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

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

[1] Department of Mathematical Sciences, Aalborg University
[2] CEA, LIST*

Abstract. State-space reduction techniques, used primarily in model-checkers, all rely on the idea that some actions are independent, hence could be taken in any (respective) order while put in parallel, without changing the semantics. It is thus not necessary to consider all execution paths in the interleaving semantics of a concurrent program, but rather some equivalence classes. The purpose of this paper is to describe a new algorithm to compute such equivalence classes, and a representative per class, which is based on ideas originating in algebraic topology. We introduce a geometric semantics of concurrent languages, where programs are interpreted as directed topological spaces, and study its properties in order to devise an algorithm for computing dihomotopy classes of execution paths. In particular, our algorithm is able to compute a control-flow graph for concurrent programs, possibly containing loops, which is "as reduced as possible" in the sense that it generates traces modulo equivalence. A preliminary implementation was achieved, showing promising results towards efficient methods to analyze concurrent programs, with very promising results compared to partial-order reduction techniques.

Introduction

Formal verification of concurrent programs is traditionally considered as a difficult problem because it might involve checking all their possible schedulings, in order to verify all the behaviors the programs may exhibit. This is particularly the case for checking for liveness or reachability properties, or in the case of verification methods that imply traversal of some important parts of the graph of execution, such as model-checking [4] and abstract testing [6]. Fortunately, many of the possible executions are equivalent (we say *dihomotopic*) in the sense that one can be obtained from the other by permuting independent instructions, therefore giving rise to the same results. In order to analyze a program, it is thus enough (and much faster) to analyze one representative in each dihomotopy class of execution traces.

We introduce in this paper a new algorithm to reduce the state-space explosion during the analysis of concurrent systems. It is based on former work of some of the authors, most notably [24] where the notion of trace space is introduced and studied, and also builds up considerably on the geometric semantics approach to concurrent systems, as developed in [13]. Some fundamentals of the mathematics involved can be

* This work has been supported by the PANDA ("Parallel and Distributed Analysis", ANR-09-BLAN-0169) French ANR project and by ESF project ACAT.

found in [19]. The main contributions of this article are the following: we develop and improve the algorithms for computing trace spaces of [24] by reformulating them in order to devise an efficient implementation for them, we generalize this algorithm to programs which may contain loops and thus exhibit an infinite number of behaviors, we apply these algorithms to a toy shared-memory language whose semantics is given in the style of [12], but in this paper, formulated in terms of d-spaces [19], and we report on the implementation and experimentation of our algorithms on trace spaces – an industrial case-study using those methods is also detailed in [3].

Stubborn sets [25], sleep sets and persistent sets [15] are among the most popular methods used for diminishing the complexity of model-checking using transition systems; they are in particular used in SPIN [1], with which we compare our work experimentally in Section 2.5. They are based on semantic observations using Petri nets in the first case and Mazurkiewicz trace theory in the other one. We believe that these are special forms of dihomotopy-based reduction as developed in this paper when cast in our geometric framework, using the adjunctions of [18]. Of course, the trace spaces we are computing have some acquaintance with traces as found in trace theory [7]: basically, traces in trace theory are points of trace spaces, and composition of traces modulo dihomotopy is concatenation in trace theory. Trace spaces are more general in that they consider general directed topological spaces and not just partially commutative monoids; they also include all information related to higher-dimensional (di-)homotopy categories, and not just the fundamental category, as in trace theory. Trace spaces are also linked with component categories, introduced by some of the authors [14,17], and connected components of trace spaces can also be computed using the algorithm introduced in [16].

Contents of the paper. We first define formally the programming language we are considering (Section 1.1) as well as an associated geometric semantics, (Section 1.2). We then introduce an algorithm for computing an effective combinatorial representation of trace spaces as well as an efficient implementation of it (Section 2), and extend this algorithm in order to handle program containing loops (Section 3). Finally, we discuss various applications, in particular to static analysis (Section 3.5) and possible extensions of the algorithm and conclude.

1 Geometric semantics of concurrent processes

1.1 A toy shared-memory concurrent language

In this paper, we consider a toy imperative shared-memory concurrent language as grounds for experimentation. In this formalism, a program can be constituted of multiple subprograms which are run in parallel. The environment provides a set of resources $\mathcal{R}$, where each resource $a \in \mathcal{R}$ can be used by at most κ_a subprograms at the same time, the integer $\kappa_a \in \mathbb{N}$ being called the *capacity* of the resource a. In particular, a *mutex* is a resource of capacity 1.

Whenever a program wants to access a resource a, it should acquire a lock by performing the action P_a which allows access to a, if the lock is granted. Once it does not need the resource anymore, the program can release the lock by performing the action V_a, following again the notation set up by Dijkstra [8]. If a subprogram tries to acquire a lock on a resource a when the resource has already been locked κ_a times,

the subprogram is stuck until the resource is released by an other subprogram. In order to be realistic even though simple, the language considered here also comprises a sequential composition operator ., a non-deterministic choice operator $+$ and a loop construct $(-)^*$, with similar semantics as in regular languages (it should be thought as a `while` construct), as well as a parallel composition operator $|$ to launch two subprograms in parallel.

Programs p are defined by the following grammar:

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

Programs are considered modulo a *structural congruence* $\equiv$ which imposes that operators ., $+$ and $|$ are associative and admit $\mathbf{1}$ as neutral element. A *thread* is a program which does not contain the parallel composition operator $|$.

1.2 Geometric semantics

We introduce here a semantics based on (directed) topological spaces. The geometric semantics will allow a different representation of n pairwise independent actions (as the surface of an n-cube) and n truly concurrent actions as the full n-cube.

We denote by $I = [0,1] \subseteq \mathbb{R}$ the standard euclidean interval. A *path* p in a topological space X is a continuous map $p : I \to X$, and the points $p(0)$ and $p(1)$ are respectively called the *source* and *target* of the path. Given two paths p and q such that $p(1) = q(0)$, we define their *concatenation* as the path $p \cdot q$ defined by

$$(p \cdot q)(t) = \begin{cases} p(2t) & \text{if } 0 \leqslant t \leqslant 1/2 \\ q(2t-1) & \text{if } 1/2 \leqslant t \leqslant 1 \end{cases}$$

A topological space can be equipped with a notion of "direction" as follows [19]:

Definition 1. *A* directed topological space *(or* d-space *for short)* $X = (X, dX)$ *consists of a topological space* X *together with a set* dX *of paths in* X *(the* directed paths*) such that*

1. constant paths*: every constant path is directed,*
2. reparametrization*:* dX *is closed under precomposition with (non necessarily surjective) increasing maps* $I \to I$*, which are called* reparametrizations*,*
3. concatenation*:* dX *is closed under concatenation.*

A morphism of d-spaces $f : X \to Y$*, a* directed map*, is a continuous function* $f : X \to Y$ *which preserves directed paths, in the sense that* $f(dX) \subseteq dY$*.*

The category of d-spaces is complete and cocomplete [19]. This allows us to abstractly define some constructions on d-spaces, which extend usual constructions on topological spaces, that we detail here explicitly by describing the associated directed paths.

