A practical application of geometric semantics to static analysis of concurrent programs

Eric Goubault[1] and Emmanuel Haucourt[2]

[1] LIST (CEA - Technologies Avancées)
DTSI-SOL, CEA F91191 Gif-sur-Yvette Cedex
Eric.Goubault@cea.fr,
[2] Preuves, Programmation, Systèmes
Université Paris 7, 175 rue Chevaleret, F75013
haucourt@cea.fr

Abstract. In this paper we show how to compress efficiently the state-space of a concurrent system (here applied to a simple shared memory model, but this is no way limited to that model). The technology used here is based on research on geometric semantics by the authors and collaborators [1]. It has been implemented in a abstract interpretation based static analyzer (ALCOOL), and we show some preliminary results and benchmarks.

1 Introduction and related work

The aim of this paper is to show how to infer some important properties of concurrent and distributed systems using geometric ideas[3]. The algorithms we describe in this paper have been implemented in a prototype "ALCOOL" briefly benchmarked and explained in Section 4, as well as in appendix A.

A class of examples arises from a toy langage manipulating semaphores. Using Dijkstra's notation [2], we consider processes to be sequences of locking operations Pa on semaphores a and unlocking operations Va. In the example where two processes share two resources a and b: $T1 = Pa.Pb.Vb.Va$ in parallel with $T2 = Pb.Pa.Va.Vb$, the geometric model is the "Swiss flag", Fig. 1, regarded as a subset of $\mathbb{R}^2$ with the componentwise partial order $(x_1, y_1) \leq (x_2, y_2)$ if $x_1 \leq y_1$ and $x_2 \leq y_2$. The (interior of the) horizontal dashed rectangle comprises global states that are such that T_1 and T_2 both hold a lock on a: this is impossible by the very definition of a binary semaphore. Similarly, the (interior of the) vertical rectangle consists of states violating the mutual exclusion property on b. Therefore both dashed rectangles form the *forbidden region*, which is the complement of the space X of (legal) states. This space with the inherited partial order provides us with a particular po-space X [3],[4], as defined in Sect. 2. This view can be generalized to more general counting semaphores, i.e. resources that can be shared by some $k > 1$ but not $k + 1$ processes (see Figure 3 for the case $k = 2$ and three processes). Moreover, legal execution paths, called *dipaths*, are

[3] Work partially funded by EDF under grant CEA/EDF 1-5-163 CE.

increasing maps from the po-space $\boldsymbol{I}$ (the unit segment with its natural order) to X. The partial order on X thus reflects (at least) the time ordering on all possible execution paths. Many different execution paths have the same global

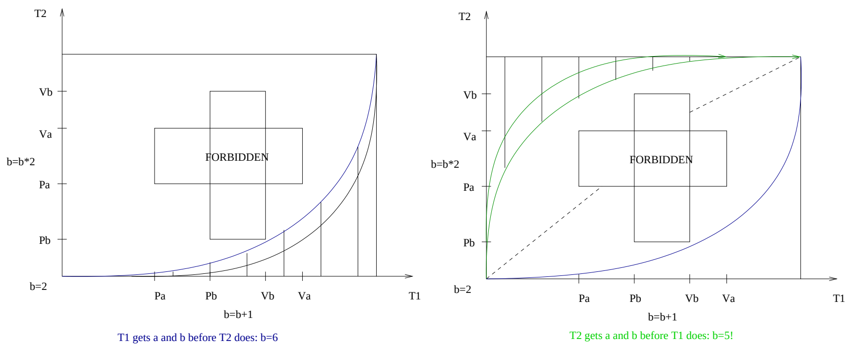

Fig. 1. Essential schedules for the swiss flag.

effect: In the “Swiss Flag” example, for any execution path shaped like the one at the left of Figure 1, T_1 gets hold of locks a and b before T_2 does. This implies that for the actual assignments on variable b that we have chosen in this example: T_1 does $b := b + 1$ and T_2 does $b = b * 2$, starting with an initial value of 2, all execution paths below the hole will end up with the value $b = 6$, since T_1 will do $b = 2 + 1 = 3$ and then only after will T_2 do $b = 3 * 2 = 6$. In fact, there are only two essentially different execution paths from the initial point $(0, 0)$ to the final point $(1, 1)$, that fully determine the computer-scientific behaviour of the system. See picture at the right hand side of Figure 1). These are in fact the only two classes of dipaths from $(0, 0)$ to $(1, 1)$ modulo “continuous deformations” that do not reverse time, i.e., up to *dihomotopy* as defined in [5]. This fact is indeed general, and is not at all limited to the example. For determining the possible outcome of a concurrent program (modelled in a suitable way, as for our PV programs), only the dihomotopy classes of dipaths count.

Other interesting dipaths, in our example space, start in the initial point $(0, 0)$ and end in a deadlock, or start in the unreachable point and end in $(1, 1)$ see the dashed paths on Figure 1.

