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

L. Fajstrup M. Raussen
Department of Mathematical Sciences,
Aalborg University
name@math.aau.dk

É. Goubault E. Haucourt S. Mimram
CEA, LIST
surname.name@cea.fr

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 better results than state of the art 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.

Contributions and related work. 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 [15]. Some fundamentals of the mathematics involved can be found in [20]. 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 [20], and we report on the implementation and experimentation of our algorithms on trace spaces.

Stubborn sets [26], sleep sets and persistent sets [16] 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 2.6. 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 [19]. Of course, the trace spaces we are computing have some acquaintance with traces as found in trace theory [8]: 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, 18], and connected components of trace spaces can also be computed using the algorithm introduced in [17].

Contents of the paper. After a giving intuitions (Section 1.1), we define formally the programming language we are considering (Section 1.2) together with a classical semantics (Section 1.3) as well as an associated geometric semantics, (Section 1.4). 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 and possible extensions of the algorithm and conclude.

1. Geometric semantics of concurrent processes

1.1 An informal introduction

Consider the following program consisting of two subprograms, which modify variables, executed in parallel:

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

where assignments are supposed to be atomic. This program might be scheduled in three different ways, respectively giving rise to the following three interleavings of the instructions:

$$\texttt{y:=3;x:=1;y:=2} \qquad \texttt{x:=1;y:=3;y:=2} \qquad \texttt{x:=1;y:=2;y:=3} \tag{2}$$

which might be represented graphically by a transition graph

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

Notice that the first two interleavings of (2) give rise to the same resulting state (in the end $\texttt{x} = 1$ and $\texttt{y} = 2$), whereas the third produces a different state ($\texttt{x} = 1$ and $\texttt{y} = 3$). The reason why the first two are equivalent is that the instructions `x:=1` and `y:=3` "commute", i.e. the way they are scheduled cannot be observed, because they modify different variables: in this sense the first two executions are equivalent, or *dihomotopic*. Using a terminology borrowed from category theory, one could say that the diagram ① commutes, whereas the diagram ② does not; or, if we see the transition graph as a 2-dimensional topological space, the square ① would be filled, whereas the square ② would be a hole. With that last view, the algebraic topological notion of continuous deformation or dihomotopy [15, 20] coincides with local commutation of actions.

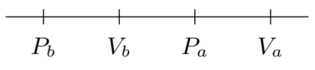
In most concurrent programming languages, the programmer is responsible for ensuring that a variable (or more generally a shared resource) will not be accessed concurrently by two processes. This is usually done by using *mutexes*, which are locks ensuring this property. For instance the program (1) should be rewritten as

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

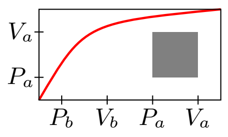
where the instruction P_a locks the mutex a and V_a unlocks it (these respectively correspond to `pthread_mutex_lock` and `pthread_mutex_unlock` functions of the POSIX thread library), and mutexes act in such a way that they cannot be locked by two processes at the same time. In order to abstract away from the irrelevant details of the programming language, we suppose that all involved variables are protected by mutexes ensuring they will not be accessed by two processes at the same time, and moreover we forget about the instructions other than control flow and mutex manipulations since they determine both the structure of the program and whether two schedulings of the program are dihomotopic or not. So, the program (1) will be simplified into

$$P_b.V_b.P_a.V_a \quad | \quad P_a.V_a \tag{4}$$

In order to devise an algorithm for computing the dihomotopy classes of interleavings, we shall use geometrical intuition and formalism by introducing a semantics in which programs are interpreted by topological spaces. For instance, the process $P_b.V_b.P_a.V_a$ will be interpreted as a finite line



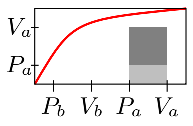
The execution of the process will be modeled as a path going from the left to the right of the figure: the progression of time imposes a direction in paths of our spaces. When the path reaches the point marked P_b, the program performs the action P_b and so on. At each point of the space, there is thus an associated usage of resources; for instance, in all the points strictly between the points P_a and V_a, the mutex a is taken but not the mutex b. In a similar fashion, the process $P_a.V_a$ is interpreted by a finite directed line, and the process (1) as a cartesian product of the interpretations of the two programs in parallel:

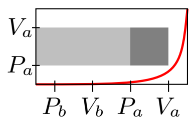


(5)

Again, an execution of the process will correspond to a continuous path going from the lower-left to the upper-right corner (the beginning and end points), which is always increasing (going up and right), such as the red path corresponding to the interleaving $P_a.P_b.V_a.V_b.P_a.V_a$. These are called dipaths, and are going to be points in trace spaces, formally introduced in Section 1.3. Resource usage is also defined in each point of the space. In particular, at the points in the interior of the gray square, the mutex a is taken twice (once by each process), and the semantics of mutexes ensures that this situation does not happen. So in fact, any valid execution path does not cross the gray square, which is called a *forbidden region* and is removed from the space (i.e. it is a hole).

In order to determine the dihomotopy classes of paths in the space, the general idea of the algorithm is to test for each hole all the possible schedulings. In our example, the mutex a is taken first either by the first or the second process. More generally, we test for each hole a possible class of scheduling by forbidding some process to take a mutex first, which amounts to removing the light gray portion of the space in the examples below, and computing whether there exists a path from the beginning to the end satisfying this scheduling.





The idea might seem simple, but it turns out to be difficult to handle correctly and efficiently in the general case, as handled in the present article.

1.2 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 [9]. 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 $|$.

1.3 Trace semantics

Suppose given an alphabet set Σ. Recall that a graph (V, E) consists of a set V of *vertices* (or *states*) and a set $E \subseteq V \times \Sigma \times V$ of *edges* (or *transitions*). We sometimes write $x \xrightarrow{A} y$ for an edge (x, A, y), and A is called the *label* of the transition. The notion of transition graph is a common tool in the study of semantics of programming languages. However, in order to properly model concurrent computations, one should also consider commutations between transitions.

