Towards Efficient Computation of Trace Spaces of Concurrent Programs

Samuel Mimram

CEA, LIST

Workshop on Computational Topology

Plan

1. Efficient implementation of the computation of the trace space
2. Extension to programs containing loops

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

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

`Homotopy classes of execution traces!`

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

`Homotopy classes of execution traces!`

Joint work with, L. Fajstrup, É. Goubault, E. Haucourt and M. Raussen

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:

$$\begin{tikzcd}
{} \arrow[r, "\texttt{x:=1}"] & {} \arrow[r, "\texttt{y:=2}"] & {} \\
{} \arrow[u, "\texttt{y:=3}"] \arrow[r, "\texttt{x:=1}"'] & {} \arrow[u, "\texttt{y:=3}"] \arrow[r, "\texttt{y:=2}"'] & {} \arrow[u, "\texttt{y:=3}"']
\end{tikzcd}$$

Programs generate trace spaces

Consider the program

$$\texttt{x:=1;y:=2} \quad | \quad \texttt{y:=3}$$

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:

$$\begin{tikzcd}
\cdot \arrow[r, "\texttt{x:=1}"] & \cdot \arrow[r, "\texttt{y:=2}"] & \cdot \\
\cdot \arrow[u, "\texttt{y:=3}"] \arrow[r, "\texttt{x:=1}"'] \arrow[ur, phantom, "\sim"] & \cdot \arrow[u, "\texttt{y:=3}"] \arrow[r, "\texttt{y:=2}"'] & \cdot \arrow[u, "\texttt{y:=3}"']
\end{tikzcd}$$

homotopy: commutation / filled square

Programs generate trace spaces

Consider the program

$$\color{red}{P_a}\,;\texttt{x:=1}\,;\color{red}{V_a}\,;\color{red}{P_b}\,;\texttt{y:=2}\,;\color{red}{V_b} \quad | \quad \color{red}{P_b}\,;\texttt{y:=3}\,;\color{red}{V_b}$$

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:

$$\begin{tikzcd}
{} \arrow[r, "\texttt{x:=1}"] & {} \arrow[r, "\texttt{y:=2}"] & {} \\
{} \arrow[u, "\texttt{y:=3}"] \arrow[r, "\texttt{x:=1}"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "\texttt{y:=3}"] \arrow[r, "\texttt{y:=2}"'] & {} \arrow[u, "\texttt{y:=3}"']
\end{tikzcd}$$

homotopy: commutation / filled square

Programs generate trace spaces

Consider the program

$$P_a \,; \qquad ; V_a \,; P_b \qquad ; V_b \quad | \quad P_b \,; \qquad ; V_b$$

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:

$$\begin{tikzcd}
{} \arrow[r, "\texttt{x:=1}"] & {} \arrow[r, "\texttt{y:=2}"] & {} \\
{} \arrow[u, "\texttt{y:=3}"] \arrow[r, "\texttt{x:=1}"'] \arrow[ur, phantom, "\sim"] & {} \arrow[u, "\texttt{y:=3}"] \arrow[r, "\texttt{y:=2}"'] & {} \arrow[u, "\texttt{y:=3}"']
\end{tikzcd}$$

homotopy: commutation / filled square

We thus consider programs p of the form

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

Geometric semantics

We thus consider programs p of the form

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

Geometric semantics

We thus consider programs p of the form

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

To every program with n threads

$$p \quad = \quad p_1|p_2|\ldots|p_n$$

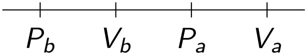
we associate a directed space, its **geometric semantics**:

- an n-dimensional directed cube
- minus l forbidden rectangular cubes (holes)

Geometric semantics

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

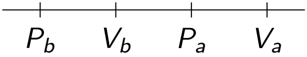
- $P_b.V_b.P_a.V_a$



Geometric semantics

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

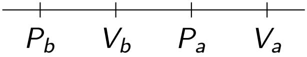
- $P_b.V_b.P_a.V_a$



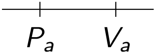
- $P_a.V_a$

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

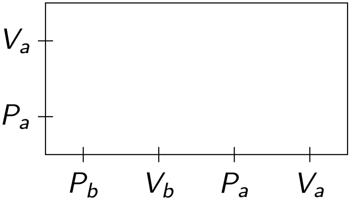
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$