In general, one of the important invariants of a concurrent system is its *fundamental category* [6],[7], defined in Section 3.1, classifiying dipaths between any pair of points up to dihomotopy, i.e, a directed version of the fundamental groupoid of a topological space. In nice cases, the relevant information in the fundamental category is essentially finite. This is shown using a construction based on categories of fractions [8], as developped in [1] and [9]. The formalism developped in these last two papers allows to decompose the fundamental category (or the state-space) into big chunks as the regions 1 to 10 in Figure 2. Basically, inside these regions, or components, nothing important happens. This produces the following compressed state-space on which, using general results of [1], one can read all temporal properties as pictured in Figure 2 (with two views, one geometric, the other, algebraic). The graph of the right hand side should be understood as generating a category [10], where morphisms represent classes of paths of execution, such that we have relations ${g'}_2 \circ {g'}_1 = g_2 \circ g_1$ and ${f'}_2 \circ {f'}_1 = f_2 \circ f_1$ (compare with e.g. [11]). In some sense, this so-called category of components finitely presents the fundamental category and the essential properties of the state-space, that can be used in a static analysis based verification tool.

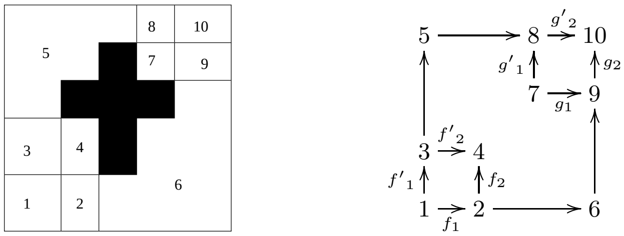

Fig. 2. The components of the Swiss flag

Some comparisons between what this type of approach should buy us with other state-space reduction techniques such as persistent sets [12],[13], stubborn sets [14],[15], Petri nets based techniques [16] etc. have been made in [7]. In this paper, we develop this line of research a bit further, giving actual algorithms to compute this component category in relevant cases, implementing them and benchmarking them.

In Section 3.2, we give an algorithm to find the components, and to enumerate the "essential traces", i.e. the traces of execution modulo dihomotopy, which correspond, on a fragment of the model, to finding representatives of the Mazurkiewicz traces [17]. This in turn can be used to compute efficiently an abstraction of the collecting semantics of parallel processes, as used in abstract interpreters [18]. This is described in Section 4. The implementation of this algorithm is for the time being rather crude, but still, one can fully handle the case of 9 philosophers, and effectively compress its state-space and its set of essential schedules (which in this case is very large anyway). This is the base of the static analyzer ALCOOL we have been developping for EDF (the main French electricity provider), that we briefly describe at the end of this section.

We should end up this introductory section by saying that this state-space reduction technique is entirely orthogonal to other techniques like *symbolic model-checking* as developped in e.g. [19],[20],[21] or like abstraction based techniques. A combination of good abstractions with this algorithm should improve performances a lot. Last but not least, other geometric criteria for state-space reduction are currently being developped, one which looks extremely promising being [22].

2 Models of concurrent computation

The main idea (see [23] for instance) is to model a *discrete* concurrency problem in a *continuous geometric* set-up: A system of n concurrent processes will be represented as a subset of Euclidean space $\mathbb{R}^n$. Each coordinate axis corresponds to one of the processes. The state of the system corresponds to a point in $\mathbb{R}^n$, whose i'th coordinate describes the state (or "local time") of the i'th processor. An execution is then a *continuous increasing path* within the subset from an initial state to a final state.

A more general framework on which this paper is based is defined below (see [5]):

Definition 1. *1. A po-space is a topological space X with a (global) closed partial order $\leq$ (i.e. $\leq$ is a closed subset of $X \times X$).*
2. A dimap $f : X \to Y$ between po-spaces X and Y is a continuous map that respects the partial orders (is non-decreasing).
3. A dipath $f : \boldsymbol{I} \to X$ is a dimap whose source is the interval $\boldsymbol{I}$ with the usual order.

Po-spaces and dimaps form a category. To a certain degree, our methods apply to the more general categories of lpo-spaces [5] , of flows [24] and of d-spaces [25].

We start with a very simplistic language, in order to explain the concepts. We will point out in Section 4 that this can be extended to more realistic languages, as used in ALCOOL.

$$Proc_d = \epsilon \mid Pa.Proc_d \mid Va.Proc_d$$

(ϵ being the empty string, a being any object of $\mathcal{O}$, defined as a binary semaphore: $s(a) = 1$ or as a counting semaphore initialized to k: $s(a) = k$). A PV program is any parallel combination of these PV processes, $Prog = Proc \quad | \quad (Prog \mid Prog)$. The typical example in shared memory concurrent programs is $\mathcal{O}$ being the set of shared variables and for all $a \in \mathcal{O}$, $s(a) = 1$. The P action is putting a lock and the V action is relinquishing it. We will suppose in the sequel that any given process can only access once an object before releasing it.

