Detecting Deadlocks in Concurrent Systems

Lisbeth Fajstrup+, Eric Goubault* and Martin Raußen+
LETI (CEA - Technologies Avancées)
DEIN-SLA, CEA F91191 Gif-sur-Yvette Cedex
+Dept of Mathematics, Aalborg University
DK-9220 Aalborg

Abstract

We study deadlocks using geometric methods based on generalized process graphs [Dij68], i.e. cubical complexes or Higher-Dimensional Automata (HDA) [Pra91, vG91, GJ92, Gun94], describing the semantics of the concurrent system of interest. A new algorithm is described and fully assessed, both theoretically and practically and compared with more well-known traversing techniques. An implementation is available, applied to a toy language. This algorithm not only computes the deadlocking states of a concurrent system but also the so-called "unsafe region" which consists of the states which will eventually lead to a deadlocking state. This is based on a real geometric characterization of deadlocks.

1 Introduction and related work

This paper deals with the detection of deadlocks motivated by applications in data engineering, e.g., scheduling in concurrent systems. Many fairly different techniques have been studied in the numerous literature on deadlock detection. Unfortunately, they very often depend on a particular (syntactic) setting, and this makes it difficult to compare them. Some authors have tried to classify them and test the existing software, like [Cor96, CCA96], but for this, one needs to translate the syntax used by each of these systems into one another, and different translation choices can make the picture entirely different. Nevertheless, we will follow their classification to put our methods in context. Notice that in this article, we go one step beyond and also derive the "unsafe region" i.e. the set of states that are bound to run into a deadlocking state after some time. This analysis is done in order to be applied to finding schedulers that help circumvent these deadlocking behaviours (and not just for proving deadlock freedom as most other techniques have been used for).

The first basic technique is a *reachability search*, i.e., the traversing of some semantic representation of a concurrent program, in general in terms of transition systems, but also

*Work done partly while at Ecole Normale Supérieure, email:goubault at aigle.saclay.cea.fr sometimes using other models, like Petri nets [MR97]. Due to the classical problem of *state-space explosion* in the verification of concurrent software, such algorithms are accompanied with state-space reduction techniques, such as *virtual coarsening* (which coalesce internal actions into adjacent external actions) [Val89], *partial-order techniques* (which alleviate the effects of representation with interleaving by pruning "equivalent" branches of search) such as *sleep sets* and *permanent (or stubborn) sets* techniques [Val91, GPS96, GHP95], and *symmetry techniques* (that reduce the state-space by consideration of symmetry). These techniques only reduce the state-space up to three or four times except for very particular applications.

The second most well-known technique is based on *symbolic model-checking* as in [BG96, BCM$^+$90, GJM$^+$97, BG96]. Deadlocking behaviors are described as a logical formula, that the model-checker tries to verify. In fact, the way a model-checker verifies such formulae is very often based on clever traversing techniques as well. In this case, the states of the system are coded in a symbolic manner (BDDs etc.) which enables a fast search.

Then many of the remaining techniques are a blend of one of these two with some abstractions, or are *compositional techniques* [YY91], or based on *dataflow analysis* [DC94], or on *integer programming techniques* [ABC$^+$91] (but this in general only relies on necessary conditions for deadlocking behaviors).

Based on some old ideas [Dij68] and some new semantic grounds [Pra91, vG91, Gun94, GJ92, Gou95a] (see §2), we have developped an enhanced sort of reachability search (§2.3). This should mostly be compared to ordinary reachability analysis and not to virtual coarsening and symmetry techniques because these can also be used on top of ours. A first approach in the direction of virtual coarsening has actually been made in [Cri95]. Some assessments about its practical use, based on a first implementation applied to simple semaphore programs are made in §2.4. Due to the page limit, we have not fully described this algorithm. We chose to focus on the really new aspect of deadlock detection using a geometric semantics.

In §3, we propose a new algorithm based on an *abstraction* (in the sense of *abstract interpretation* [CC77, CC92]) of the natural semantics, which takes advantage of the real *geometry of the executions*. This one is an entirely different method from those in the literature.

As a matter of fact, in recent years, a number of people have used ideas from geometry and topology to study concurrency: First of all, using geometric models allows one to use spatial intuition; furthermore, the well-developed machinery from geometric and algebraic topology can serve as a tool to prove properties of concurrent systems. A more detailed description of this point of view can be found in J. Gunawardena's paper [Gun94] – including many more references – which contains a first geometrical description of *safety* issues. In another direction, techniques from algebraic topology have been applied by M. Herlihy, S. Rajsbaum, N. Shavit [HS95, HS96] and others to find new *lower bounds* and *impossibility results* for distributed and concurrent computation.

We believe that this technique, which is assessed in §4.4 and §4.5 both on theoretical grounds and on the view of benchmarks, can be applied in the static analysis of "real" concurrent programs (and not only at the PV language of §2.3) by suitable compositions and reduced products with other abstract interpretations.

The authors participated in the workshop "New Connections between Mathematics and Computer Science" at the Newton Institute at Cambridge in November 1995. We thank the organizers for the opportunity to get new inspiration. This paper is the first in a series of papers resulting from the collaboration of two mathematicians (L. Fajstrup & M. Raussen) and a computer scientist (E. Goubault).

2 Models of concurrent computation

2.1 From Discrete to Continuous

A description of deadlocks in terms of the geometry of the so-called progress graph (cf. Ex. 1) has been given earlier by S. D. Carson and P. F. Reynolds [CR87], and we stick to their terminology. The main idea in [CR87] 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.

Example 1 Consider a centralized database, which is being acted upon by a finite number of transactions. Following Dijkstra [Dij68], we think of a transaction as a sequence of P and V actions known in advance – locking and releasing various records. We assume that each transaction starts at (local time) 0 and finishes at (time) 1; the P and V actions correspond to sequences of real numbers between 0 and 1, which reflect the order of the P's and V's. The initial state is $(0,\dots,0)$ and the final state is $(1,\dots,1)$. An example consisting of the two transactions $T_1 = P_aP_bV_bV_a$ and $T_2 = P_bP_aV_aV_b$ gives rise to the two dimensional *progress graph of Figure 1.*

The shaded area represents states, which are not allowed in any execution path, since they correspond to mutual exclusion. Such states constitute the *forbidden area.* An *execution path* is a path from the initial state $(0,0)$ to a final state $(1,1)$ avoiding the forbidden area and increasing in each coordinate - time cannot run backwards.

In Ex. 1, the dashed square marked "Unsafe" represents an *unsafe area*: There is no execution path from any state in that area to the final state $(1,1)$. Moreover, its extent (upper corner) with coordinates (Pb, Pa) represents a *deadlock.* Likewise, there are no execution paths starting at the initial state $(0,0)$ entering the *unreachable area* marked "Unreachable". Concise definitions of these concepts will be given in §2.2.