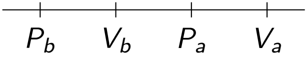
- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$



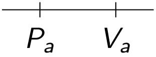
Geometric semantics

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

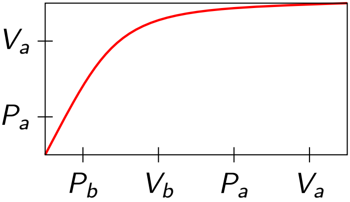
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$



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

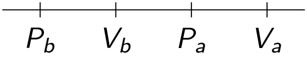


$P_a.P_b.V_a.V_b.P_a.V_a$

Geometric semantics

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

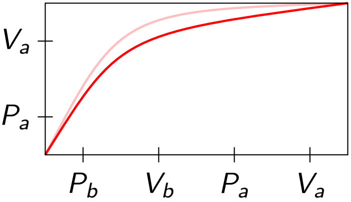
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$

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

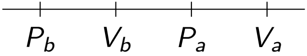
Homotopy



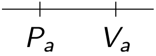
$P_a.P_b.V_a.V_b.P_a.V_a$

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

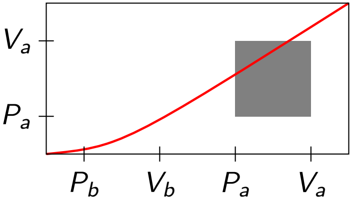
- $P_b.V_b.P_a.V_a$



- $P_a.V_a$

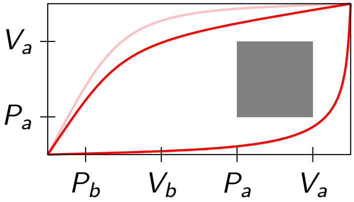


- $P_b.V_b.P_a.V_a \quad | \quad P_a.V_a$



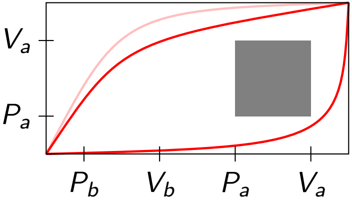
$P_b.V_b.P_a.P_a.V_a.V_a$ Forbidden region

We want to compute one path in every homotopy class:

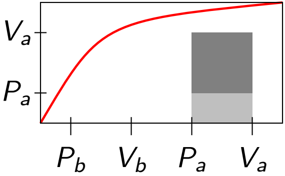


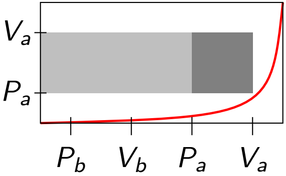
Schedulings

We want to compute one path in every homotopy class:



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

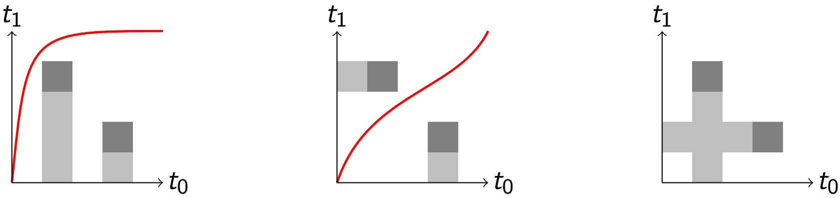




(these are called **schedulings**)

Idea of the algorithm

The main idea of the algorithm is to consider schedulings and look whether there is a path from b to e in the resulting space.



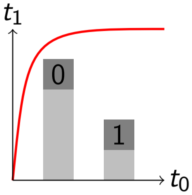
By combining those information, we will be able to compute traces modulo homotopy.

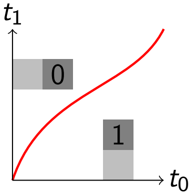
The directions in which to extend the holes will be coded by boolean matrices M.