- The *terminal d-space* $\star$ is the space reduced to one point.
- The *cartesian product* $X \times Y$ of two d-spaces X and Y has $d(X \times Y) = dX \times dY$.
- The *disjoint union* $X \uplus Y$ of two d-spaces X and Y is such that $d(X \uplus Y) = dX \uplus dY$.

– The *amalgamation* $X[x = y]$ of two points x and y in a d-space X is the d-space X where x and y have been identified, together with the expected set of directed paths.
– Given a d-space X and a topological space $Y \subseteq X$, the *subspace* Y can be canonically equipped with a structure of d-space by $dY = \{p \in dX \;/\; p(I) \subseteq Y\}$.

The geometric semantics of a program is defined using those constructions as follows:

Definition 2. *To every program p, we associate a d-space G_p together with a pair of points $b_p, e_p \in G_p$, respectively called* beginning *and* end*, and a* resource function $r_p : \mathcal{R} \times G_p \to \mathbb{Z}$ *which indicates the number of locks the program holds at a given point. The definition of these is done by induction on the structure of p as follows:*

$G_{\mathbf{1}} = \star, \quad b_{\mathbf{1}} = *, \quad e_{\mathbf{1}} = *, \quad r_{\mathbf{1}}(a, x) = 0$	
$G_{P_a} = \vec{I}, \quad b_{P_a} = 0, \quad e_{V_a} = 1,$ $r_{P_a}(b, x) = \begin{cases} -1 & \text{if } b = a \text{ and } x > 0 \\ 0 & \text{if } b \neq a \text{ or } x = 0 \end{cases}$	$G_{V_a} = \vec{I}, \quad b_{V_a} = 0, \quad e_{V_a} = 1,$ $r_{V_a}(b, x) = \begin{cases} 1 & \text{if } b = a \text{ and } x = 1 \\ 0 & \text{if } b \neq a \text{ or } x < 1 \end{cases}$
$G_{p.q} = (G_p \uplus \vec{I} \uplus G_q)[e_p = 0, 1 = b_q],$ $b_{p.q} = b_p, \quad e_{p.q} = e_q,$ $r_{p.q}(a, x) = \begin{cases} r_p(a, x) & \text{if } x \in G_p \\ r_p(a, e_p) + r_q(a, x) & \text{if } x \in G_q \end{cases}$	$G_{p+q} = (G_p \uplus G_q)[b_p = b_q, e_p = e_q],$ $b_{p+q} = b_p, \quad e_{p+q} = e_q,$ $r_{p+q}(a, x) = \begin{cases} r_p(a, x) & \text{if } x \in G_p \\ r_q(a, x) & \text{if } x \in G_q \end{cases}$
$G_{p\|q} = G_p \times G_q,$ $b_{p\|q} = (b_p, b_q), \quad e_{p\|q} = (e_p, e_q),$ $r_{p\|q}(a, (x, y)) = r_p(a, x) + r_q(a, y)$	$G_{p^*} = G_p[b_p = e_p],$ $b_{p^*} = b_p, \quad e_{p^*} = b_p,$ $r_{p^*}(a, x) = r_p(a, x)$

Given a program p, the forbidden region *is the d-space $F_p \subseteq G_p$ defined by*

$$F_p = \{x \in G_p \;/\; \exists a \in \mathcal{R},\ \kappa_a + r_p(a, x) < 0 \ \text{ or } \ r_p(a, x) > 0\}$$

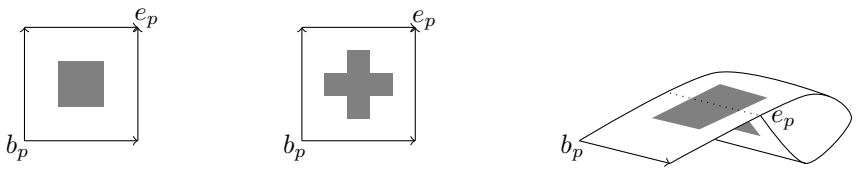
The geometric realization *of a process p, is defined as the d-space $H_p = G_p \setminus F_p$.*

We sometimes write 0 and ∞ for the beginning and the end points respectively of a geometric realization, and say that a path $p : \vec{I} \to G_p$ is *total* when it has 0 as source and ∞ as target. It is easy to show that the geometric semantics of a program is well-defined in the sense that two structurally congruent programs give rise to isomorphic geometric realizations.

Example 1. The processes

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

respectively have the following geometric realizations, which all consist of a space with some "holes", drawn in gray, induced by the forbidden region:



The space in the middle is sometimes called the "Swiss flag" because of its form and is interesting because it exhibits both a deadlock and an unreachable region [13].

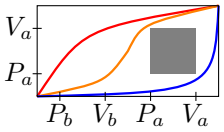
2 Computing trace spaces

2.1 Trace spaces

In topology, two paths p and q are often considered as equivalent when q can be obtained by deforming continuously p (or vice versa), this equivalence relation being called *homotopy*. The corresponding variant of this relation in the case of directed topological spaces is called *dihomotopy* and is formally defined as follows. In the category of d-spaces, the object $\vec{I}$ is *exponentiable*, which means that for every d-space Y, one can associate a d-space $Y^{\vec{I}}$ such that there is a natural bijection between morphisms $X \times \vec{I} \to Y$ and morphisms $X \to Y^{\vec{I}}$. The underlying space of $Y^{\vec{I}}$ is the set of functions $\vec{I} \to Y$ with the compact-open topology (also called uniform convergence topology), and the directed paths $h : \vec{I} \to Y^{\vec{I}}$ are the functions such that $t \mapsto h(t)(u)$ is increasing for every $u \in \vec{I}$. Finally, two paths are said to be dihomotopic when one can be continuously deformed into the other:

Definition 3. *The* dihomotopy *is defined as the smallest equivalence relation on paths such that two directed paths $p, q : \vec{I} \to X$ are dihomotopic when there exists a directed path $h : \vec{I} \to X^{\vec{I}}$ with p as source and q as target.*

Example 2. In the geometric semantics of the program $P_b.V_b.P_a.V_a \mid P_a.V_a$, the two paths above the hole are dihomotopic, whereas the path below is not dihomotopic to the two others:



The intuition underlying the geometric semantics is that two dihomotopic paths correspond to execution traces differing by inessential commutations of instructions, thus giving rise to the same result.

Given two points x and y of a d-space X, we write $X(x, y)$ for the subset of $X^{\vec{I}}$ consisting of dipaths from x to y. A *trace* is the equivalence class of a path modulo surjective reparametrization, and a *scheduling* is the equivalence class of a trace modulo dihomotopy. We write $\vec{T}(X)(x, y)$ for the *trace space* obtained from $X(x, y)$ by identifying paths equivalent up to reparametrization, and simply $\vec{T}(X)$ for $\vec{T}(X)(0, \infty)$. In particular, we have $\vec{T}(X)(x, y) \neq \emptyset$ if and only if there exists a directed path in X going from x to y.