Finding deadlocks and unsafe areas is hence the geometric problem of finding n-dimensional "corners" as the one in Ex. 1. Back in 1981, W. Lipski and C. H. Papadimitriou [LP81] attempted to exploit geometric properties of forbidden regions to find deadlocks in database-transaction systems. But the algorithm in [LP81] does not generalize to systems composed of more than two processes. S. D. Carson and P. F. Reynolds indicated in [CR87]

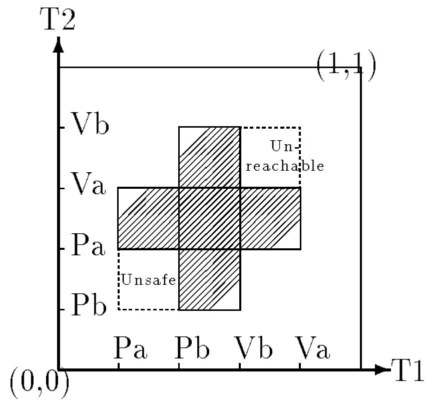


Figure 1: Example of a progress graph

an iterative procedure identifying both deadlocks and unsafe regions for systems with an arbitrary finite number of processes.

In this section, we present a streamlined path to their results in a more general situation: Basic properties of the geometry of the state space are captured in properties of a *directed graph* – back in a discrete setting. In particular, *deadlocks* correspond to *local maxima* in the associated partial order.

This set-up does not only work for semaphore programs: In general, the forbidden area may represent more complicated relationships between the processes like for instance general k-semaphores, where a shared object may be accessed by k, but not $k+1$ processes. This is reflected in the geometry of the forbidden area F, that has to be a *union of higher dimensional rectangles* or "boxes".

Furthermore, similar partially ordered sets can be defined and investigated in more general situations than those given by Cartesian progress graphs. By the same recipe, deadlocks can then be found in concurrent systems with a variable number of processes involved or with branching (tests) and looping (recursion) abilities. In that case, one has to consider partial orders on sets of "boxes" of variable dimensions. This allows the description and detection of deadlocks in the *Higher Dimensional Automata* of V. Pratt [Pra91] and R. van Glabbeek [vG91] (cf. E. Goubault [Gou95a] for an exhaustive treatment).

In the mathematical parts below, i.e., §2.2 and §2.3, the explanations have been voluntarily simplified. The full treatment of the deadlock detection method can be found on the Web (http://www.dmi.ens.fr/ goubault/analyse.html).

2.2 The continuous setup

Let I denote the unit interval, and $I^n = I_1 \times \cdots \times I_n$ the unit cube in n-space. This is going to represent the space of all local times taken by n processes. We call a subset $R = [a_1, b_1] \times \cdots \times [a_n, b_n]$ an *n-rectangle*, and we consider a set $F = \bigcup_1^r R^i$ that is a finite union of n-rectangles $R^i = [a_1^i, b_1^i] \times \cdots \times [a_n^i, b_n^i]$. The interior $\overset{\circ}{F}$ of F is the "forbidden region" of I^n; its complement is $X = I^n \setminus \overset{\circ}{F}$. Furthermore, we assume that $\mathbf{0} = (0, \ldots, 0) \notin F$, and $\mathbf{1} = (1, \ldots, 1) \notin F$.

Definition 1 • 1. A continuous path $\alpha : I \to I^n$ is called a *dipath* (directed path) if *all* compositions $\alpha_i = pr_i \circ \alpha : I \to I$, $1 \leq i \leq n$, are increasing: $t_1 \leq t_2 \Leftarrow \alpha_i(t_1) \leq \alpha_i(t_2)$, $1 \leq i \leq n$.

• 2. A point $y \in X = I^n \setminus \overset{\circ}{F}$ is in the *future* $J^+(x)$ of a point $x \in X$ if there is a dipath $\alpha : I \to X$ with $\alpha(0) = x$ and $\alpha(1) = y$. The past $J^-(x)$ is defined similarly.

• 3. A near future $J_0^+(x)$ of $x \in X$ is of the form $J^+(x) \cap ([x_1, x_1 + \varepsilon] \times \cdots \times [x_n, x_n + \varepsilon])$ where $\varepsilon < \min\{a_j^i - x_j > 0, b_j^i - x_j > 0,\ 0 \leq i \leq r, 0 \leq j \leq n\}$.

• 4. A point $x \in I^n \setminus \overset{\circ}{F}$ is called *admissible*, if $\mathbf{1} \in J^+(x)$; and *unsafe* else.

• 5. Let $\mathcal{A}(F) \subset I^n$ denote the *admissible region* containing all admissible points in X, and $\mathcal{U}(F) \subset I^n$ the *unsafe region* containing all unsafe points in X.

• 6. A point $x \in X$ is a *deadlock* if and only if $J^+(x) = \{x\}$.

In semaphore programs, the n-rectangles R^i characterize states where two transactions have accessed the same record, a situation which is *not* allowed in such programs. Such "mutual exclusion"-rectangles have the property that only two of the defining intervals are proper subintervals of the I_j. Furthermore, serial execution should always be possible, and hence F should not intersect the 1-skeleton of I^n consisting of all edges in the unit cube. These special features will *not* be used in the present paper.

A dipath represents the continuous counterparts of the traces of the concurrent system, which must not enter the forbidden regions.

2.3 Continuous to discrete - a graph theory approach

We use geometrical ideas to construct a digraph where deadlocks are the leaves and the unsafe region is found by an iterative process. The setup is as in §2.2. For $1 \leq j \leq n$, the set $\{a_j^i, b_j^i | 1 \leq i \leq r\} \subset I_j$ gives rise to a partition of I_j into at most $(2r+1)$ subintervals: $I_j = \bigcup I_{jk}$, with an obvious ordering $\leq$ on the subintervals I_{jk}. The partition of intervals gives rise to a partition $\mathcal{R}$ of I^n into n-rectangles $I_{1k_1} \times \cdots \times I_{nk_n}$ with a partial ordering given by

$$I_{1k_1} \times \cdots \times I_{nk_n} \leq I_{1k'_1} \times \cdots \times I_{nk'_n} \Leftrightarrow I_{jk_j} \leq I_{jk'_j},\ 1 \leq j \leq n.$$

The partially ordered set $(\mathcal{R}, \leq)$ can be interpreted as a *directed, acyclic graph*, denoted $(\mathcal{R}, \to)$: Two n-rectangles $R, R' \in \mathcal{R}$ are connected by an edge from R to R' – denoted $R \to R'$ – if $R \leq R'$ and if R and R' share a face. R' is then called an *upper neighbor* of R, and R a *lower neighbor* of R'. A path in the graph respecting the directions will be denoted a *directed path.*

For any subset $\mathcal{R}' \subset \mathcal{R}$ we consider the *full* directed subgraph $(\mathcal{R}', \to)$. Particularly important is the subgraph $\mathcal{R}_F$ consisting of all rectangles $R \subset X = I^n \setminus \overset{\circ}{F}$.

