Trace Spaces: an Efficient New Technique for State-Space Reduction

L. Fajstrup[1] É. Goubault[2] E. Haucourt[2]
S. Mimram[2] M. Raussen[1]

[1]Aalborg University
[2]CEA, LIST

March 26, 2012
ESOP'12

Goal

When verifying a concurrent program,
there is a priori a large number of possible interleavings to check
(exponential in the number of processes)

Goal

When verifying a concurrent program,
there is a priori a large number of possible interleavings to check
(exponential in the number of processes)

Many executions are equivalent:
we want here to provide a *minimal number of execution traces*
which describe all the possible cases
by adopting a **geometric** point of view

Programs generate trace spaces

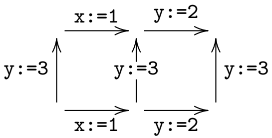
Consider the program

`x:=1;y:=2  |  y:=3`

It can be scheduled in three different ways:

`y:=3;x:=1;y:=2`   `x:=1;y:=3;y:=2`   `x:=1;y:=2;y:=3`

Giving rise to the following graph of traces:



Programs generate trace spaces

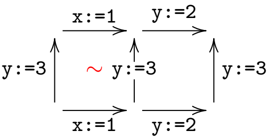
Consider the program

`x:=1;y:=2 | y:=3`

It can be scheduled in three different ways:

`y:=3;x:=1;y:=2`	`x:=1;y:=3;y:=2`	`x:=1;y:=2;y:=3`
$(x, y) = (1, 2)$	$(x, y) = (1, 2)$	$(x, y) = (1, 3)$

Giving rise to the following graph of traces:



homotopy: commutation / filled square

Mutexes

Concurrent access to shared variables should be protected using **mutexes** $a, b, \ldots$:

- P_a: lock the mutex a
- V_a: unlock the mutex a

Mutexes

Concurrent access to shared variables should be protected using **mutexes** $a, b, \ldots$:

- P_a: lock the mutex a
- V_a: unlock the mutex a

```
x:=1;y:=2  |  y:=3
```

Mutexes

Concurrent access to shared variables should be protected using **mutexes** $a, b, \ldots$:

- P_a: lock the mutex a
- V_a: unlock the mutex a

$$P_b\,;\texttt{x:=1};V_b\,;P_a\,;\texttt{y:=2};V_a \mid P_a\,;\texttt{y:=3};V_a$$

Mutexes

Concurrent access to shared variables should be protected using **mutexes** $a, b, \ldots$:

- P_a: lock the mutex a
- V_a: unlock the mutex a

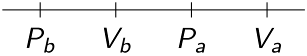
$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$

Let’s adopt a geometric point of view!

Geometric semantics

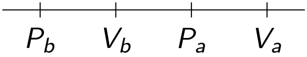
A program will be interpreted as a **directed space**:

- $P_b.V_b.P_a.V_a$

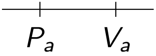


A program will be interpreted as a **directed space**:

- $P_b.V_b.P_a.V_a$



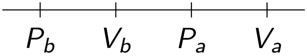
- $P_a.V_a$



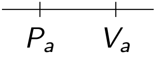
Geometric semantics

A program will be interpreted as a **directed space**:

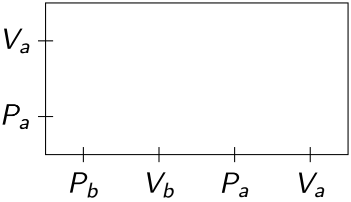
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$



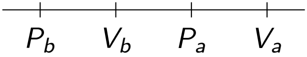
- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$



Geometric semantics

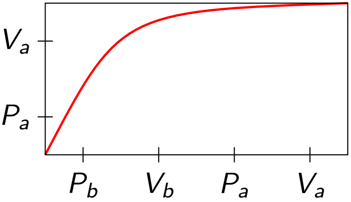
A program will be interpreted as a **directed space**:

- $P_b.V_b.P_a.V_a$



- $P_a.V_a$

- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$

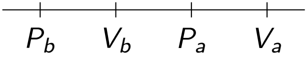


$P_a.P_b.V_a.V_b.P_a.V_a$

Geometric semantics

A program will be interpreted as a **directed space**:

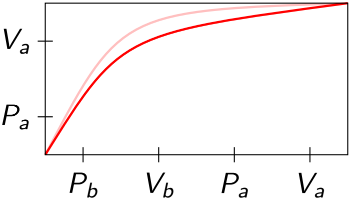
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$

- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$

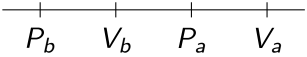
Homotopy



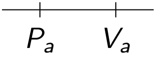
$P_a.P_b.V_a.V_b.P_a.V_a$

A program will be interpreted as a **directed space**:

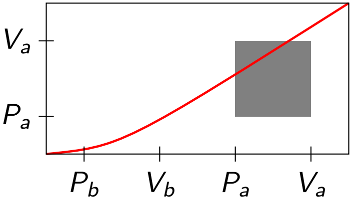
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$



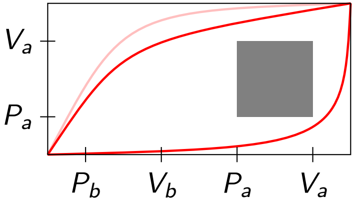
- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$



$P_b.V_b.P_a.P_a.V_a.V_a$ Forbidden region

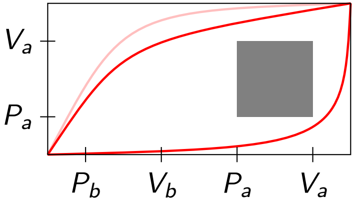
Trace

A **trace** is the homotopy class of a path.