Supposing that the length of the strings X_i ($1 \leq i \leq n$), denoting n processes in parallel in this language, are integers l_i, the semantics of $Prog$ is included in $[0, l_1] \times \cdots \times [0, l_n]$. A description of $[\![Prog]\!]$ can be given by describing inductively what should be digged into this n-rectangle (the semantics is given in terms of the set of forbidden hyper-rectangles). The semantics of our language can be described by the simple rule, $[k_1, r_1] \times \cdots \times [k_n, r_n] \in [\![X_1 \mid \cdots \mid X_n]\!]$ if there is a partition of $\{1, \cdots, n\}$ into $U \cup V$ with $card(U) = s(a) + 1$ for some object a with, $X_i(k_i) = Pa$, $X_i(r_i) = Va$ for $i \in U$ and $k_j = 0$, $r_j = l_j$ for $j \in V$.

3 Essential schedules

3.1 A bit of theory

Equivalence of dipaths, as used in the examples of Figure 1, is modelled by the notion of dihomotopy, a directed version of standard homotopy [26]. They describe accurately, in a continuous model, a generalized notion of "commutation of actions", and make available some powerful tools from algebraic topology (see [6], [27], [5] for surveys).

Dihomotopies between dipaths f and g (with fixed extremities α and β in X) are dimaps $H : \boldsymbol{I} \times I \to X$ such that for all $x \in \boldsymbol{I}$, $t \in I$, $H(x,0) = f(x)$, $H(x,1) = g(x)$, $H(0,t) = \alpha$, $H(1,t) = \beta$. Notice that here I carries the equality as order contrarily to $\boldsymbol{I}$ (another definition can be given [25], but which is equivalent in all the cases dealt with in this paper).

A dihomotopy is to be understood as a 1-parameter family of dimaps without order requirements in the second I-coordinate[4]. Now, we can define the main object of study of this paper, the fundamental category, which contains all relevant information for the study of traces of execution:

Definition 2. *The fundamental category is the category $\boldsymbol{\pi}_1(X)$ with:*

- *as objects: the points of X,*
- *as morphisms, the dihomotopy classes of dipaths: a morphism from x to y is a dihomotopy class $[f]$ of a dipath f from x to y.*

Concatenation of dipaths factors over dihomotopy and yields the composition of morphisms in the fundamental category. A dimap $f : X \to Y$ between po-spaces induces a functor $f_{\#} : \boldsymbol{\pi}_1(X) \to \boldsymbol{\pi}_1(Y)$, and we obtain thus a functor $\boldsymbol{\pi}_1$ from the category of po-spaces to the category of categories.

We formally invert some "inessential" morphisms in the fundamental category, as in [1], [9], to obtain a "compressed" component category. For instance, for a binary semaphore taken by two processes, we will obtain the category generated by the graph of Figure 6. In the case of three processes trying to get hold of a counting semaphore initialized to two (geometric semantics given by Figure 3), we would get the component category pictured in Figure 4: each of the 26 subcubes delineated by the green planes are components, and there is one morphism from each of these to neighbouring ones (in the directed order). Every four neighbours having a segment in common have their four "neighbouring" morphisms commute.

3.2 Inductive computation

In the case of the geometric semantics of the toy PV language we chose, all these component categories are in fact generated by 2-dimensional precubical sets (graphs plus a notion of 2-cell, filling some of the rectangular holes in the graph):

[4] This is slightly different for d-spaces, but coincides in important cases.

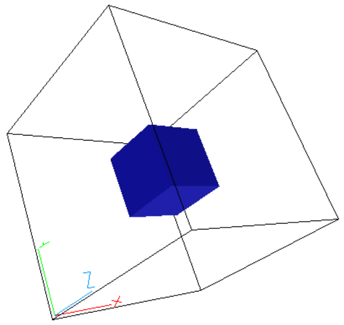

Fig. 3. Geometric semantics of a counting semaphore initialized to 2.

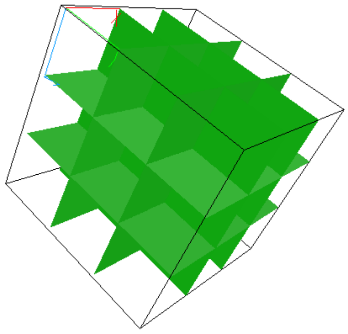

Fig. 4. Its component category

Definition 3. *A 2-dimensional precubical set is given by*

$$(X_0, X_1, X_2, (\partial_0^0, \partial_1^0, \partial_0^1, \partial_1^1 : X_2 \to X_1), (\partial_0^0, \partial_0^1 : X_1 \to X_0))$$

such that $\partial_i^k \circ \partial_j^l = \partial_{j-1}^l \circ \partial_i^k$ *for* $i < j$ *and* $k = 0, 1$, $l = 0, 1$. ∂_0^1 *and* ∂_1^1 *(respectively* ∂_0^0, ∂_1^0*) are called end (respectively start) boundary operators.*

