DETECTING DEADLOCKS IN CONCURRENT SYSTEMS

LISBETH FAJSTRUP, ERIC GOUBAULT AND MARTIN RAUSSEN

ABSTRACT. We study deadlocks using geometric methods based on generalized process graphs [11], i.e., cubical complexes or Higher-Dimensional Automata (HDA) [23, 24, 30, 35], 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. Its basis is a characterization of deadlocks using dual geometric properties of the "forbidden region".

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 [5, 6], 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 sometimes using other models, like Petri nets [29]. 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) [33], *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 [17], [18], [34], 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 [2, 3, 16]. 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* [36], or based on *dataflow analysis* [12], or on *integer programming techniques* [1] (but this in general only relies on necessary conditions for deadlocking behaviors).

Based on some old ideas [11] and some new semantic grounds [21], [23], [24], [30], [35], (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 [9]. Some assessments about its practical use, based on a first implementation applied to simple semaphore programs and also based on some general complexity reasons are made in §3.4 and §5.5.

In some ways, this deadlock detection algorithm (which determines the so-called "unsafe region" made of all states bound to run some time or another into a deadlock) is still a combinatorial search, which only takes advantage of the truly-concurrent representation of actions.

In §4, we propose a new algorithm based on an *abstraction* (in the sense of *abstract interpretation* [7, 8]) 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.

We believe that this technique, which is assessed in §5.4 and §5.5 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 §3.1) by suitable compositions and reduced products with other abstract interpretations, as sketched in §7.1.

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 [24] – 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 [25, 26] and others to find new *lower bounds* and *impossibility results* for distributed and concurrent computation.

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). E. Goubault's work was also done partly while at C.N.R.S, Ecole Normale Supérieure and while visiting Aalborg University.

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 [4], and we stick to their terminology. The main idea in [4] 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 $\mathbb{R}^n$ with the usual partial order. 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 [11], 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 (local 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 $(0, \ldots, 0)$ and the final state is $(1, \ldots, 1)$. An example consisting of the two transactions $T_1 = P_a P_b V_b V_a$ and $T_2 = P_b P_a V_a V_b$ gives rise to the two dimensional *progress graph* of Figure 1.

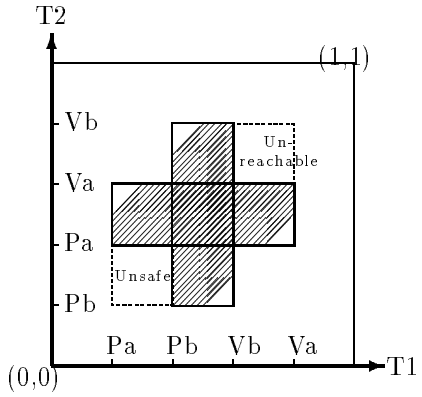


FIGURE 1. Example of a progress graph

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 $(1, 1)$ or $Pb, Pa)$ 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 [28] attempted to exploit geometric properties of forbidden regions to find deadlocks in database-transaction systems. But the algorithm in [28] does not generalize to systems composed of more than two processes. S. D. Carson and P. F. Reynolds indicated in [4] 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 [30] and R. van Glabbeek [35] (cf. E. Goubault [21] 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 is done entirely in the algorithmic and implementation part, §3.

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 $R = [a_1, b_1] \times \cdots \times [a_n, b_n]$ a *hyperrectangle*[1], and we consider a set $F = \bigcup_1^r R^i$ that is a finite union of hyperrectangles $R^i = [a_1^i, b_1^i] \times \cdots \times [a_n^i, b_n^i]$. 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 $\mathbf{1} = (1, \ldots, 1) \notin F$.

Remark 1. We consider the interior of F as a subspace of I^n; e.g. the interior of $R = [1/4, 1/2] \times [0, 1]$ in I^2 is $\overset{\circ}{R} =]1/4, 1/2[\times [0, 1]$

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

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

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

• 4. 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.

• 5. A point $\mathbf{x} \in X$ is a *deadlock* if $\uparrow \mathbf{x} = \{\mathbf{x}\}$ and $\mathbf{x} \neq \mathbf{1}$.

In semaphore programs, the hyperrectangles 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 (i.e., a directed graph) where deadlocks are leaves (i.e., the nodes of the digraph, if any, that have no successors) 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 hyperrectangles $I_{1k_1} \times \cdots \times I_{nk_n}$ with a partial ordering given by

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

[1] which has the property that all its faces are parallel to the coordinate axes. In dimension 2 this is called an isothetic rectangle [31]

Remark 2.
1. We will not worry about the fact, that there are nonempty intersections of the hyperrectangles. Defining everything with halfopen intervals would be an unnecessary complication.
2. Admissibility with respect to the forbidden region F can be defined in terms of these hyperrectangles: Two points in the same hyperrectangle of the partition above are either both admissible or both unsafe points.
3. The hyperrectangle $R_{\mathbf{1}}$ containing $\mathbf{1}$ is the *global maximum* for $\mathcal{R}$, the hyperrectangle $R_{\mathbf{0}}$ containing $\mathbf{0}$ is the *global minimum.*

The partially ordered set $(\mathcal{R}, \leq)$ can be interpreted as a *directed, acyclic graph*, denoted $(\mathcal{R}, \rightarrow)$: Two hyperrectangles $R, R' \in \mathcal{R}$ are connected by an edge from R to R' – denoted $R \rightarrow R'$ – 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 $(\mathcal{R}', \rightarrow)$. Particularly important is the subgraph $\mathcal{R}_{\bar{F}}$ consisting of all hyperrectangles $R \subset X = I^n \setminus \overset{\circ}{F}$.

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. A hyperrectangle $R \in \mathcal{R}_{\bar{F}}$ is called a* deadlock *hyperrectangle if $R \neq R_{\mathbf{1}}$, and if R is a local maximum with respect to $\mathcal{R}_{\bar{F}}$. An* unsafe *hyperrectangle $R \in \mathcal{R}_{\bar{F}}$ is characterized by the fact, that* any *directed path α starting at R hits a deadlock hyperrectangle sooner or later* [4].

Remark 3.
1. An element $R \in \mathcal{R}_{\bar{F}}$ is a deadlock if $R \neq R_{\mathbf{1}}$, and if all its upper neighbors in $\mathcal{R}$ are contained in F. Deadlocks in $\mathcal{R}_{\bar{F}}$ are the maximal corners of the unsafe regions.
2. *Unreachable* hyperrectangles can be defined similarly. Local minima ($\neq R_{\mathbf{0}}$) are their minimal corners.

In order to find the set $\mathcal{U}$ of all unsafe points – which is the union of *all* unsafe hyperrectangles – 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 hyperrectangles (local maxima distinct from $R_{\mathbf{1}}$) 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 hyperrectangles in $I^n \setminus F_1$, i.e., after removing S_1. (4) Find the set S_2 of all deadlock hyperrectangles with respect to $\mathcal{R}_{F_1}$. Let $F_2 = F_1 \cup S_2$. Carry on with 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* hyperrectangles, 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 $R_{\mathbf{1}}$.*

Proof. Only the last assertion has still to be shown. The set $\mathcal{A}(F)$ is non-empty since it contains the global maximum $R_{\mathbf{1}}$. Now fix any directed path starting from an arbitrary hyperrectangle in $\mathcal{A}(F)$. It will run through (finitely many) hyperrectangles in $\mathcal{A}(F)$ until it reaches a local maximum. This local maximum must be the global maximum $R_{\mathbf{1}}$, since $\mathcal{A}(F)$ does not contain any deadlock. □

In order to show the applicability of the previous method, we explain how to give semantics of a toy language in terms of these forbidden regions, how to implement it, and how to implement the deadlock detection algorithm.