Trace

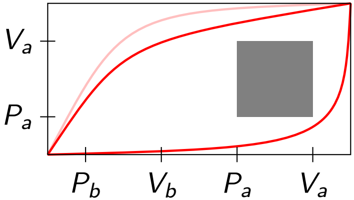
A **trace** is the homotopy class of a path.



We want to compute *a path in every trace*

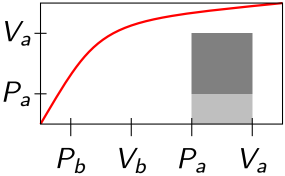
Trace

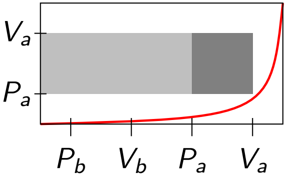
A **trace** is the homotopy class of a path.



We want to compute *a path in every trace*

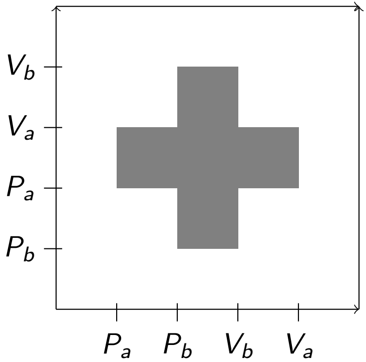
We do this by testing possible ways to go around forbidden regions:





The Swiss flag

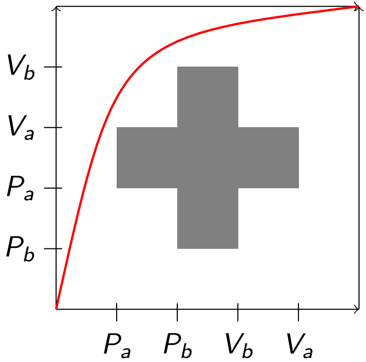
$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



A forbidden region

The Swiss flag

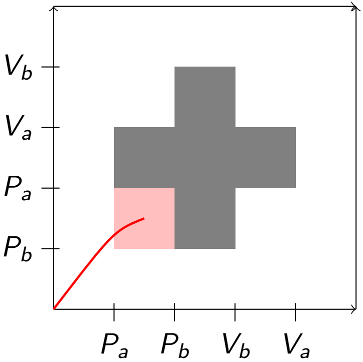
$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



A path: $P_b.P_a.V_a.P_a.V_b.P_b.V_b.V_a$

The Swiss flag

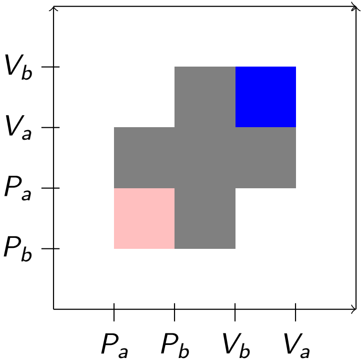
$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



A deadlock: $P_b.P_a$

The Swiss flag

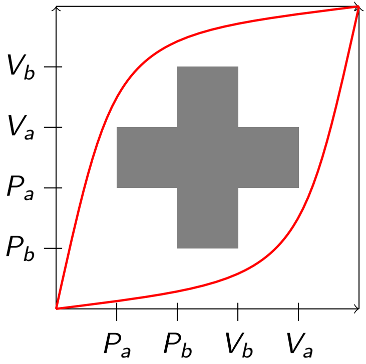
$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$



An unreachable region

The Swiss flag

$$P_a.P_b.V_b.V_a \quad | \quad P_b.P_a.V_a.V_b$$



Here we are interested in maximal paths modulo homotopy

Plan

1. Trace semantics of programs
2. Geometric semantics of programs
3. Computation of the trace space

Programs

We consider programs of the form:

$$p \quad ::= \quad \mathbf{1} \quad | \quad P_a \quad | \quad V_a \quad | \quad p.p \quad | \quad p|p \quad | \quad p+p \quad | \quad p^*$$

Programs

We consider programs of the form:

$$p \quad ::= \quad \mathbf{1} \quad | \quad P_a \quad | \quad V_a \quad | \quad p.p \quad | \quad p|p$$

We omit non-deterministic choice, loops

Programs

We consider programs of the form:

$$p \quad ::= \quad \mathbf{1} \quad | \quad P_a \quad | \quad V_a \quad | \quad p.p \quad | \quad p|p$$