Definition 2 *Let $\mathcal{R}' \subset \mathcal{R}$ be a subgraph. An element $R \in \mathcal{R}'$ is a* local maximum *if it has no upper neighbors in $\mathcal{R}'$.* Local minima *have no lower neighbors. An n-rectangle $R \in \mathcal{R}_{\bar{F}}$ is called a* deadlock *rectangle if $R \neq R_{\mathbf{1}}$, and if R is a local maximum with respect to $\mathcal{R}_{\bar{F}}$. An* unsafe *n-rectangle $R \in \mathcal{R}_{\bar{F}}$ is characterized by the fact, that* any *directed path α starting at R hits a deadlock rectangle sooner or later [CR87].*

In order to find the set $\mathcal{U}$ of all unsafe points – which is the union of *all* unsafe n-rectangles – apply the following. (1) Remove F from I^n giving rise to the directed graph $(\mathcal{R}_{\bar{F}}, \rightarrow)$. (2) Find the set S_1 of all deadlock n-rectangles (local maxima) with respect to $\mathcal{R}_{\bar{F}}$. Let $F_1 = F \cup S_1$. (3) Let $\mathcal{R}_{\overline{F_1}}$ denote the full directed subgraph on the set of rectangles in $I^n \setminus F_1$, i.e., after removing S_1. (4) Find the set S_2 of all deadlock n-rectangles with respect to $\mathcal{R}_{F_1}$. Let $F_2 = F_1 \cup S_2$. Carry on the same completion mechanism etc.

Notice that it is enough to search among the lower neighbors of elements in F in step 2, and that the only candidates for deadlocks in step 4 are the lower neighbors of elements of S_1. Since there are only *finitely many* rectangles, this process stops after a finite number of steps, ending with S_r and yielding the following result:

Theorem 1 • *1. The unsafe region is determined by $\mathcal{U}(F) = \bigcup_1^r S_i$.*

• *2. The set of admissible points is $\mathcal{A}(F) = I^n \setminus (\overset{\circ}{F} \cup \mathcal{U}(F))$. Moreover, any directed path in $\mathcal{A}(F)$ will eventually reach $R_{\mathbf{1}}$.*

A prototype analyser has been programmed on the base of an HDA semantics of PV programs with the following syntax: Given a set of objects $\mathcal{O}$ (like shared memory locations, synchronization barriers, semaphores, control units, printers etc.) and a function $s : \mathcal{O} \rightarrow \mathbb{N}^+$ associating to each object a, the maximum number of processes $s(a) > 0$ which can access it at the same time, any process $Proc$ can try to access an object a by action Pa or release it by action Va, any finite number of times. In fact, processes are defined by means of a finite number of recursive equations involving process variables X in a set $\mathcal{V}$: they are of the form $X = Proc_d$ where $Proc_d$ is the process definition formally defined as,

$$\begin{array}{rclll} Proc_d & = & \epsilon \mid & Pa.Proc_d & \mid Va.Proc_d \\ & & & Proc_d + Proc_d & \mid Y \end{array}$$

(ϵ being the empty string, a being any object of $\mathcal{O}$, Y being any process variable in $\mathcal{V}$). A PV program is any parallel combination of these PV processes, $Prog = Proc \quad | \quad (Proc \mid Proc)$. 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. We also suppose that the recursive equations are "guarded" in the sense that for all process variables X, $Proc_X$ does not contain a summand of the form $X.T$, T being any non-empty term.

We deliver here only the theoretical and practical assessment of this "reference" algorithm (with which we are going to compare our new algorithm).

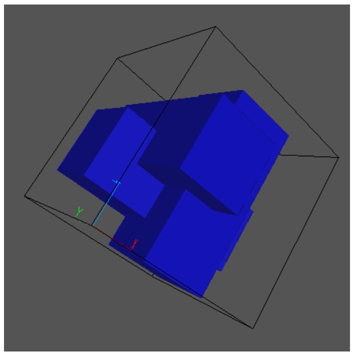


Figure 2: The forbidden regions for 3phil

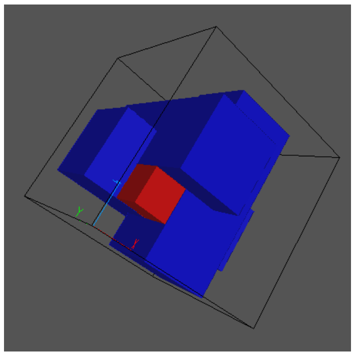


Figure 3: Unsafe (red) region for 3phil

2.3.1 Algorithmic issues

We let the *volume* $Vol(S)$ of a set S of nodes (n-rectangles) in $\mathcal{R}$ be the number of its elements. For every element $R \in R_i$ one has to check whether R has to be added to the unsafe region. Only if the answer is yes, the $2n$ operations of disconnecting R form its n sons and n parents and possibly, a single addition to, resp. removal from, the list of unsafe rectangles, has to be performed. This implies:

Proposition 1 *For a pure term consisting of n transactions with a forbidden region $F = \bigcup_1^r R_i$, the worst case complexity of the algorithm of is of order $nVol(F) + \Sigma_1^r Vol(R_i)$.*

Examples reaching the worst case have a high amount of global synchronization, which should be avoided in the programs involved. Hence one would expect a much better behaviour in the average situation. In fact, if $nVol(F)$ is the dominating part, the complexity is at most nN.

2.4 Benchmarks

The program has been written in C and compiled using `gcc -O2` on an Ultra Sparc 170E with 496 Mbytes of RAM, 924 Mbytes of cache. All times have been measured using the `ddi.h` library and the virtual times as provided by the command `gethrvtime()`. The dynamic data (the graph of cubes itself for instance) was created using the standard `malloc()` function of the `bsdmalloc` library. No particular optimization was made here. Timings have been rounded to the nearest hundredth of a second but are not more precise than a couple of hundredths of a second.

In the table below, dim is the dimension of the program considered (look at Appendix A for explanations), #face is the number of all faces that are actually represented, same for #cube but for the n-rectangles (including the forbidden ones) and for #forb, for the n-rectangles that are in the forbidden region. Then t_{sem} is the time needed to construct the whole semantics, t_{dead} is the time needed from then to compute the unsafe region and #d is the number of n-rectangles found to be in the unsafe region.

program	dim	#face	#cube	#forb	t_{sem}	t_{dead}	#d
example	2	112	79	14	0	0	13
stair2	2	152	105	16	0.01	0	41
stair3	3	1614	960	290	0.18	0.01	356
stair3'	3	6027	2314	80	0.64	0.02	0
lipsky	3	939	556	158	0.08	0	0
3phil	3	190	123	32	0	0	1
4phil	4	1152	611	190	0.09	0	1
5phil	5	6298	2899	1048	0.82	0.02	1
6phil	6	32596	13455	5482	5.82	0.13	1
7phil	7	162990	61703	27668	42.35	0.86	1

3 Continuous to discrete - invoking the geometry

The first algorithm uses very little of the rich geometry available. In fact there is a much better way to look at deadlocks.