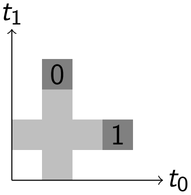
The index poset

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

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





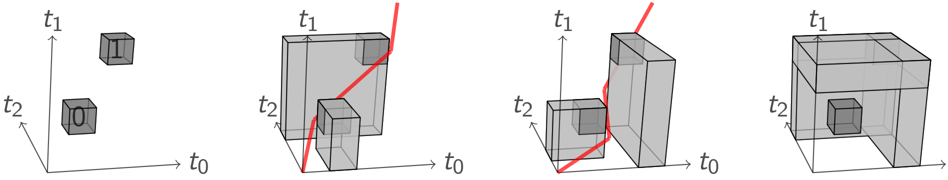


$\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}$ $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

- M is alive if there is a path $b \to e$
- M is dead if there is no path $b \to e$

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 algorithm

The algorithm proceeds as follows:

1. Compute the minimal dead matrices.

The algorithm

The algorithm proceeds as follows:

1. Compute the minimal dead matrices.
2. Deduce the maximal alive matrices.

The algorithm

The algorithm proceeds as follows:

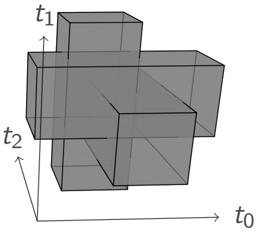
1. Compute the minimal dead matrices.
2. Deduce the maximal alive matrices.
3. The set of maximal alive matrices
quotiented by the *connexity* equivalence relation
is in bijection with homotopy classes of paths!

Definition

Two matrices M and N are **connected** when their intersection $M \wedge N$ does not contain any row filled with zeros.

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	∞	∞

How do we extend this methodology to program with loops?

Loops

Given a thread p, we write p^* for its looping: `while(...){`p`}`.

Loops

Given a thread p, we write p^* for its looping: `while(...){`p`}`.

Given a program p with n threads:

$$p \quad = \quad p_1|p_2|\ldots|p_n$$

we write p^* for

$$p^* \quad = \quad p_1^*|p_2^*|\ldots|p_n^*$$

Loops

Given a thread p, we write p^* for its looping: `while(...){`p`}`.

Given a program p with n threads:

$$p \quad = \quad p_1|p_2|\ldots|p_n$$

we write p^* for

$$p^* \quad = \quad p_1^*|p_2^*|\ldots|p_n^*$$

Notice that the geometric semantics X_{p^*} can be deduced from the semantics of p by glueing copies of X_p in every direction:

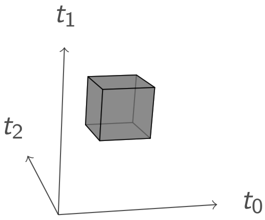
$$p_i^* \quad = \quad p_i.p_i.p_i\ldots$$

Deloopings

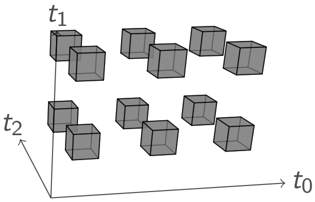
Notice that the geometric semantics X_{p^*} can be deduced from the semantics of p by glueing copies of X_p in every direction.

Example

Consider the program $p = q|q|q$ with $q = P_a.V_a$ (and a of arity 3):



X_p



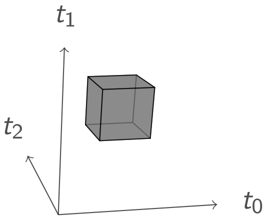
X_{p^*}

Deloopings

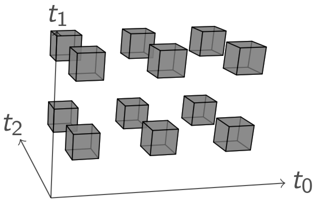
Notice that the geometric semantics X_{p^*} can be deduced from the semantics of p by glueing copies of X_p in every direction.

Example

Consider the program $p = q|q|q$ with $q = P_a.V_a$ (and a of arity 3):



X_p



$X_{p^{(3,2,2)}}$

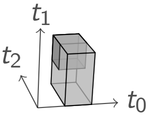
Finite deloopings:

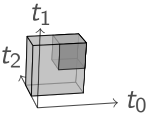
$$X_{p^{(3,2,2)}} \quad = \quad (Y \oplus_1 Y) \oplus_2 (Y \oplus_1 Y) \qquad \text{with} \qquad Y = X_p \oplus_0 X_p \oplus_0 X_p$$