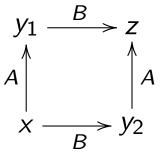
We omit non-deterministic choice, loops and thread creation:

$$\begin{array}{llll} A & ::= & P_a \quad | \quad V_a & \textit{actions} \\ t & ::= & A.t \quad | \quad \mathbf{1} & \textit{threads} \\ p & ::= & t|t|\ldots|t & \textit{programs} \end{array}$$

Trace semantics

The trace semantics of a program will be an **asynchronous graph**:

- a graph $G = (V, E)$ labeled by actions
- with an *independence relation* I

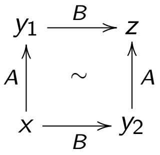


relating paths of length 2

Trace semantics

The trace semantics of a program will be an **asynchronous graph**:

- a graph $G = (V, E)$ labeled by actions
- with an *independence relation* I

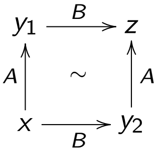


relating paths of length 2

Trace semantics

The trace semantics of a program will be an **asynchronous graph**:

- a graph $G = (V, E)$ labeled by actions
- with an *independence relation* I



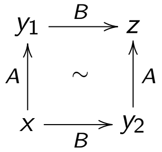
relating paths of length 2

Homotopy is the smallest congruence on paths containing I.

Trace semantics

The trace semantics of a program will be an **asynchronous graph**:

- a graph $G = (V, E)$ labeled by actions
- with an *independence relation* I



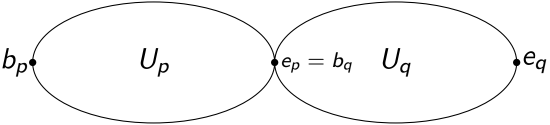
relating paths of length 2

- together with a beginning and an end vertex

Homotopy is the smallest congruence on paths containing I.

To every program p we associate (U_p, b_p, e_p) defined by:

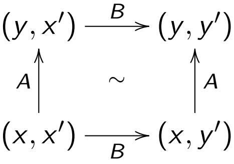
- $U_{\mathbf{1}}$: terminal graph
- U_{P_a}: $b_{P_a} \xrightarrow{P_a} e_{P_a}$ $\qquad$ U_{V_a}: $b_{P_a} \xrightarrow{V_a} e_{V_a}$
- $U_{p.q}$:



- $U_{p|q}$ is the "cartesian product" of U_p and U_q:

$$(x, y) \xrightarrow{A} (x', y) \qquad \text{when } \; x \xrightarrow{A} x' \; \in U_p$$

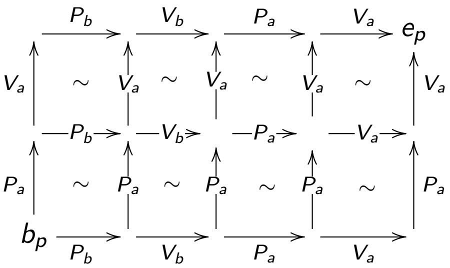
$$(x, y') \xrightarrow{B} (x, y') \qquad \text{when } \; y \xrightarrow{B} y' \; \in U_q$$



Trace semantics

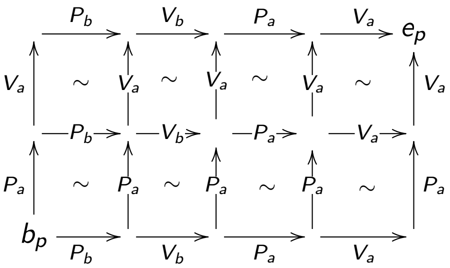
Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$



Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$



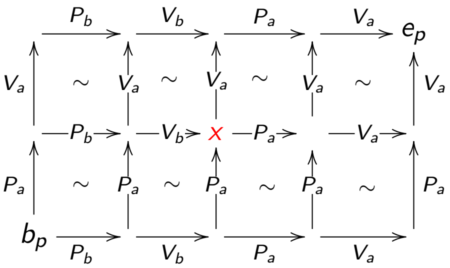
The **resource function** r_a associates to every vertex x:

number of releases of a - number locks of a

Trace semantics

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$



The **resource function** r_a associates to every vertex x:

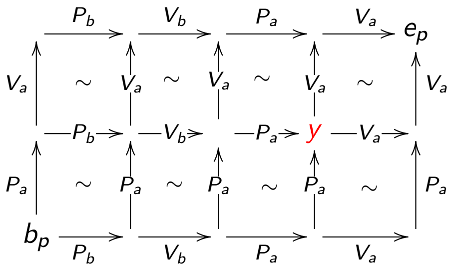
number of releases of a - number locks of a

Ex: $r_a(x) = -1$, $r_b(x) = 0$

Trace semantics

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$



The **resource function** r_a associates to every vertex x:

number of releases of a - number locks of a

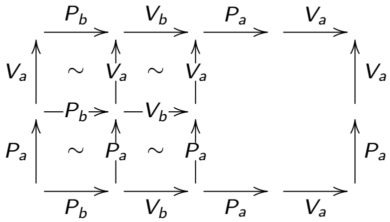
Ex: $r_a(y) = -2$, $r_b(y) = 0$

Trace semantics T_p:
U_p where we remove vertices x which do not satisfy

$$-1 \leqslant r_a(x) \leqslant 0$$

Example:

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$$



Geometric semantics

The trace semantics is difficult to use to build intuitions...