3.1 The boundary of the forbidden area

To study dipaths and futures of points in $X = I^n \setminus \overset{\circ}{F}$ efficiently, we need a closer geometric/combinatorial examination of the boundary of the forbidden area. Moreover, this analysis will be helpful in analyzing dihomotopy relations between dipaths; this has an interest in studying equivalence of execution paths, cf. [Gou95a, Gou95b], and, in particular, safety issues, cf. [Gun94]. Details in that direction will be worked out elsewhere.

Let $R = [a_1, b_1] \times \cdots \times [a_n, b_n]$ denote an n-rectangle. Its boundary $\partial(R)$ decomposes into

- the *lower boundary* $\partial_-(R) := \{\mathbf{x} \in R | \forall j : x_j < b_j, \ \exists j : x_j = a_j\}$;
- the *upper boundary* $\partial_+(R) := \{\mathbf{x} \in R | \forall j : x_j > a_j, \ \exists j : x_j = b_j\}$;
- the *intermediate boundary* $\partial_\pm(R) := \{\mathbf{x} \in R | \exists j_1, j_2 : x_{j_1} = a_{j_1}, \ x_{j_2} = b_{j_2}\}$.

Let again $\overset{\circ}{F} \subset I^n$ denote the forbidden region and let $X = I^n \setminus \overset{\circ}{F}$. In the sequel, we need the following *genericity* property of the rectangles in F:

If $\overset{\circ}{R^{i_1}} \cap \overset{\circ}{R^{i_2}} \neq \emptyset$, then $a_j^{i_1} = a_j^{i_2} \Rightarrow a_j^{i_1} = 0$ and $b_j^{i_1} = b_j^{i_2} \Rightarrow b_j^{i_1} = 1, \ 1 \leq j \leq n$.

This property is obviously satisfied for forbidden regions for "mutually exclusion" models, in particular for PV-models.

Points in I^n with at least one coordinate 0 or 1 play a special role: In a mutual exclusion model they stand for situations where not all processors have started their execution or where some of them already have terminated. These points require special attention. To circumvent lengthy case studies in the mathematical part, we slightly change our model in order to include the upper boundary $\partial_+(I^n)$ of I^n into the forbidden region. To this end, let $\tilde{I} = [0, 2]$ and $I^n \subset \tilde{I}^n$.

Slightly changing the notation, let $R^i = [0,2]^{i-1}\times[1,2]\times[0,2]^{n-i}$, $1 \leq i \leq n$, and shifting indices by n, $R^{n+1},\dots,R^{n+r}$ will denote the n-rectangles used in the previous model F of the forbidden region modified to maintain genericity: If $b^i_j = 1$, then let $b^{i+n}_j = 2$. Then $\bigcup_1^n R^i = \tilde{I}^n\setminus \overset{\circ}{I^n}$, and $\tilde{F} = F\cup\bigcup R^i = \bigcup_{i=1}^{n+r} R^i$. By an abuse of notation, we will from now on write F for $\tilde{F}$.

The *boundary* $\partial F \subset F$ decomposes as $\partial F = \partial_- F\cup\partial_+ F\cup\partial_\pm F$ with $\partial F = (\bigcup_i \partial R^i)\setminus \overset{\circ}{F}$, $\partial_- F = (\bigcup_i \partial_- R^i)\setminus \overset{\circ}{F}$, $\partial_+ F = (\bigcup_i \partial_+ R^i)\setminus \overset{\circ}{F}$ and $\partial_\pm F = (\bigcup_i \partial_\pm R^i)\setminus \overset{\circ}{F}$.

Looking at dipaths starting from a point $\mathbf{x}\in X$, we can concentrate attention on points $\mathbf{x}\in\partial_- F$, since there are no local obstructions for all the other points:

Lemma 1 *For* $\mathbf{x} = (x_1,\dots,x_n)\in(X\setminus\partial_- F)$*, the future* $J^+(\mathbf{x})$ *contains a* complete cone $[x_1, x_1+\varepsilon]\times\cdots\times[x_n,x_n+\varepsilon]$ *for some* $\varepsilon > 0$. □

For points $\mathbf{x}\in\partial_- F$, the structure of the near future $J_0^+(\mathbf{x})$ can be explained in terms of a *boundary stratification*:

Let $R^i = [a^i_1, b^i_1]\times\cdots\times[a^i_n,b^i_n]$, and for any nonempty index set $J = \{i_1,\dots,i_k\}\subseteq \{1,\dots,n+r\}$ define $R^J = R^{i_1}\cap\cdots\cap R^{i_k}$, i.e., $R^J = [a^J_1, b^J_1]\times\cdots\times[a^J_n, b^J_n]$ with $a^J_j = \max\{a^i_j | i\in J\}$ and $b^J_j = \min\{b^i_j|i\in J\}$. This set is again an $n-$rectangle unless it is empty (if $a^k_j > b^l_j$ for some $1\leq j\leq n$ and $k,l\in J$). To the index set J we associate $\partial_- R^J = \partial_- R^{i_1}\cap\cdots\cap\partial_- R^{i_k}$ and the *boundary stratum* (in $\partial_- F$) $\partial^J_- F = R^J\cap\partial_- F = \partial_- R^J\setminus \overset{\circ}{F}$.

An index set $\emptyset\neq J\subseteq\{1,\dots,n+r\}$ is called *f-relevant* (f for future) if $\partial^J_- F\neq\emptyset$, i.e., $R^J\neq\emptyset$ and $\mathbf{a}^J\notin\overset{\circ}{F}$.

Lemma 2 *If* $I \subsetneq J$ *are both f-relevant, then* $\partial^J_- F\subsetneq\partial^I_- F$*; i.e., for every* $i\in J$ *there is at least one coordinate such that* $a^J_j = a^i_j\geq a^k_j$ *for all* $k\in J$. □

In particular, we obtain the *boundary stratification* $\partial_- F = \bigcup_{J \text{ f-relevant}} \partial^J_- F$.

Every f-relevant subset $\emptyset\neq J\subseteq\{1,\dots,n+r\}$ comes with a *partition* p^J of the set $\{1,\dots,n\}$: $p^J(i) = \{j | 1\leq j\leq n,\ a^J_j = a^i_j\}$. In other words: $j\in p^J(i)$ if and only if $a^i_j = a^J_j = \max\{a^k_j | k\in J\}$.

Lemma 3 • *1. For every f-relevant subset* $\emptyset\neq J\subseteq\{1,\dots,n+r\}$*,* p^J *is in fact a partition of* $\{1,\dots,n\}$.

• *2. The stratification (3.1) of* $\partial_- F$ *above can be described as follows:*

$$\mathbf{x}\in\partial^J_- F \Leftrightarrow \forall i\in J\quad\exists j\in p^J(i) : x_j = a^J_j = a^i_j.$$

In other words:

$\mathbf{x}\in\partial^J_- F$ *if and only if* x_j *is minimal in* R^J $(x_j = a^J_j)$ *for at least one* $j\in p^J(i)$.