Similarly, given schedulings

$$M = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \qquad \text{and} \qquad N = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$$

of the previous program p

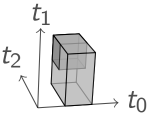


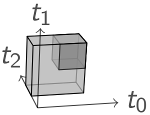


Similarly, given schedulings

$$M = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \qquad \text{and} \qquad N = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$$

of the previous program p

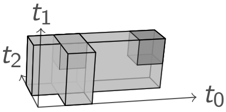




we write

$$M \oplus_0 N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

for the following scheduling of $X_p^{(2,1,1)} = X_p \oplus_0 X_p$

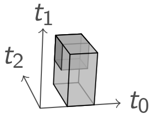
$X_{M \oplus_0 N} =$ 

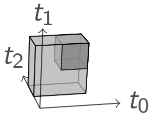
Schedulings

Similarly, given schedulings

$$M = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \qquad \text{and} \qquad N = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$$

of the previous program p



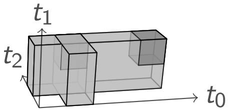


we write

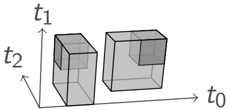
$$M \oplus_0 N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

for the following scheduling of $X_p^{(2,1,1)} = X_p \oplus_0 X_p$

$X_{M \oplus_0 N} =$



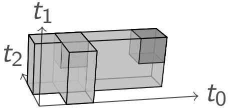
$\neq$

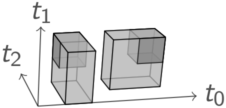


$= X_M \oplus_0 X_N$

Shadows

In fact, scheduling drop "*shadows*" on previous schedulings

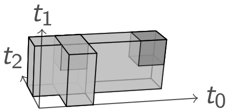
$X_{M \oplus_0 N} =$ 

$\neq$ 

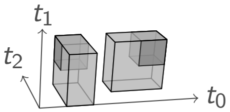
$= X_M \oplus_0 X_N$

In fact, scheduling drop "*shadows*" on previous schedulings

$X_{M\oplus_0 N} =$



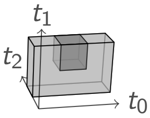
$\neq$



$= X_M \oplus_0 X_N$

Write $X_{M|_j}$ for the **shadow** projected by scheduling M in direction j:

$X_{N|_0} \quad =$



so that

$$X_{M\oplus_j N} \quad = \quad (X_M \cap X_{N|_j}) \otimes_j X_N$$

Alive matrices for programs with loops

Every scheduling M of a delooping of X_p is composed by glueing *submatrices* $(M_{i_1,\ldots,i_n})$.

Alive matrices for programs with loops

Every scheduling M of a delooping of X_p is composed by glueing *submatrices* $(M_{i_1,\ldots,i_n})$.

If X_M contains a deadlock then some subspace $X_{(M_{i_1,\ldots,i_n})}$ contains a deadlock:

Lemma

If a matrix M is alive then all its submatrices are alive.

Alive matrices for programs with loops

Every scheduling M of a delooping of X_p is composed by glueing *submatrices* $(M_{i_1,\ldots,i_n})$.

If X_M contains a deadlock then some subspace $X_{(M_{i_1,\ldots,i_n})}$ contains a deadlock:

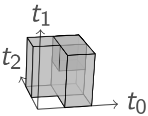
Lemma

If a matrix M is alive then all its submatrices are alive.

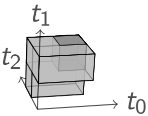
`The converse is not true!`

Shadows can create deadlocks

The following matrices P and Q coding the schedulings



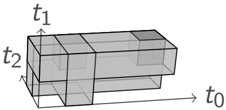
X_P



X_Q

of p are alive, however the matrix $P \oplus_0 Q$ is dead:

$X_{P \oplus_0 Q} \quad =$



The shadow automaton

We construct an automaton which describes all the schedulings possible in the future (which won't create deadlocks by their shadow): given a scheduling M and a direction j, it describes all the matrices N such that $M \oplus_j N$ is alive.

The shadow automaton

Definition

The **shadow automaton** of a program p is a non-deterministic automaton whose

- states are shadows
- transitions $N \xrightarrow{j,M} N'$ are labeled by a direction j (with $0 \leqslant j < n$) and a scheduling M

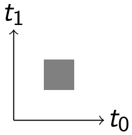
defined as the smallest automaton

- containing the empty scheduling $\emptyset$
- and such that for every state N', for every direction j and for every scheduling M such that the scheduling $M \cup N'$ is alive, and M is maximal with this property, there is a transition $N \xrightarrow{j,M} N'$ with $N = (M \cup N')|_j$.

