Partial Commutation and Traces

Jiří Novotňák

Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 00 Brno, CZ
www.fit.vutbr.cz/∼inovotnak

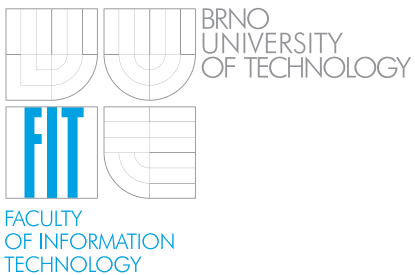


Table of contents

Introduction

Motivation

- Model of parallelism and concurrency
- Rigorous mathematical model

What is the trace

- A sequence of letters
- One process containing jobs
- Some jobs can be commuted or executed simultaneously
- Some jobs have to be execute in order

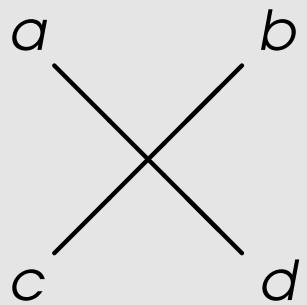
Independence relation

Definition

- Let Σ a finite alpabet of letters
- $I \subseteq \Sigma \times \Sigma$ is a independece relation
- If $(a, b) \in I$, then their order doesn't matter
- $I(a) = \{b \in \Sigma | (a, b) \in I\}$

Graphic representation

- (Σ, I) is called *independence alphabet*
- For example
  $\Sigma = \{a, b, c, d\}$,
  $I = \{(a, d), (d, a), (b, c), (c, b)\}$

$(\Sigma, I) =$ 

Dependence relation

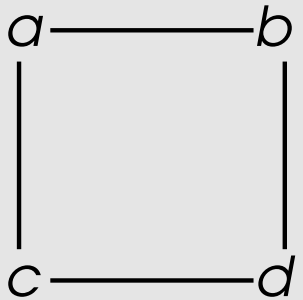
Definition

- $D \subseteq \Sigma \times \Sigma$ is a dependence relation
- If $(a, b) \in D$, then there exist dependency between a and b and order is important
- $D = \Sigma \times \Sigma \setminus I$
- $D(a) = \{b \in \Sigma \mid (a, b) \notin I\}$
- Since I is irreflexive, there is $a \in D(a)$

Graphic representation

- (Σ, D) is called *Dependence alphabet*

$$(\Sigma, D) =$$



Equivalence of traces

Equivalence relation

- The relation I induces $\sim_I$ over Σ^*
- Two words x and y are equivalent under $\sim_I$ (denoted by $x \sim_I y$), if there exist a sequence $z_1, z_2, ..., z_k$ of words such that $x = z_1$ and $y = z_k$ and for all i, $1 \leq i < k$ there exists words z_i', z_i'' and letters a_i, b_i satisfying:
  $z_i = z_i' a_i b_i z_i''$, $z_{i+1} = z_i' b_i a_i z_i''$ and $(a, b) \in I$

Example

- $\Sigma = \{a, b, c, d\}$
- $I = \{(a, d), (d, a), (b, c), (c, b)\}$
- $acdb \sim_I cabd$
- $acdb$ and $cabd$ are the same trace
- $abcd$ and $cabd$ are different traces

Trace monoids

Free partially commutative monoids

- $\sim_I$ is a congruence over Σ^*
- Monoid induced by the relation I is the **Free partially commutative monoid** $M(\Sigma, I)$
- The elements of $M(\Sigma, I)$ which are equivalence classes of words of Σ^* under the relation $\sim_I$ are called **traces**
- $M(\Sigma, I)$ is called **Trace monoid** too

Normal forms

Normal forms of traces

- There are two normal forms
- Lexicographic normal form
- Foata normal form
- Σ must be totally ordered

Lexicographic normal form

- A word x is the lexicographic normal form of a trace if and only if for all factorization $x = ybuaz$, where $y, u, z \in \Sigma^*, (a, b) \in I$ and $a < b$, there exist a letter of u which does not commute with a.

Normal forms

Foata normal form

- A word x of Σ^* is in the Foata normal form, if it is the empty word or if there exist an integer $n > 0$ and non-empty words x_i $(1 \leq i \leq n)$ such that
  - $x = x_1 \ldots x_n$
  - for each i, the word x_i is a product of pairwise independent letters and x_i is minimal with respect to the lexicographic ordering
  - For each $1 \leq i < n$ and for each letter a of x_{i+1} there exist a letter b of x_i such that $(a, b) \in D$
- Every trace has a unique Foata normal form.

Computing normal forms

Algorithm to prepare data structures

- There is a simple algorithm to compute normal forms
- Let $M(\Sigma, I)$ is a partially commutative monoid
- We use a stack for each letter of the alphabet Σ
- We scan word x ($x \in \Sigma^*$) from right to left
- When processing a letter a it is pushed on its stack and a marker is pushed on the stack of all the letters b ($b \neq a$) which do not commute with a
- When all letters has been processed we can compute either the lexicographic normal form or the Foata normal form

Retrieve normal forms

How to get lexicographic normal form

- We get a letter from top of stack of minimal letters
- We we pop a marker on each stack corresponding to a letter b ($b \neq a$) which does not commute with a
- We repeat this loop until all stack are empty

How to get foata normal form

- We take within a loop the set formed by letters being on the top of stacks
- Arrange the letters in the lexicographic order
- Pop corresponding markers (for each letter)
- Repeat this loop until all stacks are empty

Normal forms - lexNF example

Lexicographic normal form

- Let $\Sigma = a, b, c, d$
- Let $I = (a, d), (d, a), (b, c), (c, b)$
- Let $w = badacb$

*	b		
a	*	*	*
a	*	*	d
*	*	*	*
*	b	c	*

- $w' = baadbc$
- $w \sim_I w'$

Recognizable trace languages

Definition

- Let M be a monoid with the unit element 1
- A subset $T \subseteq M$ is said to be *recognizable* if there exist a homomorphism η from M to a finite monoid S such that $T = \eta^{-1}(F)$ for some subset $F \subseteq S$.
- Homomorphism η recognizes T.

