Extending Trace Theory for Concurrent Program Analysis

S. Karageorgieva, N. Kumar, R. Liu, A. Roze

November 12, 2022

High Level Overview

- Important problem in computer science: modelling types of programs using mathematical structures, e.g. for program verification/analysis.

High Level Overview

- Important problem in computer science: modelling types of programs using mathematical structures, e.g. for program verification/analysis.
- One idea is to represent a basic step as a letter, and a program as a string.

High Level Overview

- Important problem in computer science: modelling types of programs using mathematical structures, e.g. for program verification/analysis.
- One idea is to represent a basic step as a letter, and a program as a string.
- Concurrent programs: need to consider steps that execute concurrently, or independently of each other.

High Level Overview

- Important problem in computer science: modelling types of programs using mathematical structures, e.g. for program verification/analysis.
- One idea is to represent a basic step as a letter, and a program as a string.
- Concurrent programs: need to consider steps that execute concurrently, or independently of each other.
- To represent concurrency,we allow some substrings to commute when adjacent.

High Level Overview

- Important problem in computer science: modelling types of programs using mathematical structures, e.g. for program verification/analysis.
- One idea is to represent a basic step as a letter, and a program as a string.
- Concurrent programs: need to consider steps that execute concurrently, or independently of each other.
- To represent concurrency,we allow some substrings to commute when adjacent.
- Concatenating strings represents running two programs in succession—we obtain a monoid.

Content

- *Trace monoids.*
- *Regular languages.*
- *Generalized trace monoids.*
- *Our progress so far.*

Trace Theory

Formally, trace theory is the study of partially commutative free monoids.

Notation

Trace Theory

Formally, trace theory is the study of partially commutative free monoids.

Notation

- The (finite) set of all symbols Σ is called an **alphabet** and the symbols are called **letters**.

Trace Theory

Formally, trace theory is the study of partially commutative free monoids.

Notation

- The (finite) set of all symbols Σ is called an **alphabet** and the symbols are called **letters**.
- let x be a string of letters (ex. abc), we call x a **word**

Trace Theory

Formally, trace theory is the study of partially commutative free monoids.

Notation

- The (finite) set of all symbols Σ is called an **alphabet** and the symbols are called **letters**.
- let x be a string of letters (ex. abc), we call x a **word**
- Σ^* is the set of all strings over Σ

Trace Theory

Formally, trace theory is the study of partially commutative free monoids.

Notation

- The (finite) set of all symbols Σ is called an **alphabet** and the symbols are called **letters**.
- let x be a string of letters (ex. abc), we call x a **word**
- Σ^* is the set of all strings over Σ
- A **language** T is a set of strings ($T \subseteq \Sigma^*$)

Trace Theory

Formally, trace theory is the study of partially commutative free monoids.

Notation

- The (finite) set of all symbols Σ is called an **alphabet** and the symbols are called **letters**.
- let x be a string of letters (ex. abc), we call x a **word**
- Σ^* is the set of all strings over Σ
- A **language** T is a set of strings ($T \subseteq \Sigma^*$)
- The **independence relation** I is a symmetric and irreflexive subset of $\Sigma \times \Sigma$.

Trace Monoids

- I induces an **equivalence relation** $\sim_I$ over Σ^*.

Trace Monoids

- I induces an **equivalence relation** $\sim_I$ over Σ^*.
- Strings $u, v \in \Sigma^*$ are equivalent if u can be transformed into v via a sequence of strings which differ by a single commutation of adjacent letters.

Trace Monoids

- I induces an **equivalence relation** $\sim_I$ over Σ^*.
- Strings $u, v \in \Sigma^*$ are equivalent if u can be transformed into v via a sequence of strings which differ by a single commutation of adjacent letters.
- Formally: there exist strings $w_1, \ldots, w_n \in \Sigma^*$ such that $w_1 = u$, $w_n = v$, and $w_i = w_i' ab w_i''$ and $w_{i+1} = w_i' ba w_i''$, where $(a, b) \in I$, for each i.

Trace Monoids

- I induces an **equivalence relation** $\sim_I$ over Σ^*.
- Strings $u, v \in \Sigma^*$ are equivalent if u can be transformed into v via a sequence of strings which differ by a single commutation of adjacent letters.
- Formally: there exist strings $w_1, \ldots, w_n \in \Sigma^*$ such that $w_1 = u$, $w_n = v$, and $w_i = w_i' ab w_i''$ and $w_{i+1} = w_i' ba w_i''$, where $(a, b) \in I$, for each i.
- Furthermore, $\sim_I$ respects string concatenation, i.e. $\sim_I$ induces a congruence $\equiv_I$.