All the states of the automaton are both initial and final.

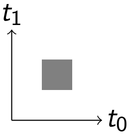
The shadow automaton

For instance consider the program $p = P_a.V_a | P_a.V_a$

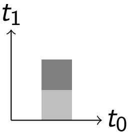
$X_p \quad =$ 

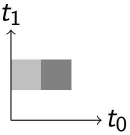
The shadow automaton

For instance consider the program $p = P_a.V_a|P_a.V_a$

$X_p \quad =$ 

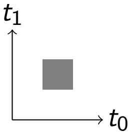
There are two maximal schedulings



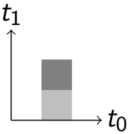


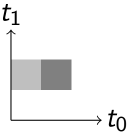
The shadow automaton

For instance consider the program $p = P_a.V_a|P_a.V_a$

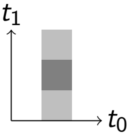
$X_p \quad =$ 

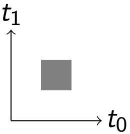
There are two maximal schedulings

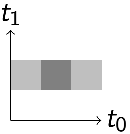




which can drop three possible shadows

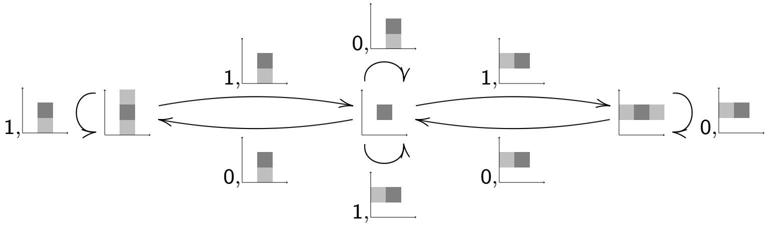






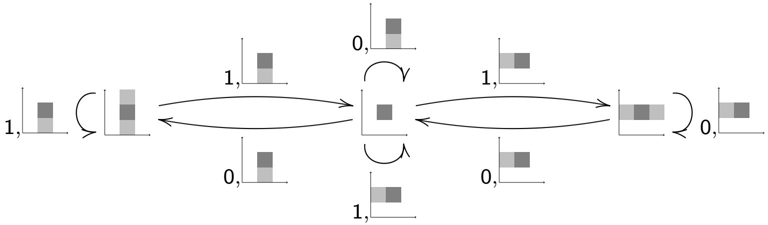
The shadow automaton

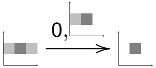
The shadow automaton of p is

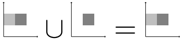


The shadow automaton

The shadow automaton of p is



For instance, the transition  is computed as follows:

- consider the shadow $M =$ 
- compute its shadow in direction 0: 

The shadow automaton

Theorem

Given a program p to any total path in a delooping of p is represented by a path in the shadow automaton of p such that

- *every path in the automaton comes from a total path in* $X_{p^?}$
- *if two total paths in* $X_{p^?}$ *correspond to the same path in the automaton then they are homotopic*

Paths in the shadow automaton describe homotopy classes in deloopings of p.

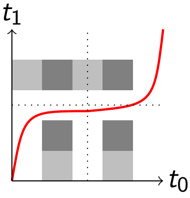
The shadow automaton

Theorem

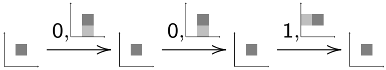
Given a program p to any total path in a delooping of p is represented by a path in the shadow automaton of p such that

- *every path in the automaton comes from a total path in $X_{p^?}$*
- *if two total paths in $X_{p^?}$ correspond to the same path in the automaton then they are homotopic*

Paths in the shadow automaton describe homotopy classes in deloopings of p.