3. Implementation of the combinatorial approach

3.1. **The language.** We consider in the following the language PV whose syntax is defined below. Given a set of objects $\mathbf{O}$ (like shared memory locations, synchronization barriers, semaphores, control units, printers etc.) and a function $s : \mathbf{O} \to \mathbb{N}^+$ 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 $\mathbf{V}$: they are of the form $X = Proc_d$ where $Proc_d$ is the process definition formally defined as,

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

(ϵ being the empty string, a being any object of $\mathbf{O}$, Y being any process variable in $\mathbf{V}$) A PV program is any parallel combination of these PV processes, $Prog = Proc \mid (Proc \mid Proc)$. The typical example in shared memory concurrent programs is $\mathbf{O}$ being the set of shared variables and for all $a \in \mathbf{O}$, $s(a) = 1$. 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.

3.2. **The semantics.** The semantics of the PV language as a graph of hyperrectangles is as follows[2]. An environment is a function $\rho : \mathbf{O} \to \mathbb{N}$, whose value for an object a represents the number of times a can still be accessed by the processes. A hyperrectangle or state of the program is a pair (C, ρ) where C is an element of the language, ρ is a context. Basically, C represents the program that remains to be executed and ρ is the current context in which C has to be executed.

The representation of the graph of hyperrectangles is done by explicitly representing the glueing faces which define the "neighboring" relation between hyperrectangles (as in §2.3). Look at Figure 2 for an explanation in the case of the semantics of $(Pa.Va \mid Pa.Va)$. The collection of faces of each hyperrectangle is separated in n start faces, here for example for the 2-rectangle (i.e., a hyperrectangle of dimension 2) A, $d^0_0(A)$ and $d^0_1(A)$, and n end faces, here $d^1_0(A)$ and $d^1_1(A)$. The order between the different hyperrectangles, as sketched in this example by the graph at the right-hand side of Figure 2, is generated by the relation "having a d^1 face equal to a d^0 face". Here $A \leq B$ because $c = d^0_1(B) = d^1_0(A)$. This encoding is standard in the HDA framework where faces are $(n-1)$-transitions and hyperrectangles are n-transitions (see [21] for more explanations).

Let us separate out our semantics in two distinct phases. Consider first the "pure" terms consisting of those terms for which the syntactic tree of each process begins by a sequential composition of a P or a V with any term. Any set of k PV processes in parallel $X_1 \mid \cdots \mid X_k$ may generate k-rectangles according to the environment it is executed in. Supposing none of these processes are empty, we write $X_i = Q_i a_i.Y_i$, $1 \leq i \leq k$, where Q_i is P or V, $a_i \in \mathbf{O}$ and Y_i is a process. We then have the following semantic equation describing the semantics $[\![X_1 \mid \cdots \mid X_k]\!]\rho$ in environment ρ. If for all $a \in \mathbf{O}$, $\rho(a) \geq 0$,

$$\begin{array}{rl} [\![X_1 \mid \cdots \mid X_k]\!]\rho = (X_1 \mid \cdots \mid X_k, \rho) + & [\![Y_1 \mid X_2 \mid \cdots \mid X_k]\!]\rho_1 + \cdots \\ & + [\![X_1 \mid \cdots \mid X_{k-1} \mid Y_k]\!]\rho_k \end{array}$$

[2]This had already been "pictured" under the name of process graphs by E.W.Dijkstra [11], Carson and Reynolds [4], J. Gunawardena [24] in the case of terms with no choice operator nor recursive equations. The formal semantics in terms of this graph of hyperrectangles, or HDA [21] is new here.

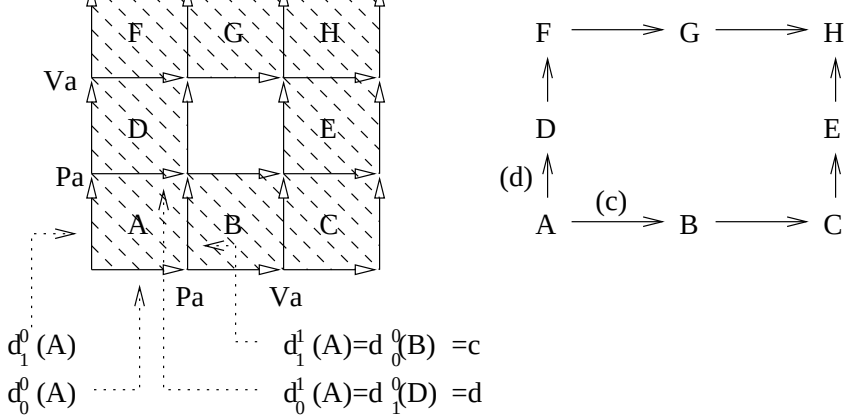


FIGURE 2. Semantics of $(Pa.Va \mid Pa.Va)$ as a discretisation of its geometry (left), as a graph of hyperrectangles (right).

where ρ_i, $1 \leq i \leq k$ is such that $\rho_i(b) = \rho(b)$ for all $b \in \mathbf{O}$, $b \neq a_i$, and $\rho_i(a_i) = \rho(a_i) - 1$ if $Q_i = P$ or $\rho_i(a_i) = \rho(a_i) + 1$ if $Q_i = V$. If there is an $a \in \mathbf{O}$, $\rho(a) < 0$,

$$[\![X_1 \mid \cdots \mid X_k]\!]\rho = [\![Y_1 \mid X_2 \mid \cdots \mid X_k]\!]\rho_1 + \cdots + [\![X_1 \mid \cdots \mid X_{k-1} \mid Y_k]\!]\rho_k$$

with the same environments ρ_i, $1 \leq i \leq k$.

These equations should be understood as follows. $(X_1 \mid \cdots \mid X_k, \rho)$ is a k-rectangle, which is not forbidden if and only if all k processes can progress. This is not the case if one of the processes is waiting for an object to be released (in the second case, there is an $a \in \mathbf{O}$ such that $\rho(a) < 0$). If we want to generate only reachable states, then it is enough to forget the second semantic equation. In the first case, the k start boundaries and the k end boundaries of dimension $k-1$ of this k-rectangle are[3], $d_i^0(X_1 \mid \cdots \mid X_k, \rho) = (X_1 \mid \cdots \mid \hat{X}_i \mid \cdots \mid X_k, \rho, i)$, (the face at the right-hand side is defined if the hyperrectangle at the left-hand side is defined), and $d_i^1(X_1 \mid \cdots \mid X_k) = (X_1 \mid \cdots \mid \hat{X}_i \mid \cdots \mid X_k, \rho_i, i)$. This last component for the faces is not needed in general, but it permits to unfold entirely the graph of cubes (thus the semantics does not create fake unfoldings that the verification algorithms would believe to be divergences – see the discussion of §3.3.1 and §3.3.2).