In a similar way, one can define a **geometric semantics** where programs are interpreted by *directed spaces*.

Geometric semantics

A **path** in a topological space X is a continuous map $I = [0, 1] \to X$.

Definition

A **d-space** (X, dX) consists of

- a topological space X
- a set dX of paths in X, called *directed paths*, such that
  - *constant paths*: every constant path is directed,
  - *reparametrization*: dX is closed under precomposition with increasing maps $I \to I$, which are called *reparametrizations*,
  - *concatenation*: dX is closed under concatenation.

Geometric semantics

A **path** in a topological space X is a continuous map $I = [0,1] \to X$.

Definition

A **d-space** (X, dX) consists of

- a topological space X
- a set dX of paths in X, called *directed paths*, such that
  - *constant paths*: every constant path is directed,
  - *reparametrization*: dX is closed under precomposition with increasing maps $I \to I$, which are called *reparametrizations*,
  - *concatenation*: dX is closed under concatenation.

Example

$(X, \leqslant)$ space with a partial order, $dX = \{\text{increasing maps } I \to X\}$

$\vec{I}$: d-space induced by $[0,1]$

Geometric semantics

A **path** in a topological space X is a continuous map $I = [0,1] \to X$.

Definition

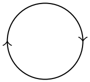
A **d-space** (X, dX) consists of

- a topological space X
- a set dX of paths in X, called *directed paths*, such that
  - *constant paths*: every constant path is directed,
  - *reparametrization*: dX is closed under precomposition with increasing maps $I \to I$, which are called *reparametrizations*,
  - *concatenation*: dX is closed under concatenation.

Example

$S^1 = \{e^{i\theta}\} 0 \leqslant \theta < 2\pi$

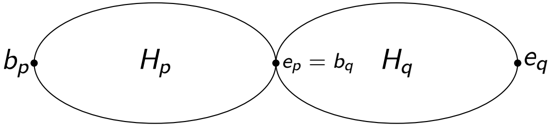
dS^1: $p(t) = e^{i f(t)}$ for some increasing function $f : I \to \mathbb{R}$



Geometric semantics

To each program p we associate a d-space (H_p, b_p, e_p):

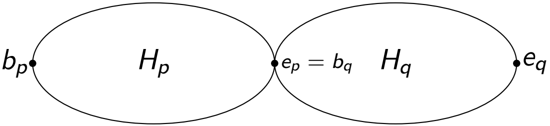
- $H_{\mathbf{1}}$: $\bullet$
- $H_{P_a} = \vec{I}$ $\quad\quad$ $H_{V_a} = \vec{I}$
- $H_{p.q}$:



- $H_{p|q}$: $H_p \times H_q$, $b_{p|q} = (b_p, b_q)$, $e_{p|q} = (e_p, e_q)$

To each program p we associate a d-space (H_p, b_p, e_p):

- $H_{\mathbf{1}}$: $\bullet$
- $H_{P_a} = \vec{I}$ $\quad$ $H_{V_a} = \vec{I}$
- $H_{p.q}$:



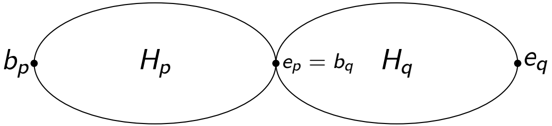
- $H_{p|q}$: $H_p \times H_q$, $b_{p|q} = (b_p, b_q)$, $e_{p|q} = (e_p, e_q)$

Resource function: $r_a(x) \in \mathbb{Z}$ for each $a \in \mathcal{R}$ and point x

Geometric semantics

To each program p we associate a d-space (H_p, b_p, e_p):

- $H_\mathbf{1}$: $\bullet$
- $H_{P_a} = \vec{I}$ $\qquad H_{V_a} = \vec{I}$
- $H_{p.q}$:



- $H_{p|q}$: $H_p \times H_q$, $b_{p|q} = (b_p, b_q)$, $e_{p|q} = (e_p, e_q)$

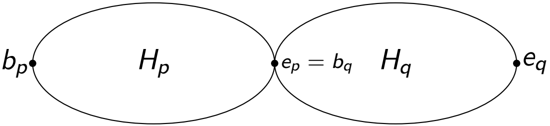
Resource function: $r_a(x) \in \mathbb{Z}$ for each $a \in \mathcal{R}$ and point x

Forbidden region:
$F_p = \{x \in H_p \ / \ \exists a, \quad r_a(x) < -1 \quad \text{or} \quad r_a(x) > 0\}$

Geometric semantics

To each program p we associate a d-space (H_p, b_p, e_p):

- $H_{\mathbf{1}}$: $\bullet$
- $H_{P_a} = \vec{I}$ $\qquad H_{V_a} = \vec{I}$
- $H_{p.q}$:



- $H_{p|q}$: $H_p \times H_q$, $b_{p|q} = (b_p, b_q)$, $e_{p|q} = (e_p, e_q)$

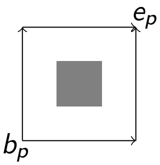
Resource function: $r_a(x) \in \mathbb{Z}$ for each $a \in \mathcal{R}$ and point x

Forbidden region:
$F_p = \{x \in H_p \ / \ \exists a, \quad r_a(x) < -1 \quad \text{or} \quad r_a(x) > 0\}$

Geometric semantics: $G_p = H_p \setminus F_p$

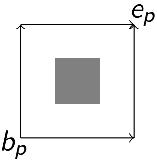
Examples of geometric semantics

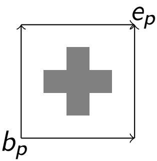
$P_a.V_a|P_a.V_a$



Examples of geometric semantics

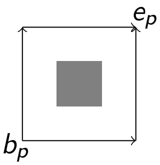
$P_a.V_a|P_a.V_a$ $\quad$ $P_a.P_b.V_b.V_a|P_b.P_a.V_a.V_b$

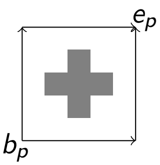


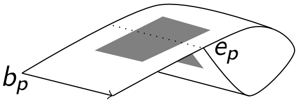


Examples of geometric semantics

$P_a.V_a|P_a.V_a$ $\quad$ $P_a.P_b.V_b.V_a|P_b.P_a.V_a.V_b$ $\quad$ $P_a.(V_a.P_a)^*|P_a.V_a$

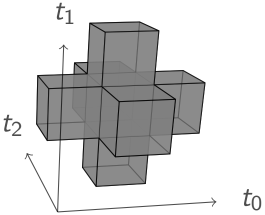




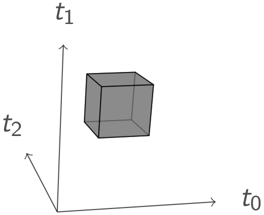


Examples of geometric semantics

$P_a.V_a|P_a.V_a|P_a.V_a$
$(\kappa_a = 1)$



$P_a.V_a|P_a.V_a|P_a.V_a$
$(\kappa_a = 2)$



Geometric realization

The two semantics are "essentially the same": the geometric semantics is the **geometric realization** of a *cubical set*

$$G_p \quad = \quad \int^{n \in \square} T_p(n) \cdot \vec{I}^n$$

Proposition

Given a program p, with T_p as trace semantics and G_p as geometric semantics,

- *every path $\pi : b \to e$ in T_p induces a path $\overline{\pi} : b \to e$ in G_p,*
- *$\pi \sim \rho$ in T_p implies $\overline{\pi} \sim \overline{\rho}$ in G_p*
- *every path ρ of G_p is homotopic to a path $\overline{\pi}$ (π path in G_p)*

Computing the trace space

Goal

Given a program p, we describe an algorithm to compute a trace in each equivalence class of traces $\pi : b_p \to e_p$ *up to homotopy in* G_p.

The algorithm

Suppose given a program

$$p \quad = \quad p_0 | p_1 | \ldots | p_{n-1}$$

with n *threads*.

The algorithm

Suppose given a program

$$p \quad = \quad p_0|p_1|\ldots|p_{n-1}$$

with *n* *threads*.

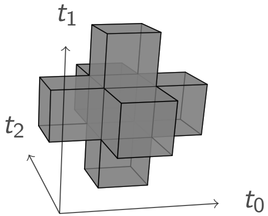
Under mild assumptions, the geometric semantics is of the form

$$G_p \quad = \quad \vec{I}^n \setminus \bigcup_{i=0}^{l-1} R^i$$

where

$$R^i = \prod_{j=0}^{n-1}]x_j^i, y_j^i[$$

are *l* *open rectangles*.



Under mild assumptions, the geometric semantics is of the form

$$G_p \quad = \quad \vec{I}^n \setminus \bigcup_{i=0}^{l-1} R^i$$

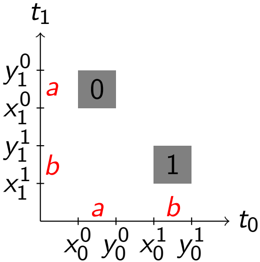
where

$$R^i = \prod_{j=0}^{n-1}]x_j^i, y_j^i[$$

are l *open rectangles*.