In this section, we reformulate the algorithm for computing the trace space $\vec{T}(X)$ up to dihomotopy equivalence, originally introduced in [24], in order to achieve an efficient implementation of it. For simplicity, we restrict here to spaces which are geometric realizations of programs of the form

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

where the p_i are built up only from $\mathbf{1}$, concatenation, resource locking and resource unlocking (extending the algorithm to programs which may contain loops requires significant generalizations which are described in Section 3). In this case, the geometric realization is of the form

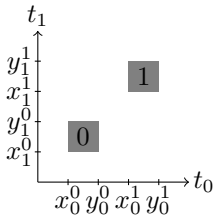
$$G_p \quad = \quad \vec{I}^n \setminus \bigcup_{i=0}^{l-1} R^i$$

where $\vec{I}^n$ denotes the cartesian product of n copies of $\vec{I}$, and each $R^i = \prod_{j=0}^{n-1} \vec{I}_j^i$ is a rectangle. We suppose here that each R^i is homothetic to the n-dimensional open rectangle, i.e. each directed interval $\vec{I}_j^i$ is of the form $\vec{I}_j^i =]x_j^i, y_j^i[$, and generalize this at the end of the section. The restrictions on the form of the programs are introduced here only to simplify our exposition: programs with choice can be handled by computing the trace spaces on each branch and program with loops can be handled by suitably unfolding the loops so that all the possible behaviors are exhibited (a detailed presentation of this is given in Section 3, which will enable to handle the full language). We suppose fixed a program with n threads and l forbidden open rectangles, and consistently use the notations above.

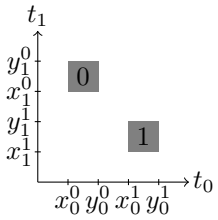
Example 3. The geometric realization of the programs

$$P_a.V_a.P_b.V_b|P_a.V_a.P_b.V_b \qquad \text{and} \qquad P_a.V_a.P_b.V_b|P_b.V_b.P_a.V_a$$

are respectively



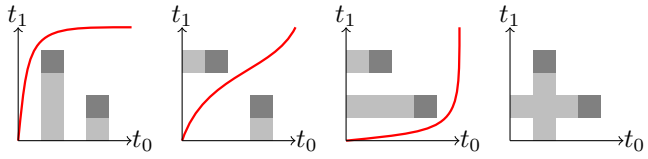
and



2.2 The index poset

Let us come back to the second program of Example 3. We will determine the different traces, and their relationships in the trace space, by combinatorially looking at the way they can turn around holes. To see this in that example, we extend each hole in parallel to the axes, below or leftwards from the holes, until they reach the boundary of the state space. These new obstructions impose traces to go the other way around each hole: the existence of deadlocks, given these new constraints in the trace space allows us to determine whether traces going one way or the other around each hole exist. In fact, this combinatorial information precisely computes all of the trace space [24].

In the second program of Example 3, there are four possibilities to extend once each of the two holes:



(2)

Notice that there exists a total path in the first three spaces (as depicted above), whereas there is none in the last one.

A simple way to encode the combinatorial information about the extension of holes is through boolean matrices. We write $\mathcal{M}_{l,n}$ for the poset of $l \times n$ matrices, with l rows (the number of holes R^i) and n columns (the dimension of the space, i.e. the number of threads in the program), with coefficients in $\mathbb{Z}/2\mathbb{Z}$, with the pointwise ordering such that $0 \leqslant 1$: we have $M \leqslant N$ whenever

$$\forall (i,j) \in [0:l[\times[0:n[, \qquad M(i,j) \leqslant N(i,j) \tag{3}$$

where $[m : n[$ denotes the set $\{m, \ldots, n-1\}$ of integers and $M(i,j)$ denotes the (i,j)-th coefficient of M. We also write $\mathcal{M}^R_{l,n}$ for the subposet of $\mathcal{M}_{l,n}$ consisting of matrices whose row vectors are all different from the zero vector, and $\mathcal{M}^C_{l,n}$ for the subposet of $\mathcal{M}_{l,n}$ consisting of matrices whose column vectors are all unit vectors (containing exactly one coefficient 1).

Given a matrix $M \in \mathcal{M}_{l,n}$, we define X_M as the subspace of X obtained by extending downwards each forbidden rectangle R^i in every direction j' different from j for every j such that $M(i,j) = 1$. Formally,

$$X_M \quad = \quad \vec{I}^n \setminus \bigcup_{M(i,j)=1} \tilde{R}^i_j$$

where $\tilde{R}^i_j = \prod_{j'=0}^{j-1}[0, y^i_{j'}[\times]x^i_j, y^i_j[\times \prod_{j'=j+1}^{n-1}[0, y^i_{j'}[$, see (2) and Example 4 below.

In order to study whether there is a total path in the space associated to a matrix, we define a map $\Psi : \mathcal{M}_{l,n} \to \mathbb{Z}/2\mathbb{Z}$ by $\Psi(M) = 1$ iff $\vec{T}(X_M) = \emptyset$, i.e. there is no total path in X_M. A matrix M is *dead* when $\Psi(M) = 1$ and *alive* otherwise. The map Ψ can easily be shown to be order preserving.

Definition 4. *We write*

$$\mathcal{D}(X) \quad = \quad \{M \in \mathcal{M}^C_{l,n} \,/\, \Psi(M) = 1\}$$

for the set of (column) dead matrices and

$$\mathcal{C}(X) \quad = \quad \{M \in \mathcal{M}^R_{l,n} \,/\, \Psi(M) = 0\}$$

for the set of alive matrices (with non-empty rows), which is called the index poset *– it is implicitly ordered by the relation* (3)*.*

Example 4. In the example above, the three extensions of holes (2) are respectively encoded by the following matrices:

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

The last matrix is dead and the three others are alive. The last matrix being dead indicates that there is no way a trace can pass left of the upper left hole and carry on passing below the lower right hole.

A reason why the matrices in the index poset are convenient objects to study the schedulings is that they are topologically very simple [24]:

Proposition 1. *For any matrix $M \in \mathcal{M}^R_{l,n}$, the space $X_M(x,y)$ is either empty or contractible: any two paths with the same source x and target y are dihomotopic. In particular, for any matrix $M \in \mathcal{C}(X)$, the space $X_M(0,\infty)$ is always contractible.*

Our main interest in the index poset is that it enables us to compute the schedulings (i.e. maximal paths modulo dihomotopy) of the space: these schedulings are in bijection with alive matrices in $\mathcal{C}(X)$ modulo an equivalence relation called *connexity*, which is defined as follows. Given two matrices $M, N \in \mathcal{M}_{l,n}$, their *intersection* $M \wedge N$ is defined as the matrix $M \wedge N$ such that $(M \wedge N)(i,j) = \min(M(i,j), N(i,j))$.

Definition 5. *Two matrices M and N are* connected *when their intersection does not contain any row filled with* 0.

The dihomotopy classes of total paths in X can finally be computed thanks to the following property:

Proposition 2. *The connected components of $\mathcal{C}(X)$ are in bijection with schedulings in X.*

Example 5. Consider the program $p = q|q|q$ where $q = P_a.V_a$. The associated trace space X_p is a cube minus a cube (as shown in Example 8). The matrices in $\mathcal{C}(X_p)$ are

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

and they are all (transitively) connected. For instance, $\begin{pmatrix}0&1&1\end{pmatrix} \wedge \begin{pmatrix}1&0&1\end{pmatrix} = \begin{pmatrix}0&0&1\end{pmatrix}$. The program p thus has exactly one total scheduling, as expected.