Now for the "non-pure" terms, we use the following two rules in order to get pure terms,

(Elimination of process variables)

$$[\![X_1 \mid \cdots \mid Y.Y_i \mid \cdots \mid X_k]\!]\rho = [\![X_1 \mid \cdots \mid Proc_Y.Y_i \mid \cdots \mid X_k]\!]\rho$$

(Elimination of plus)

$$[\![X_1 \mid \cdots \mid Y_i + Z_i \mid \cdots \mid X_k]\!]\rho = [\![X_1 \mid \cdots \mid Y_i \mid \cdots \mid X_k]\!]\rho +_i [\![X_1 \mid \cdots \mid Z_i \mid \cdots \mid X_k]\!]\rho$$

The first equation eliminates the process variable Y by its definition $Proc_Y$. The second equation eliminates the choice operator in the definition of the ith process. The plus symbol at the right hand-side of this equation denotes an amalgamated sum (i.e., a union) of its two arguments, identifying the face $(X_1 \mid \cdots \mid Y_i \mid \cdots X_k, \rho, i)$ with the face $(X_1 \mid \cdots \mid Z_i \mid \cdots X_k, \rho, i)$.

Notice that using this semantic definition, we can define directly the n-transitions of a program consisting of n processes in parallel, generating also the $(n-1)$-transitions, but not the transitions of lower dimension.

3.3. **Implementation of the first deadlock algorithm.** A general purpose C library has been written to generate and manipulate graphs of hyperrectangles (in fact, any HDA). Basically such a graph is described by incidence matrices. To be more precise, $\mathbf{R}$ is represented by a 4uple $(R_{n-1}^0, R_{n-1}^1, R_n^0, R_n^1)$. R_n^i is the (sparse) matrix whose rows $R_n^i(x)$ are indexed by the hyperrectangles x (states of dimension

[3]The notation $X_1, \cdots, \hat{X}_i, \cdots$ means that we have the collection $X_1, X_2, \cdots$ except X_i.

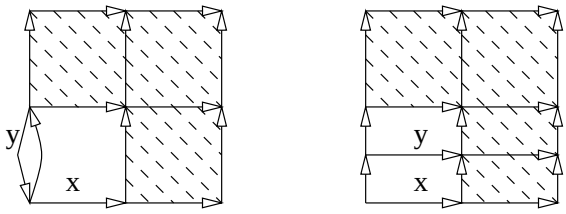


FIGURE 3. An example of cyclic behavior and its 1-unfolding

n as described in the semantics), and which contain the corresponding lower (for $i = 0$) and upper (for $i = 1$) boundaries of x. R^i_{n-1} is the co-incidence matrix whose rows $R^i_{n-1}(y)$ are indexed by the faces y (states of dimension $n-1$) and consist of the hyperrectangles whose lower boundary (for $i = 0$) contains y or whose upper boundary (for $i = 1$) contains y.

We describe here how to compute the subset D of the set of ascendants of a given set S of states such that all its descendants finally (only) reach S. We suppose that S is organized into a FIFO queue q. We can perform operations *empty*?, *enq* (for enqueue) and *deq* (for dequeue) on it which should have an obvious semantics. We suppose that S is only composed of hyperrectangles of dimension n, n fixed. It can be constructed once and for all or it can be constructed on the fly, when boundaries are demanded by the algorithm. This corresponds to the deadlock algorithm sketched in §2.3 when S is taken to be the set of forbidden hyperrectangles.

3.3.1. *Cycles as divergences.* The standard way of constructing D is to compute the ascendants as the transitive closure of the "parent" relation (by iteration) and similarly for the descendants. It is actually quite expensive and is not necessary in our case. To be more precise, the algorithm below is sound and complete, in the sense that it computes faithfully D if *there is no cycle* in the semantics, or if we consider cycles to represent finite *and* infinite paths (i.e., cycles contain non-deadlocking paths). We discuss the case when cycles represent only finite paths in §3.3.2.

We suppose that an integer m_x is associated to each hyperrectangle x generated by the semantics, such that,

- for any n-cube x in S the integer m_x is initialized to 0,
- for any other hyperrectangle, m_x is initialized to its number of sons

Then,

- the multiset P_x of hyperrectangles, parents of a given hyperrectangle x is the union of the lists $R^1_{n-1}(y)$ for $y \in R^0_n(x)$.
- the algorithm for finding D is as follows. D is empty at the beginning, then,

  [(1)] if *empty*? then we have reached the result.

  [(2)] decrement m_z by one for all $z \in P_{deq}$.

  [(3)] if in this process, one of the z considered has m_z equal to zero then add z to D and $enq(z)$.

  [(4)] loop back at point (1).

3.3.2. *Cycles as finite iterations.* Look at Figure 3 (notice that here, the forbidden region is represented by the dashed lines). If we use the deadlock algorithm of §3.3.1 on the picture at the left, then we detect no deadlock nor unsafe region. Then x has $m_x = 3$ because it has two sons in the forbidden region and the third one is y. Canceling the two forbidden 2-rectangles leaves $m_x = 1$ at the end of the algorithm and x is not detected as an unsafe 2-rectangle. It is true that x has one non-forbidden son (y) but it allows for a non-deadlocking behaviour only if we consider infinite paths through x and y. If we are only considering finite paths, then we are bound to end up blocked by the forbidden region.

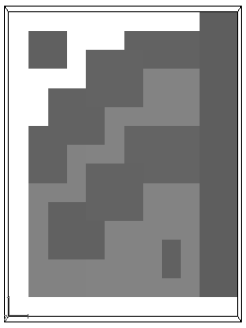


FIGURE 4. unfold once

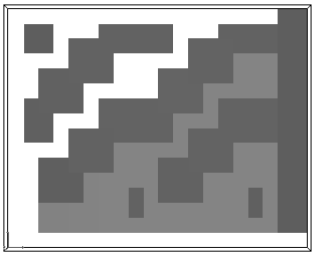


FIGURE 5. unfold twice

In fact there is no way to determine unsafe regions in that case without looking at all unfoldings of the terms, as the following example shows. We will see in Section 7.1 an answer to this problem (in the case of the second algorithm).

Example 2. `A=Pa.Vd.Pb.Va.Pd.Vb.Pe.Ve.A`
`B=Pd.A.Pf.Vd.Vf`
`C=Pf.Pe.Pa.Ve.Pb.Va.Pd.Vb.Pa.Vd.Pb.Va.Pd.Vb.Vd.Vf`
`PROG=B|C`

The unfolding of the term A just once gives Figure 4. The unfolding of the term A twice gives Figure 5. The red regions delimit the unsafe regions. Notice how the unsafe region evolves after each unfolding. The two apparent deadlocks are not deadlocks - they just force reentrance of the loop, and hence there is no unsafe region.