The stratification (3.1) above allows us to describe the local future $J^+_0(\mathbf{x})$ of a point $\mathbf{x}\in\partial_- F$:

Proposition 2 *Let* $\mathbf{x}\in\partial^J_- F$*. Then,* $J^+_0(\mathbf{x})\subset\partial^J_- F$*:*

$$\mathbf{y} = (y_1,\dots,y_n)\in J^+_0(\mathbf{x})\Rightarrow:\quad \forall i\in J\ \exists j\in p^J(i) : x_j = y_j = a^i_j.$$

3.2 Deadlocks

Using the geometrical insight gained from the stratification, we give another description of deadlocks and unsafe areas. Deadlock points can now be found as those $\mathbf{x} \in \partial_- F$ with $J^+(\mathbf{x}) = J_0^+(\mathbf{x}) = \{\mathbf{x}\}$.

Proposition 3 *A point $\mathbf{x} \in \partial_- F$ is a deadlock if and only if $\mathbf{x} \neq \mathbf{1}$ and there is an f-relevant n-*element *index set $J = \{i_1, \ldots, i_n\}$, and $\mathbf{x} = \mathbf{a}^J = [a_1^J, \ldots, a_n^J] = \min(R^{i_1} \cap \cdots \cap R^{i_n})$. In that case, $\partial_-^J(F)$ is the one point set $\{\mathbf{a}^J\}$.*

As an immediate consequence, we get a method to avoid deadlocks that is easy to check:

Corollary 1 *A forbidden region $F = \bigcup_1^{n+r} R^i \subset I^n$ has a deadlock-free complement $X = I^n \setminus F$ if and only if for any index set $J = \{i_1, \ldots, i_n\}$ with $|J| = n$*

$$R^J = R^{i_1} \cap \cdots \cap R^{i_n} = \emptyset \text{ or } R^J = \{1\} \text{ or } \min R^J \in \overset{\circ}{F} .$$

$\Box$

3.3 Unsafe regions

The boundary stratification gives a very efficient way of describing n-rectangles "under" a deadlock that consist entirely of unsafe points:

Let $J = \{i_1, \ldots, i_n\} \subset \{1, \ldots, n+r\}$ denote an n-element index set with $\partial_-^J(F) = \{\mathbf{a} = (a_1^J, \ldots, a_n^J) = (a_1^{i_1}, \ldots, a_n^{i_n}) = \min R^J, \}$, i.e, $\mathbf{a}$ is a deadlock. For every $1 \leq j \leq n$, we choose $\widetilde{a_j^J}$ as the "second largest" of the $a_j^{i_k}$, i.e.,

$$\widetilde{a_j^J} = a_j^{i_s} \text{ with } a_j^{i_k} \leq a_j^{i_s} < a_j^J \text{ for } a_j^{i_k} \neq a_j^J.$$

We associate to $\mathbf{a}$ the n-rectangle $U_{\mathbf{a}} = [\widetilde{a_1^J}, a_1^J] \times \cdots \times [\widetilde{a_n^J}, a_n^J]$.

Proposition 4 *The "half-open" $n-rectangle$ $U_{\mathbf{a}} \setminus \partial_-(U_{\mathbf{a}}) =]\widetilde{a_1^J}, a_1^J] \times \cdots]\widetilde{a_n^J}, a_n^J]$ is* unsafe*, i.e., every dipath in I^n from a point $\mathbf{x} \in (U_{\mathbf{a}} \setminus \partial_-(U_{\mathbf{a}}))$ will enter $\overset{\circ}{F}$.*

In general, the n-rectangle $U_{\mathbf{a}}$ will be considerably larger than the n-rectangles from the graph algorithm; it will contain several of the n-rectangles in the partition $\mathcal{R}$. This is where we gain in efficiency: look at Figures 4, 5, 6 and 7. They describe the 3 iterations needed in the following streamlined algorithm, whereas the first algorithm needed 26 iterations (two for each thirteen unsafe 2-rectangles).

In analogy with the graph algorithm we can now describe an algorithm finding the *complete unsafe region $U \subset I^n$* as follows: Find the set $\mathcal{D}$ of deadlocks in X and, for every deadlock $\mathbf{a} \in \mathcal{D}$, the unsafe n-rectangle $U_{\mathbf{a}}$. Let $F_1 = F \cup \bigcup_{\mathbf{a} \in \mathcal{D}} U_{\mathbf{a}}$. Find the set $\mathcal{D}_1$ of deadlocks in $X_1 = X \setminus F_1 \subset X$, and, for every deadlock $\mathbf{a} \in \mathcal{D}_1$, the unsafe n-rectangle $U_{\mathbf{a}}$. Let $F_2 = F_1 \cup \bigcup_{\mathbf{a} \in \mathcal{D}_1} U_{\mathbf{a}}$ etc.

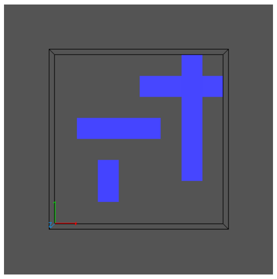


Figure 4: The forbidden region

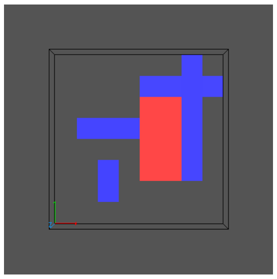


Figure 5: First step of the algorithm

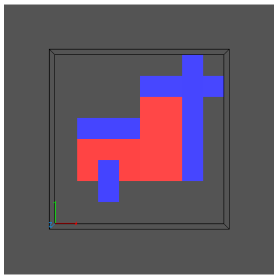


Figure 6: Second step of the algorithm

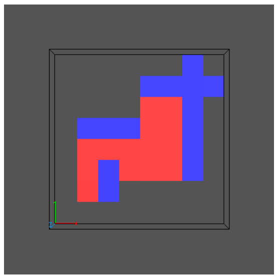


Figure 7: Last step of the algorithm

This algorithm stops after a finite number n of loops ending with a set $U = F_n$ and such that $X_n = X \setminus U$ does no longer contain any deadlocks. The set $U \setminus \partial_-(U)$ consists precisely of the forbidden and of the unsafe points.

The example of Figure 4 demonstrates that there may be arbitrarily many loops in this second algorithm – even in the case of a 2-dimensional forbidden region associated to a simple PV-program: Obviously, the "staircase" in Figure 4 (corresponding to the PV term `example`, see Appendix A) producing more and more unsafe n-rectangles can be extended ad libitum by introducing extra rectangles R^i to F along the "diagonal".

We now show the applicability of the method by exemplifying it on our toy PV language.