More general versions of these precubical sets have been used to model concurrent processes [6]. These 2-dimensional precubical sets are somehow the analogues of asynchronous transition systems [28], [29]. Elements of X_n $(n = 0, 1, 2)$ are called n-transitions. A simple example of a 2-dimensional pre-cubical set (which should represent a in parallel with b) is given below:

$$\begin{tikzcd}
s_0 \arrow[r, "a"] \arrow[d, "b"'] \arrow[dr, phantom, "A"] & s_1 \arrow[d, "b'"] \\
s_2 \arrow[r, "a'"'] & s_3
\end{tikzcd}$$

where A is a 2-transition, a, b, a', b' are 1-transitions and s_0, s_1, s_2 and s_3 are all 0-transitions (or states). We have $\partial_0^0(A) = a$, $\partial_0^1(A) = a'$, $\partial_1^0(A) = b$, $\partial_1^1(A) = b'$, $\partial_0^0(a) = \partial_0^0(b) = s_0$, $\partial_0^1(a) = s_1 = \partial_0^0(b')$, $\partial_0^1(b) = \partial_0^0(a') = s_2$ and $\partial_1^1(b') = \partial_1^1(a') = s_3$. One can readily check the commutation rules of the definition, for instance, $\partial_0^0\partial_1^1(A) = \partial_0^0(b') = s_1 = \partial_0^1(a) = \partial_0^1\partial_0^0(A)$. We should think in the sequel, of A as representing the independance of a and b.

Example We know from [1] that the po-space and the component category corresponding to the PV program (where `a` is a binary semaphore) $A = Pa.Va$

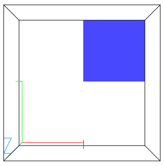

Fig. 5. Po-space corresponding to a simple PV program

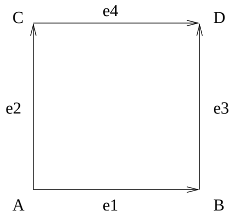

Fig. 6. Its component category

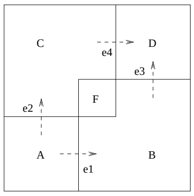

Fig. 7. The components, geometrically

in parallel with $B = Pa.Va$ are those pictured at Figure 5, respectively, of Figure 6.

As a matter of fact, the precubical set (here of dimension 1, since there is no relation between morphisms here) corresponding to this component category can be pictured as in Figure 7.

Intuition of the inductive algorithm Now, what if we dig in a new hole in the po-space of Figure 5? We get the po-space pictured in Figure 8 and should obtain the component category (where solid squares represent relations) pictured in Figure 9. This po-space corresponds to the PV program $A = Pa.Va.Pb.Vb$ in parallel with $B = Pb.Vb.Pa.Va$ and the component category corresponds to the precubical set of dimension 2, pictured "geometrically" in Figure 10. The idea is that digging a new hole, creates new isothetic hyperplanes, coming out from the min and max points of this hole. These hyperplanes cut the previous components into new components; the orthogonal of these hyperplanes will create new edges in the component graph, or morphisms in the component category. A new phenomenon here is that the intersection of two hyperplanes (here lines), which give a codimension 2 linear variety in general (here, points), correspond to relations between newly created morphisms. Here, in Figure 10, F_2 is the new hole. The morphisms of the component category for the only hole F_1 are denoted here by f_1, f_2, f_3 and f_4. We see[5] in Figure 10 that we have two codimension 2 varieties of interest, namely the two intersections $e_1 \cap f_2$ and $e_3 \cap f_4$ which give the two relations, hence the two 2-cells of the component category, pictured in Figure 9.

In the case of the 3 philosophers problem, $A = Pa.Pb.Va.Vb$ parallel $B = Pb.Pc.Vb.Vc$ parallel $C = Pc.Pa.Vc.Va$, we get the very nice component category pictured in Figure 11 for instance, where the central point represents both the deadlocking and the unreachable regions.

[5] The intersection $a \cap b$ in this figure are denoted by the pair a, b.

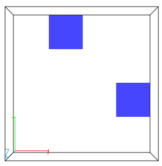

Fig. 8. Po-space with two incomparable holes

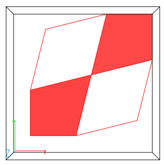

Fig. 9. Component category (squares are relations)

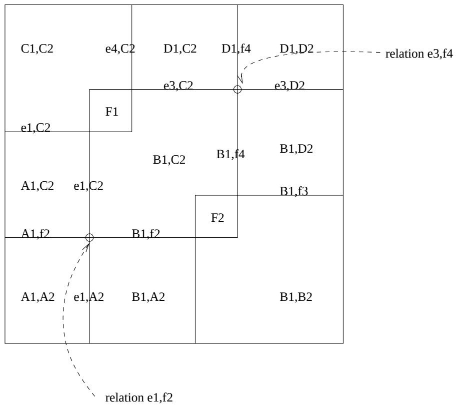

Fig. 10. The precubical set corresponding to the component category, geometrically

Inductive computation - the algorithm We start inductively by a component category of $[0,1]^n \backslash R$, generated by a 2-dimensional precubical set, that we write in short as $(Y_0, Y_1, Y_2, \delta^0, \delta^1)$. We define a new structure $(Z_0, Z_1, Z_2, \partial^0, \partial^1)$ as follows, which will generate (an "approximation" of) the component category of $U \backslash R$:

- $Z_0 = \{A \cap B \mid A \in X_0, B \in Y_0, A \cap B \neq \emptyset\}$
- $Z_1 = \begin{array}{l} \{A \cap f \mid A \in X_0, f \in Y_1, A \cap f \neq \emptyset\} \\ \cup \{e \cap B \mid e \in X_1, B \in Y_0, e \cap B \neq \emptyset\} \end{array}$
- $Z_2 = \begin{array}{l} \{e \cap f \mid e \in X_1, f \in Y_1, e \cap f \neq \emptyset\} \\ \cup \{R \cap B \mid R \in X_2, B \in Y_0, R \cap B \neq \emptyset\} \\ \cup \{A \cap S \mid A \in X_0, S \in Y_2, A \cap S \neq \emptyset\} \end{array}$
- $\partial^*_* : Z_1 \to Z_0$ are defined by:
  - $\partial^0_0(A \cap f) = A \cap \delta^0_0(f)$,
  - $\partial^1_0(A \cap f) = A \cap \delta^1_0(f)$,
  - $\partial^0_0(e \cap B) = d^0_0(e) \cap B$,
  - $\partial^1_0(e \cap B) = d^1_0(e) \cap B$.

  $\partial^*_* : Z_2 \to Z_1$ are defined by:
  - $\partial^0_0(e \cap f) = d^0_0(e) \cap f$,
  - $\partial^0_1(e \cap f) = e \cap \delta^0_0(f)$,
  - $\partial^1_0(e \cap f) = d^1_0(e) \cap f$,
  - $\partial^1_1(e \cap f) = e \cap \delta^1_0(f)$,
  - $\partial^k_l(R \cap B) = d^k_l(R) \cap B$, $k, l = 0, 1$,
  - $\partial^k_l(A \cap S) = A \cap \delta^k_l(S)$.

One can show that this gives an "over-approximation" of the component category in general, i.e. that one will get a compressed state-space, which might not be as optimal as the component category defined in [1]. Similarly, one can check easily that this, applied to the case of Figure 8 starting with the case of Figure 6 gives the right result of Figure 10.

3.3 Syntactic lift

From the component category, we can deduce the maximal morphisms (or equivalently, the equivalences classes of maximal dipaths, or put it differently the maximal essential traces), basically from some traversing of the underlying graph modulo 2 cells. In the case of the maximal dipaths modulo dihomotopy for the 3 philosophers, we find 7 paths, the $3! = 6$ non-deadlocking paths, 3 of which are represented as blue lines in Figures 11, 12 and 13, one deadlocking path.

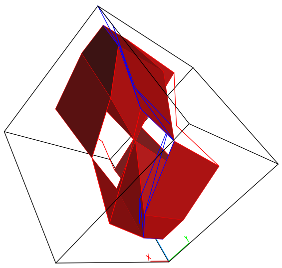

Fig. 11. Paths (1)

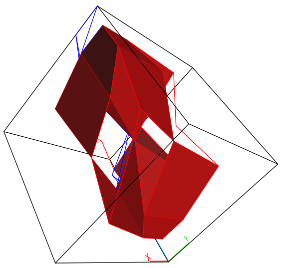

Fig. 12. Paths (2)

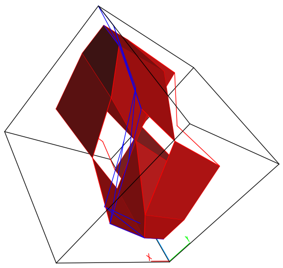

Fig. 13. Paths (3)

Now, we want to get back from these "continuous" paths to "discrete" paths. This "discrete" path should be an interleaving path corresponding to this idealized execution, which can then be analyzed by any standard sequential analyzer.

We remark, essentially by [1], that (1): every component has a trivial $\vec{\pi}_1$ and (2): there exists a path (unique) from the *minimum* (or infimum in general) from a component to the minimum of the next component (essentially by the lifting property). Given the morphisms of the component category, we compute:

- (a) the infimum of the components (i.e. of hyperrectangles minus the forbidden region)
- (b) the *program* comprising the possible executions between the minimum of a component, and the minimum of the next
- (c) we use the interleaving semantics for finding *just one path* in this program (using (1), in a very economical manner)

We exemplify this in Figures 14, 15, 16, 17, 18 and 19 for the 3 philosophers' problem. Forbidden regions are represented in blue, and components are represented from green to red, in a graded manner. We represent only maximal paths in the component category as sequences of such components in these figures.

