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 \textit{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

\[ x := 1; y := 2 \quad | \quad y := 3 \]

It can be scheduled in three different ways:

\[ y := 3; x := 1; y := 2 \quad x := 1; y := 3; y := 2 \quad x := 1; y := 2; y := 3 \]

Giving rise to the following graph of traces:
Programs generate trace spaces

Consider the program

\[
\begin{align*}
x &:= 1; y := 2 & \mid & y := 3
\end{align*}
\]

It can be scheduled in three different ways:

\[
\begin{align*}
y &:= 3; x := 1; y := 2 & & x := 1; y := 3; y := 2 & & x := 1; y := 2; y := 3 \\
(x, y) & = (1, 2) & (x, y) & = (1, 2) & (x, y) & = (1, 3)
\end{align*}
\]

Giving rise to the following graph of traces:

\[
\begin{array}{c}
\xymatrix@R=2em{\bullet \ar[r] & \bullet \\
\bullet & \bullet & \bullet \\
\bullet \ar[u] \ar[r] & \bullet & \bullet \ar[u] \\
\bullet \ar[u] & \bullet & \bullet \ar[u]}
\end{array}
\]

homotopy: commutation / filled square
**Programs generate trace spaces**

Consider the program

\[
P_a; x := 1; V_a; P_b; y := 2; V_b \quad | \quad P_b; y := 3; V_b
\]

It can be scheduled in three different ways:

\[
\begin{align*}
\text{y := 3; x := 1; y := 2} & \quad (x, y) = (1, 2) \\
\text{y := 3} & \quad \sim \quad \text{y := 3} \\
\text{y := 3; x := 1; y := 2} & \quad (x, y) = (1, 2) \\
\text{y := 3; x := 1; y := 3} & \quad (x, y) = (1, 3)
\end{align*}
\]

Giving rise to the following graph of traces:

\[\text{homotopy: commutation / filled square}\]
Programs generate trace spaces

Consider the program

\[ P_a ; V_a ; P_b \quad ; V_b \mid P_b ; V_b \]

It can be scheduled in three different ways:

\[
\begin{align*}
  y := 3 &; x := 1 &; y := 2 \\
  (x, y) &= (1, 2)
\end{align*}
\]

\[
\begin{align*}
  x := 1 &; y := 3 &; y := 2 \\
  (x, y) &= (1, 2)
\end{align*}
\]

\[
\begin{align*}
  x := 1 &; y := 2 &; y := 3 \\
  (x, y) &= (1, 3)
\end{align*}
\]

Giving rise to the following graph of traces:

\[
\begin{array}{c}
\xymatrix{
& x := 1 \ar[dr] & \ar[dl] y := 2 \\
 y := 3 \ar[rru] & & y := 3 \\
x := 1 \ar[u] & \sim y := 3 & y := 2 \ar[uu]}
\end{array}
\]

homotopy: commutation / filled square
We thus consider programs $p$ of the form

$$p ::= 1 \mid P_a \mid V_a \mid p.p \mid p|p$$
Geometric semantics

We thus consider programs $p$ of the form

\[ p ::= 1 \mid P_a \mid V_a \mid p.p \mid p|p \mid p^* \]
We thus consider programs $p$ of the form

$$p ::= 1 \mid P_a \mid V_a \mid p.p \mid p|p \mid p^*$$

To every program with $n$ threads

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

we associate a directed space, its geometric semantics:

- an $n$-dimensional directed cube
- minus / forbidden rectangular cubes (holes)
Geometric semantics

A program will be interpreted as a directed space:

- $P_b \cdot V_b \cdot P_a \cdot V_a$

\[ P_b \quad V_b \quad P_a \quad V_a \]

Forbidden region
Geometric semantics

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

- $P_b \cdot V_b \cdot P_a \cdot V_a$

\[ P_b \quad V_b \quad P_a \quad V_a \]

- $P_a \cdot V_a$

\[ P_a \quad V_a \]
Geometric semantics

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

- $P_b \cdot V_b \cdot P_a \cdot V_a$

  ![Diagram 1]

- $P_a \cdot V_a$

  ![Diagram 2]

- $P_b \cdot V_b \cdot P_a \cdot V_a$  |  $P_a \cdot V_a$

  ![Diagram 3]
Geometric semantics

A program will be interpreted as a directed space:

- $P_b \cdot V_b \cdot P_a \cdot V_a$

- $P_a \cdot V_a$

- $P_b \cdot V_b \cdot P_a \cdot V_a \mid P_a \cdot V_a$

$P_a \cdot P_b \cdot V_a \cdot V_b \cdot P_a \cdot V_a$
Geometric semantics

A program will be interpreted as a directed space:

- \( P_b \cdot V_b \cdot P_a \cdot V_a \)

\[
\begin{array}{cccc}
P_b & V_b & P_a & V_a \\
\end{array}
\]

- \( P_a \cdot V_a \)

\[
\begin{array}{cc}
P_a & V_a \\
\end{array}
\]

- \( P_b \cdot V_b \cdot P_a \cdot V_a \mid P_a \cdot V_a \) Homotopy

\[
\begin{array}{cccc}
P_b & V_b & P_a & V_a \\
\end{array}
\]