Intuitively, alive matrices describe sets of dihomotopic total paths (Proposition 1) and the fact that two matrices have non-zero rows in their intersection means that there are paths which satisfy the constraints imposed by both matrices, i.e. the two matrices describe the same dihomotopy class of total paths.

2.3 Computing dihomotopy classes

The computation of the dihomotopy classes of total paths in the geometric semantics X of a given program will be performed in three steps:

1. we compute the set $\mathcal{D}(X)$ of dead matrices,
2. we use $\mathcal{D}(X)$ to compute the index poset $\mathcal{C}(X)$,
3. we deduce the homotopy classes of total paths by quotienting $\mathcal{C}(X)$ by the connexity relation.

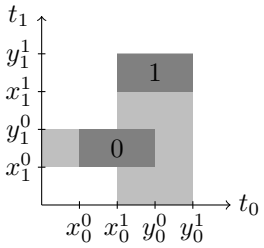
These steps are detailed below.

Given a subset I of $[0:l[$ and an index $j \in [0:n[$, we write $y^I_j = \min\{y^i_j \;/\; i \in I\}$ (by convention $y^\emptyset_j = \infty$). Given a matrix $M \in \mathcal{M}_{l,n}$, we define the set of *non-zero rows* of M by $R(M) = \{i \in [0:l[\;/\; \exists j \in [0:n[,\ M(i,j) \neq 0\}$. It can be shown that a matrix M is dead if and only if the space X_M contains a deadlock. From the characterization of deadlocks in geometric semantics given in [11], the following characterization of dead matrices can therefore be deduced:

Proposition 3. *A matrix $M \in \mathcal{M}_{l,n}^C$ is in $\mathcal{D}(X)$ iff it satisfies*

$$\forall (i,j) \in [0:l[\times[0:n[, \quad M(i,j) = 1 \Rightarrow x_j^i < y_j^{R(M)} \tag{4}$$

Example 6. In the example below with $l = 2$ and $n = 2$, the matrix $M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ is dead (we suppose that $x_j^i = 1 + i(j+1)$ and $y_j^i = 3 + i(j+1) - j$):



$$x_1^0 = 1 < 2 = y_1^{\{0,1\}}$$
$$x_0^1 = 2 < 3 = y_0^{\{0,1\}}$$

The above proposition enables us to compute the set of dead matrices, for instance by enumerating all matrices and checking whether they satisfy condition 4 (a more efficient method is described in Section 2.4). From this set, the index poset $\mathcal{C}(X)$ can be determined using the following property:

Proposition 4. *A matrix $M \in \mathcal{M}_{l,n}$ is not in $\mathcal{C}(X)$ iff there exists a matrix $N \in \mathcal{D}(X)$ such that $N \leqslant M$. In other words, $M \in \mathcal{C}(X)$ iff for every matrix $N \in \mathcal{D}(X)$ there exists indexes $i \in [0:l[$ and $j \in [0:n[$ such that $M(i,j) = 0$ and $N(i,j) = 1$.*

Notice that the poset $\mathcal{C}(X)$ is downward closed (because Ψ is order preserving) and one is naturally interested in the subset $\mathcal{C}_{\max}(X)$ of *maximal* matrices in order to describe it. Proposition 4 provides a simple-minded algorithm for computing (maximal) matrices in $\mathcal{C}(X)$. We write $\mathcal{D}(X) = \{D_0, \ldots, D_{p-1}\}$. We then compute the sets C_k of maximal matrices M such that for every $i \in [0:k[$ we have $D_i \not\leqslant M$. We start from the set $C_0 = \{\mathbf{1}\}$ where $\mathbf{1}$ is the matrix containing only 1 as coefficients. Given a matrix M, we write $M^{\neg(i,j)}$ for the matrix obtained from M by replacing the (i,j)-th coefficient by $1 - M(i,j)$. The set C_{k+1} is then computed from C_k by doing the following for all matrices $M \in C_k$ such that $D_k \leqslant M$:

1. remove M from C_k,
2. for every (i,j) such that $D_k(i,j) = 1$,
   - remove every matrix $N \in C_k$ such that $N \leqslant M^{\neg(i,j)}$,
   - if there exists no matrix $N \in C_k$ such that $M^{\neg(i,j)} \leqslant N$, add $M^{\neg(i,j)}$ to C_k.

The set $\mathcal{C}_{\max}(X)$ is obtained as C_p. If we remove the second point and replace it by

2'. for every (i,j) such that $D_k(i,j) = 1$ and $M^{\neg(i,j)} \in \mathcal{M}_{l,n}^R$, add $M^{\neg(i,j)}$ to C_k.

we compute a set C_p such that $\mathcal{C}_{\max}(X) \subseteq C_p \subseteq \mathcal{C}(X)$, which is enough to compute connected components and has proved faster to compute in practice.

Example 7. Consider again Example 3. The algorithm starts with

$$C_0 \quad = \quad \left\{ M_0 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \right\}$$

For C_1, we must have $D_0 \not\leqslant M_0$ so we swap any of the two ones in the first row:

$$C_1 \quad = \quad \left\{ M_1 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}, M_2 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \right\}$$

Similarly for C_2, we have to swap the bits on the second row so that $D_1 \not\leqslant M_i$:

$$C_2 = \left\{ M_3 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}, M_4 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, M_5 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, M_6 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \right\}$$

Finally, we have $D_2 \not\leqslant M_i$, excepting $D_2 \leqslant M_5$, so we swap the bits in position $(1,1)$ and in position $(2,2)$:

$$M_5' = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \leqslant M_3 \qquad\qquad M_5'' = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \leqslant M_6$$

Since we are only interested in maximal matrices, we end up with $C_3 = \{M_6, M_4, M_3\}$. The trace spaces corresponding to those matrices are the three first depicted in (2). None of those matrices being connected, the trace space up to dihomotopy consists of exactly 3 distinct points.

Other implementations of the algorithm can be obtained by reformulating the computation of $\mathcal{C}_{\max}(X)$ as finding a minimal transversal in a hypergraph, for which efficient algorithms have been proposed [21].

We have supposed up to now that the forbidden region was a union of rectangles R^i, each such rectangle being a product of open intervals $\vec{I}_j^i =]x_j^i, y_j^i[$. The algorithm given above can easily be generalized to the case where the rectangles R^i can "touch the boundary" in some dimensions, i.e. the intervals $\vec{I}_j^i$ are either of the form $]x_j^i, y_j^i[$ or $[0, y_j^i[$ or $]x_j^i, \infty]$ or $[0, \infty]$. For example, the process $P_a.V_a|P_a.V_a|P_a.V_a$, with $\kappa_a = 1$, generates such a forbidden region. We write $B \in \mathcal{M}_{l,n}$ for the *boundary matrix*, which is the matrix such that $B(i,j) = 0$ whenever $x_j^i = 0$ (i.e. the i-th interval touches the lowest boundary in dimension j) and $B(i,j) = 1$ otherwise. The matrices of $\mathcal{D}(X)$ are the matrices $M \in \mathcal{M}_{n,l}$ of the form $M = N \wedge B$, for some matrix $N \in M_{n,l}^C$, which satisfy (4) and such that

$$\forall j \in C(M), \qquad y_j^{R(M)} = \infty \tag{5}$$

where $C(M)$ is the set of indexes of null columns of M.