4 Implementation of the geometric algorithm

4.1 The semantics

Now we have a dual view on PV terms. Instead of representing the allowed n-rectangles, we represent the forbidden n-rectangles only. First, let $T = X_1 \mid \cdots \mid X_n$ (for some $n \geq 1$) be a pure term (i.e. no recursion nor plus operator) of our language such that all its subterms are pure as well. We consider here the X_i $(1 \leq i \leq n)$ to be strings made out of letters of the form Pa or Vb, $(a, b \in \mathcal{O})$. $X_i(j)$ will denote the jth letter of the string X_i. Supposing that the length of the strings X_i $(1 \leq i \leq n)$ are integers l_i, the semantics of $Prog$ is included in $[0, l_1] \times \cdots \times [0, l_n]$. A description of $[\![Prog]\!]$ from above can be given by describing inductively what should be digged into this n-rectangle. The semantics of our language can be described by the simple rule, $[k_1, r_1] \times \cdots \times [k_n, r_n] \in [\![X_1 \mid \cdots \mid X_n]\!]_2$ 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 $j \in V$.

Now we have to take care of unpure terms. Geometrically, a branching between two sets of n concurrent processes can be represented in an $\mathbb{R}^{n+s}$, with s big enough, with the coordinate-wise ordering as in the "pure case". In our language, a branching comes from a choice operator in a sequential process, so s can be taken equal to one. Formally, the forbidden n-rectangles in $[\![X_1 \mid \cdots \mid Y_i + Z_i \mid \cdots \mid X_n]\!]_2$ are $[0, 0] \times [\![X_1 \mid \cdots \mid Y_i \mid \cdots \mid$

$X_n]\!]_2 \bigcup [\![X_1 \cdots \mid Z_i \mid \cdots \mid X_n]\!]_2 \times [0,0]$.

Things are more complex when it comes to recursive equations. A loop (with the right pre-order indicating the progress of time) cannot be embedded into an $\mathbb{R}^n$ with the partial order induced by the order on each coordinate. But it can be embedded into a quotient of this partial order. So we have to change the semantic domain we use to be a pair of a set of forbidden n-rectangles together with a sequence of n equivalence relations, describing the identifications of the local times (or the foldings, or the cycles) that the recursive equations enforce.

The semantics of pure terms is unchanged, except we have an extra component in the semantics, $([k_1, r_1] \times \cdots \times [k_n, r_n], (\emptyset, \cdots, \emptyset)) \in [\![X_1 \mid \cdots \mid X_n]\!]_2$ 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 $j \in V$. This means that for pure terms, no identification of local times is made so all relations are empty.

When a recursive call to the same process variable is found $X_i(j) = X_i$ for some local time $j \geq 1$ then the ith equivalence relation is updated to contain also the equivalence $1 R_i j$.

4.2 The implementation

A general purpose library for manipulating finite unions of n-rectangles (for any n) has been implemented in C. A n-rectangle is represented as a list of n closed intervals. Regions (like the forbidden region) are represented as lists of n-rectangles. We also label some n-rectangles by associating to them a region. Labeled regions are then lists of such labeled n-rectangles.

Let us look first at the semantics of pure terms. Three arrays are constructed from the syntax in the course of computation of the forbidden region. For a process named `i` and an object (semaphore) named `j`, `tP[i][j]` is updated during the traversing of the syntactic tree to be equal to the ordered list of times at which process `i` locks semaphore `j`. Similarly `tV[i][j]` is updated to be equal to the ordered list of times at which process `i` unlocks semaphore `j`. Finally, an array `t[i]` gives the maximal (local) time that process `i` runs.

For all objects a, we build recursively all partitions as in §4.1 of $\{1, \cdots, n\}$ into a set U of $s(a) + 1$ processes that lock a and V such that $U \cup V = \{1, \cdots, n\}$ and $U \cap V = \emptyset$. For each such partition (U, V) we list all corresponding pairs (Pa, Va) in each process X_i, $i \in U$. As we have supposed that in our programs, all processes must lock exactly once an item before releasing it, these pairs correspond to pairs (`tP[`i`][`a`]`$_j$,`tV[`i`][`a`]`$_j$) for j ranging over the elements of the lists `tP[`i`][`a`]` and `tV[`i`][`a`]`. Then we deduce the n-rectangle in the forbidden region for each partition and each such pair.

For the unpure terms, we choose first a representation of the sequence of equivalence relations $(R_1, \cdots, R_n)$. As they are finitely generated by simple foldings, each of these relations R are implemented as a list l_j $(j = 1, \cdots, l)$ of ordered lists $l_{jk} \in \mathbb{N}$ $(k = 1, \cdots, m_j)$. The set $\{l_{jk} | k = 1, \cdots, m_j\}$ is exactly an equivalence class in R. We also construct this so that l_{j1} is an ordered list. The operations `min(x,y)` and `max(x,y)` in coordinate i are then quite simple. We determine for `x` and `y` their minimal representatives

x_m and y_m under R_i using the representation above: this is a l_{j1} for some suitable j or x (resp. y) themselves. Then min(x,y)$= min(x_m, y_m)$. Similarly, we can determine the maximal representatives x_M and y_M of x and y and then max(x,y)$= max(x_M, y_M)$.

Now we have to handle extra-coordinates induced by the operator plus. In fact, instead of using the mathematical representation of n-rectangles, we can describe the branching structure of the processes in a separate manner. Basically, we represent the pre-order determining the time flow together with the forbidden regions by a tree whose leaves consist of an n-rectangle together with an equivalence relation (represented as explained above). Each branching in this tree represents a plus operation. At the leaves is the semantics of all terms with no plus. In order to do that, we unfold the syntactic tree (just once for the moment) of the processes, and each time we traverse a plus node, we create a branching in this tree. Then at some point we end with a pure term whose subterms are pure (or contain process variables). We apply the rule for the semantics of such terms for each leaf, also deriving the equivalence relation for each process.

4.3 Implementation of the second deadlock algorithm

The implementation uses a global array of labeled regions called pile: pile[0],...,pile[n-1] (n being the dimension we are interested in). The idea is that pile[0] contains at first the initial forbidden region, pile[1] contains the intersection of exactly two distinct regions of pile[0], etc., pile[n-1] contains the intersection of exactly n distinct regions of pile[0].

The algorithm is incremental. In order to compute the effect of adding a new forbidden n-rectangle S the program calls the procedure complete(S,∅). This calls an auxiliary function derive also described in pseudo-code below, in charge of computing the unsafe region generated by a possible deadlock created by adding S to the set of existing forbidden regions. The resulting forbidden and unsafe region is contained in pile[0].

```
complete(S,l)
   if S is included into a X in pile[0] return
   for i=n-2 to 0 by -1 do pile[i+1]=intersection(pile[i]\l,S)
   pile[0]=union(pile[0],S)
   for all X in pile[n-1] do pile[n-1]=pile[n-1]\X
                             derive(X)
```

The intersection of a labeled region R (such as pile[i] above) with a n-rectangle S gives the union of all intersections of n-rectangles X in R (which are also n-rectangles) labeled with the concatenation of the label of X with S (which is a region). Therefore labels of elements of regions in pile are the regions whose intersection is exactly these elements.