Definition 1. An *asynchronous graph* $G = (V, E, I)$ consists of a graph (V, E) together with a set I of *independence tiles* which are pairs of paths of length 2, with the same source and target, and with labels of the form $A.B$ and $B.A$, which we sometimes draw as

$$\begin{tikzcd}
y_1 \arrow[r, "B"] & z \\
x \arrow[u, "A"] \arrow[r, "B"'] & y_2 \arrow[u, "A"']
\end{tikzcd}$$

These are close to transition systems with independence [2, 25]. Intuitively, a tile relating two such paths means that the transitions A and B can be permuted in the program, as in the tile ① in the introductory example (3). For the sake of simplicity, we only present asynchronous graphs here, but it should be noted that they are particular cases of a more general notion called *cubical sets* [19], which is able to model commutations between any number of events. All the developments carried on here can be generalized to those.

Given two asynchronous graphs G_1 and G_2, their *asynchronous tensor product* $G_1 \otimes G_2$ is defined as follows. Its underlying graph G is the so called "cartesian product of graphs" (which is not actually a cartesian product in the category of graphs but only a tensor product) defined by $V = V_1 \times V_2$ and the transitions are of the form $(x, x') \overset{A}{\to} (y, x')$ or $(x', x) \overset{A}{\to} (x', y)$ when there exists a transition $x \overset{A}{\to} y$ in G_1 or in G_2 respectively (i.e. every transition in G either comes from G_1 or from G_2). Its independence tiles relate every two paths of the form

$$\begin{tikzcd}
(y,x') \arrow[r, "B"] & (y,y') \\
(x,x') \arrow[u, "A"] \arrow[r, "B"'] & (x,y') \arrow[u, "A"']
\end{tikzcd}$$

where the transitions $x \overset{A}{\to} y$ and $x' \overset{B}{\to} y'$ come from G_1 and G_2 respectively.

From now on, suppose that $\Sigma = \{P_a, V_a \ / \ a \in \mathcal{R}\}$ is the set of *actions*. To every program p we associate an asynchronous graph G_p and two vertices b_p and e_p of G_p (the *beginning* and the *end*) defined inductively by

– $G_\mathbf{1}$ is the terminal graph (with one vertex and no edge),

– G_{P_a} is the graph $b_{P_a} \overset{P_a}{\to} e_{P_a}$ (with two vertices and one edge),

– G_{V_a} is the graph $b_{V_a} \overset{V_a}{\to} e_{V_a}$ (with two vertices and one edge),

– $G_{p.q}$ is the graph obtained from the disjoint union of G_p and G_q by identifying e_p with b_q, such that $b_{p.q} = b_p$ and $e_{p.q} = e_q$,

– G_{p+q} is the graph obtained from the disjoint union of G_p and G_q by identifying b_p with b_q and e_p with e_q, such that $b_{p+q} = b_p = b_q$ and $e_{p+q} = e_p = e_q$,

– G_{p^*} is obtained from G_p by identifying e_p with b_p, such that $b_{p^*} = e_{p^*} = b_p = e_p$,

– $G_{p|q}$ is the graph $G_p \otimes G_q$ with $b_{p|q} = (b_p, b_q)$ and $e_{p|q} = (e_p, e_q)$.

A *total path* in such a graph is a path from the beginning to the end.

We write Σ^* for the free monoid of words over Σ. Every path s in an asynchronous graph G_p (also called a *trace*) is labeled by a word $\ell(s)$ in Σ^*. The set $\mathbb{Z}^\mathcal{R}$ of functions $\mathcal{R} \to \mathbb{Z}$ can be equipped with a structure of additive monoid with the constant function equal to 0 as unit, and the sum $f + g$ of two functions f and g being defined pointwise, i.e. as the function which to every resource $a \in \mathcal{R}$ associates $f(a) + g(a)$. The *resource function* $r : \Sigma^* \to \mathbb{Z}^\mathcal{R}$ is the morphism of monoids such that

$$r(P_a)(b) = \begin{cases} -1 & \text{if } b = a \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad r(V_a)(b) = \begin{cases} 1 & \text{if } b = a \\ 0 & \text{otherwise} \end{cases}$$

In the following, we always suppose that the graph G_p is such that for every two paths $s_1 : b_p \twoheadrightarrow x$ and $s_2 : b_p \twoheadrightarrow x$ with b_p as source and the same target, we have $r(\ell(s_1)) = r(\ell(s_2))$. This property can be enforced on programs by a simple syntactic criterion [12], based on a well-bracketing condition (if we see resource locking and unlocking as an opening and closing bracket respectively). Given a state x reachable from b_p, we write $r(x) = r(\ell(s))$ for any path $s : b_p \twoheadrightarrow x$.

The *asynchronous transition system* H_p of a program p is the asynchronous graph obtained from G_p by removing all the vertices x not satisfying $0 \leqslant \kappa_a + r(x) \leqslant \kappa_a$ for some resource $a \in \mathcal{R}$, as well as all edges and independence tiles involving them. For instance the asynchronous graph associated to the program (4) is the graph

$$\begin{tikzcd}
{} \arrow[r, "P_b"] & {} \arrow[r, "V_b"] & {} \arrow[r, "P_a"] & {} \arrow[r, "V_a"] & {} \\
{} \arrow[u, "V_a"] \arrow[r, "P_b"] \arrow[ur, phantom, "I"] & {} \arrow[u, "V_a"] \arrow[r, "V_b"] \arrow[ur, phantom, "I"] & {} \arrow[u, "V_a"] & & {} \arrow[u, "V_a"'] \\
{} \arrow[u, "P_a"] \arrow[r, "P_b"'] \arrow[ur, phantom, "I"] & {} \arrow[u, "P_a"] \arrow[r, "V_b"'] \arrow[ur, phantom, "I"] & {} \arrow[u, "P_a"] \arrow[r, "P_a"'] & {} \arrow[r, "V_a"'] & {} \arrow[u, "P_a"']
\end{tikzcd}$$

with all the squares marked I as independence tiles. We write $\sim$ for the congruence on paths generated by I, called *dihomotopy*: it is the smallest equivalence relation such that $s \sim t$ for every pair of paths $(s, t) \in I$, and if $s \sim t$ then $s_1 \cdot s \cdot s_2 \sim s_1 \cdot t \cdot s_2$ for every paths s_1 and s_2 for which the concatenations make sense. The *schedulings* of a program p is the set of paths $s : b_p \twoheadrightarrow e_p$ quotiented by dihomotopy. As we will see in Section 1.3, this describes the connected components of the trace space. In order to compute this trace space, it turns out to be convenient to adopt a more geometrical point of view and replace the asynchronous graphs by topological spaces (their geometric realizations).