3.4. **Complexity issues.** We let the *volume* $Vol(S)$ of a set S of nodes (hyperrectangles) in $\mathbf{R}$ be the number of its elements. The dominant part of the algorithm is the removal of F and finding the deadlocks. To remove F and find S_1 one has to check for each $R \in R^i$ whether it is already marked in F. Only if the answer is no, the $2n$ operations of disconnecting R from its n sons and n parents and possibly, a single addition (of a parent) to, resp. removal (of R) from, the list of potential deadlocks, has to be performed. This implies:

Proposition 1. *For a pure term (i.e., no + nor any recursion) consisting of n transactions with a forbidden region* $F = \bigcup_1^r R^i$*, the worst case complexity of the algorithm is of order* $nVol(F) + \Sigma_1^r Vol(R^i)$*.*

Remark 4. *Examples reaching the worst case have a high amount of global synchronization, which in general should be avoided for good programming practice. 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 *(where* N *is the number of states).*

4. CONTINUOUS TO DISCRETE - INVOKING THE GEOMETRY

Using the combinatorial geometry of the *boundary* ∂F of the forbidden region, we are now going to describe the deadlocks in X and the unsafe regions associated to them in a more efficient way.

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 hyperrectangles in F:

If $i_1 \neq i_2$ and $\overset{\circ}{R}{}^{i_1} \cap \overset{\circ}{R}{}^{i_2} \neq \emptyset$, 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 ("no interior faces at the same level") is obviously satisfied for forbidden regions for "mutually exclusion" models, in particular for PV-models.

We want to include deadlocks on the boundary ∂I^n into our description: In a mutual exclusion model, points on ∂I^n stand for situations where not all processors have started their execution or where some of them already have terminated. To circumvent lengthy case studies – and with an eye to implementation – we slightly change our model in order to include the upper boundary $\partial_+(I^n) = \{\mathbf{x} \in I^n | \exists j : x_j = 1\}$ 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 $\widetilde{R^i} = [0,2]^{i-1} \times [1,2] \times [0,2]^{n-i}$, $1 \le i \le n$, and shifting indices by n, $\widetilde{R^{n+1}}, \ldots, \widetilde{R^{n+r}}$ will denote the hyperrectangles 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 $\bigcup_1^n \widetilde{R^i} = \tilde{I}^n \setminus \overset{\circ}{I}{}^n$, and $\tilde{F} = F \cup \bigcup_1^n \widetilde{R^i} = \bigcup_{i=1}^{n+r} \widetilde{R^i}$. By an abuse of notation, we will from now on write R^i for $\widetilde{R^i}$ and F for $\tilde{F}$.

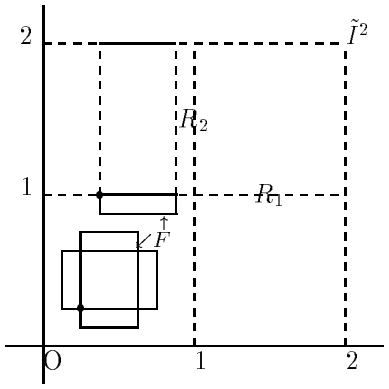


FIGURE 6. Extending the model

For any nonempty index set $J = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n+r\}$ define

$$R^J = R^{i_1} \cap \cdots \cap R^{i_k} = [a^J_1, b^J_1] \times \cdots \times [a^J_n, b^J_n]$$

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 $1 \le j \le n$ and $k, l \in J$). Let $\mathbf{a}^J = [a^J_1, \ldots, a^J_n] = \min R^J$ denote the minimal point in that hyperrectangle.

For every $1 \le j \le n$, we choose $\widetilde{a^J_j}$ as the "second largest" of the $a^{i_l}_j$, i.e., $\widetilde{a^J_j} = a^{i_s}_j$ with $a^{i_l}_j \le a^{i_s}_j < a^J_j$ for $a^{i_l}_j \ne a^J_j$, and consider the "half-open" hyperrectangle $U^J =]\widetilde{a^J_1}, a^J_1] \times \cdots \times]\widetilde{a^J_n}, a^J_n]$ "below" R^J.

Theorem 2. 1. *A point $\mathbf{x} \in X$ is a deadlock if and only if $\mathbf{x} \ne \mathbf{1}$ and there is an n-*element *index set $J = \{i_1, \ldots, i_n\}$, with $R^J \ne \emptyset$ and $\mathbf{x} = \mathbf{a}^J = \min R^J$.*

2. *If $\mathbf{x} = \min R^J$ is a deadlock, then the "half-open" $n-$rectangle U^J is* unsafe, *i.e., every dipath in I^n from a point $\mathbf{y} \in U^J$ will eventually enter $\overset{\circ}{F}$.*

Proof.

1. Let $\mathbf{x} = \mathbf{a}^J = \min R^J$. Every element $\mathbf{y} = [a^J_1 + \varepsilon_1, \ldots, a^J_n + \varepsilon_n]$, $\varepsilon_j \ge 0$ and $0 < \sum_1^n \varepsilon_i$ small, is contained in at least one of the sets $\overset{\circ}{R}{}^{j_i}$ and thus in $\overset{\circ}{F}$.

   On the other hand, let $\mathbf{x} = [x_1, \ldots, x_n] \in X$ be a deadlock. Then, for small values $\varepsilon > 0$, the element $\mathbf{x}^i = [x_1, \ldots, x_i + \varepsilon, \ldots, x_n]$ is contained in one of the sets $\overset{\circ}{R}{}^{j_i}$. Hence, $\mathbf{x} \in R^J$ with $J = \{j_1, \ldots, j_n\}$. This set contains n different elements: If, e.g., $R^{j_1} = R^{j_2}$, then $\mathbf{x}^1 \notin \overset{\circ}{R}{}^{j_1}$!

Moreover, $\mathbf{x}$ is an element of the set $R^J \setminus \bigcup \overset{\circ}{R}{}^{j_i}$ consisting of the 2^n points with all coordinates either a_i^J or b_i^J. Obviously, the only possible deadlock point in this set is $\mathbf{x} = \mathbf{a}^J = \min R^J$.

2. Let $\alpha : I \to X$ be a dipath with $\alpha(t_0) \in U^J$ and $\alpha(t_2) \notin U^J$ for some $t_0 < t_2$. There has to be a maximal value $t_0 \leq t_1 < t_2$ such that $\alpha(t_1) \in U^J$. Moreover, $\alpha(t_1) \in \partial_+ U^J = \{\mathbf{y} \in U^J | \exists k : \ y_k = a_k^J\}$, and thus $\alpha(t_1 + \varepsilon)$ is contained in one of the sets $\overset{\circ}{R}{}^{j_i}$ and thus in $\overset{\circ}{F}$. Contradiction!

□

As an immediate consequence, we get a criterion for deadlockfreeness that is easy to check:

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

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

Remark 5.

1. In geometric terms, U^J is the "corner under $\mathbf{a}^J$", i.e., a hyperrectangle whose "upper boundary" $\partial_+(U^J)$, i.e., the faces containing $\mathbf{a}^J$, consists of faces contained in the "lower boundaries" of the $R^i, i \in J$.
2. In general, the hyperrectangle U^J will be considerably larger than the hyperrectangles from the graph algorithm; it will contain several of the hyperrectangles in the partition $\mathcal{R}$.
3. It is possible that $\overset{\circ}{U}{}^J$ has non-empty intersection with $\overset{\circ}{F}$ – cf. Figure 9.
4. The n points $\mathbf{a_i} = (a_1^J, \ldots, \hat{a_i^J}, \ldots, a_n^J)$ are *critical points of coindex 1* of the sum function $f(x_1, \ldots, x_n) = x_1 + \cdots + x_n$ restricted to ∂F. We were led to Thm. 2 by this type of differential geometric considerations.