Trace Monoids

Trace Monoid

- A partially commutative free monoid, or **trace monoid**, is the quotient monoid $\mathbb{M}(\Sigma, I) = \Sigma^*/_{\equiv_I}$.

Trace Monoids

Trace Monoid

- A partially commutative free monoid, or **trace monoid**, is the quotient monoid $\mathbb{M}(\Sigma, I) = \Sigma^*/_{\equiv_I}$.
- The elements of $\mathbb{M}(\Sigma, I)$ are called **traces**.

Trace Monoids

Trace Monoid

- A partially commutative free monoid, or **trace monoid**, is the quotient monoid $\mathbb{M}(\Sigma, I) = \Sigma^*/_{\equiv_I}$.
- The elements of $\mathbb{M}(\Sigma, I)$ are called **traces**.
- The binary operation is **concatenation**.

Trace Monoids

Trace Monoid

- A partially commutative free monoid, or **trace monoid**, is the quotient monoid $\mathbb{M}(\Sigma, I) = \Sigma^*/_{\equiv_I}$.
- The elements of $\mathbb{M}(\Sigma, I)$ are called **traces**.
- The binary operation is **concatenation**.
- Subsets of $\mathbb{M}(\Sigma, I)$ are called **trace languages**.

Trace Monoids

Trace Monoid

- A partially commutative free monoid, or **trace monoid**, is the quotient monoid $\mathbb{M}(\Sigma, I) = \Sigma^*/_{\equiv_I}$.
- The elements of $\mathbb{M}(\Sigma, I)$ are called **traces**.
- The binary operation is **concatenation**.
- Subsets of $\mathbb{M}(\Sigma, I)$ are called **trace languages**.

Example

Let $\Sigma = \{a, b, c, d\}$ and $I = \{(a, d), (d, a), (b, c), (c, b)\}$. An example of a trace is

$$[baadcb]_I = \{baadcb, baadbc, badacb, badabc, bdaabc, bdaacb\}$$

Regularity of Trace Languages

Regular Language

- Language can be recognized by an algorithm using **constant space**
- **Goal:** To efficiently find out whether a particular concurrent system works
- **Method**: By determining whether a trace language is regular

Regularity of Trace Languages

Regular Language

- Language can be recognized by an algorithm using **constant space**
- **Goal:** To efficiently find out whether a particular concurrent system works
- **Method**: By determining whether a trace language is regular

- **Example**: $(a)^* = \{a^n \mid n \in \mathbb{N}\}$ (the language of all strings consisting only of the letter a)
  - **Algorithm:** Walk through the string left-to-right and check if every letter is a
  - Runs in constant space since no need to keep track of any other letter

$$aa\overset{\downarrow}{a}aaa \to aaa\overset{\downarrow}{a}aa$$

Regularity of Trace Languages

Regular Language

- Language can be recognized by an algorithm using **constant space**
- **Goal:** To efficiently find out whether a particular concurrent system works
- **Method**: By determining whether a trace language is regular

- **Example**: $(a)^* = \{a^n \mid n \in \mathbb{N}\}$ (the language of all strings consisting only of the letter a)
  - **Algorithm:** Walk through the string left-to-right and check if every letter is a
  - Runs in constant space since no need to keep track of any other letter

$$aa\overset{\downarrow}{a}aaa \to aaa\overset{\downarrow}{a}aa$$

- **Counterexample**: $\{a^n b^n \mid n \in \mathbb{N}\}$
  - **No** constant-space algorithm recognizes this language
  - Need to remember the number of as while counting number of bs

Properties of Regular Languages

Consider two regular languages L_1 and L_2.

- Closure Properties
  - **Union**: Run the algorithm for L_1 and L_2 on the word and take the OR of the results.
  - **Intersection**: Run the algorithm for L_1 and L_2 on the word and take the AND of the results.
  - **Complement**: Run the algorithm for L_1 on the word and take the NOT of the results.

Properties of Regular Languages

Consider two regular languages L_1 and L_2.

- Closure Properties
  - **Union**: Run the algorithm for L_1 and L_2 on the word and take the OR of the results.
  - **Intersection**: Run the algorithm for L_1 and L_2 on the word and take the AND of the results.
  - **Complement**: Run the algorithm for L_1 on the word and take the NOT of the results.
- Finiteness Properties
  - Regular languages can be written as expressions of the languages $\emptyset, \Sigma^*$ and the language consisting only of a single letter, (a)
  - Regular languages are recognized by machines which have finite memory, known as Finite State Automata

What Languages Are We Interested In?