1.4 Geometric semantics

The notion of trace semantics introduced in previous section is quite convenient to work with and has lead to many developments [27], but it is sometimes difficult to build intuitions about the behavior of concurrent programs. In order to overcome this, we introduce a semantics based on (directed) topological spaces. Moreover, 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 [20]:

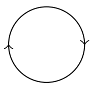
Definition 2. A *directed topological space* (or *d-space* for short) $X = (X, dX)$ consists of a topological space X together with a set dX of paths in X (the *directed paths*) such that

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 $f(dX) \subseteq dY$.

Example 3. Every topological space X equipped with a partial order $\leqslant$ defines a d-space by taking dX the set of paths $p : I \to X$ which are increasing. In particular, we often write $\vec{I}$ for the d-space induced by the unit interval $I = [0,1]$ equipped with the usual total order. Notice that given a d-space X, the maps $p : \vec{I} \to X$ are the directed paths in dX and the maps $r : \vec{I} \to \vec{I}$ are the reparametrizations.

The circle $S^1 = \{e^{i\theta} \ / \ 0 \leqslant \theta < 2\pi\}$ in the complex plane can be equipped with a structure of d-space with dS^1 being the set of paths p of the form $p(t) = e^{if(t)}$ for some increasing function $f : I \to \mathbb{R}$. Notice that in this case, the structure of directed spaces is not induced by a partial order on the space, which makes d-spaces a more general notion.



The category of d-spaces is complete and cocomplete [20]. 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 is such that $d(X \times Y) = dX \times dY$.
- The *disjoint union* $X \uplus Y$ of two d-spaces X and Y is such that $d(X \uplus Y) = dX \uplus dY$.
- The *amalgamed sum* $(X \uplus Y)/(x \sim y)$ of two d-spaces X and Y on points $x \in X$ and $y \in Y$ is the disjoint union $X \uplus Y$ where the points x and y have been identified. A directed path is a reparametrization of a path $p \cdot q$, where $(p,q) \in dX \times dY$ (resp. $(q,p) \in dX \times dY$) are paths such that $p(1) = x$ and $q(0) = y$ (resp. $p(1) = y$ and $q(0) = x$).
- The *difference* $X \setminus Y$ of two d-spaces X and Y has the set $d(X \setminus Y) = \{p \in dX \ / \ p(I) \cap Y = \emptyset\}$ 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 $dY = \{p \in dX \ / \ p(I) \subseteq Y\}$.

These constructions enable us to define the geometric semantics of a program as follows.

Definition 4. 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

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

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

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

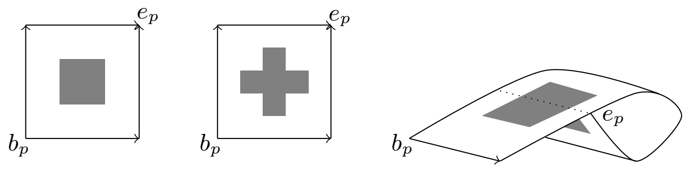
The *geometric realization* of a process p, is defined as the d-space $H_p = G_p \setminus F_p$.

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 5. 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:



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.

The fact that the definition of the geometric semantics resembles a lot the trace semantics introduced in Section 1.3 can be explained by the fact that it is in fact a "geometrization" of the trace semantics. Namely, if we see a vertex as a point, an edge as a directed segment $\vec{I}$, an independence tile as a directed square $\vec{I} \times \vec{I}$, and glue these topological spaces according to how they are connected in the asynchronous graphs, then we recover a subset of the geometric semantics (this process can be formally expressed category using a coend): this process is called the *geometric realization* of a cubical set. In particular, this implies that the schedulings in trace and geometric semantics are essentially the same:

Proposition 6. *Given a program p, there is a (well-behaved) injection ι from the set of total paths of the trace semantics of p to the set of total paths of the geometric semantics of p. Moreover, every total path in the geometric semantics is dihomotopic to a total path in the image of ι; and two total paths in the trace semantics are dihomotopic if and only if their images under ι are dihomotopic.*

The notion of dihomotopy in geometric semantics is formally introduced in Definition 7 below. We call any total path in the image of ι, dihomotopic to p in the geometric semantics, a *lifting* of p.

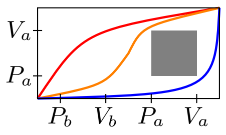
2. Computing trace spaces

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 7. 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 8. In the geometric semantics of the program (4) described in the introduction, the two paths above the hole are dihomotopic, whereas the path below is not dihomotopic to the two others:



As explained in the introduction (Section 1.1), 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

$$p \quad = \quad p_0 \mid p_1 \mid \dots \mid p_{n-1} \tag{6}$$

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

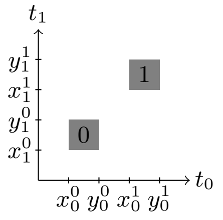
$$G_p \quad = \quad \vec{I}^n \setminus \bigcup_{i=0}^{l-1} R^i$$

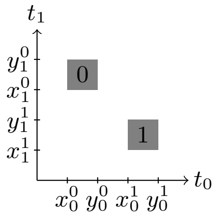
$\vec{I}^n$ denoting the cartesian product of n copies of $\vec{I}$, where each $R^i = \prod_{j=0}^{n-1} \vec{I}^i_j$ is a rectangle. We suppose here that each R^i is homothetic to the n-dimensional open rectangle, i.e. each directed interval $\vec{I}^i_j$ is of the form $\vec{I}^i_j =]x^i_j, y^i_j[$, 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). We suppose fixed a program with n threads and l forbidden open rectangles, and consistently use the notations above.

Example 9. The geometric realization of the programs

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

are respectively

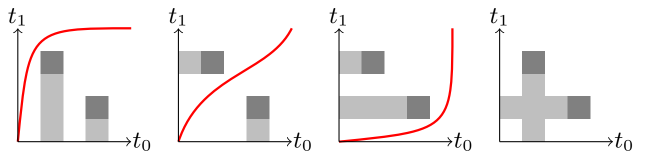


and


2.2 The index poset

Let us come back to the second program of Example 9. 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 9, there are four possibilities to extend once each of the two holes:



(7)

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

$$\forall (i, j) \in [0 : l[\times [0 : n[, \qquad M(i, j) \leqslant N(i, j) \tag{8}$$

where $[m : n[$ denotes the set $\{m, \dots, n-1\}$ 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,

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

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 Examples 11 and 12 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 10. We write

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

for the set of (column) dead matrices and

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

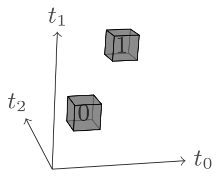
for the set of alive matrices (with non-empty rows), which is called the *index poset* – it is implicitly ordered by the relation (8).

