_0, \epsilon\rangle$$

Figure 28: Mozilla: Transitions of $\mathcal{P}_1$

$$\langle s_0, n_0\rangle \overset{lock}{\hookrightarrow} \langle s_0, n_1\rangle \qquad \langle s_0, n_1\rangle \overset{g:=null}{\hookrightarrow} \langle s_0, n_2\rangle$$
$$\langle s_0, n_2\rangle \overset{unlock}{\hookrightarrow} \langle s_0, \epsilon\rangle$$

Figure 29: Mozilla: Transitions of $\mathcal{P}_2$

The first thread consists of two atomic blocks. An error occurs if thread two starts running between these atomic blocks and sets `gScript` to `null`. We translate these two threads to two PDA, $\mathcal{P}_1$ and $\mathcal{P}_2$, whose transitions are shown in Figures 28 and 29. We shuffle these grammars together to obtain the shuffle grammar $\mathcal{S} = \mathcal{P}_1 \sqcup\!\sqcup_1 \mathcal{P}_2$.

In this example, we try another approach to shared variables. The idea is that if we have a variable with very small domains (e.g. a boolean variable), we can shuffle the languages of the programs sharing that variable, and then intersect the shuffle with a NFA that tracks the state of the variable and allows certain statements only in certain states. For example, the conditional `v!=0)` could only be read in a NFA state which represents that `v` is not 0.

We intersect the grammar $\mathcal{S}$ with the NFA in Figures 30, 31 and 32. These enforce correct locking, track the variable g, and give us only runs which contain the error label. The resulting grammar is non-empty; it produces (assuming we tagged the terminals) the word

$< lock, \mathcal{P}_1 >< g := aspt, \mathcal{P}_1 >< unlock, \mathcal{P}_1 >< lock, \mathcal{P}_2 >< g := null, \mathcal{P}_2 >< unlock, \mathcal{P}_2 >< lock, \mathcal{P}_1 >< null, \mathcal{P}_1 >< error, \mathcal{P}_1 >< unlock, \mathcal{P}_1 >$

6.4 Results

Table 1 contains the timing results for constructing the final intersection grammar and checking emptiness. The tests were performed on a 2.5 Ghz Intel Xeon with 6GB of RAM.

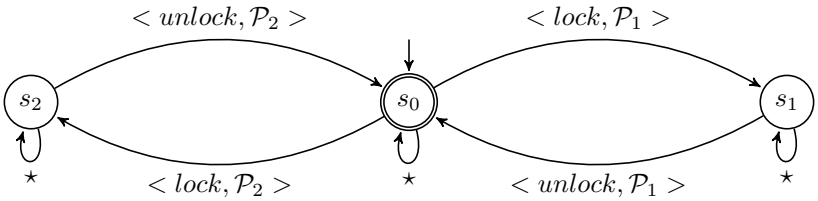

Figure 30: Mozilla: Enforcing lock semantics

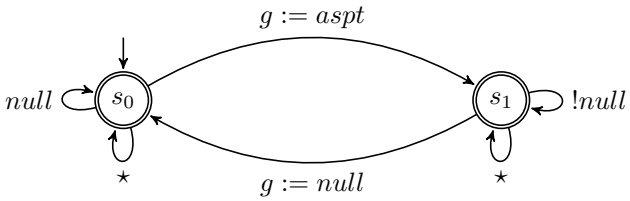

Figure 31: Mozilla: Enforcing variable semantics

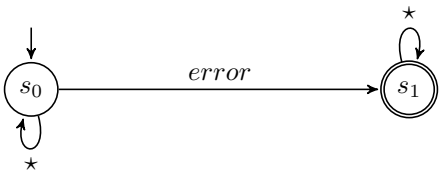

Figure 32: Mozilla: Error traces

Program	Time
Simple Counter	3.2s
Bluetooth	3m40s
Mozilla	1.7s

Table 1: Results

7 Discussion and conclusions

Our prototype tool implements the shuffle up to 1. We have shown that this crude underapproximation is sufficient to detect bugs in some concurrent programs. However, our current prototype has several limitations, beside the fact that we only support $k = 1$:

- **Manual translation**: We manually translate the concurrent systems to pushdown automata. This is a time consuming process which is only

feasible for small programs. The lack of automatic translation is the major shortcoming of the tool. If we want to build a usable tool which works on real-world programs, we need to automate the translation.