Now, derive(X) takes care of deriving an unsafe region from an intersection X of n forbidden or unsafe distinct n-rectangles. Therefore X is a labeled n-rectangle, whose labels is X1,...,Xn (the set of the n n-rectangle which it is the intersection of). We call X(i) the projection of X on coordinate i.

```
derive(X)
   for all i do yi=max({Xj(i) / j=1,...,n}\{X(i)})
   Y=[y1,X(1)]x...x[yn,X(n)]
   if Y is not included in one of the Xj complete(Y,(X1,...,Xn))
```

This last check is done when computing all `yi`. We use for each `i` a list `ri` of indexes `j` such that `yi=Xj(i)` (there might be several). If the intersection of all `ri` is not empty then `Y` is included into one of the `Xj`. It is to be noticed that this algorithm considers cycles (recursive calls) as representing (unbounded) finite computations.

4.4 Complexity issues

The entire algorithm consists of 3 parts: The first establishes the initial list `pile[0]` of forbidden n-rectangles, the second works out the complete array `pile` – including the deadlocks encoded in `pile[n]` –, and the third adds pieces of the unsafe regions, recursively.

Let again n denote the number of processes (the dimension of the state space), and r the number of n-rectangles. From a complexity viewpoint, the first step is negligeable; finding the n-rectangles involves $C^n_{s(a)+1}$ searches in the syntactic tree for every shared object a – in each of the n coordinates.

The array `pile` involves the calculation of $S(r,n) = \sum_{i=1}^n C^r_i$ intersections, each of them needing comparisons in n coordinates. Note that these comparisons show which of the intersections are empty, as well. To find the deadlocks, one has to compare (n coordinates of) the at most C^r_n non-empty elements in `pile[n]` with the r elements in `pile[0]`. Adding pieces of unsafe regions in the third step involves the same procedures with an increased number r of n-rectangles. The worst-case figure $S(r,n)$ above can be crudely estimated as follows: $S(r,n) \leq 2^r$ for all n, and $S(r,n) \leq nC^r_n$ for $r > 2n$ – which is a better estimate only for $r >> 2n$.

Remark that the algorithm above has a total complexity roughly proportional to the *geometric complexity* of the forbidden region. The latter may be expressed in terms of the *number of non-empty intersections* of elementary n-rectangles in the forbidden region. This figure reflects the degree of synchronization of the processes, and will be much lower that $S(n,r)$ for a well-written program. We conjecture, that the number of steps in *every* algorithm detecting deadlocks and unsafe regions is bounded below by this geometric complexity. On the other hand, for the analysis of big concurrent programs, this geometric complexity will be tiny compared to the number of states to be searched through by a traversing strategy.

4.5 Benchmarks

The program has been written in C and compiled using `gcc -O2` on an Ultra Sparc 170E with 496 Mbytes of RAM, 924 Mbytes of cache. All times have been measured using the `ddi.h` library and the virtual times as provided by the command `gethrvtime()`. The dynamic data was created using the standard `malloc()` function of the `bsdmalloc` library.

No particular optimization was made here. Timings have been rounded to the nearest hundredth of a second but are not more precise than a couple hundredths of a second.

In the following table, dim represents the dimension of the program checked, #forbid is the number of forbidden n-rectangles found in the semantics of the program, t semantics is the time it took to find these forbidden n-rectangles, t unsafe is the time it took to find the unsafe region and #unsafe is the number of n-rectangles found to be unsafe (they now encapsulate many of the "unit" n-rectangles found by the first deadlock detection algorithm). These measures have been taken on a first implementation which does not include yet the branching and looping constructs.

program	dim	#forbid	t semantics	t unsafe	#unsafe
example	2	4	0.020	0	3
stair2	2	6	0.020	0	15
stair3	3	18	0.010	0	4
stair3'	3	6	0.030	0	0
lipsky	3	6	0.020	0	0
3phil	3	3	0.020	0	1
4phil	4	4	0.030	0	1
5phil	5	5	0.030	0	1
6phil	6	6	0.030	0	1
16phil	16	16	0.030	0.030	1
32phil	32	32	0.030	0.420	1
64phil	64	64	0.040	1.520	1
128phil	128	128	0.100	26.490	1

5 Conclusion and future work

We have presented two algorithms for deadlock detection, including the computation of the set of states (the unsafe region) that will eventually lead to a deadlock. These algorithms were based on geometric intuition and techniques. They have been implemented, and the first one shows good comparison with ordinary reachability search with some state-space reduction techniques. But due to its complexity, this does not seem to be easily usable for very big programs (except if combined with clever abstract interpretations) or for a big number of processes (6 or 7 seems to be a maximum in general for practical use). The second algorithm has shown much better promise. Its complexity depends on the complexity of the synchronization of the processes, and not on a fake number of global states, as in most techniques used. In this regard it is much more practical. Dealing with 128 processes is not a problem if they are not synchronizing too much (as in the dining philosophers problem), but this is certainly intractable for reachability search with no clever partial order techniques(there are more than 10^{85} global states in that case). It should be noted also that these two algorithms could be enhanced by the use of some other well-known technique, like symmetry and (for the first one) some state-space reduction techniques. As the second algorithm is based on an abstract interpretation of the semantics, it should be developed for the use on real concurrent languages in conjunction with other well-known abstract interpretations. This is for future work. Also this should be linked with a full description of "schedules" and verification of safety properties of concurrent programs as hinted in [Gun94, Gou95b, FR96] using the geometric notions developed in this article.

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).

A The examples detailed

You can check the implementations and the examples at http://www.dmi.ens.fr/~goubault.

- The dining philosophers' problem. The source below is for three philosophers, the next one is for five. The way others of these examples are generated should be obvious from these examples.

  ```
  /* 3 philosophers ''3phil'' */
  A=Pa.Pb.Va.Vb
  B=Pb.Pc.Vb.Vc
  C=Pc.Pa.Vc.Va
  ```

  The output giving the unsafe region is then,

  ```
  (P(b).V(a).V(b)|P(c).V(b).V(c)|P(a).V(c).V(a),[c,0][b,0][a,0])
  ```

  ```
  /* 5 philosophers ''5phil'' */
  A=Pa.Pb.Va.Vb
  B=Pb.Pc.Vb.Vc
  C=Pc.Pd.Vc.Vd
  D=Pd.Pe.Vd.Ve
  E=Pe.Pa.Ve.Va
  ```

- This is example of Figure 4.

  ```
  /* ''example'' */
  A=Pa.Pb.Vb.Pc.Va.Pd.Vd.Vc
  B=Pb.Pd.Vb.Pa.Va.Pc.Vc.Vd
  ```

- This is the classical Lipsky/Papadimitriou example (see [Gun94]) which produces no deadlock.