Example 11. In the example above, the three extensions of holes (7) 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 other are alive. The last matrix being dead shows that there is no way a trace can go left and up of the upper left hole and carry on going right of the lower right hole.

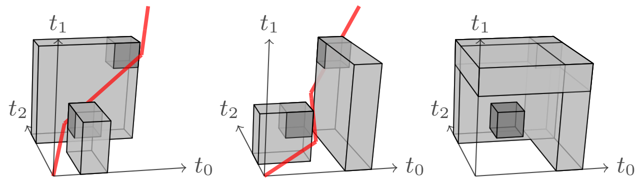
Example 12. The geometric semantics of the program constituted of three copies of the thread $P_a.V_a.P_b.V_b$ in parallel, with $\kappa_a = \kappa_b = 2$, is



The spaces X_M corresponding to the matrices

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

are respectively



The first two matrices are alive, as shown by the drawn total paths.

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 13. *For any matrix $M \in \mathcal{M}^R_{l,n}$, any two paths with the same source x and target y are dihomotopic: the space $X_M(x,y)$ is either empty or contractible. 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 14. 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 15. *The connected components of $\mathcal{C}(X)$ are in bijection with schedulings in X.*

Example 16. 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 in Example 12 but with only one forbidden cube). The matrices in $\mathcal{C}(X_p)$ are

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

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 13) and the fact that two matrices have non-empty lines 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.

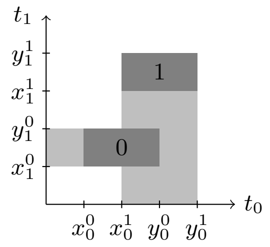
Given a subset I of $[0 : l[$ and an index $j \in [0 : n[$, we write $y^I_j = \min\{y^i_j \ / \ i \in I\}$ (by convention $y^\emptyset_j = \infty$). Given a matrix $M \in \mathcal{M}_{l,n}$, we define the set of *non-zero rows* of M by $R(M) = \{i \in [0 : l[\ / \ \exists j \in [0 : n[, \ M(i,j) \neq 0\}$. 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 [13], the following characterization of dead matrices can therefore be deduced:

Proposition 17. *A matrix $M \in \mathcal{M}^C_{l,n}$ is in $\mathcal{D}(X)$ iff it satisfies*

$$\forall (i,j) \in [0 : l[\times [0 : n[, \quad M(i,j) = 1 \Rightarrow x^i_j < y^{R(M)}_j \tag{9}$$

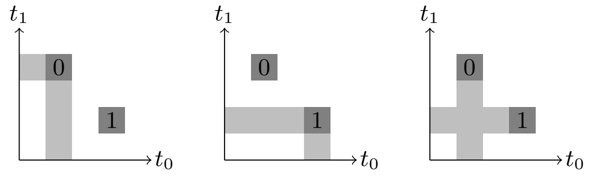
Example 18. In the example below with $l = 2$ and $n = 2$, the matrix $M = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ is dead (we suppose that $x^i_j = 1 + i(j+1)$

and $y_j^i = 3 + i(j+1) - j$):



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

Example 19. Consider the geometric semantics of the second program of Example 9. The minimal dead matrices are



$$D_0 = \begin{pmatrix}1 & 1\\ 0 & 0\end{pmatrix} \qquad D_1 = \begin{pmatrix}0 & 0\\ 1 & 1\end{pmatrix} \qquad D_2 = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

The above proposition enables us to compute the set of dead matrices, for instance by enumerating all matrices and checking whether they satisfy condition 9 (a more efficient method is described in Section 2.5). From this set, the index poset $\mathcal{C}(X)$ can be determined using the following property:

Proposition 20. *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 20 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 $M^{\neg(i,j)} \in \mathcal{M}^R_{l,n}$, 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 21. Consider again Example 19. 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 $D_1 \not\leqslant M_i$:

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

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 $(2,2)$:

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

Since we are only interested in minimal 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 (7). 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. Recall that an *hypergraph* $H = (V, E)$ consists of a set V of *vertices* and a set E of edges, where an *edge* is a subset of V. A *transversal* T of H is a subset of V such that $T \cap e \neq \emptyset$ for every edge $e \in E$. The set $\mathcal{D}(X)$ can be regarded as a hypergraph H, whose set of vertices is $[0:l[\times[0:n[$ and whose hyperedges are the sets $\{(i,j) \;/\; D(i,j) = 1\}$, where D is a matrix in $\mathcal{D}(X)$. It is easy to see that the sets $\{(i,j) \;/\; M(i,j) = 0\}$, where M is a maximal matrix of $\mathcal{C}(X)$, correspond to *minimal transversals* wrt inclusion order (or *hitting sets*) of the hypergraph H. The computation of those transversals was the subject of many studies, for which efficient algorithms have been proposed [21]. Also, we are currently exploring links with transversal matroids, which are likely to provide useful new theoretical tools to study trace spaces.

2.4 Extension to cubes touching boundaries

In Section 2.1, we have supposed that the forbidden region was a union of rectangles R^i, each such rectangle being a product of open intervals $\vec{I}^i_j =]x^i_j, y^i_j[$. 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 $\vec{I}^i_j$ are either of the form $]x^i_j, y^i_j[$ or $[0, y^i_j[$ or $]x^i_j, \infty]$ 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 $x^i_j = 0$ (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 $N \in M^C_{n,l}$, which satisfy (9) and such that