In general, the hyperrectangle U^J will be considerably larger than the hyperrectangles from the graph algorithm; it will contain several of the hyperrectangles in the partition $\mathcal{R}$. This is where we gain in efficiency: look at Figures 7, 8, 9 and 10. They describe the 3 iterations needed in the following streamlined algorithm, whereas the first algorithm needed 26 iterations (two for each of the 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 $\mathbf{a}^J \in \mathcal{D}$, the unsafe hyperrectangle U^J. Let $F_1 = F \cup \bigcup_{\mathbf{a}^J \in \mathcal{D}} U^J$. Find the set $\mathcal{D}_1$ of deadlocks in $X_1 = X \setminus F_1 \subset X$, and, for every deadlock $\mathbf{a}^I \in \mathcal{D}_1$, the unsafe hyperrectangle U^I. Let $F_2 = F_1 \cup \bigcup_{\mathbf{a}^I \in \mathcal{D}_1} U^I$ etc.

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

The example "s2" in Appendix A 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, this "staircase" producing more and more unsafe hyperrectangles 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.

FIGURE 7

FIGURE 8

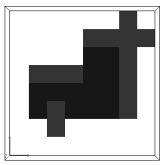


FIGURE 9

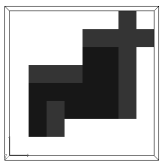


FIGURE 10

5. Implementation of the geometric algorithm

5.1. **The semantics.** Now we have a dual view on PV terms. Instead of representing the allowed hyperrectangles, we represent the forbidden hyperrectangles only. Notice that up to now, we have only implemented the algorithm on pure terms (i.e. no recursion nor plus operator). The full treatment of the PV language is postponed until Section 7.1. 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, $(a, b \in \mathbf{O})$. $X_i(j)$ 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 hyperrectangle. 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 $j \in V$.

5.2. **The implementation.** A general purpose library for manipulating finite unions of hyperrectangles (for any n) has been implemented in C. A hyperrectangle is represented as a list of n closed intervals. Regions (like the forbidden region) are represented as lists of hyperrectangles. We also label some hyperrectangles by associating a region to them. Labeled regions are then lists of such labeled hyperrectangles. Notice that all this is quite naively implemented up to now. Much better algorithms can be devised (inspired by algorithms on isothetic hyperrectangles [31]) that reduce the complexity of intersection calculations a lot. This will be discussed in Section 6.

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 §5.1 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 hyperrectangle in the forbidden region for each partition and each such pair.

5.3. **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 hyperrectangle 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 an 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 hyperrectangle S gives the union of all intersections of hyperrectangles X in R (which are also hyperrectangles) 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 hyperrectangles. Therefore `X` is a labeled hyperrectangle, whose labels are `X1,...,Xn` (the set of the n hyperrectangles 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.

5.4. **Complexity issues.** The entire algorithm consists of 3 parts: The first establishes the initial list `pile[0]` of forbidden hyperrectangles, the second works out the complete array `pile` – including the deadlocks encoded in `pile[n-1]` –, 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 hyperrectangles. From a complexity viewpoint, the first step is negligeable; finding the hyperrectangles 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-1]` 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 hyperrectangles. 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 hyperrectangles in the forbidden region. This figure reflects the degree of synchronization of the processes, and will be much lower that $S(r,n)$ 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.

5.5. **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.

In the following table, dim represents the dimension of the program checked, #fbd2 is the number of forbidden hyperrectangles found in the semantics of the program (to be compared with #fbd1, the number of unit cubes forbidden in the first semantics), t s2 is the time it took to find these forbidden hyperrectangles (respectively t s1 is the time taken for the first semantics, looking at the enabled transitions), t uns2 is the time it took to find the unsafe region in the second algorithm (respectively in the first algorithm) and #uns is the number of hyperrectangles found to be unsafe (they now encapsulate many of the "unit" hyperrectangles 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.

P	dim	#fbd2	#fbd1	t s2	t s1	t uns2	t uns1	#uns
ex	2	4	14	0.02	0	0	0	3
s2	2	6	16	0.02	0.01	0	0	15
s3	3	18	290	0.01	0.18	0	.01	4
s3'	3	6	80	0.03	0.64	0	0.02	0
l	3	6	158	0.02	0.08	0	0	0
3p	3	3	32	0.02	0	0	0	1
4p	4	4	190	0.03	0.09	0	0	1
5p	5	5	1048	0.03	0.82	0	0.02	1
6p	6	6	5482	0.03	5.82	0	0.13	1
7p	7	7	27668	0.04	42.35	0	0.86	1
16p	16	16	NA	0.03	NA	0.03	NA	1
32p	32	32	NA	0.03	NA	0.42	NA	1
64p	64	64	NA	0.04	NA	1.52	NA	1
128p	128	128	NA	0.10	NA	26.49	NA	1

6. A sketch for a better implementation

The deadlock detection program can be made much more efficient by replacing the algorithm in charge of reporting the non-trivial intersections of hyperrectangles. What we have implemented is based on the following simple operations: let $R^1 = \Pi_{i=1,\dots,n}[a_i^1, b_i^1]$ and $R^2 = \Pi_{i=1,\dots,n}[a_i^2, b_i^2]$ be two hyperrectangles (sometimes called n-ranges [32]), we check if $R^1 \cap R^2 \neq \emptyset$ by checking that,

- $a_1^1 \leq b_1^2$ and $a_1^2 \leq b_1^1$,
- $a_2^1 \leq b_2^2$ and $a_2^2 \leq b_2^1$,
- ...
- $a_n^1 \leq b_n^2$ and $a_n^2 \leq b_n^1$.

Each time we want to add one forbidden hyperrectangle to a forbidden region composed of N forbidden hyperrectangles, we check these $2n$ inequalities up to N times. If we suppose that the coordinates of the hyperrectangles are independent random variables, the average number of operations needed is $O(nN)$ inequalities. If we

want reporting but also the actual hyperrectangles at the intersection, we need to add up $2nK$ operations, where K is the number of intersections found. In fact, there is theoretically a way that makes the reporting of all new intersections found of order $O(log_2(N))$ and of order $O(log_2(N)+2nK)$ if we want their values as well. Practically, the best algorithms known report intersections when adding an hyperrectangle in time of order $O(log_2^d(N))$ [31, 32]. As a matter of fact, we need more than just reporting of intersections between two hyperrectangles, namely intersections between at most n hyperrectangles. So the complexity of the algorithm we use for that purpose (using `pile`) is of order $O(n\sum_{i=0,\ldots,n-1} N_i)$ where $N_0 = N$ and N_i is the number of of intersections of exactly $i+1$ distinct hyperrectangles. The worst case is $N_i = C_N^{i+1} = \frac{N!}{(i+1)!(N-i-1)!}$ and worst case complexity (attained by the N philosophers' problem) is bounded by $nO(2^N)$. In fact there is a much better algorithm for reporting the intersections between i hyperrectangles ($i \leq n$) whose heart is in the algorithms of H. W. Six, D. Wood and H. Edelsbrunner [13, 32]. The structure involved in this algorithm is an "interval tree" or "segment tree" that we slightly customize for our purpose.