Point (c) is done by taking any interleaving path for some program, extracted from *Prog* in a very easy manner, using the coordinates of the two consecutive

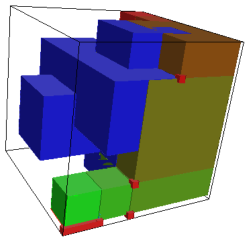

Fig. 14. First step

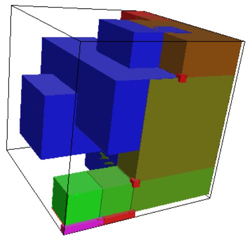

Fig. 15. Second step

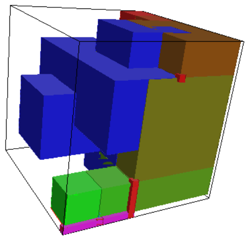

Fig. 16. Third step

infima points (represented as red dots) as intervals, in each coordinate, or equivalently for each process, of instructions to fire (this is represented as red chunks). The first step of the lifting is (Figure 14) amounts to interpreting `0 | 0 | P(c)` in context $sem(c) = 1$, $sem(b) = 1$, and $sem(a) = 1$. We use the notation $sem(x) = k$ to express that x is a semaphore which can be taken (by P) by at most k processes. This describes the state of our concurrent machine. The 2^{nd}, 3^{rd}, 4^{th}, 5^{th} and 6^{th} steps are respectively described in Figures 15, 16, 17, 18 and 19.

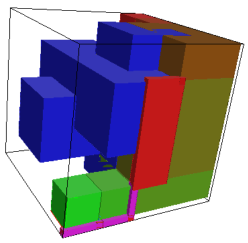

Fig. 17. Fourth step

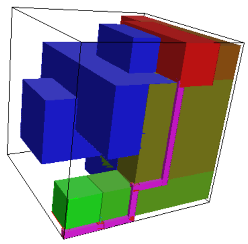

Fig. 18. Fifth step

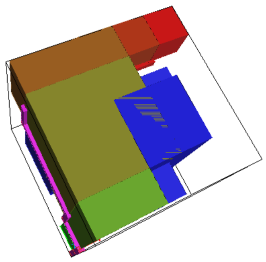

Fig. 19. Sixth step

The final interleaving representative is given at Figure 20 and corresponds to[6]:

$$P_3(c).P_3(a).P_2(b).V_3(c).P_2(c).V_2(b).V_2(c).V_3(a).P_1(a).P_1(b).V_1(a).V_1(b).$$

[6] Where we put the number of the process which takes the step as a subscript of the P and V actions.

4 Application to static analysis

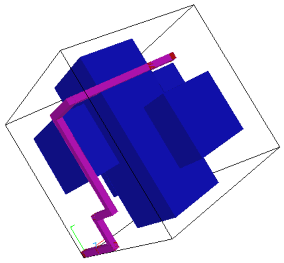

Fig. 20. The interleaving representative

A static analyzer (ALCOOL) based on these principles has been implemented, it consists of about 25000 lines of C. It relies on Hans Boehm garbage collector [30] for memory allocation and QT for the graphical user interface. ALCOOL analyzes programs written in a high-level language, to be described elsewhere, extending the one of Section 2: binary semaphores, general counting semaphores but also synchronisation barriers and bounded and unbounded FIFO message passing queues (with various blocking/unblocking policies for sending and receiving) are modelled. Numeric variables are allowed, and guards (tests) are allowed in non-deterministic choices. General expressions on variables are understood, as well as iteration schemes. As such, this language is not far from the level of expressiveness of PROMELA [31], with a different syntax, aimed at the particular geometric models we have developped. A comparison with PROMELA, and SPIN, will be published elsewhere. An example of the syntax can be found in appendix A. The analyzer first represents an abstraction of the set of forbidden regions (as products of intervals in some subspace of R^n), from the syntax of the program to analyze. It then computes inductively, using the algorithm presented in Section 3.2, the component category, as a 2-dimensional precubical set both geometrically (meaning that the objects, morphisms and relations are represented as their corresponding 0-, 1- and 2-codimensional geometric varieties) and combinatorially, using the boundary operators. It represents internally also the duals of the boundary operators, the "coboundary" operators, mapping each i-dimensional object to the $(i+1)$-dimensional objects it is the boundary of. Using these coboundary operators for edges, and a simple depth-first or breadth-first traversal of the underlying graph, it can determine the maximal dipaths modulo 2-cells (modulo dihomotopy), that is, the essential paths. From the essential paths, it determines using a simple abstract interpreter [32] (using intervals of values again), an over-approximation of the local invariants of the program. It has then to iterate this process again, since knowing more about the values of the variables at each reachable state, enables to qualify more precisely whether all the synchronisation that have been modelled as forbidden regions are actually done (because of the guards of the choices for instance). More about the look and feel of the analyzer can be found in appendix A. The analyzer has been applied to a variety of academic examples. For instance, the enumeration of the compressed state space of the n-philosophers' problem is shown for different values of n, on a standard PC with 512Mb of memory and 1GHz clock:

n	time	mem	# o	# m	# r	# p	#s	# t
3	0.38s	$\leq$ 10 Mb	27	48	18	6	576	1475
4	0.43s	$\leq$ 15 Mb	85	200	132	24	3966	13450
5	0.69	19 Mb	263	770	730	120	27265	113938
6	3.49	23 Mb	807	2832	3516	720	184876	914019
7	96.76s	42 Mb	2467	10094	15484	5040	?	?
8	1656.9s	100Mb	7533	35216	64312	40320	?	?
9	13739s	319Mo	22995	120924	256158	362880	$\sim$2996970*	$\sim$22698700*

where # o, # m and # r denote respectively the number of objects, morphisms and relations of the component category, # p is the number of maximal terminating paths (not counting the deadlocking path for instance) and # s (respectively # t) is the number of states (respectively transitions) used in the translation of the n-philosophers' problem for SPIN with the partial-order reduction package (in PROMELA) of [33][7]. For the 9 philosophers' problem, we have only an estimate $\sim \ldots^*$, using the bit state hashing reduction technique.