$$\forall j \in C(M), \qquad y_j^{R(M)} = \infty \tag{10}$$

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

2.5 An efficient implementation

In order to compute the set $\mathcal{D}(X)$ of dead matrices, the general idea is to enumerate all the matrices $M \in \mathcal{M}^C_{l,n}$ and check whether they satisfy the condition (9). Of course, a direct implementation of this idea would be highly inefficient since there are l^n matrices in $\mathcal{M}^C_{l,n}$. In order to improve this, we try to detect "as soon as possible" when a matrix 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. Namely, we have to check that every coefficient (i,j) such that $M(i,j) = 1$ satisfies $x^i_j < y_j^{R(M)}$. Now, suppose

```
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 $m.(j) \leftarrow$None else $m.(j) \leftarrow$Some $i$;
        (match $m.(j)$ with
            | Some $i \rightarrow$ if $x^i_j \geqslant yrows.(j)$ then raise Exit
            | None $\rightarrow$ if $yrows.(j) \neq \infty$ then raise Exit);
        let $yrows$ =
          let $j' = j$ in
          if not $changed\_rows$ then $yrows$ else
            Array.mapi (fun $j$ yrj $\rightarrow$
                if yrj $\leqslant y^i_j$ then yrj else
                  match $m.(j)$ with
                    | None $\rightarrow$
                        if $j \leqslant j'$ && $y^i_j \neq \infty$ then raise Exit; $y^i_j$
                    | Some $i \rightarrow$
                        if $x^i_j \geqslant y^i_j$ then raise Exit; $y^i_j$
                        ) $yrows$
        in
        compute_dead $(j+1)$ $m$ $rows$ $yrows$
      with Exit $\rightarrow$ ()
    done
```

Figure 1. Algorithm for computing dead matrices.

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-null 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 (9) because $x^i_j \geqslant y^I_j \geqslant y^{R(M)}_j$. A similar reasoning can be held for condition (10).

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 $yrows.(j)= y^{rows}_j$. The function is initially called with the following parameters:

compute_dead 0 (Array.make n None) Set.empty (Array.init n ∞)

The matrix $bounds$ is the matrix previously noted B used to perform the check (10). 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^{rows}_j 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 21. We have also implemented a simple hypergraph transversal algorithm [3] 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 15) 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, as explained for the path of (5) for example.

2.6 A benchmark: 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 [10] whose schedulings are difficult to compute and is thus often used as a benchmark for concurrency tools. 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 state of the art partial order reduction techniques. If 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 fine 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 Figure 1 could be improved by finding a heuristic to suitably sort holes so that failures to satisfy condition (9) are detected earlier. The present algorithm is also significantly faster than some of the author's previous contribution [17]: for instance, it was unable to generate these maximals dipaths because of memory requirements, for n philosophers with $n > 8$ (in the benchmarks of [17], 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 wost-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

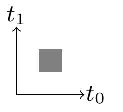
One of the most challenging part of verifying concurrent programs consists in verifying programs with loops since those contain a priori an infinite number of possible execution traces. We extend here the previous methodology and, given a program containing loops, we compute a (finite!) automaton whose accepted paths will be exactly the schedulings of the program: this automaton, can thus be considered as a "reduced" 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 3.3). This section builds on some ideas being currently developed by Fajstrup [11], however most of the properties presented in this section are entirely new. We have tried to give an idea of the proof for most properties we stated, these should be formally presented in a long version of the article.

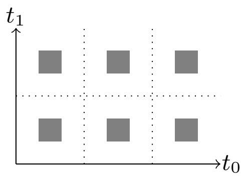
In the following, we suppose fixed a program of the form $p = p_0|p_1|\ldots|p_{n-1}$ as in (6), with n threads. We write

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

for the associated "looping program". Our goal in this section is to describe the schedulings of such a program p^* (more general programs could have also been considered, at the cost of a more technical presentation). Following Section 1.4, 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 $v = (v_0, \ldots, v_{n-1})$ with coefficients in $\mathbb{N}$, the *v-delooping* of p, written p^v, is the program $p_0^{v_0}|p_1^{v_1}|\ldots|p_{n-1}^{v_{n-1}}$, 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 22. Consider the program $p = q|q$ where $q = P_a.V_a$. Its geometric realization X_p is pictured on the left, and its $(3,2)$-delooping $X_{p^{(3,2)}}$ is pictured on the right.





Given two spaces X and Y which are hypercubes with holes (which is typically the case for the geometric realization of the programs we are considering), 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 1.4 as $X \oplus_j Y = X \uplus Y / \sim$ where $\sim$ 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 23. The $(3,2)$-delooping of Example 22 is

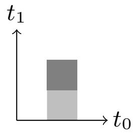
$$X_{p^{(3,2)}} \quad = \quad (X_p \oplus_0 X_p \oplus_0 X_p) \oplus_1 (X_p \oplus_0 X_p \oplus_0 X_p)$$

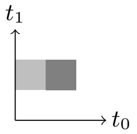
More generally, any v-delooping p^v of a program p of the form (6) can be obtained by gluing copies of X_p.

Given two 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 $X_p \oplus_j X_p$.

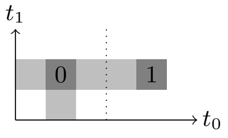
Example 24. The program p of Example 22 admits two maximal alive matrices:

$$M_0 = \begin{pmatrix}1 & 0\end{pmatrix} \qquad\qquad M_1 = \begin{pmatrix}0 & 1\end{pmatrix}$$





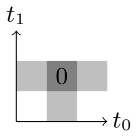
The matrix $M_0 \oplus_0 M_1 = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$ encodes the following extension $X_{M_0 \oplus_0 M_1}$ of holes in $X_{p^{(2,1)}} = X_p \oplus_0 X_p$:



(11)

(the dotted line represents the face on which the two squares have been glued).

In the above example, notice that the "first copy" of X_p (the part corresponding to M_0) gets a new (horizontal) hole inherited by the extension of the second hole due to M_1:



In the following, we write $X_{M_0} \backslash_0 X_{M_1}$ for this space and more generally, given matrices M and N and a dimension j, $X_M \backslash_j X_N$ denotes the space obtained as the subspace of $X_{M \oplus_j N}$ corresponding to X_M, which is called the *j-residual* of X_M before X_N.