\( P_a \cdot P_b \cdot V_a \cdot V_b \cdot P_a \cdot V_a \)
Geometric semantics

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

- $P_b \cdot V_b \cdot P_a \cdot V_a$

- $P_a \cdot V_a$

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

Forbidden region

$P_b \cdot V_b \cdot P_a \cdot P_a \cdot V_a \cdot V_a$
We want to compute one path in every homotopy class:

\( V_b \)

\( V_a \)

\( P_a \)

\( P_b \)
Schedulings

We want to compute one path in every homotopy class:

\[ \begin{array}{c}
\mathcal{V}_a \\
\mathcal{P}_a \\
\mathcal{P}_b \\
\mathcal{V}_b \\
\mathcal{P}_a \\
\mathcal{V}_a \\
\end{array} \]

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

(these are called **schedulings**)

(These diagrams illustrate the concept of schedulings in the context of homotopy classes, showing how paths can be computed around forbidden regions.)
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 \( j \)

- \( M \) is alive if there is a path \( b \rightarrow e \)
- \( M \) is dead if there is no path \( b \rightarrow e \)
The index poset

\[ P_a \cdot V_a \cdot P_b \cdot V_b \mid P_a \cdot V_a \cdot P_b \cdot V_b \mid P_a \cdot V_a \cdot P_b \cdot V_b \]

\[
\begin{pmatrix}
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
1 & 0 & 0 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
0 & 0 & 1 \\
1 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
0 & 0 & 0 \\
1 & 1 & 1
\end{pmatrix}
\]

alive alive alive dead
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 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.
$n$ processes $p_k$ in parallel:

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>sched.</th>
<th>ALCOOL (s)</th>
<th>ALCOOL (MB)</th>
<th>SPIN (s)</th>
<th>SPIN (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>254</td>
<td>0.1</td>
<td>0.8</td>
<td>0.3</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>510</td>
<td>0.8</td>
<td>1.4</td>
<td>1.5</td>
<td>41</td>
</tr>
<tr>
<td>10</td>
<td>1022</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>179</td>
</tr>
<tr>
<td>11</td>
<td>2046</td>
<td>32</td>
<td>9</td>
<td>42</td>
<td>816</td>
</tr>
<tr>
<td>12</td>
<td>4094</td>
<td>227</td>
<td>26</td>
<td>313</td>
<td>3508</td>
</tr>
<tr>
<td>13</td>
<td>8190</td>
<td>1681</td>
<td>58</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>14</td>
<td>16382</td>
<td>13105</td>
<td>143</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
How do we extend this methodology to program with loops?
Loops

Given a thread $p$, we write $p^*$ for its looping: \texttt{while(...)}{$p$}.
Loops

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

Given a program $p$ with $n$ threads:

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

we write $p^*$ for

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

Given a thread $p$, we write $p^*$ for its looping: $\text{while}(\ldots)\{p\}$.

Given a program $p$ with $n$ threads:

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

we write $p^*$ for

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

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

$$p_i^* = p_i.p_i.p_i\ldots$$
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):

\[ X_p = (X_p \oplus 0) \oplus (Y \oplus 1 Y) \oplus (Y \oplus 2 (Y \oplus 1 Y)) \]

with $Y = X_p \oplus 0 X_p \oplus 0 X_p$.
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 $q = P_a \cdot V_a$ (and $a$ of arity 3):

Finite deloopings:

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

\[ M = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad N = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \]

of the previous program \( p \)
Similarly, given schedulings

\[ M = (1 \ 0 \ 0) \quad \text{and} \quad N = (0 \ 0 \ 1) \]

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 \( X_p^{(2,1,1)} = X_p \oplus_0 X_p \)
Similarly, given schedulings

\[ M = (1 \ 0 \ 0) \quad \text{and} \quad N = (0 \ 0 \ 1) \]

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 \( X_p^{(2,1,1)} = X_p \oplus_0 X_p \)

\[ X_{M \oplus_0 N} = \neq \]

\[ = X_{M \oplus_0 N} \]
Shadows

In fact, scheduling drop “shadows” on previous schedulings

\[ X_{M \oplus_0 N} = \neq = X_{M \oplus_0 X_N} \]
Shadows

In fact, scheduling drop “shadows” on previous schedulings

\[ X_{M \oplus_0 N} = \]

so that

\[ X_{M \oplus_j N} = (X_M \cap X_{N|_j}) \otimes_j X_N \]
Alive matrices for programs with loops

Every scheduling $M$ of a delooping of $X_p$ is composed by glueing submatrices $(M_{i_1},...,i_n)$.
Alive matrices for programs with loops

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

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

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

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

$$X_{P \oplus_0 Q} =$$
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.
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 $0 \leq j < n$) 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 $N = (M \cup N')\upharpoonright j$.