This analyzer has also been applied to a real industrial example, for the french electricity provider EDF. The code to analyze was a 100000 lines program written in C, comprising a dozen threads running on top of the VXWORKS operating system. These threads communicate through FIFO queues, and synchronize using several dozens of semaphores and monitors. First, this code has been translated in the ALCOOL language (an extract of a typical example is given in Appendix A). This can now be done using the tool MIEL, by Jean-Michel Collart (CEA/LIST), which will be described elsewhere. The first analysis has been made using a handmade translation, taking into account a subgroup of six processes, accounting for 1966 lines of process algebra code, much like the ones shown in Appendix A. The analyzer could prove (using some restrictive assumptions though) that there is no deadlock, no loss of message in 497.43 seconds, for a maximal memory consumption of 47 Mb. In order to do this, it enumerated the class of execution paths (about 6.2 Mb in textual form) and interpret them using a simple interval abstract interpreter.

5 Conclusion and future work

We have described a first step towards using geometric invariants for efficient static analysis of concurrent programs. Much work is still to do. For instance,

[7] Some other implementations of the n-philosophers' problem may find different numbers of states and transitions: our experience is that it can vary from 1 to 10.

the computation of components is still sub-optimal (in size). We could also use static/dynamic segment trees to improve the computation of intersections, or simpler geometric constraints to prune the intersection search. For the computation of morphisms in the component category, we could think of getting some help from the first homology group. We can also approximate relations using some techniques used in persistent sets [34] for instance. On the longer run, we think that the consideration of higher-dimensional analogues of the fundamental category (see [6] and in particular [35]) should help us having smaller retracts of the state space. We should also point out that some other methods used to compress the state space being entirely orthogonal to our technique, we should combine the latter with symbolic methods, use of symmetry, on the fly traversal etc. We would also like to generalize our current ALCOOL analyzer so that it can deal with more general temporal logic formulas. For the time being, loops (and non-deterministic branchings) are interpreted in a simplistic way, by just unravelling them. We are currently trying to see if we can extend our method to local po-spaces [5] directly.

Acknowledgments
We used Geomview, see the Web page
`http://freeabel.geom.umn.--edu/software/download/geomview.html/`
to make the 3D pictures of this article (in a fully automated way, from ALCOOL). Acknowledgments are due to Fabrice Derepas for his help with comparing ALCOOL with SPIN. Acknowledgments to Jean-Michel Collart (CEA), Alain Ourghanlian and Jean-Baptiste Chabannes (EDF).

References

1. Fajstrup, L., Goubault, E., Haucourt, E., Raussen, M.: Components of the fundamental category. Applied Categorical Structures (2004)
2. Dijkstra, E.: Cooperating Sequential Processes. Academic Press (1968)
3. Nachbin, L.: Topology and Order. Van Nostrand, Princeton (1965)
4. Johnstone, P.T.: Stone Spaces. Cambridge University Press (1982)
5. Fajstrup, L., Goubault, E., Raussen, M.: Algebraic topology and concurrency. submitted to Theoretical Computer Science, also technical report, Aalborg University (1999)
6. Goubault, E.: Some geometric perspectives in concurrency theory. Homology Homotopy and Applications (2003)
7. Goubault, E., Raussen, M.: Dihomotopy as a tool in state space analysis. In Rajsbaum, S., ed.: LATIN 2002: Theoretical Informatics. Volume 2286 of Lect. Notes Comput. Sci., Cancun, Mexico, Springer-Verlag (2002) 16 – 37
8. Gabriel, P., Zisman, M.: Calculus of fractions and homotopy theory. Number 35 in Ergebnisse der Mathematik und ihrer Grenzgebiete. Springer Verlag (1967)
9. Haucourt, E.: A framework for component categories. ENTCS (to appear, 2005)
10. Mac Lane, S.: Categories for the working mathematician. Springer-Verlag (1971)
11. Gaucher, P., Goubault, E.: Topological deformation of higher dimensional automata. Technical report, arXiv:math.AT/010760, to appear in HHA (2001)