Lemma 25. *Suppose given matrices M, N, P, and a dimension j. The residuation operation defined above is commutative and idempotent in the sense that*

$$(X_M \backslash_j X_N) \backslash_j X_P \quad = \quad (X_M \backslash_j X_P) \backslash_j X_N$$

and

$$(X_M \backslash_j X_N) \backslash_j X_N \quad = \quad X_M \backslash_j X_N$$

Moreover, there exists a smallest matrix $N \backslash_j P$ such that

- *$N \leqslant N \backslash_j P$*
- *$X_M \backslash_j X_{(N \backslash_j P)} = (X_M \backslash_j X_N) \backslash_j X_P$*
- *$N \backslash_j P$ is alive if $N \oplus_j P$ is alive*

Proof. The commutativity, idempotence and distributivity properties are easy to show. The matrix $N \backslash_j P$ can be defined as follows. Consider the matrix P' defined by $P'(i, j') = 0$ if $j' \neq j$ and $P'(i, j) = P(i, j)$, and define $N \backslash_j P = N \vee P'$ (where $\vee$ is computed pointwise). The matrix $N \backslash_j P$ thus defined can be checked to satisfy the three properties. □

Example 26. With the notations of Example 24, we have the matrices $M_0 \backslash_0 M_1 = \begin{pmatrix}1 & 1\end{pmatrix}$ and $M_1 \backslash_0 M_0 = \begin{pmatrix}0 & 0\end{pmatrix}$.

By generalizing Example 23, it can easily be shown that any matrix M for a v-delooping p^v of a program p can be expressed as a suitable gluing of matrices M_w for p, indexed by n-dimensional $\mathbb{N}$-vectors such that for every dimension j we have $0 \leqslant w_j < v_j$. We say that the matrix M *contains* a matrix N when there exists a vector w such that $N = M_w$. The following lemma will prove useful in order to deduce the alive matrices of the deloopings p^v from the alive matrices of p:

Lemma 27. *Given a vector v, if M is a matrix for p^v which is alive, then any matrix N for p contained in M is also alive.*

Proof. The result can be deduced from the following series of equivalences: the matrix M is dead iff the space X_M contains a deadlock iff it contains a deadlock in some subspace X_{M_w} (since those form a partition of X_M) iff there exists a vector w such that M_w is dead. □

Of course, the converse implication is not true: Example 24 shows that the gluing (11) of two alive matrices is not necessarily alive.

3.2 The scheduling 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 a quotient of the set of paths of an automaton that we call the *reduced scheduling automaton*: this automaton provides us with a *finite presentation* of the set of schedulings.

Definition 28. The *scheduling automaton* S_p of a looping program p^* with n threads is the automaton, i.e. graph labeled by the alphabet $[0 : n[$, whose

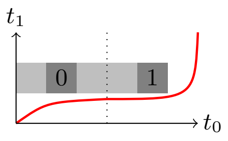
- vertices are the maximal alive matrices of X_p, i.e. elements of the index poset $\mathcal{C}(X_p)$,
- there is a transition labeled by $j \in [0 : n[$ from M to N whenever $M \oplus_j N$ is alive.

A path in this automaton is called a *transition path*.

Example 29. Consider the program $p = q|q$ with $q = P_a.V_a$ introduced in Example 22. The vertices of S_p are the two maximal alive matrices M_0 and M_1 described in Example 24, and the transitions are the following:

$$\begin{tikzcd} M_0 \arrow[loop above, "0,1"] \arrow[r, bend left, "1"] & M_1 \arrow[loop above, "1,0"] \arrow[l, bend left, "0"] \end{tikzcd}$$

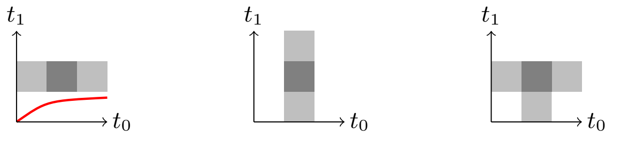
For instance, the transition $M_1 \xrightarrow{0} M_1$ reflects the fact that the matrix $M_1 \oplus_0 M_1$ is alive:



However, there is no transition $M_0 \xrightarrow{0} M_1$ because the matrix $M_0 \oplus_0 M_1$ is not alive, as figured in (11).

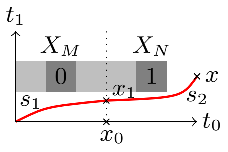
Our goal is to show that schedulings of p^* correspond to paths in S_p. However it is not the case, because two transition paths in S_p can encode the same scheduling, and we will introduce a variant of the scheduling automaton, called the *reduced scheduling automaton* for which this property is verified. The following definitions and lemmas are established in order to do so. Given a space X which is an n-dimensional hypercube with holes, and a dimension $j \in [0 : n[$, we say that X is *j-reachable* when there exists a path whose source is the beginning point of the hypercube and target is a point lying on the j-th target face of the hypercube.

Example 30. The first space is 0-reachable (as shown by the drawn path) but the last two are not (there is no path from the beginning to the right face):

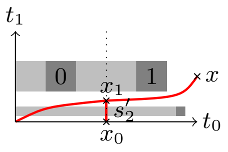


Lemma 31. *Given a point x of X_N, there is a path from 0 to x in $M \oplus_j N$, where x is a point of X_N if and only if X_M is j-reachable and there exists a path from 0 to x in X_N.*

Proof. We illustrate the proof on the example where p is the program of Example 29, $M = N = M_1$, and $j = 0$. Suppose that there exists a path s from 0 to some point x in the target 0-border of X_N:

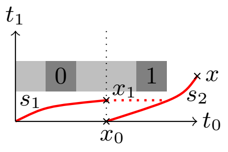


We write x_1 for the intersection of the path with the gluing face and x_0 for the beginning point of X_N. The path s can be expressed as the concatenation of two paths $s = s_1 \cdot s_2$ such that $s_1(1) = s_2(0) = x_1$. Moreover, the path s_1', from x_0 to x_1, obtained as the projection of s_1 onto the j-th target face of X_M is well-defined (i.e. does not go through a forbidden region): if it was not the case, it would mean that there exists an obstruction which was extended, but then the path s_1 itself would not be valid since it would cross the forbidden region:



Therefore, the path s_1 shows that X_M is j-reachable, and the path $s_1' \cdot s_2$ is a path in X_N from the beginning point x_0 to x.