2.4 An efficient implementation

In order to compute the set $\mathcal{D}(X)$ of dead matrices, the general idea is to enumerate all the matrices $M \in \mathcal{M}_{l,n}^C$ and check whether they satisfy the condition (4). Of course, a direct implementation of this idea would be highly inefficient since there are l^n matrices in $\mathcal{M}_{l,n}^C$. In order to improve this, we try to detect "as soon as possible" when a matrix

```
let rec compute_dead j m rows yrows =
  if j = n then dead := m :: !dead else
    for i = 0 to l − 1 do
      try
        let changed_rows = not (Set.mem i rows) in
        let rows = Set.add i rows in
        let m = Array.copy m in
        if bounds(i,j) = 1 then m.(j) ←None else m.(j) ←Some i;
        (match m.(j) with
            | Some i →if x^i_j ⩾ yrows.(j) then raise Exit
            | None → if yrows.(j) ≠∞ then raise Exit);
        let yrows =
          let j' = j in
          if not changed_rows then yrows else
            Array.mapi (fun j yrj →
                if yrj ⩽ y^i_j then yrj else
                  match m.(j) with
                    | None →
                        if j ⩽ j' && y^i_j≠∞ then raise Exit; y^i_j
                    | Some i →
                        if x^i_j ⩾ y^i_j then raise Exit; y^i_j
                        ) yrows
        in
        compute_dead (j+1) m rows yrows
      with Exit → ()
    done
```

Fig. 1. Algorithm for computing dead matrices.

does not satisfy the condition: we first fix the coefficient in the first column of M and check whether it is possible for a matrix with this first column to be dead, then we fix the second column and so on. In fact, we have to check that every coefficient (i, j) such that $M(i, j) = 1$ satisfies $x^i_j < y^{R(M)}_j$. Now, suppose that we know some of the coefficients (i, j) for which $M(i, j) = 1$. We therefore know a subset $I \subseteq R(M)$ of the non-zero rows. If for one of these coefficients we have $x^i_j \geqslant y^I_j$, we know that the matrix cannot satisfy the condition (4) because $x^i_j \geqslant y^I_j \geqslant y^{R(M)}_j$. A similar reasoning can be held for condition (5).

The actual function computing the dead matrices is presented in Figure 1, in pseudo-OCaml code. This recursive function fills j-th column of the matrix M (whose columns with index below j are supposed to be already fixed) and performs the check: it tries to set the i-th coefficient to 1 (and all the others to 0) for every $i \in [0 : l[$. If a matrix beginning as M (up to the j-th column) cannot be dead, the computation is aborted by raising the Exit exception. When all the columns have been computed the matrix is added to the list $dead$ of dead matrices. Since a matrix $M \in \mathcal{M}^C_{l,n}$ has at most one non-null coefficient in a given column, it will be coded as an array of length n whose j-th element is either None when all the elements of the j-th column are null, or Some i when the i-th coefficient of the j-th column is 1 and the others are 0. The argument $rows$ is the set of indexes of known non-null rows of M and $yrows$ is an array of

length n such that $yrows.(j) = y_j^{rows}$. The matrix $bounds$ is the matrix previously noted B used to perform the check (5). Notice that the algorithm takes advantage of the fact that when the coefficient i chosen for the j-th column is already in $rows$ (i.e. when the variable $changed\_rows$ is false) then many computations can be spared because the coefficients y_j^{rows} are not changed.

Once the set of dead matrices computed, the set $\mathcal{C}(X)$ of alive matrices is then computed using the naive algorithm of Section 2.3, exemplified in Example 7. We have also implemented a simple hypergraph transversal algorithm [2] but it did not bring significant improvements, more elaborate algorithms might give better results though. Finally, the representatives of traces are computed as the connected components (in the sense of Proposition 2) of $\mathcal{C}(X)$, in a straightforward way. An explicit sequence of instructions corresponding to every representative M can easily be computed: it corresponds to the sequence of instructions crossed by any increasing total path in the d-space X_M.

2.5 An example: the n dining philosophers

In order to illustrate the performances of our algorithm, we present below the computation times for the well-known n dining philosophers program [9] whose schedulings are in $O(2^n)$, hence is pushing any algorithm that would determine the essential schedules to its (exponential) limits. It is constituted of n processes p_k in parallel, using n mutexes a_i, defined by $p_k = P_{a_k}.P_{a_{k+1}}.V_{a_k}.V_{a_{k+1}}$, where the indexes on mutexes a_i are taken modulo n. Such a program generates $2^n - 2$ distinct schedulings, which our program finds correctly. The table below summarizes the execution time and memory consumption for our tool ALCOOL (programmed in OCaml), as well as for the model checker SPIN [1] implementing partial order reduction techniques. Whereas SPIN is not significantly slower, it consumes much more memory and starts to use swap from $n = 12$ (thus failing to give an answer in a reasonable time for $n > 12$). Notice that the implementation of SPIN is finely tuned and also benefits from `gcc` optimizations, whereas there is room for many improvements in ALCOOL. In particular, most of the time is spent in computing dead matrices and the algorithm of Section 2.4 could be improved by finding a heuristic to suitably sort holes so that failures to satisfy condition (4) are detected earlier. The present algorithm is also significantly faster than some of the author's previous contribution [16]: for instance, it was unable to generate these maximal dipaths because of memory requirements, for n philosophers with $n > 8$ (in the benchmarks of [16], it was taking already 13739s, on a 1GHz laptop computer though, to generate just the component category for 9 philosophers).

n	sched.	ALC. (s)	ALC. (MB)	SP. (s)	SP. (MB)
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	∞	∞

Since the size of the output is generally exponential in the size of the input, there is no hope to find an algorithm which has less than an exponential worst-case complexity (which our algorithm clearly has). However, since our goal is to program actual tools to very concurrent programs, practical improvements in the execution time or memory

consumption are really interesting from this point of view. We have of course tried our tool on many more examples, which confirm the improvement trend, and shall be presented in a longer version of the article.

3 Programs with loops

3.1 Paths in deloopings

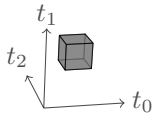
One of the most challenging part of verifying concurrent programs consists in verifying programs with loops since those contain a priori an infinite number of possible execution traces. We extend here the previous methodology and, given a program containing loops, we compute a (finite!) automaton whose accepted paths describe the schedulings of the program: this automaton, can thus be considered as a control flow graph of the concurrent program. Of course, we are then able to use the traditional methods in static analysis, such as abstract interpretation, to study the program (this is briefly presented in Section 3.5). This section builds on some ideas being currently developed by Fajstrup [10], however most of the properties presented in this section are entirely new. To the best of our knowledge, this is the first works in which geometric methods are used in order devise a practical algorithm to handle programs containing loops. A particularly interesting feature of our method lies in the fact that it consider the broad "geometry of holes" and can thus associate a small control flow graph to a given program, see Section 3.4.