- **Grammar representation:** We currently represent a grammar straightforwardly as a start variable and a set of production rules. This is not feasible for larger programs. In order to become competitive, we need to find an efficient symbolic representation, so that we don't need to store every production rule explicitly.

- **NFA intersection**: The algorithm for NFA intersection results in unreasonably large grammars for all but the smallest NFA. This is not a problem if the possible synchonization statements can be synchronized with small NFA. For larger programs, the number of production rules might explode because of the intersection. This problem might be solved by a symbolic representation.

- **Variable ranges**: Currently, we can only handle variables with a small range. Each variable's range affects the size of the PDA and, consequently, the size of the resulting grammar. For boolean programs, this is not a problem. For other types of programs, we cannot hope to build the corresponding PDA due to its size. Once again, an efficient symbolic representation could solve this.

We believe that this method complements bounded model checking, which finds all errors within some constant k context-switches. Using our technique, we can find errors for an arbitrary number of context-switches. As we increase k, the shuffle up to k gets more fine-grained.

There are two challenges to making this technique useful in practice; one minor and one major. The minor challenge is to automate the process of extracting a grammar from a program. This amounts to automating the translation from source code to PDA. The major challenge is to find a suitable symbolic representation that is both concise and supports efficient grammar operations.

8 Related work

Bouajjani, Esparza and Touli [1] use over-approximations of context-free languages to verify the non-reachability of error states. Given two context-free

languages L_1 and L_2, one computes over-approximations A_1 and A_2 in a class of languages that are closed under intersection, and for which emptiness is decidable. If it is the case that $A_1 \cap A_2 = \emptyset$, then $L_1 \cap L_2 = \emptyset$.

Chaki et al. [2] extend this approach with two CEGAR [3, 5] schemes; one which derives and refines abstract systems of pushdown automata from C code, and one which automatically refines the over-approximations A_1 and A_2 in case $A_1 \cap A_2 \neq \emptyset$.

KISS [10] under-approximates the behavior of a concurrent program by a sequential program, but cannot handle more than 2 context switches. Still, this low bound has been enough to detect race conditions in windows device drivers.

The model checker Zing [8] uses procedure summaries to check assertions on multithreaded recursive programs. This approach allows analysis results for procedures to be reused. However, since the underlying problem is undecidable, Zing is not guaranteed to terminate.

Qadeer and Rehof [9] introduce a context-bounded model checking technique that is both sound and complete up to the bound, even for an unbounded number of threads.

Moped [12, 14] is a model checker for sequential and concurrent pushdown automata. It uses an efficient BDD-based symbolic representation. Its concurrent analysis is based on context-bounded model checking.

References

[1] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *ACM SIGPLAN Notices*, volume 38, pages 62–73. ACM, 2003.

[2] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing c programs with recursive calls. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 334–349, 2006.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169. Springer, 2000.

[4] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*, volume 3. Addison-wesley Reading, MA, 1979.

[5] RP Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton Univ Pr, 1994.

[6] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. *ACM SIGPLAN Notices*, 41(11):37–48, 2006.

[7] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI'07*, pages 446–455. ACM, 2007.

[8] S. Qadeer, S.K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *ACM SIGPLAN Notices*, volume 39, pages 245–255. ACM, 2004.

[9] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, 2005.

[10] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. *ACM SIGPLAN Notices*, 39(6):14–24, 2004.

[11] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[12] S. Schwoon. *Model-checking pushdown systems*. PhD thesis, Technische Universität München, Universitätsbibliothek, 2002.

[13] M. Sipser. Introduction to the theory of computation. 1996.

[14] Dejvuth Suwimonteerabuth. *Reachability in Pushdown Systems: Algorithms and Applications*. PhD thesis, Technische Universität München, 2009.