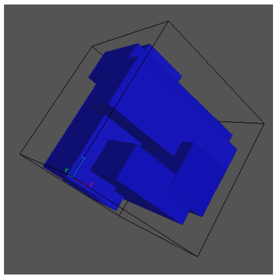


Figure 8: The Lipsky/Papadimitriou example

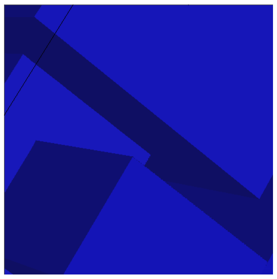


Figure 9: A close-up to a hole in the forbidden region

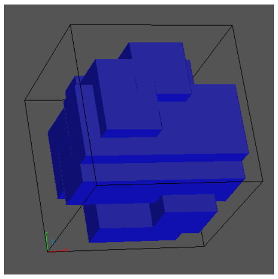


Figure 10: Turning around

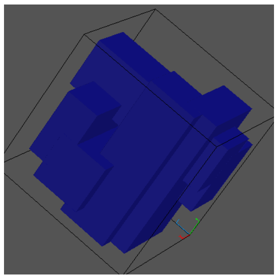
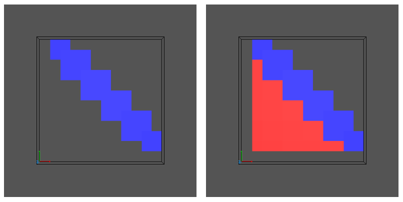


Figure 11: Behind, notice the exit in the hole

```
/* ``lipsky'' */
A=Px.Py.Pz.Vx.Pw.Vz.Vy.Vw
B=Pu.Pv.Px.Vu.Pz.Vv.Vx.Vz
C=Py.Pw.Vy.Pu.Vw.Pv.Vu.Vv
```

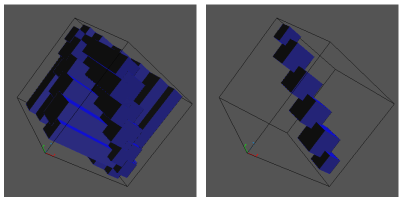
- This is a staircase (worst complexity case for the second algorithm).

```
/* ``stair2'' */
A=Pa.Pb.Va.Pc.Vb.Pd.Vc.Pe.Vd.Pf.Ve.Vf
B=Pf.Pe.Vf.Pd.Ve.Pc.Vd.Pb.Vc.Pa.Vb.Va
```



- This is a 3-dimensional staircase. Notice that if you declare all semaphores used (a, b, c, d, e and f) to be initialized to 2 (example "stair3"'), there is no 3-deadlock.

```
/* ``stair3'' */
A=Pa.Pb.Va.Pc.Vb.Pd.Vc.Pe.Vd.Pf.Ve.Vf
B=Pf.Pe.Vf.Pd.Ve.Pc.Vd.Pb.Vc.Pa.Vb.Va
C=Pf.Pe.Vf.Pd.Ve.Pc.Vd.Pb.Vc.Pa.Vb.Va
```



References

[ABC+91] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Soft. Eng.*, 17(11):1204–1222, November 1991.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the Fifth Annual IEEE Symposium on Logic and Computer Science*, pages 428–439. IEEE Press, 1990.

[BG96] B. Boigelot and P. Godefroid. Model checking in practice: An analysis of the access.bus protocol using spin. In *Proceedings of Formal Methods Europe'96*, volume 1051, pages 465–478. Springer-Verlag, Lecture Notes in Computer Science, March 1996.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. *Principles of Programming Languages 4*, pages 238–252, 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CCA96] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin. An empirical comparison of static concurrency analysis techniques. Technical Report 96-84, Department of Computer Science, University of Massachusetts, August 1996.

[Cor96] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), March 1996.

[CR87] S.D. Carson and P.F. Reynolds. The geometry of semaphore programs. *ACM TOPLAS*, 9(1):25–53, 1987.

[Cri95] R. Cridlig. Semantic analysis of shared-memory concurrent languages using abstract model-checking. In *Proc. of PEPM'95*, La Jolla, June 1995. ACM Press.

[DC94] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proc. of the Second Symposium on Foundations of Software Engineering*, pages 62–75, December 1994.

[Dij68] E.W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–110. Academic Press, New York, 1968.

[FR96] L. Fajstrup and M. Raußen. Some remarks concerning monotopy of increasing paths. unpublished manuscript, Aalborg University, 1996.

[GHP95] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. In *Formal Methods and System Design*, volume 7, pages 1–15. Kluwer Academic Publishers, November 1995.

[GJ92] E. Goubault and T. P. Jensen. Homology of higher-dimensional automata. In *Proc. of CONCUR'92*, Stonybrook, New York, August 1992. Springer-Verlag.

[GJM$^+$97] H. Garavel, M. Jorgensen, R. Mateescu, Ch. Pecheur, M. Sighireanu, and B. Vivien. Cadp'97 – status, applications and perspectives. Technical report, Inria Alpes, 1997.

[Gou95a] E. Goubault. *The Geometry of Concurrency.* PhD thesis, Ecole Normale Supérieure, 1995. to be published, 1997, also available at http://www.dmi.ens.fr/~goubault.

[Gou95b] E. Goubault. Schedulers as abstract interpretations of HDA. In *Proc. of PEPM'95*, La Jolla, June 1995. ACM Press, also available at http://www.dmi.ens.fr/~goubault.

[GPS96] P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, July 1996.

[Gun94] J. Gunawardena. Homotopy and concurrency. *Bulletin of the EATCS*, 54:184–193, 1994.

[HS95] M. Herlihy and S.Rajsbaum. Algebraic Topology and Distributed Computing. A Primer. volume 1000 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

[HS96] M. Herlihy and N. Shavit. The topological structure of asynchronous computability.

Technical report, Brown University, Providence, RI, January 1996.

[LP81] W. Lipski and C.H. Papadimitriou. A fast algorithm for testing for safety and detecting deadlocks in locked transaction systems. *Journal of Algorithms*, 2:211–226, 1981.

[MR97] S. Melzer and S. Roemer. Deadlock checking using net unfoldings. In *Proc. of Computer Aided Verification.* Springer-Verlag, 1997.

[Pra91] V. Pratt. Modeling concurrency with geometry. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages.* ACM Press, 1991.

[Val89] A. Valmari. Eliminating redundant interleavings during concurrent program verification. In *Proc. of PARLE*, volume 366, pages 89–103. Springer-Verlag, Lecture Notes in Computer Science, 1989.

[Val91] A. Valmari. A stubborn attack on state explosion. In *Proc. of Computer Aided Verification*, number 3, pages 25–41. AMS DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1991.

[vG91] R. van Glabbeek. Bisimulation semantics for higher dimensional automata. Technical report, Stanford University, Manuscript available on the web as http://theory.stanford.edu/~rvg/hda, 1991.

[YY91] W. J. Yeh and M. Young. Compositional reachability analysis using process algebras. In *Proc. of the symposium on Testing, Analysis and Verification*, pages 178–187. ACM Press, October 1991.