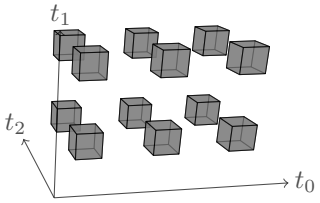
In the following, we suppose fixed a program of the form $p = p_0|p_1|\dots|p_{n-1}$ as in (1), with n threads. We write

$$p^* \quad = \quad p_0^* \mid p_1^* \mid \ \dots \ \mid p_{n-1}^*$$

for the associated "looping program". Our goal in this section is to describe the schedulings of such a program p^* (the restriction on the form of the programs considered here was only done to simplify our presentation and the methodology can be extended to handle all well-bracketed programs generated by the grammar, without any essential technical difficulty added). Following Section 1.2, its geometrical semantics consists of an n-dimensional torus with rectangular holes. As previously, for simplicity, we suppose that these holes do not intersect the boundaries, i.e. that p satisfies the hypothesis of Section 2.1. Given an n-dimensional vector $v = (v_0, \dots, v_{n-1})$ with coefficients in $\mathbb{N}$, the *v-delooping* of p, written p^v, is the program $p_0^{v_0}|p_1^{v_1}|\dots|p_{n-1}^{v_{n-1}}$, where $p_j^{v_j}$ denotes the concatenation of v_j copies of p_j. A *scheduling* in p is a scheduling in the previous sense (i.e. a total path modulo homotopy) in p^v for some vector v.

Example 8. Consider the program $p = q|q|q$ of Example 5, where $q = P_a.V_a$. Its geometric realization X_p is pictured on the left, and its $(3,2,2)$-delooping $X_{p^{(3,2,2)}}$ is pictured on the right.





Given two spaces X and Y which are hypercubes with holes (which is the case for the geometric realizations of the programs we are considering here), we write $X \oplus_j Y$ for the space obtained by identifying the j-th target face of the hypercube X with the j-th source face of the hypercube Y, and call it the *j-gluing* of X and Y. Formally, this can be defined as in Section 1.2 as $X \oplus_j Y = X \uplus Y / \sim$, where the relation $\sim$ identifies points $x \in X$ and $y \in Y$ such that $x_j = \infty$, $y_j = 0$ and $x_{j'} = y_{j'}$ for every dimension $j' \neq j$, and directed paths are defined in a similar fashion. Notice that, by definition, there is a canonical embedding of X (resp. Y) into $X \oplus_j Y$, which will allow us to implicitly consider X (resp. Y) as a subspace of $X \oplus_j Y$ in the following.

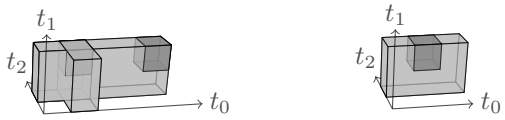
Example 9. The $(3, 2, 2)$-delooping of Example 8 is

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

More generally, any v-delooping p^v of a program p of the form (1) can be obtained by gluing copies X_p^w of X_p, indexed by a vector w such that for every dimension i with $0 \leqslant i < n$, we have $0 \leqslant w_i < v_i$ (what we will simply write $0 \leqslant w < v$).

Given two scheduling matrices M and N encoding extensions of holes of such a program p (cf. Section 2.2), we reuse the notation and write $M \oplus_j N$ for the obvious matrix coding extension of holes in the space $X_p \oplus_j X_p$. At this point, it is crucial to notice that the holes described by N in the second copy of X_p can have an effect on the first copy of X_p (when they are extended to 0 in the direction j), what we call the *j-shadow of N*, and write $X_{N|_j}$.

Example 10. With the program p of Example 8, consider the matrices $M = (1\ 0\ 0)$ and $N = (0\ 0\ 1)$. We have $M \oplus_0 N = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, the space $X_{M \oplus_0 N}$ is pictured on the left, and the 0-shadow $X_{N|_0}$ of N is pictured on the right:



The above example makes clear that the space corresponding to a scheduling $M \oplus_j N$ is of the form $X_{M \oplus_j N} = (X_M \cap X_{N|_j}) \otimes_j X_N$, i.e. the holes in the first copy come either from M or from shadows of N. Moreover, the holes in the space $X_{N|_j}$ are hypercubes which are products of intervals of the form $\prod_{0 \leqslant j < n} \vec{I_j}$, where each interval $\vec{I_j}$ is of the form $]x_j^i, y_j^i[$ or $[0, y_j^i[$ or $[0, \infty]$, with $0 \leqslant i < l$. The shadows can therefore be coded as matrices (using a slightly different coding from the one used up to now, the precise way they are coded being quite irrelevant) and we write $N|_j$ for the matrix coding the j-shadow of n, which can easily be computed from N and j. A scheduling matrix M can obviously be seen as a particular "shadow", enabling us to use the same notation for both, and we write $M \cup N$ for the union of two shadows M and N, so that $X_{M \cup N} = X_M \cap X_N$. Finally, given a shadow M, the algorithm described in Section 2.3 can easily be adapted to the new coding in order to determine whether the space X_M is alive.

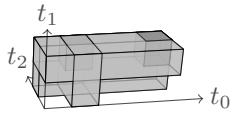
3.2 The shadow automaton

The trace space of a program p^* is not finite in the general case. We show here that it can however be described as the set of paths of an automaton that we call the *shadow automaton*: this automaton provides us with a *finite presentation* of the set of schedulings.

Consider the v-delooping p^v of a program p. The space X_{p^v} consists of the gluing of copies of X_p indexed by vectors w such that $0 \leqslant w < v$ and similarly, a scheduling M of X_{p^v} consists of the gluing of matrices M^w. Clearly, if some submatrix M^w is dead then the whole matrix M is dead:

Lemma 1. *If a matrix M is alive then all its submatrices M^w are alive.*

However, the converse is not true because a scheduling M^w might create a deadlock with the shadows coming from matrices above it. For instance in Example 8, the matrix $M = (1\ 0\ 0) \oplus_0 (0\ 1\ 1)$ is not alive because the space $X_{M^{(0,0,0)}}$ induced by the submatrix $M^{(0,0,0)}$ is contained in the space X_N, where $N = (1\ 1\ 1)$ is a dead matrix:



In order to generate all the possible schedulings M^w visited by a total path in X_{p^v}, we therefore have to take in account the shadows dropped by scheduling of copies of X_p in its future. We will construct an automaton which will consider the visited schedulings of the path, starting from the end, and maintains the shadow they produce on the next state in a given direction j, so that we can compute the possible previous matrices in direction j such that the whole matrix is not dead. Formally,

Definition 6. *The* shadow automaton *of a program p is a non-deterministic automaton whose*

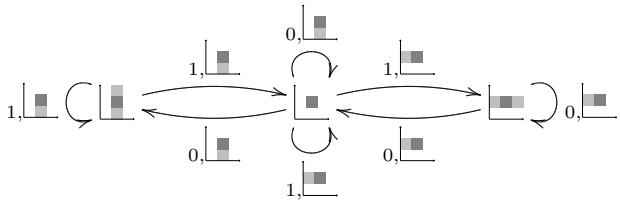
- *states are shadows*
- *transitions $N \xrightarrow{j,M} N'$ are labeled by a direction j (with $0 \leqslant j < n$) and a scheduling M*

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.