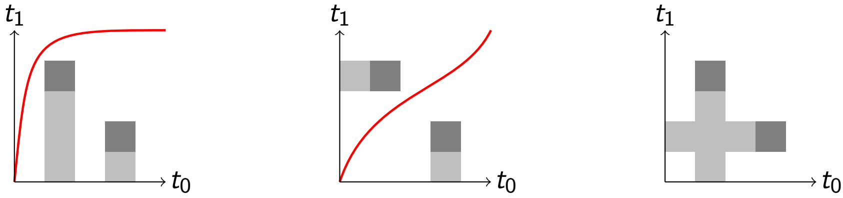
Example

$$P_a.V_a.P_b.V_b | P_b.V_b.P_a.V_a$$



The algorithm

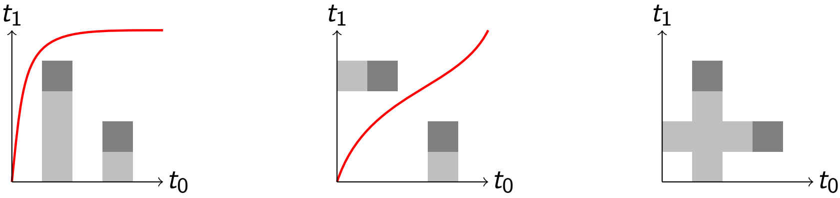
The main idea of the algorithm is to extend the forbidden cubes downwards in various directions and look whether there is a path from b to e in the resulting space.



By combining those information, we will be able to compute traces modulo homotopy.

The algorithm

The main idea of the algorithm is to extend the forbidden cubes downwards in various directions and look whether there is a path from b to e in the resulting space.



By combining those information, we will be able to compute traces modulo homotopy.

The directions in which to extend the holes will be coded by boolean matrices M.

The index poset

$\mathcal{M}_{l,n}$: boolean matrices with l rows and n columns.

The index poset

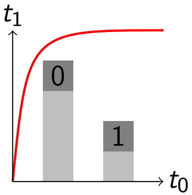
$\mathcal{M}_{l,n}$: boolean matrices with l rows and n columns.

X_M: space obtained by *extending*
for every (i, j) such that $M(i, j) = 1$
the forbidden cube i downwards
in every direction other than j

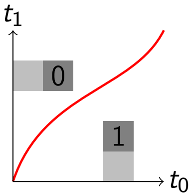
The index poset

$\mathcal{M}_{l,n}$: boolean matrices with l rows and n columns.

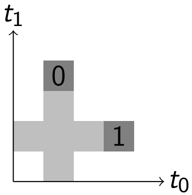
X_M: space obtained by *extending* for every (i,j) such that $M(i,j) = 1$ the forbidden cube i downwards in every direction other than j



$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$$



$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

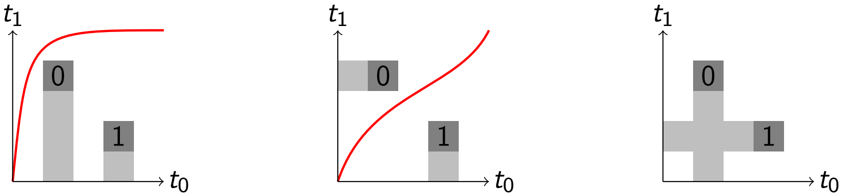


$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The index poset

$\mathcal{M}_{l,n}$: boolean matrices with l rows and n columns.

X_M: space obtained by *extending* for every (i,j) such that $M(i,j) = 1$ the forbidden cube i downwards in every direction other than j



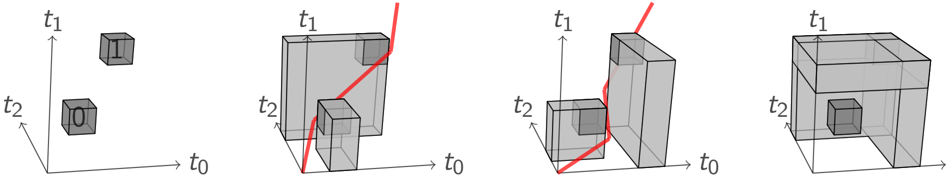
$$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$\Psi : \mathcal{M}_{l,n} \to \{0,1\}$:

- $\Psi(M) = 0$ if there is a path $b \to e$: M is alive
- $\Psi(M) = 1$ if there is no path $b \to e$: M is dead

The index poset

$$P_a.V_a.P_b.V_b \quad | \quad P_a.V_a.P_b.V_b \quad | \quad P_a.V_a.P_b.V_b$$



$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

alive *alive* *alive* *dead*

The index poset

- $\mathcal{M}_{l,n}$ is equipped with the pointwise ordering
- Ψ is increasing: more 1 $\Rightarrow$ more obstructions
- $\mathcal{M}^R_{l,n}$: matrices with non-null rows
- $\mathcal{M}^C_{l,n}$: matrices with unit column vectors

The index poset

- $\mathcal{M}_{l,n}$ is equipped with the pointwise ordering
- Ψ is increasing: more 1 $\Rightarrow$ more obstructions
- $\mathcal{M}^R_{l,n}$: matrices with non-null rows
- $\mathcal{M}^C_{l,n}$: matrices with unit column vectors

Definition

The **index poset** $\mathcal{C}(X) = \{M \in \mathcal{M}^R_{l,n} \ / \ \Psi(M) = 0\}$ (the alive matrices).