1. The Lexicographic Language $Lex(\Sigma^*)$

- Consider any trace, say $t = \{abc, bac, acb\}$ where $I = \{(a, b), (b, c)\}$
- Order the elements of t in **dictionary order**, ie. $abc < acb < bac$
- Then the minimal element $abc = Lex(t)$
- The language of **minimal elements of traces** is $Lex(\Sigma^*)$

Both these languages are **regular**.

What Languages Are We Interested In?

1. The Lexicographic Language $Lex(\Sigma^*)$

- Consider any trace, say $t = \{abc, bac, acb\}$ where $I = \{(a, b), (b, c)\}$
- Order the elements of t in **dictionary order**, ie. $abc < acb < bac$
- Then the minimal element $abc = Lex(t)$
- The language of **minimal elements of traces** is $Lex(\Sigma^*)$

2. The Ordered Language

- Consider the language $\{aub \mid a \prec_{I,aub} b\}$
- Language of all words of the form aub where a and b cannot be exchanged so that a appears in front of b

Both these languages are **regular**.

Generalized Trace Monoids

- $\mathbb{I} \subseteq \Sigma^* \times \Sigma^*$ (instead of $\mathbb{I} \subseteq \Sigma \times \Sigma$)

Generalized Trace Monoids

- $\mathbb{I} \subseteq \Sigma^* \times \Sigma^*$ (instead of $\mathbb{I} \subseteq \Sigma \times \Sigma$)

Example

Let $\Sigma := \{a, b, c, d\}$ and let $\mathbb{I} := \{(abc, da), (da, abc)\}$. Then

$$bc\color{blue}{abc}\color{red}{da}\color{black}{dd} \equiv bc\color{red}{da}\color{blue}{abc}\color{black}{dd}$$

Generalized Trace Monoids

- $\mathbb{I} \subseteq \Sigma^* \times \Sigma^*$ (instead of $\mathbb{I} \subseteq \Sigma \times \Sigma$)

Example

Let $\Sigma := \{a, b, c, d\}$ and let $\mathbb{I} := \{(abc, da), (da, abc)\}$. Then

$$bc\color{blue}{abc}\color{red}{da}\color{black}{dd} \equiv bc\color{red}{da}\color{blue}{abc}\color{black}{dd}$$

- The set of equivalence classes forms a monoid with the operation of concatenation.

Generalized Trace Monoids

- $\mathbb{I} \subseteq \Sigma^* \times \Sigma^*$ (instead of $\mathbb{I} \subseteq \Sigma \times \Sigma$)

Example

Let $\Sigma := \{a, b, c, d\}$ and let $\mathbb{I} := \{(abc, da), (da, abc)\}$. Then

$$bc\color{blue}{abc}\color{red}{da}\color{black}{dd} \equiv bc\color{red}{da}\color{blue}{abc}\color{black}{dd}$$

- The set of equivalence classes forms a monoid with the operation of concatenation.

Example

$\Sigma := \{a, b, c, d\}$, $\mathbb{I} := \{(abc, da), (da, abc)\}$

$$ab \cdot cda = \color{blue}{abc}\color{red}{da} \color{black}{=} \color{red}{da}\color{blue}{abc} \color{black}{= d \cdot aabc}$$

What goes wrong?

- The cancellation property fails. e.g. if $\mathbb{I} = \{(a, ab), (ab, a)\}$, then

$$aab \equiv aba \qquad \text{but} \qquad ab \not\equiv ba$$

Example

Let $\mathbb{I} = \{(a, c), (ad, c), (ad, b)\}$. Then

$$bacc \ldots cd \equiv bcc \ldots cad \equiv badcc \ldots c \equiv adbcc \ldots c$$

What goes wrong? cont.

The languages which interest us are no longer necessarily regular.

- Suppose $\mathbb{I} = \{(ab, cd), (a, cd), (c, d)\}$. Then $(\Sigma^* \setminus Lex(\Sigma^*)) \cap ac^*d^*b = \{ac^nd^nb : n \geq 1\}$.

What goes wrong? cont.

The languages which interest us are no longer necessarily regular.