Conversely, suppose that X_M is j-reachable, i.e. there exists a path s_1 from the beginning of X_M to a point x_1 in the j-th target face of X_M, and there exists a path from the beginning x_0 of X_N to x:



The space X_M can be shown under (pointwise) supremums [24]. Therefore the path s_2' defined for any $t \in \vec{I}$ by $s_2'(t) = s_2(t) \vee x_1$ (drawn with a dotted line) is well-defined in X_N and has x_1 as source and x as target. The path $s_1 \cdot s_2'$, which is from 0 to x, enables us to conclude. □

Previous lemma shows in particular that there is a transition path $M \xrightarrow{j} M'$ in the automaton S_p if and only if $M \backslash_j M'$ is j-reachable. This result enables us to "lift" any transition path in the automaton to a trace of the process p^*:

Lemma 32. *For any transition path in S_p*

$$M_{k_0} \xrightarrow{j_1} M_{k_1} \xrightarrow{j_2} \cdots \xrightarrow{j_k} M_{k_m} \qquad (12)$$

there exists a total path in X_{p^v}, called the lifting *of the transition path, where v is the n-dimensional vector such that v_j is one plus the number of occurrences of j in the labels of the edges occurring in* (12), *with forbidden regions in copies of X_p extended according to the matrices M_{k_i} (see Example 33 below).*

Proof. If the transition path (12) is of length $m = 0$, the result is immediate since the matrix M_{k_0} is alive and therefore there exists a total path in $X_{M_{k_0}}$. If the transition path (12) is of length $m = 1$, then by definition of the transitions in S_p there exists a total path in $M_{k_0} \oplus_{j_1} M_{k_1}$. We give here the idea of the proof for a path

$$M_{k_0} \xrightarrow{j_1} M_{k_1} \xrightarrow{j_2} M_{k_2}$$

of length $m = 2$, which can easily be generalized to any longer path. If $j_1 \neq j_2$ then the result is easily obtained: by definition of the automaton, we know that there is a total path in both $X_{M_{k_0}} \oplus_{j_1} X_{M_{k_1}}$ and $X_{M_{k_1}} \oplus_{j_2} X_{M_{k_2}}$, i.e. by Lemma 31 the space

$X_{M_{k_0}} \backslash_{j_1} X_{M_{k_1}}$ is j_1-reachable and the space $X_{M_{k_1}} \backslash_{j_2} X_{M_{k_2}}$ is j_2-reachable, and we can conclude by Lemma 31 again. Now, suppose that $j_1 = j_2$. The situation is more subtle here since it might a priori happen that there is a total path in both $X_{M_{k_0}} \oplus_{j_1} X_{M_{k_1}}$ and $X_{M_{k_1}} \oplus_{j_2} X_{M_{k_2}}$ but not in $X_{M_{k_0}} \oplus_{j_1} X_{M_{k_1}} \oplus_{j_2} X_{M_{k_2}}$, because M_{k_2} creates a deadlock with M_{k_0}, i.e. we have that the space $X_{M_{k_0}} \backslash_{j_1} X_{M_{k_1}} \backslash_{j_1} X_{M_{k_2}}$ contains a deadlock. However, this case cannot occur. Namely, by Lemma 25, we have

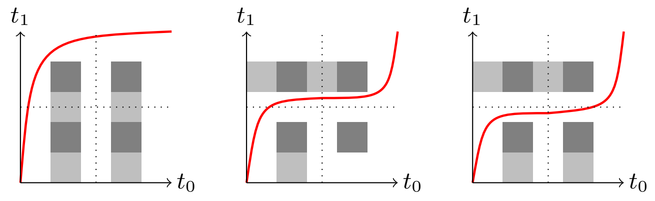
$$X_{M_{k_0}} \backslash_{j_1} X_{M_{k_1}} \backslash_{j_1} X_{M_{k_2}} = X_{M_{k_0}} \backslash_{j_1} X_{(M_{k_1} \backslash_{j_1} M_{k_2})} \qquad (13)$$

and $M_{k_1} \backslash_{j_1} M_{k_2}$ is alive and above M_{k_1}. Moreover, since by definition of S_p, the matrix is M_{k_1} is *maximally* alive, we deduce $M_{k_1} \backslash_{j_1} M_{k_2} = M_{k_1}$. Therefore the space (13) does not contain any deadlock. □

Example 33. Considering again the automaton of Example 29, the transition paths

$$M_0 \xrightarrow{1} M_0 \xrightarrow{0} M_0 \qquad M_0 \xrightarrow{1} M_1 \xrightarrow{0} M_0 \qquad M_0 \xrightarrow{0} M_0 \xrightarrow{1} M_1$$

respectively imply that there are total paths in the following spaces:



Notice that since the label of the first two transition paths are 10, the witnessing paths are first going up (direction 1) and then right (direction 0); and in the other way for the last transition path. Moreover, the holes are extended according to the M_i. For instance in the second space, the lower left copy of X_p is extended according to M_0, the upper left copy of X_p according to M_1 and the upper right copy according to M_0 (and the lower right copy is not extended).

Lemma 34. *Suppose given a transition path of the form* (12)*, then any two paths in X_{p^v} lifting this transition path (as described in Lemma 32) are dihomotopic.*

Proof. This can be deduced from the fact that in the space generated by an alive matrix, two paths with the same source and target are dihomotopic (Proposition 13). □

Lemma 35. *Every path in X_{p^v} is a lifting of some transition path in S_p.*

So, Lemma 32 states that every transition path in S_p can be *lifted* into a path in X_p and by Lemma 34 this lifting is uniquely defined up to dihomotopy. The *lifting* operation which to every transition path of S_p associate the dihomotopy class of the paths lifting it is thus well-defined, and is surjective by Lemma 35. However, the lifting operation is not injective. For instance, the two last paths of Example 33 are homotopic and yet are respectively the lifting the distinct transition paths

$$M_0 \xrightarrow{1} M_1 \xrightarrow{0} M_1 \quad \text{and} \quad M_0 \xrightarrow{0} M_0 \xrightarrow{1} M_1$$

This suggests that paths in the automaton should be considered modulo a congruence identifying the two paths above: we write R_p for the set of pairs of transition paths of length 2 of the form

$$M_1 \xrightarrow{j} M_2 \xrightarrow{j'} M_3 \quad \text{and} \quad M_1 \xrightarrow{j'} M_2' \xrightarrow{j} M_3$$

which lift into paths in

$$(M_1 \oplus_j M_2) \oplus_{j'} (M_2' \oplus_j M_3)$$

These are called the *relations* of the automaton S_p. We write $\equiv$ for the smallest congruence (wrt concatenation) on paths of S_p containing these relations. In the following, we will consider paths in S_p modulo this congruence (in fact S_p should be formalized as an asynchronous graph whose independence tiles are specified by R_p but we do not detail this here).