Definition 3. *An interval tree is a rooted binary tree whose nodes u contain an interval $I(u)$, and verifying the following conditions, given that u_l and u_r are respectively the left son and right son of a node u,*

(a) $I(u_l) \cap I(u_r)$ *is a singleton,*
(b) $I(u) = I(u_l) \cup I(u_r)$,
(c) $\forall x \in I(u_l)$, $\forall y \in I(u_r)$, $x \leq y$.

In fact, we will have in the future to relax conditions (a) and (b) a bit. This definition corresponds to a "static" interval tree (as used in [32, 31]) a "dynamic" version needs condition (a) to be deleted and condition (b) to be understood as the union $\cup$ being the convex union of the intervals (the least upper bound in the lattice of intervals). This will enable us to use only the bounds of the intervals that are actually given by the forbidden hyperrectangles. But in this report we will only use the "static" interval trees. We refer the reader to [13] for some hints about a dynamic version.

We need some information associated to the nodes of interval trees, therefore we have to make the following definition,

Definition 4. *A labeled interval tree is an interval tree together with labels $l(u)$ associated with each node u.*

In the following we will use labels of the form $l(u) = \{k^u, (R^{i_1}, \ldots, R^{i_{k^u}})\}$ where $R^{i_1}, \ldots, R^{i_{k^u}}$ is a set of distinct hyperrectangles (coming from the forbidden region), and k^u is its cardinal. We use a list notation since for implementation matters we will order the collection of names of hyperrectangles in such sets using the order on their indexes. This means that we will require $i_1 < i_2 < \ldots < i_{k^u}$. This will enable fast comparisons of such sets. Also for the same purpose, we can add to the label $l(u)$ an entry h^u given by a hash function H on the sequence $i_1, \ldots, i_{k^u}$, like, $H(i_1, \ldots, i_{k^u}) = \sum_{j=1,\ldots,k^u} B^{i_j} \bmod C$, where B and C are two integers (relatively prime in general).

Let us first explain what happens when $n = 1$ in order to describe the basic operations on the labeled interval trees.

Suppose we are given a labeled interval tree T and an interval R^k. We want to insert R^k into T. The different steps of the insertion algorithm are,

(1) Begin with node u equal to the root of T,
(2) If $I(u) \subseteq R^k$ then insert R^k in the list $l(u)$, set $k^u = k^u + 1$, update h^u (by adding $B^k \bmod C$ in our example). Add similarly R^k to the lists $l(v)$ where v is any son of u in T.

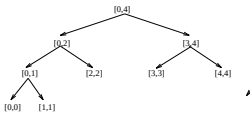


FIGURE 11

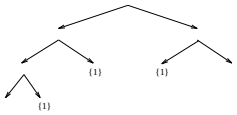


FIGURE 12

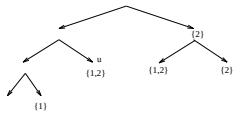


FIGURE 13

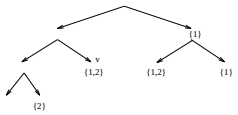


FIGURE 14

(3) If $I(u) \not\subseteq R^k$ then we should do at least one and maybe both of the following,
   (i) If $I(u_l) \cap R^k \neq \emptyset$, go to (2) with $u = u_l$,
   (ii) If $I(u_r) \cap R^k \neq \emptyset$, go to (2) with $u = u_r$,

For use of this structure for our deadlock algorithm, at the beginning we generate the static interval trees T_i whose root nodes u_i are such that $I(u_i)$ are the intervals of local times we are using for giving the semantics of process P_i $(i = 1, \ldots, n)$. Then we generate the forbidden intervals R_i^j (for each process P_i) and insert them in the corresponding T_i.

This ensures the following property,

Lemma 1.
- *Suppose* $l(u) = \{k, \{l_1, \ldots, l_k\}, h\}$ *for a node* u *in* T_i*, then,* $I(u) \subseteq \cap_{j=l_1,\ldots,l_k} R_i^j$,
- *Suppose that we have inserted rectangles* R_i^j *in a (originally empty) labeled interval tree* T*, such that* $\cap_{j=l_1,\ldots,l_k} R_i^j \neq \emptyset$*. Then there is a node* v *of* T *with* $l(u) = \{k, \{l_1, \ldots, l_k\}, h\}$ *(for some* h*).*

Then, to detect deadlocks, it is enough to do the following,

(1) When inserting hyperrectangles R_i^j, detect (using the labelling of nodes) if we have a possible intersection of n distinct intervals at node u of T_i,
(2) If so, put the corresponding node in a list L_i (ordered by the hash value) of plausible candidates for deadlocks,
(3) Traverse the lists L_k, $k \neq i$ to find nodes with the hash value equal to the hash value of node u. If it fails, the node u does not represent a deadlock. If not, we see if the labels of the nodes found is the same as the label of u. If it is so for all i, then we have a deadlock.
(4) Then if we have a deadlock we generate a new hyperrectangle, and then we go back to (1).

Lemma 1 ensures the correctness of this variant of the deadlock algorithm.

Example 3. *Consider the program,*

```
X=Px.Py.Vx.Vy
Y=Py.Px.Vy.Vx
PROG=X|Y
```

We generate T_1 *(for process* `X`*) and* T_2 *(for process* `Y`*) so that* $I(u_1) = [0, 4]$ *and* $I(u_2) = [0, 4]$*. Thus at first, they are the same and look like what is pictured in Figure 11. Then the semantics of the PV language prescribes that the forbidden region is* $F = \{R^1 = [1, 3] \times [2, 4], R^2 = [2, 4] \times [1, 3]\}$*. We first add* $R_1^1 = [1, 3]$ *to* T_1*. We find the interval tree pictured in Figure 12 (we have written the label associated to nodes when it was not empty, ignoring the hash number and the count number).*

After adding up R_2^1 *to* T_2 *we add up* R_1^2 *to* T_1 *generating the list* $L_1 = (u, u')$ *where* u *and* u' *are shown in Figure 13. Finally when adding* R_2^2*, we find* $L_2 = (v, v')$*, where* v *and* v' *are shown in Figure 14. It appears instantly that* $l(u) = l(v)$ *leading to a deadlock.*

This algorithm has not been tested yet but we expect much better performances from programs handling a big number of processes.

7. Abstraction of forbidden regions

Here we state a few preliminary ideas about how to generalize the geometric approach to more complex languages, that include in particular branchings and loops.

7.1. **A simple framework.** We want to formalize here how we can (upper-) approximate safely the unsafe regions in the cases where we cannot hope for an exact computation (in particular when we are dealing with recursive processes etc.). A first application will be detailed in Section 7.2.