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 \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \q
The shadow automaton

For instance consider the program \( p = P_a \cdot V_a | P_a \cdot V_a \)

\[ X_p = \]

There are two maximal schedulings
The shadow automaton

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

There are two maximal schedulings

which can drop three possible shadows
The shadow automaton of $p$ is

For instance, the transition $0 \rightarrow$ is computed as follows:

- Consider the shadow $M = \cup$
- Compute its shadow in direction 0:
The shadow automaton

The shadow automaton of $p$ is

For instance, the transition $\begin{array}{c} 1, \text{gray} \\ \rightarrow \\ 0, \text{gray} \end{array}$ is computed as follows:

- consider the shadow $M = \begin{array}{c} 1, \text{gray} \\ \cup \\ 0, \text{gray} \end{array} = \begin{array}{c} 1, \text{gray} \end{array}$
- compute its shadow in direction 0: $\begin{array}{c} 0, \text{gray} \end{array}$
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 \( p \).
Reducing the size of the automaton

The shadow automaton is too big:

- we can determinize it:

```
\( \begin{array}{c}
\_ \Downarrow \_ \Downarrow \_ \Downarrow \_ \Downarrow 0
\_ \Uparrow \_ \Uparrow \_ \Uparrow \_ \Uparrow 1
\_ \Leftarrow \_ \Leftarrow \_ \Leftarrow \_ \Leftarrow 0
\_ \Rightarrow \_ \Rightarrow \_ \Rightarrow \_ \Rightarrow 1
\end{array} \)
```
Reducing the size of the automaton

The shadow automaton is too big:

- we can determinize it:

```
\begin{center}
\begin{tikzpicture}
  \node (0) at (0, 0) {0};
  \node (1) at (1, 0) {1};
  \draw [->] (0) to node [above] {0} (1);
  \draw [->] (1) to node [above] {1} (0);
  \draw [->] (0) to node [left] {1} (1);
  \draw [->] (1) to node [right] {1} (0);
  \end{tikzpicture}
\end{center}
```

- 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

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

induces the automaton

![Automaton Diagram]

which admits a least fixed point

$$A_\infty^0 = A_\infty^1 = \ldots$$
An application to static analysis

The program
\[ p^* = (P_a.a := a - 1.V_a)^* \parallel (P_a.(a := \frac{a}{2}).V_a)^* \]

induces the automaton

\[
\begin{array}{c}
\text{[a:=a-1]} \\
0 \quad \Rightarrow \quad \text{[a:=a/2]} \\
\Rightarrow \quad \text{[a:=a-1]} \\
1
\end{array}
\]

and thus the set of equations

\[
\begin{align*}
A_0 &= I \cup (A_0 - 1) \cup (A_1 - 1) \\
A_1 &= I \cup \frac{A_1}{2} \cup \frac{A_0}{2}
\end{align*}
\]
An application to static analysis

The program

\[ p^* = \left( P_a \cdot a := a - 1 \cdot V_a \right)^* \mathbin| \left( P_a \cdot \left( a := \frac{a}{2} \right) \cdot V_a \right)^* \]

induces the automaton

\[
\begin{align*}
[a := a - 1] & \quad \quad \quad \quad [a := \frac{a}{2}] \\
0 & \quad \quad \quad \quad \quad 1
\end{align*}
\]

and thus the set of equations

\[
\begin{align*}
A_0 &= I \cup (A_0 - 1) \cup (A_1 - 1) \\
A_1 &= I \cup \frac{A_1}{2} \cup \frac{A_0}{2}
\end{align*}
\]

which admits a least fixed point

\[
A_0^\infty = A_1^\infty = \left[ -\infty, 1 \right]
\]
An example: the two-phase protocol

We consider $n$ programs locking $l$ resources:

$$p_{n,l} = q|q| \ldots |q \quad \text{with} \quad q = P_{a_1} \ldots P_{a_l} \cdot V_{a_1} \ldots V_{a_l}$$
An example: the two-phase protocol

We consider $n$ programs locking $l$ resources:

$$p_{n,l} = q_1|q_2|\ldots|q_l \quad \text{with} \quad q = P_{a_1}\ldots P_{a_l} \cdot V_{a_1}\ldots V_{a_l}$$

For instance, $p_{2,2} = q_1|q_2$ is
An example: the two-phase protocol

We consider \( n \) programs locking \( l \) resources:

\[
p_{n,l} = q | q | \ldots | q \quad \text{with} \quad q = P_{a_1} \ldots P_{a_l} \cdot V_{a_1} \ldots V_{a_l}
\]

For instance, \( p_{2,2} = q | q \) is

![Diagram](attachment:image.png)

We get the following results compared to spin:

<table>
<thead>
<tr>
<th>( n )</th>
<th>( l )</th>
<th>( s )</th>
<th>( t )</th>
<th>( s' )</th>
<th>( t' )</th>
<th>( s'' )</th>
<th>( t'' )</th>
<th>( s_{\text{SPIN}} )</th>
<th>( t_{\text{SPIN}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>58</td>
<td>65</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>112</td>
<td>129</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>180</td>
<td>209</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>19</td>
<td>90</td>
<td>4</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>171</td>
<td>218</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>19</td>
<td>90</td>
<td>4</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>441</td>
<td>602</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>19</td>
<td>90</td>
<td>4</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>817</td>
<td>1128</td>
</tr>
</tbody>
</table>
Conclusion

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