- Suppose $\mathbb{I} = \{(ab, cd), (a, cd), (c, d)\}$. Then $(\Sigma^* \setminus Lex(\Sigma^*)) \cap ac^*d^*b = \{ac^n d^n b : n \geq 1\}$.
- For instance:

$$acddb \equiv cdadb \qquad \text{(not enough } c\text{'s)}$$

$$accdb \equiv acdcb \equiv acdcb \equiv cdacb \qquad \text{(not enough } d\text{'s)}$$

$$accddb \equiv acdcdb \equiv cdacdb \equiv cdcdab \equiv abcdcd$$

(enough c's and d's; not in lexicographical order!)

What goes wrong? cont.

The languages which interest us are no longer necessarily regular.

- Suppose $\mathbb{I} = \{(ab, cd), (a, cd), (c, d)\}$. Then $(\Sigma^* \setminus Lex(\Sigma^*)) \cap ac^*d^*b = \{ac^nd^nb : n \geq 1\}$.
- For instance:

$$acddb \equiv cdadb \qquad \text{(not enough } c\text{'s)}$$

$$accdb \equiv acdcb \equiv acdcb \equiv cdacb \qquad \text{(not enough } d\text{'s)}$$

$$accddb \equiv acdcdb \equiv cdacdb \equiv cdcdab \equiv abcdcd$$

(enough c's and d's; not in lexicographical order!)

- Why? $ac^nd^nb \equiv a(cd)^nb \equiv (cd)^nab \equiv ab(cd)^n$

What goes wrong? cont.

The languages which interest us are no longer necessarily regular.

- Suppose $\mathbb{I} = \{(ab, cd), (a, cd), (c, d)\}$. Then $(\Sigma^* \setminus Lex(\Sigma^*)) \cap ac^*d^*b = \{ac^nd^nb : n \geq 1\}$.
- For instance:

$$acddb \equiv cdadb \qquad \text{(not enough } c\text{'s)}$$

$$accdb \equiv acdcb \equiv acdcb \equiv cdacb \qquad \text{(not enough } d\text{'s)}$$

$$accddb \equiv acdcdb \equiv cdacdb \equiv cdcdab \equiv abcdcd$$

$$\text{(enough } c\text{'s and } d\text{'s; not in lexicographical order!)}$$

- Why? $ac^nd^nb \equiv a(cd)^nb \equiv (cd)^nab \equiv ab(cd)^n$
- On the other hand, if $n \neq m$, then $ac^nd^mb \equiv w$ implies $w = uavb$, where $u, v \in \{c, d\}^*$, $|v|_c - |v|_d = n - m \neq 0$

What goes wrong? cont.

The languages which interest us are no longer necessarily regular.

- Suppose $\mathbb{I} = \{(ab, cd), (a, cd), (c, d)\}$. Then $(\Sigma^* \setminus Lex(\Sigma^*)) \cap ac^*d^*b = \{ac^nd^nb : n \geq 1\}$.
- For instance:

$$acddb \equiv cdadb \qquad \text{(not enough } c\text{'s)}$$

$$accdb \equiv acdcb \equiv acdcb \equiv cdacb \qquad \text{(not enough } d\text{'s)}$$

$$accddb \equiv acdcdb \equiv cdacdb \equiv cdcdab \equiv abcdcd$$

$$\text{(enough } c\text{'s and } d\text{'s; not in lexicographical order!)}$$

- Why? $ac^nd^nb \equiv a(cd)^nb \equiv (cd)^nab \equiv ab(cd)^n$
- On the other hand, if $n \neq m$, then $ac^nd^mb \equiv w$ implies $w = uavb$, where $u, v \in \{c, d\}^*$, $|v|_c - |v|_d = n - m \neq 0$
- $\{ac^nd^nb : m \geq 1\}$ is irregular, since the automaton has to remember the number of c's. Hence, $Lex(\Sigma^*)$ is irregular.

Our progress so far

- Our goal is to characterize when these languages, $\mathcal{L}_{a \prec b}$ and $Lex(\Sigma^*)$, are regular.

Our progress so far

- Our goal is to characterize when these languages, $\mathcal{L}_{a \prec b}$ and $Lex(\Sigma^*)$, are regular.
- Let $\tilde{\mathbb{I}} := \{u : (u, v) \in \mathbb{I} \text{ for some } v\}$, i.e. $\tilde{\mathbb{I}}$ is the set of "swappable" strings.

Our progress so far

- Our goal is to characterize when these languages, $\mathcal{L}_{a \prec b}$ and $Lex(\Sigma^*)$, are regular.
- Let $\tilde{\mathbb{I}} := \{u : (u, v) \in \mathbb{I} \text{ for some } v\}$, i.e. $\tilde{\mathbb{I}}$ is the set of "swappable" strings.
- Idea 1: restrict the definition of $\mathbb{I}$.