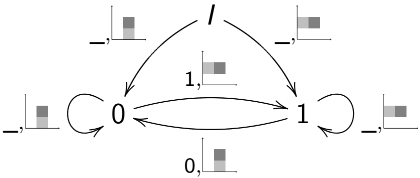
is represented by



Reducing the size of the automaton

The shadow automaton is too big:

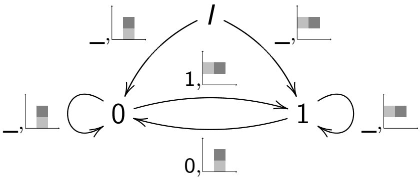
- we can determinize it:



Reducing the size of the automaton

The shadow automaton is too big:

- we can determinize it:



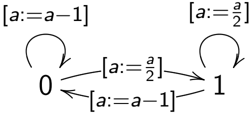
- two distinct paths in the automaton can represent the same homotopy class of paths: we can quotient paths under connexity.

An application to static analysis

The program

$$p^* \quad = \quad \Big(P_a.a := a - 1.V_a\Big)^* \Big| \Big(P_a.\Big(a := \frac{a}{2}\Big).V_a\Big)^*$$

induces the automaton



An application to static analysis

The program

$$p^* \quad = \quad \Big(P_a.a := a - 1.V_a\Big)^* \Big| \Big(P_a.\Big(a := \frac{a}{2}\Big).V_a\Big)^*$$

induces the automaton

$$\begin{tikzcd} 0 \arrow[loop above, "{[a{:=}a-1]}"] \arrow[r, bend left, "{[a{:=}\frac{a}{2}]}"] & 1 \arrow[loop above, "{[a{:=}\frac{a}{2}]}"] \arrow[l, bend left, "{[a{:=}a-1]}"] \end{tikzcd}$$

and thus the set of equations

$$\begin{cases} A_0 & = & I \cup (A_0 - 1) \cup (A_1 - 1) \\ A_1 & = & I \cup \frac{A_1}{2} \cup \frac{A_0}{2} \end{cases}$$

An application to static analysis

The program

$$p^* \quad = \quad \Big(P_a.a := a - 1.V_a\Big)^* \Big| \Big(P_a.\Big(a := \frac{a}{2}\Big).V_a\Big)^*$$

induces the automaton

$$\begin{tikzcd}
0 \arrow[loop above, "{[a{:=}a-1]}"] \arrow[r, bend left, "{[a{:=}\frac{a}{2}]}"] & 1 \arrow[loop above, "{[a{:=}\frac{a}{2}]}"] \arrow[l, bend left, "{[a{:=}a-1]}"]
\end{tikzcd}$$

and thus the set of equations

$$\begin{cases} A_0 & = & I \cup (A_0 - 1) \cup (A_1 - 1) \\ A_1 & = & I \cup \frac{A_1}{2} \cup \frac{A_0}{2} \end{cases}$$

which admits a least fixed point

$$A_0^\infty \quad = \quad A_1^\infty \quad = \quad]-\infty, 1]$$

An example: the two-phase protocol

We consider n programs locking l resources:

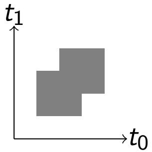
$$p_{n,l} = q|q|\ldots|q \qquad \text{with} \qquad q = P_{a_1}.\ldots.P_{a_l}.V_{a_1}.\ldots.V_{a_l}$$

An example: the two-phase protocol

We consider n programs locking l resources:

$$p_{n,l} = q|q|\dots|q \qquad \text{with} \qquad q = P_{a_1}.\dots.P_{a_l}.V_{a_1}.\dots.V_{a_l}$$

For instance, $p_{2,2} = q|q$ is

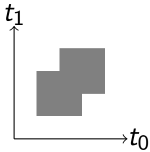


An example: the two-phase protocol

We consider n programs locking l resources:

$$p_{n,l} = q|q|\ldots|q \qquad \text{with} \qquad q = P_{a_1}.\ldots.P_{a_l}.V_{a_1}.\ldots.V_{a_l}$$

For instance, $p_{2,2} = q|q$ is



We get the following results compared to spin:

n	l	s	t	s'	t'	s''	t''	s_{SPIN}	t_{SPIN}
2	1	3	8	3	10	1	1	58	65
2	2	3	8	3	10	1	1	112	129
2	3	3	8	3	10	1	1	180	209
3	1	19	90	4	24	1	1	171	218
3	2	19	90	4	24	1	1	441	602
3	3	19	90	4	24	1	1	817	1128

Conclusion

- Geometric methods can help to devise efficient algorithms to study concurrent programs
- Lots of works remain to be done...