M-automaton

- $A = (M, Q, \delta, q_0, F)$
- $\delta : Q \times M \to Q$
- $\forall q \in Q \;\; \delta(q, 1) = q$
- $\forall q \in Q \;\; \forall m_1, m_2 \in M \;\; \delta(q, m_1 m_2) = \delta(\delta(q, m_1), m_2)$
- The subset T of M recognized by the automaton A is defined by $T = \{m \in M \mid \delta(q_0, m) \in F\}$

Dependence graphs

Dependence graph

- $G(V, E, \lambda)$
- G is directed node-labeled acyclic graph
- V is an at most countable set of vertices
- $E \subseteq V \times V$ is the directed edge relation
- $\lambda : V \to \Sigma$ is the node-labeling such that $(\lambda(x), \lambda(y)) \in D$ if and only if $(x, y) \in E \bigcup E^{-1} \bigcup id_V$

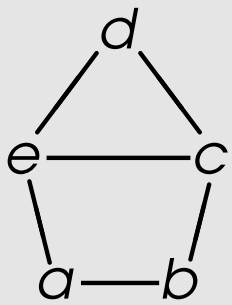
Hasse diagram

- Allows visualization of factors
- Allows visual working with trace

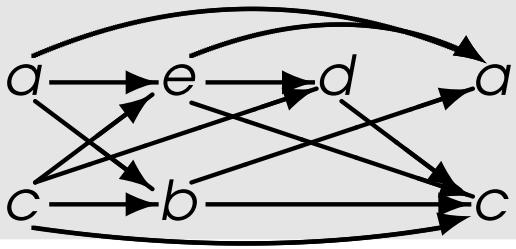
Example

Dependency and input word

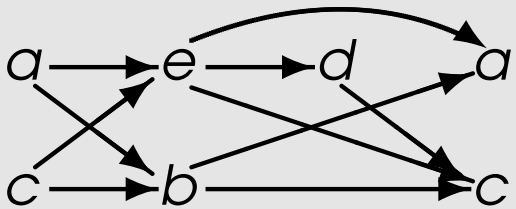
$t = [acebdac]$

$(\Sigma, D) =$ 

Dependence graph



Hasse diagram



Asynchronous automata

Asynchronous automata A

- Has a distributed finite state control such that independent actions may be performed in parallel
- The set of global states is modeled as a Cartesian product $Q = \prod_{i \in J} Q_i$, where the Q_i are states of the local component $i \in J$ and J is some index set.
- With each letter $a \in \Sigma$ we associate a *read domain* $R(a) \subseteq J$ and a *write domain* $W(a) \subseteq J$
- $W(a) \subseteq R(a)$

Transition function

Local transition function

$$\left(\delta_a : \prod_{i \in R(a)} Q_i \to \prod_{i \in W(a)} Q_i\right)_{a \in \Sigma}$$

Global transition function

$$\delta : \left(\prod_{i \in J} Q_i\right) \times \Sigma \to \prod_{i \in J} Q_i$$

Where

- δ is partially defined transition function
- $\delta\left((q_i)_{i \in J}, a\right) = \left(q_i'\right)_{i \in J}$ is defined if and only if $\delta_a\left((Q_i)_{i \in R(a)}\right)$ is defined

Read-write conflicts

Allowed conflicts

- Concurrent Read Exclusive Write
  if $R(a) \bigcap W(b) = \emptyset$ for all $(a, b) \in I$
- Concurrent Read Owner Write
  if $R(a) \bigcap W(b) = \emptyset$ for all $(a, b) \in I$ and $W(a) \bigcap W(b) = \emptyset$ for all $a \neq b$
- Exclusive Read Exclusive Write
  if $R(a) \bigcap R(b) = \emptyset$ for all $(a, b) \in I$
- Exclusive Read Owner Write
  if $R(a) \bigcap R(b) = \emptyset$ for all $(a, b) \in I$ and $W(a) \bigcap W(b) = \emptyset$ for all $a \neq b$

Changing between conflicts types

Changing between conflicts types

- Concurrent Read can be changed to Exclusive Read
- Exclusive Write can be changed to Owner Write

Changing between whole conflict types

- The original definition an asynchronous automaton demands an EREW (Exclusive Read Exclusive Write) type with $R(a) = W(a)$ for all $a \in \Sigma$.
- EROW (Exclusive Read Owner Write) type is even a stronger condition
- Each of the four described types can be changed to EROW type.

Asynchronous cellular automata

Definition of asynchronous cellular automata

- An asynchronous automation A is called asynchronous cellular, if the state space Q can be decomposed as $Q = \prod_{a \in \Sigma} Q_a$ such that $W(a) = a$ and $R(a) = D(a) = \{b \in \Sigma | (a, b) \in D\}$ for all $a \in \Sigma$

Remark

- Every *CROW*-type asynchronous automata can be viewed as asynchronous cellular by trivial transformation (regrouping components) which does not change the number of reachable global states

Practical use

Asynchronous automata

- Parallel control mechanism
- One-phase updating in cellular automata.

Traces self

- Optimization of executable code (Foata normal form)
- Comparing executables

References

G. Rozenberg and A. Salomaa, editors.
Handbook of formal languages, vol. 3: beyond words.
Springer-Verlag New York, Inc., New York, NY, USA, 1997.

Thank for your attention.