Example 11. Consider the program $p = q|q$ with $q = P_a.V_a$ whose geometric semantics is a square with a square hole. The associated shadow automaton is



For instance the transition  is computed as follows: we take the shadow $M =$  and compute its shadow in direction 0, i.e. on the left, to compute the source of the transition. This shadow is  , namely:  .

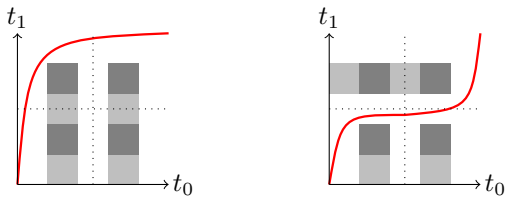
The interest of the automaton lies in the fact that fully describes the possible schedulings crossed by a total path in a scheduling of a delooping X_{p^v}:

Theorem 1. *Suppose that M is a scheduling of X_{p^v}, obtained by gluing schedulings M^w of X_p. Then there exists a total path in X_M going through the subspaces $X_{M^{w_0}}, X_{M^{w_1}}, \dots, X_{M^{w_m}}$ in this order, such that w_k and w_{k+1} only differ by one coordinate j_k (i.e. the path exits from $X_{M^{w_k}}$ through its j_k-th face), if and only if there exists a path labeled as follows in the shadow automaton:*

$$N_0 \xrightarrow{j,M^{w_0}} N_1 \xrightarrow{j_0,M^{w_1}} N_2 \quad \cdots \quad N_m \xrightarrow{j_{m-1},M^{w_m}} N_{m+1}$$

for some states N_i and dimension j.

Example 12. With the program p of Example 11, the following paths in the $(2,2)$-delooping



are respectively witnessed by the following paths of the shadow automaton:

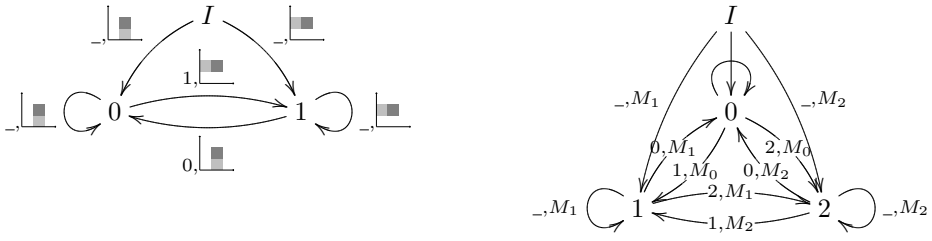
3.3 Reducing the size of the shadow automaton

The size of the shadow automaton grows very quickly when the complexity of the trace space grows. For instance, for the program p of Example 8, the shadow automaton has already 19 states and 80 transitions. We describe here some ways to reduce the automaton while preserving Theorem 1. Namely, we should remark that the automaton is not minimal in the following sense. By Proposition 1, given a scheduling M two total paths X_M are necessarily homotopic: an alive scheduling thus describes an homotopy

class of total paths. By Theorem 1, the schedulings "visited" by a total path in X_{p^v} are described by a path in the shadow automaton, therefore every homotopy class of total paths in X_{p^v} is described by at least one path in the scheduling automaton. The shadow automaton is not minimal in the sense that generally, an homotopy class is described by more than one path in the scheduling automaton.

Determinization. First, our non-deterministic automaton can be determinized using classical algorithms of automata theory, which in practice greatly reduce their size: the determinized automaton for the program of Example 8 has only 4 states and 24 transitions.

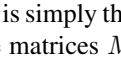
Example 13. The determinized automata for Examples 11 and 8 are respectively:



with $M_0 =$  $M_1 =$  $M_2 =$ 

where "_" means any direction j. The state I is initial and all the states are final.

Quotient under connexity. A way to further reduce the automaton consists in quotienting the scheduling matrices labeling the arrows of the automaton under the connexity relation of Definition 5 before determinizing the automaton, which is formally justified by Proposition 2.

Example 14. The shadow automaton corresponding to the program Example 8 quotiented under connexity, determinized and minimized is simply the automaton 

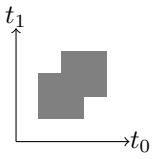
where $M = M_1 = M_2 = M_3$ up to connexity (the matrices M_i are those defined in Example 13).

We are currently investigating further conditions in order to construct the minimal automaton describing the trace space associated to a looping program, but the conditions mentioned above are already providing us with promisingly small automata.

3.4 Preliminary implementation and benchmark

A preliminary implementation of the computation of the shadow automaton was done. The algorithm implemented is currently quite simple, but we plan to generalize the algorithm of Section 2.4 soon, which is not complicated from a theoretical point of view but much more involved technically, in order to achieve better performances. Most experiments lead so far are already promising and make it clear that taking in account the geometry of the state-space enables us to reduce, sometimes drastically, the size of the control flow graph corresponding to the program to be analyzed.

Example 15. The *two-phase locking protocol* is a simple discipline for distributed databases, in which the processes first lock all the mutexes for the resources they are going to use and free all of them in the end [20]. This can be modeled as a program $q_{n,l}$ consisting of n copies of the process $p = P_{a_1} \dots P_{a_l} . V_{a_1} \dots V_{a_l}$ in parallel (each of these process is using l resources). For instance, the geometric semantics of $q_{2,2} = p|p$ is shown below. Notice that this state space is equivalent to a space with only one hole up to dihomotopy. More generally, given $l \geqslant 1$, it can be shown that the geometric semantics of $q_{n,l}$ is equivalent to $q_{n,1}$, which our algorithm is able to take into account! Namely, the size of the shadow automaton associated to $q^*_{n,l}$ only depends on n whereas the number of states of the automaton produced by SPIN is exponential in l (with n fixed). Below are presented the size (states, transitions) of the non-deterministic automaton (s, t), determinized automaton (s',t') and SPIN's automaton (s_{SPIN}, t_{SPIN}) for the two-phase locking process described in Example 15, for some values of n and l.



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

3.5 An Application to static analysis

Now that we have the reduced shadow automaton, we can explain how one can perform static analysis by *abstract interpretation* [5] on concurrent systems, in an economic way. The systematic design and proof of correctness of such abstract analysis is left for a future article, the aim of this section is to give an intuition why the computations of Section 3 are relevant to static analysis by abstract interpretation. The idea is to associate, to each node n of the shadow automaton, a set of values A_n that program variables can take if computation follows a transition path whose last vertex is n. Among the actions the program can take along this scheduling, we consider only the *greedy* ones, that is the ones which execute all possible actions permitted by the dihomotopy class of schedulings ending by n.