Proposition 2. *Let $F = \cup_{i=1,\dots,k} R^i$ be a set of forbidden hyperrectangles in $\mathbb{R}^n$. Suppose we are given S^i, for $i = 1$ to m, hyperrectangles such that $F \subseteq S = \cup_{i=1,\dots,m} S_i$. Then,*

$$\mathbf{U}(F) \subseteq \mathbf{U}(S)$$

where $\mathbf{U}(F)$ denotes the union of the unsafe regions of F with F.

Proof.

Let $x \in \mathbf{U}(F)$. Then,

- if $x \in S$ then $x \in \mathbf{U}(S)$,
- otherwise, $x \notin F$. Suppose we have a dipath γ starting at x and reaching the final point $\mathbf{1}$ without entering S. This means in particular γ is a dipath from x reaching the final point without going through F. This contradicts the fact that x is in the unsafe region of F.

□

Now, the second deadlock algorithm can be applied to the set of "abstract" hyperrectangles S and this will give an upper-approximation of the unsafe region.

7.2. **An abstraction: Regular Expressions.** The object of this section is to treat "non-pure" terms of our PV language by defining a suitable abstraction of forbidden regions. We already know that with these "non-pure" terms we should unfold as much as possible the executions to determine the unsafe region. An answer is to abstract the unfoldings by a finite number of abstract hyperrectangles. For this we will motivate and define the use of regular expressions.

We consider regular expression on the alphabet made of Px and Vx where x is any variable name (in the set $\mathbf{O}$). We recall that regular expressions on an alphabet $\mathbf{A}$ are made up of elements of $\mathbf{A}$ plus 1 and 0 and stable under the following operations,

- concatenation ".". 1 is the neutral element for concatenation,
- union "+" (for which 0 is the neutral element),
- star "(.)*".

Regular expressions form a dioid (i.e. an idempotent semi-ring with + and .).

A regular expression has a meaning in terms of subsets of strings on the alphabet $\mathbf{A}$. We recall that,

- $[\![X + Y]\!] = [\![X]\!] \cup [\![Y]\!]$,
- $[\![X.Y]\!] = \{x.y / x \in [\![X]\!], y \in [\![Y]\!]\}$,
- $[\![X^*]\!] = \cup_{i \geq 0} X^n$ where $X^n = X.X^{n-1}$ for $n \geq 1$ and $X^0 = \{1\}$,
- $[\![a]\!] = \{a\}$ where $a \in \mathbf{A}$,
- $[\![1]\!] = \{1\}$,
- $[\![0]\!] = \emptyset$.

There is also a relation between regular expressions and graphs.

A (labeled, directed) graph on an alphabet $\mathbf{A}$ is $G = (N, E)$ consisting of,

- a set of node N,
- a set of edges $E \subseteq N \times \mathbf{A} \times N$ between nodes.

A labeled graph has a representation as a matrix of elements in the dioid freely generated by $\mathbf{A}$. Given $G = (N, E)$, the corresponding matrix M^G is a square matrix of size cardinal of N, such that the entry corresponding to the pair of nodes (n, n') is,

$$M^G_{n,n'} = \begin{cases} a & \text{if } (n', a, n) \in E \\ 0 & \text{otherwise} \end{cases}$$

Then the matrix $(M^G)^+ = M^G + (M^G)^2 + \ldots + (M^G)^k + \ldots$ has as entries $(M^G)^+_{n,n'}$ a regular expression denoting the set of paths of length at least one from n' to n in the graph G. Even more, $(M^G)^+$ is the solution of the equation $M^G(Id + X) = X$ which can be algorithmically solved by a classical version of Gauss' algorithm (see [20]),

Lemma 2. *Let A be an $n \times n$ matrix with coefficients in a complete*[4] *dioid and $A^0, \ldots, A^n$ be the matrices defined by,*

$$\begin{aligned} A^0 &= A \\ A^k_{i,j} &= A^{k-1}_{i,j} + A^{k-1}_{i,k} \left(A^{k-1}_{k,k}\right)^* A^{k-1}_{k,j} \quad i, j = 1, \ldots, n \; k = 1, \ldots, n \end{aligned}$$

Then $A^n = A^+$.

We are now considering again our full PV programming language. It is easy to see that terms derivable with the grammar of PV processes is exactly the same as regular expressions on our alphabet $\mathbf{A}$. Therefore dipaths in the semantics of PV terms project on each coordinate on regular expressions on $\mathbf{A}$ ("local times") as defined below.

Given any PV program composed of k processes $P_1, \ldots, P_k$, we do the following,

(A) construct the graph of transitions G_i for process P_i $(i = 1, \ldots, k)$. This is done using the following semantics,
  - $[\![X]\!] = [\![Proc_d]\!]$ if $X = Proc_d$ is a valid definition,
  - $[\![Px.Q]\!]$ is a graph with initial node $Px.Q$ and transition from it to the node Q of $[\![Q]\!]$, labeled Px,
  - $[\![Vx.Q]\!]$ is a graph with initial node $Vx.Q$ and transition from it to the node Q of $[\![Q]\!]$, labeled Vx,
  - $[\![P + Q]\!]$ is the union of the graphs $[\![P]\!]$ and $[\![Q]\!]$ quotiented by $P = Q$.

  We also minimize the graph (by classical minimalization of automata),

(B) construct the incidence matrices M^{G_i}, $i = 1, \ldots, n$,

(C) compute $(M^{G_i})^+$, $i = 1, \ldots, n$ using Gauss' algorithm

Every graph G_i built as above has a well identified "initial node" I_i (whose name is $Proc_d$ the process definition corresponding to P_i). The "local time" corresponding to I_i is no longer 0 since we might go through it over and over again. It is straightforward to see that the good candidate to represent such a local time is now an element of the dioid on $\mathbf{A}$ which is the regular expression of paths from I_i to itself, i.e. $(M^{G_i})^*_{I_i,I_i}$. Similarly, the local time corresponding to any other node n is $(M^{G_i})^*_{n,I_i}$. The order on the local times is not the dioid order but the usual prefix ordering (lifted to the powerset of strings on $\mathbf{A}$). This prefix ordering can be tested directly on the graphs G_i. Notice that as G_i are all deterministic graphs (as they are minimal) distinct nodes have distinct local times. Let us exemplify all this on a simple example first.

Example 4. *Consider the following program,*

```
X=Px.Vx.Py.Vy.X
Y=Py.Vy.Px.Vx.Y
Prog=X | Y
```

[4] This is in particular the case of the dioids we are considering in this paper.