Example 36. The relations of the automaton of Example 29 are

$$\begin{aligned}
M_0 \xrightarrow{0} M_0 \xrightarrow{1} M_0 &\equiv M_0 \xrightarrow{1} M_1 \xrightarrow{0} M_0\\
M_0 \xrightarrow{0} M_0 \xrightarrow{1} M_1 &\equiv M_0 \xrightarrow{1} M_1 \xrightarrow{0} M_1\\
M_1 \xrightarrow{0} M_0 \xrightarrow{1} M_0 &\equiv M_1 \xrightarrow{1} M_1 \xrightarrow{0} M_0\\
M_1 \xrightarrow{0} M_0 \xrightarrow{1} M_1 &\equiv M_1 \xrightarrow{1} M_1 \xrightarrow{0} M_1
\end{aligned}$$

Finally, two vertices of an automaton which are connected matrices in the sense of Definition 14 should be identified because of Proposition 15. We thus define

Definition 37. The *reduced automaton* $\tilde{S}_p$ associated to a program p is the automaton S_p where connected vertices have been identified.

This automaton thus enables us to provide a *finite presentation* of the set of schedulings, in a sense close to group presentations [7] (i.e. a finite set of generators, the automaton $\tilde{S}_p$, and a relation $\equiv$ finitely generated by R_p):

Theorem 38. *The transition paths in the reduced automaton $\tilde{S}_p$ considered modulo the congruence $\equiv$ generated by R_p are in bijection with schedulings in p^*.*

The previously developed algorithm (Section 2) can be adapted in order to compute the reduced scheduling automaton $\tilde{S}_p$ associated to a looping program p (in particular, the procedure for determining whether a matrix is alive can be modified in order to determine whether a matrix is j-reachable). For the lack of space, we do not detail this here.

3.3 Application to static analysis

Now that we have the reduced scheduling automaton, we are in a position to 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 M of the scheduling automaton $\tilde{S}_p$, a set of values A_M that program variables can take if computation follows a scheduling lifting a transition path whose last vertex is M. 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 M.

Suppose that we want to analyze the program

$$p^* \quad = \quad \Big(P_a.\,(a := a-1)\,.V_a\Big)^* \Big| \Big(P_a.\Big(a := \frac{a}{2}\Big).V_a\Big)^* \qquad (14)$$

What are the possible sets of values reached, for a, starting with $a \in [0,1]$? The associated scheduling automaton S_p has been determined in Example 29 (this automaton is reduced) together with relations, that we will not be using in this article, yet. In many ways, this reduced scheduling automaton plays the role of a compact *control flow graph* for the program we are analyzing.

We are now in a position to interpret the arrows of the scheduling 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. Remember that we determined the two maximal alive matrices M_0 and M_1

corresponding to the program p of (14) in Example 24. The greedy executions of the program in X_{M_0} and X_{M_1} have the following effect on a: respectively $a := a/2$ and $a := a - 1$. This interprets the transfer functions associated to the self arrows on M_0 and M_1. The transfer function associated to the arrows from M_0 to M_1 (resp. from M_1 to M_0) are identity functions: they simply indicate that we have to take the union of the values coming from M_0 (resp. from M_1) with the effect of the transfer function associated to the self arrow on M_0 (resp. on M_1). This can be graphically pictured as

$$\begin{tikzcd} M_0[a := \frac{a}{2}] \arrow[loop above, "0{,}1"] \arrow[r, shift left, "0"] & M_1[a := a - 1] \arrow[loop above, "1{,}0"] \arrow[l, shift left, "1"] \end{tikzcd}$$

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

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

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 is thus that we are looking for $A^\infty = \bigvee_{[0,1]} F$ where:

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

where $[0, 1] \subseteq A$. A simple 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 scheduling 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. Future works

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] M. A. Bednarczyk. *Categories of asynchronous systems*. PhD thesis, University of Sussex, 1988.

[3] C. Berge. *Hypergraphs*, volume 445. North Holland Mathematical Library, 1989.

[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] H. Coxeter and W. Moser. *Generators and relations for discrete groups*. Springer-Verlag, 1980.

[8] V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific, 1995.

[9] E. Dijkstra. The structure of the the operating system. *Communication of the ACM*, 11(15):341–436, 1968.

[10] E. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971. ISSN 0001-5903.

[11] L. Fajstrup. Trace spaces of directed tori with rectangular holes. Technical Report R-2011-08, Aalborg University, 2001.

[12] L. Fajstrup and S. Sokolowski. Infinitely running concurrent processes with loops from a geometric viewpoint. *ENTCS*, 39(2), 2000. ISSN 1571-0661.

[13] L. Fajstrup, E. Goubault, and M. Raußen. Detecting deadlocks in concurrent systems. *CONCUR'98 Concurrency Theory*, pages 332–347, 1998.

[14] L. Fajstrup, M. Raußen, E. Goubault, and E. Haucourt. Components of the fundamental category. *Appl. Cat. Struct.*, 12(1):81–108, 2004.

[15] L. Fajstrup, M. Raußen, and E. Goubault. Algebraic topology and concurrency. *Theor. Comput. Sci.*, 357(1-3):241–278, 2006.

[16] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proc. of the Third Workshop on Computer Aided Verification*, volume 575, pages 417–428. Springer-Verlag, LNCS, 1991.

[17] E. Goubault and E. Haucourt. A practical application of geometric semantics to static analysis of concurrent programs. In *CONCUR*, pages 503–517, 2005.

[18] E. Goubault and E. Haucourt. Components of the fundamental category II. *Applied Categorical Structures*, 15(4):387–414, 2007.

[19] E. Goubault and S. Mimram. Formal relationships between geometrical and classical models for concurrency. *CoRR*, abs/1004.2818, 2010.

[20] M. Grandis. *Directed Algebraic Topology, Models of Non-Reversible Worlds*. Number 13 in New Mathematical Monographs. Cambridge University Press, 2009.

[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] S. Peltier, L. Fuchs, and P. Lienhardt. Simploidals sets: Definitions, Operations and Comparison with Simplicial Sets. *Discrete Applied Mathematics*, 157:542–557, 2009.

[24] M. Raussen. Simplicial models of trace spaces. *Alg. & Geom. Top.*, 10:1683–1714, 2010.

[25] M. Shields. Concurrent machines. *Computer Journal*, 28, 1985.

[26] A. Valmari. A stubborn attack on state explosion. In *Proc. of CAV'90*. LNCS, 1990.

[27] G. Winskel and M. Nielsen. Models for Concurrency, 1995.