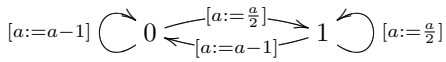
Suppose that we want to analyze the program

$$p^* \quad = \quad \Big(P_a . (a := a - 1) . V_a\Big)^* \Big| \Big(P_a . \Big(a := \frac{a}{2}\Big) . V_a\Big)^* \tag{6}$$

What are the possible sets of values reached, for a, starting with $a \in [0, 1]$? The associated shadow automaton S_p has been determined in Example 13 (this automaton is reduced) together with relations, that we will not be using in this article, yet. In many ways, this reduced shadow automaton plays the role of a compact *control flow graph* for the program we are analyzing. Calling $M_0 =$  and $M_1 =$  , X_{M_0} has the effect on environment: $a := a/2$ and X_{M_1} has as effect: $a := a - 1$.

We are now in a position to interpret the arrows of the shadow automaton as simple *abstract transfer functions* and produce a system of equations for which we want to

determine a least-fixed point, to get the invariant of the program at the (multi-)control point which is the pair of the heads of the loops of each process. The interpretation on the shadow automaton now gives (ignoring the initial state I in that picture, for simplicity's sake) can be graphically pictured as:



Given the abstract transfer functions on each edge of the shadow automaton, we produce as customary the abstract semantic equations, one per node, by joining all transfer functions correspond to ingoing edges to that node:

$$\begin{pmatrix} A_0 \\ A_1 \end{pmatrix} \quad = \quad F \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} \quad = \quad \begin{pmatrix} I \cup (A_0 - 1) \cup (A_1 - 1) \\ I \cup \frac{A_1}{2} \cup \frac{A_0}{2} \end{pmatrix} \tag{7}$$

This set of semantic equations can be seen as a least-fixed point equation, that we can solve using any of our favorite tool, for instance Kleene iteration and widening/narrowing, on any abstract domain, such as the domain of intervals as in the example below. The least-fixed point formulation that we are looking for is thus $A^\infty = \bigvee_{[0,1]} F$, where F is the function defined in (7) and $I = [0, 1]$. A Kleene iteration on this monotonic function F on the lattice of intervals over $\mathbb{R}$ reveals that $A_0^\infty = A_1^\infty =]-\infty, 1]$.

We have presented this example in order to show how the reduced shadow automaton can be used in order to use usual static analysis methods on concurrent programs, avoiding state-space explosion as much as possible. It has the advantage of being short, however it does not really show the main interest of our technique: the scheduling automaton allows us to take in account properties which tightly depend on the way the synchronizations constraint the executions of the programs.

4 Conclusion and Future work

We have presented an algorithm in order to compute a finite presentation of the trace space of concurrent programs, which may contain loops. An application to abstract interpretation has also described but remains to be implemented. In order to give a simple presentation of the algorithm, we have restricted ourselves here to programs of a simple form (in particular, we have omitted non-determinism). We shall extend our algorithm to more realistic programming languages in a subsequent article. Our approach can also be applied to languages with other synchronization primitives (monitors, send/recv, etc.), for which there are simple geometric semantics available. There are also many possible general improvements of the algorithm; the most appealing one would perhaps be to find a way to have a more modular way of computing the total schedulings by combining locally computed schedulings in $\vec{T}(X)(x, y)$ with varying endpoints x and y. In a near future, the schedulings provided by the algorithm will be used by our tool ALCOOL to analyze concurrent programs using abstract interpretation, thus providing one of the first tools able to do such a static analysis on concurrent programs without forgetting most of the possible synchronizations during their execution.

On the theoretical side, we envisage to study in details and use the structure of the index poset $\mathcal{C}(X)$ which contains much more information than only the schedulings of the program. Namely, it can be equipped with a structure of *prodsimplicial set* [22]

(a structure similar to simplicial sets but whose elements are products of simplexes), whose geometric realization provides a topological space which is homotopy equivalent to the trace space $\vec{T}(X)$ [24]. This essentially means that $\mathcal{C}(X)$ contains all the geometry of the trace space and we plan to try to benefit from all the information it provides about the possible computations of a program. Our ALCOOL prototype actually implements this computation – using a combinatorial presentation of the prodsimplicial sets known as *simploidal sets* [23] – which will be reported elsewhere.

References

1. The SPIN Model-Checker. `http://spinroot.com/`.
2. C. Berge. *Hypergraphs*, volume 445. North Holland Mathematical Library, 1989.
3. R. Bonichon and al. Rigorous evidence of freedom from concurrency faults in industrial control software. In *Comp. Safety, Reliab. & Secu.*, volume 6894 of *LNCS*, pages 85–98. 2011.
4. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of Principles Of Programming Languages*, pages 269–282. ACM Press, 1979.
6. P. Cousot and R. Cousot. Abstract interpretation based program testing. In *Proc. of the SSGRR 2000 Computer & eBusiness International Conference*, 2000.
7. V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific, 1995.
8. E. Dijkstra. The Structure of the THE Operating System. *Com. of the ACM*, 11(15), 1968.
9. E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2), 1971.
10. L. Fajstrup. Trace spaces of directed tori with rectangular holes. Technical Report R-2011-08, Aalborg Univ., 2001.
11. L. Fajstrup, E. Goubault, and M. Raußen. Detecting deadlocks in concurrent systems. *CONCUR'98 Concurrency Theory*, pages 332–347, 1998.
12. L. Fajstrup and S. Sokolowski. Infinitely running concurrent processes with loops from a geometric viewpoint. *ENTCS*, 39(2), 2000.
13. Lisbeth Fajstrup, Martin Raußen, and Eric Goubault. Algebraic topology and concurrency. *Theor. Comput. Sci.*, 357(1-3):241–278, 2006.
14. Lisbeth Fajstrup, Martin Raußen, Eric Goubault, and Emmanuel Haucourt. Components of the fundamental category. *Appl. Cat. Struct.*, 12(1):81–108, 2004.
15. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. of CAV*, volume 575, pages 417–428. LNCS, 1991.
16. Eric Goubault and Emmanuel Haucourt. A practical application of geometric semantics to static analysis of concurrent programs. In *CONCUR*, pages 503–517, 2005.
17. Eric Goubault and Emmanuel Haucourt. Components of the fundamental category II. *Applied Categorical Structures*, 15(4):387–414, 2007.
18. Eric Goubault and Samuel Mimram. Formal relationships between geometrical and classical models for concurrency. *CoRR*, abs/1004.2818, 2010.
19. Marco Grandis. *Directed Algebraic Topology, Models of Non-Reversible Worlds*. Number 13 in New Mathematical Monographs. Cambridge University Press, 2009.
20. J. Gunawardena. Homotopy and concurrency. *Bulletin of the EATCS*, 54:184–193, 1994.
21. D. Kavvadias and E. Stavropoulos. Evaluation of an algorithm for the transversal hypergraph problem. *Algorithm Engineering*, pages 72–84, 1999.
22. D. Kozlov. *Combinatorial Algebraic Topology*. Springer-Verlag, 2007.
23. Samuel Peltier, Laurent Fuchs, and Pascal Lienhardt. Simploidals sets: Definitions, Operations and Comparison with Simplicial Sets. *Discrete Applied Math.*, 157:542–557, 2009.
24. M. Raussen. Simplicial models of trace spaces. *Alg. & Geom. Top.*, 10:1683–1714, 2010.
25. A. Valmari. A stubborn attack on state explosion. In *Proc. of CAV'90*. LNCS, 1990.