The index poset

- $\mathcal{M}_{l,n}$ is equipped with the pointwise ordering
- Ψ is increasing: more 1 $\Rightarrow$ more obstructions
- $\mathcal{M}^R_{l,n}$: matrices with non-null rows
- $\mathcal{M}^C_{l,n}$: matrices with unit column vectors

Definition

The **index poset** $\mathcal{C}(X) = \{M \in \mathcal{M}^R_{l,n} \ / \ \Psi(M) = 0\}$ (the alive matrices).

Definition

The **dead poset** $D(X) = \{M \in \mathcal{M}^C_{l,n} \ / \ \Psi(M) = 1\}$.

The index poset

- $\mathcal{M}_{l,n}$ is equipped with the pointwise ordering
- Ψ is increasing: more 1 $\Rightarrow$ more obstructions
- $\mathcal{M}^R_{l,n}$: matrices with non-null rows
- $\mathcal{M}^C_{l,n}$: matrices with unit column vectors

Definition
The **index poset** $\mathcal{C}(X) = \{M \in \mathcal{M}^R_{l,n} \;/\; \Psi(M) = 0\}$ (the alive matrices).

Definition
The **dead poset** $D(X) = \{M \in \mathcal{M}^C_{l,n} \;/\; \Psi(M) = 1\}$.

$D(X) \quad \rightsquigarrow \quad \mathcal{C}(X) \quad \rightsquigarrow \quad$ homotopy classes of traces

The dead poset

Proposition

A matrix $M \in \mathcal{M}_{l,n}^{C}$ is in $D(X)$ iff it satisfies

$$\forall (i,j) \in [0:l[\times[0:n[, \qquad M(i,j) = 1 \quad \Rightarrow \quad x_j^i < \min_{i' \in R(M)} y_j^{i'}$$

where $R(M)$: indexes of non-null rows of M.

The dead poset

Proposition

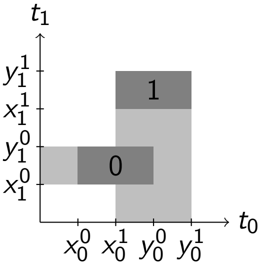
A matrix $M \in \mathcal{M}^C_{l,n}$ is in $D(X)$ iff it satisfies

$$\forall (i,j) \in [0:l[\times[0:n[, \qquad M(i,j) = 1 \quad \Rightarrow \quad x^i_j < \min_{i' \in R(M)} y^{i'}_j$$

where $R(M)$: indexes of non-null rows of M.

Example

M is dead:



$$M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$x^0_1 = 1 < 2 = \min(y^0_1, y^1_1)$$
$$x^1_0 = 2 < 3 = \min(y^0_0, y^1_0)$$

The index poset

Proposition

A matrix M is in $\mathcal{C}(X)$ iff for every $N \in D(X)$, $N \nleq M$.

Proposition

A matrix M is in $\mathcal{C}(X)$ iff for every $N \in D(X)$, $N \nleq M$.

Remark

$N \nleq M$: *there exists* (i,j) *s.t.* $N(i,j) = 1$ *and* $M(i,j) = 0$.

Remark

Since $\mathcal{C}(X)$ is downward closed it will be enough to compute the set $\mathcal{C}_{\max}(X)$ of maximal alive matrices.

Connected components

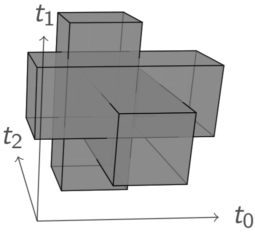
$M \wedge N$: pointwise min of M and N

Definition
Two matrices M and N are **connected** when $M \wedge N$ does not contain any null row.

Proposition
The connected components of $\mathcal{C}(X)$ are in bijection with homotopy classes of traces $b \to e$ in X.

Dining philosophers

n processes p_k in parallel:



$$p_k = P_{a_k}.P_{a_{k+1}}.V_{a_k}.V_{a_{k+1}}$$

n	sched.	ALCOOL (s)	ALCOOL (MB)	SPIN (s)	SPIN (MB)
8	254	0.1	0.8	0.3	12
9	510	0.8	1.4	1.5	41
10	1022	5	4	8	179
11	2046	32	9	42	816
12	4094	227	26	313	3508
13	8190	1681	58	∞	∞
14	16382	13105	143	∞	∞

Handling programs with loops

Consider the following program:

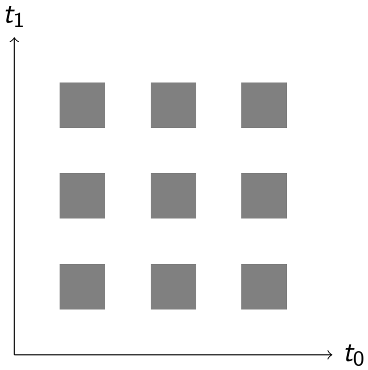
$$p \quad = \quad (P_a.V_a)^*|(P_a.V_a)^*$$