12. Godefroid, P., Peled, D., Staskauskas, M.: Using partial-order methods in the formal validation of industrial concurrent programs. IEEE Transactions on Software Engineering **22** (1996) 496–507
13. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-space caching revisited. In: Formal Methods and System Design. Volume 7., Kluwer Academic Publishers (1995) 1–15
14. Valmari, A.: A stubborn attack on state explosion. In: Proc. of CAV'90, Springer Verlag, LNCS (1990)
15. Valmari, A.: Eliminating redundant interleavings during concurrent program verification. In: Proc. of PARLE. Volume 366., Springer-Verlag, Lecture Notes in Computer Science (1989) 89–103
16. Melzer, S., Roemer, S.: Deadlock checking using net unfoldings. In: Proc. of Computer Aided Verification, Springer-Verlag (1997)
17. Mazurkiewicz, A.: Basic notions of trace theory. In: Lecture notes for the REX summer school in temporal logic, Springer-Verlag (1988)
18. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. Principles of Programming Languages 4 (1977) 238–252
19. Boigelot, B., Godefroid, P.: Model checking in practice: An analysis of the access.bus protocol using spin. In: Proceedings of Formal Methods Europe'96. Volume 1051., Springer-Verlag, Lecture Notes in Computer Science (1996) 465–478
20. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proc. of the Fifth Annual IEEE Symposium on Logic and Computer Science, IEEE Press (1990) 428–439
21. Garavel, H., Jorgensen, M., Mateescu, R., Pecheur, C., Sighireanu, M., Vivien, B.: Cadp'97 – status, applications and perspectives. Technical report, Inria Alpes (1997)
22. Raussen, M.: Deadlocks and dihomotopy in mutual exclusion models. Technical report, Aalborg University (2005) available at http://www.math.aau.dk/index_en.html.
23. Carson, S., Reynolds, P.: The geometry of semaphore programs. ACM TOPLAS **9** (1987) 25–53
24. Gaucher, P.: A convenient category for the homotopy theory of concurrency. preprint available at math.AT/0201252 (2002)
25. Grandis, M.: Directed homotopy theory, I. the fundamental category. Cahiers Top. Gom. Diff. Catg, to appear, Preliminary version: Dip. Mat. Univ. Genova, Preprint 443 (2001)
26. Spanier, E.J.: Algebraic Topology. McGraw Hill (1966)
27. Goubault, E.: Geometry and concurrency: A users' guide. Mathematical Structures in Computer Science (2000)
28. Goubault, E.: Cubical sets are generalized transition systems. Technical report, pre-proceedings of CMCIM'02, also available at http://www.di.ens.fr/~goubault (2001)
29. Fahrenberg, U.: A category of higher-dimensional automata. In: Foundations of Software Science and Computation Structures (FOSSACS) : 8th International Conference. LNCS, Springer (2005) to appear.
30. Boehm, H.: Bounding space usage of conservative garbage collector. In: Principles Of Programing Language. (2002) see http://www.hpl.hp.com/personal/Hans_Boehm/gc/.
31. Holzmann, G.J.: SPIN Model Checker : The Primer and Reference Manual. Addison Wesley (2003)

32. Cousot, P., Cousot, R.: Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. BIGRE **74** (1991) 107–110
33. Demartini, C., Iosif, R., Sisto, R.: Modeling and validation of java multithreading applications using spin. In: SPIN Workshop. (1998)
34. Godefroid, P., Wolper, P.: 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., Springer-Verlag, Lecture Notes in Computer Science (1991) 417–428
35. Grandis, M.: The shape of a category up to directed homotopy. Technical Report preprint 509, Dip. Mat. Univ. Genova (2004) available at http://www.dima.unige.it/ grandis/rec.public_grandis.html.

A ALCOOL analyzer

Let us give a simple example, in the language used by ALCOOL. Here we define two FIFO queues containing at most one entry, x and y, and two semaphores z and evt. `INIT` is a reserved keyword for initializing, before starting any process (see PROMELA) the context of execution. `@(a,5)` stands for setting value 5 to variable `a`. We can also use general interval expressions, such as `[0,2]`. `PROG` is a reserved keyword to express which are the processes put in parallel. `R(x,z)` stands for (blocking) receive on channel `x`, and put the received value in z (value "protected" by semaphore `z`). `A+[x=0]-B` stands for: do `A` is guard (here `x=0`) is true, otherwise, do `B`. The definition of `automate` as a "matrix" of actions times events is typical of actuation and control software. `S(x,7)` stands for (non-blocking) send on channel `x`, of value 7.

```
#fifo x
#fifo y
#sem z
#sem evt

INIT=@(a,5).@(z,0).@(evt,[0,2])
PROG=automate|tache

act1=R(x,z).@(z,z*2)
act2=R(y,z).@(z,z*3+1)
act3=Pa.@(a,1).Va
ligneA=act1+[a=0]-(act2+[a=1]-(act3+[a=2]-))
ligneB=act2+[a=0]-(act3+[a=1]-(act1+[a=2]-))
ligneC=act3+[a=0]-(act1+[a=1]-(act2+[a=2]-))
matrice=ligneA+[evt=0]-(ligneB+[evt=1]-(ligneC+[evt=2]-))
automate=matrice.automate

tache=S(x,7).S(y,9).Pa.@(a,0).Va.Pa.@(a,2).Va
```