Our progress so far

- Our goal is to characterize when these languages, $\mathcal{L}_{a \prec b}$ and $Lex(\Sigma^*)$, are regular.
- Let $\tilde{\mathbb{I}} := \{u : (u, v) \in \mathbb{I} \text{ for some } v\}$, i.e. $\tilde{\mathbb{I}}$ is the set of "swappable" strings.
- Idea 1: restrict the definition of $\mathbb{I}$.

Theorem

Let $\mathbb{I}$ be finite. If $\Sigma = \Sigma_1 \sqcup \Sigma_2$, $\mathbb{I} \subseteq \Sigma_1^* \times \Sigma_2^* \cup \Sigma_2^* \times \Sigma_1^*$, and no string $u \in \tilde{\mathbb{I}}$ is a prefix or suffix of another string $v \in \tilde{\mathbb{I}}$, then $\mathcal{L}_{a \prec b}$ is regular.

Our progress so far

- Our goal is to characterize when these languages, $\mathcal{L}_{a \prec b}$ and $Lex(\Sigma^*)$, are regular.
- Let $\tilde{\mathbb{I}} := \{u : (u, v) \in \mathbb{I} \text{ for some } v\}$, i.e. $\tilde{\mathbb{I}}$ is the set of "swappable" strings.
- Idea 1: restrict the definition of $\mathbb{I}$.

Theorem

Let $\mathbb{I}$ be finite. If $\Sigma = \Sigma_1 \sqcup \Sigma_2$, $\mathbb{I} \subseteq \Sigma_1^* \times \Sigma_2^* \cup \Sigma_2^* \times \Sigma_1^*$, and no string $u \in \tilde{\mathbb{I}}$ is a prefix or suffix of another string $v \in \tilde{\mathbb{I}}$, then $\mathcal{L}_{a \prec b}$ is regular.

Theorem

Let $\mathbb{I}$ be finite. Suppose that no string $u \in \tilde{\mathbb{I}}$ has a prefix which is a suffix of another string $v \in \tilde{\mathbb{I}}$, then $\mathcal{L}_{a \prec b}$ is regular.

Our progress so far

- Our goal is to characterize when these languages, $\mathcal{L}_{a\prec b}$ and $Lex(\Sigma^*)$, are regular.
- Let $\tilde{\mathbb{I}} := \{u : (u,v) \in \mathbb{I} \text{ for some } v\}$, i.e. $\tilde{\mathbb{I}}$ is the set of "swappable" strings.
- Idea 1: restrict the definition of $\mathbb{I}$.

Theorem

Let $\mathbb{I}$ be finite. If $\Sigma = \Sigma_1 \sqcup \Sigma_2$, $\mathbb{I} \subseteq \Sigma_1^* \times \Sigma_2^* \cup \Sigma_2^* \times \Sigma_1^*$, and no string $u \in \tilde{\mathbb{I}}$ is a prefix or suffix of another string $v \in \tilde{\mathbb{I}}$, then $\mathcal{L}_{a\prec b}$ is regular.

Theorem

Let $\mathbb{I}$ be finite. Suppose that no string $u \in \tilde{\mathbb{I}}$ has a prefix which is a suffix of another string $v \in \tilde{\mathbb{I}}$, then $\mathcal{L}_{a\prec b}$ is regular.

- Both theorems have sufficient conditions that allow the expression of strings u as a unique concatenation of swappable substrings $u_1 \dots u_n$.

Our progress so far cont.

- Idea 2: study less complex but similar languages.

Our progress so far cont.

- Idea 2: study less complex but similar languages.

Theorem

Let $\mathbb{I} \subseteq \Sigma_1^* \times \Sigma_2^*$. *Let* $a \in \Sigma_1$. *Let* $\tilde{\mathbb{I}}$ *be regular. Then, the following languages are regular:*

1. $\mathcal{L}_a := \{u \in \Sigma_2^* : au \equiv ua\}$.
2. $\mathcal{L}_{a,v} := \{u \in \Sigma_2^* : uav \equiv uva\}$, *where* $v \in \Sigma_2^*$.
3. $\mathcal{L}_{\textvisiblespace a \textvisiblespace} := \{uaw \in \Sigma_2^* a \Sigma_2^* : uaw \equiv uwa\}$.

Questions