Handling programs with loops

Consider the following program:

$$p \quad = \quad (P_a.V_a)^*|(P_a.V_a)^*$$

Its trace space is

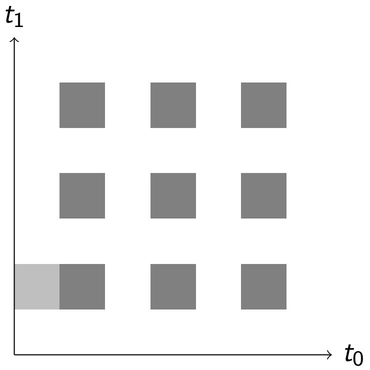


Handling programs with loops

Consider the following program:

$$p = (P_a.V_a)^*|(P_a.V_a)^*$$

Its trace space is

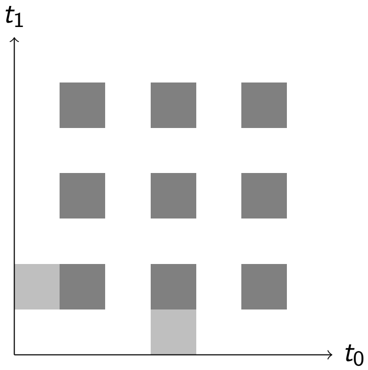


Handling programs with loops

Consider the following program:

$$p \quad = \quad (P_a.V_a)^*|(P_a.V_a)^*$$

Its trace space is

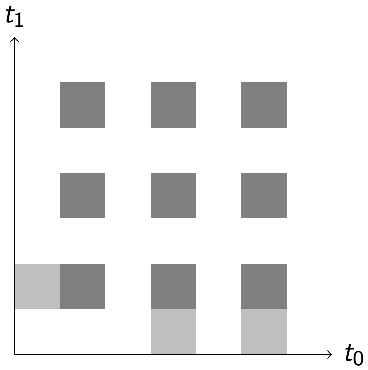


Handling programs with loops

Consider the following program:

$$p \quad = \quad (P_a.V_a)^*|(P_a.V_a)^*$$

Its trace space is

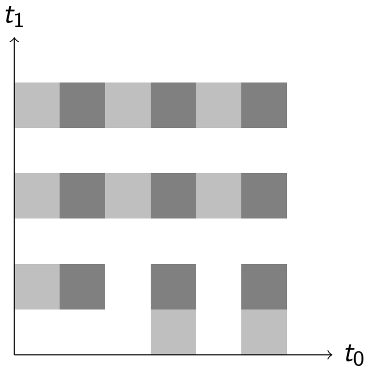


Handling programs with loops

Consider the following program:

$$p \quad = \quad (P_a.V_a)^*|(P_a.V_a)^*$$

Its trace space is

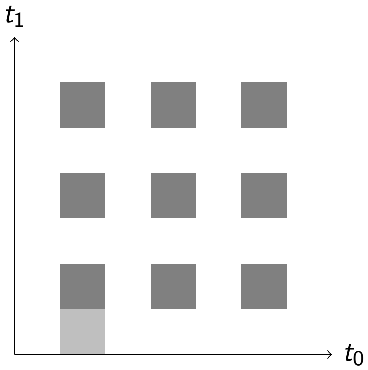


Handling programs with loops

Consider the following program:

$$p \quad = \quad (P_a.V_a)^*|(P_a.V_a)^*$$

Its trace space is

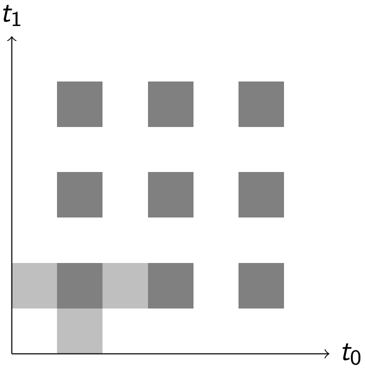


Handling programs with loops

Consider the following program:

$$p \quad = \quad (P_a.V_a)^*|(P_a.V_a)^*$$

Its trace space is

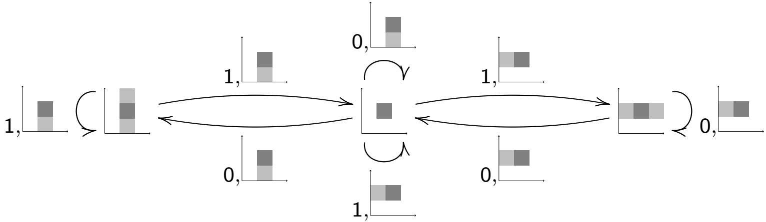


Handling programs with loops

Consider the following program:

$$p \quad = \quad (P_a.V_a)^*|(P_a.V_a)^*$$

Its trace space can be described by the following automaton:



(which can be reduced a bit more)

Summary

- The computation of trace space through boolean matrices is quite efficient
- We compute a "most reduced CFG" which can be then be analyzed through usual techniques (abstract interpretation, model checking, etc.)
- Geometric semantics can be useful in order to reason about concurrency

Future works

- Interface with static analyzers
- Speed and implementation improvements (algorithms, GPU)
- Precise relation with partial-order reduction (joint work with T. Heindel)
- Lots of work remain to be done on the theoretical side in order to really understand the geometry of concurrency

Thanks!

Questions?