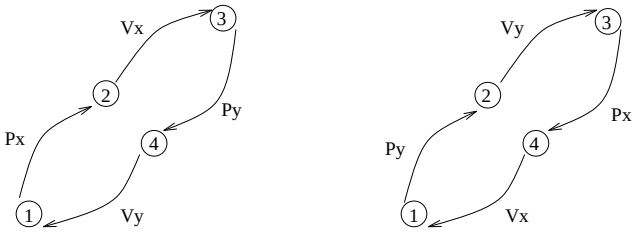


FIGURE 15. Graphs G_1 and G_2

Let P_1 be the process corresponding to the first equation (dealing with `X`*) and P_2 be the process corresponding to the second equation (dealing with* `Y`*). The corresponding graphs are pictured in Figure 4 (vertices X_i are represented as a circled i). The corresponding matrices are,*

$$M^{G_1} = \begin{pmatrix} 0 & 0 & 0 & Vy \\ Px & 0 & 0 & 0 \\ 0 & Vx & 0 & 0 \\ 0 & 0 & Py & 0 \end{pmatrix} \quad M^{G_2} = \begin{pmatrix} 0 & 0 & 0 & Vx \\ Py & 0 & 0 & 0 \\ 0 & Vy & 0 & 0 \\ 0 & 0 & Px & 0 \end{pmatrix}$$

Therefore, by using Gauss' algorithm, the local times for G_1 are,

- *corresponding to X_1 is $(Px.Vx.Py.Vy)^*$,*
- *corresponding to X_2 is $(Px.Vx.Py.Vy)^*.Px$,*
- *corresponding to X_3 is $(Px.Vx.Py.Vy)^*.Px.Vx$,*
- *corresponding to X_4 is $(Px.Vx.Py.Vy)^*.Px.Vx.Py$.*

and for G_2,

- *corresponding to X_1 is $(Py.Vy.Px.Vx)^*$,*
- *corresponding to X_2 is $(Py.Vy.Px.Vx)^*.Py$,*
- *corresponding to X_3 is $(Py.Vy.Px.Vx)^*.Py.Vy$,*
- *corresponding to X_4 is $(Py.Vy.Px.Vx)^*.Py.Vy.Px$.*

Now, for each graph G_i we are constructing the list of intervals of local times comprising the states where a lock has been acquired on a given object x and when this lock has been released. But intervals of local times do not represent accurately what we want: $[a^*, a^*.b]$ does represent the set of intervals $\{[1,b], [1,a.b], \ldots, [a, a.b], [a, a^2.\ b], \ldots\}$. What we need to represent is in general a set like $\{[1,b], [a,a.b], [a^2, a^2.b], \ldots\}$. This is representable exactly in the regular interval expressions. As a matter of fact the intervals on regular expressions form a monoid with $[a,b].[c,d] = [a.c, b.d]$. Therefore we can construct regular interval expressions and the latter set of intervals is then $[a,a]^*.[1,b]$. Notice here that $[a,a]^*$ is distinct from $[a^*, a^*]$. We can use a particular form of regular interval expressions for our algorithm,

Definition 5. *A regular term-interval is any expression of the form,*

$$\sum_{i=1,\ldots,k} X_i.[1, B_i]$$

where X_i and B_i are regular expressions.

The order on regular term-intervals is derived from the order on regular interval expressions which in term is the set inclusion order.

The regular term-intervals abstracting (in the sense of Section 7.1) the forbidden hyperrectangles projected on the ith coordinate for variable x, are determined as follows,

(D) Determine the set of nodes of G_i onto which goes an edge labeled Px. Call this set N_P,

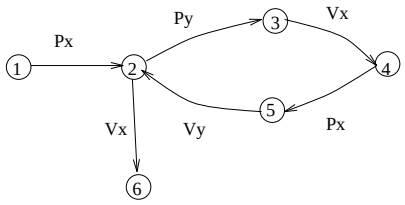


FIGURE 16. Graph for process Y

(E) Determine the set of node of G_i onto which goes an edge labeled Vx. Call this set N_V,
(F) The set of intervals we are seeking is $L_{i,x}$ with

$$L = \sum \{(M^{G_i})^*_{n,I_i}.[1, K_{n,n'}]/n \in N_P, n' \in N_V\}$$

where $K_{n,n'}$ is the regular expression corresponding to the paths from n to n' not going through any P_x: $K_{n,n'} = (M^{G_i})^*_{n',n} \backslash \mathbf{A}^*.Px.\mathbf{A}^*$.

Let us continue with our example,

Example 5. *We find,*

- *For* G_1,
  - *For* x, $N_P = \{X_2\}$, $N_V = \{X_3\}$ *and* $L_{1,x} = (Px.Vx.Py.Vy)^*.Px.[1, Vx]$,
  - *For* y, $N_P = \{X_4\}$, $N_V = \{X_1\}$ *and* $L_{1,y} = (Px.Vx.Py.Vy)^*.Px.Vx.Py.[1, Vy]$.
- *For* G_2,
  - *For* x, $N_P = \{X_4\}$, $N_V = \{X_1\}$ *and* $L_{2,x} = (Py.Vy.Px.Vx)^*.Py.Vy.Px.[1, Vx]$,
  - *For* y, $N_P = \{X_2\}$, $N_V = \{X_3\}$ *and* $L_{2,y} = (Py.Vy.Px.Vx)^*.Py.[1, Vy]$.

Now, as usual we only have to carry on with the usual algorithm to find the unsafe region (an upper-approximation of it in fact).

Example 6. *We have here two "families" of forbidden regions,*

- $R^1 = (Px.Vx.Py.Vy)^*.Px.[1, Vx] \times (Py.Vy.Px.Vx)^*.Py.Vy.Px.[1, Vx]$,
- $R^2 = (Px.Vx.Py.Vy)^*.Px.Vx.Py.[1, Vy] \times (Py.Vy.Px.Vx)^*.Py.[1, Vy]$.

Our deadlock algorithm remains valid on this abstract representation of the forbidden region since we are in the hypothesis of Proposition 2. The interest is that we have fast algorithms for manipulating regular expressions.

Example 7. *We compute* $R^1 \cap R^2 = \emptyset$

A more intricate example to end up this section,

Example 8.
```
X=Py.Vx.Px.Vy.X
Y=Px.X.Vx
Z=Px.Vx
PROG=Y | Z
```

Then the graph for process Y *is represented in Figure 16 and variable* x *enters a critical region for times between* X_2 *and* X_4 *and between* X_5 *and* X_4 *and between* X_5 *and* X_6 *and between* X_2 *and* X_6*. This gives the four following regular interval expressions,*

- $R^1 = Px.[1, Py.Vx] \times Px.[1, Vx] = [Px, Px.Py.Vx] \times Px.[1, Vx]$,
- $R^2 = Px.(Py.Vx.Px.Vy)^*.Py.Vx.Px.[1, Vy.Py.Vx] \times Px.[1, Vx]$,
- $R^3 = Px.(Py.Vx.Px.Vy)^*.Py.Vx.Px.[1, Vy.Vx] \times Px.[1, Vx]$,
- $R^4 = [Px, Px.Vx] \times Px.[1, Vx]$.

The only intersection is,

$$R^2 \cap R^3 = Px.(Py.Vx.Px.Vy)^*.Py.Vx.Px.[1, Vy] \times Px.[1, Vx]$$

This does not *give rise to a deadlock - after all there is only one shared object.*

This example shows we are only computing an upper-approximation and not the exact unsafe region. We believe in the case of the PV language (with no values involved) that it is possible to compute the exact unsafe region. This is left for future work.

In fact, as one can guess, we can interpret the forbidden regions in any intervals of abstract local times, which are elements of a given dioid. Some examples of further generalizations follow.

8. Some future extensions

8.1. **Non-regular languages.** Based on some mathematical ideas (unitary-prefix monomial relations of [14]) and of their applications in alias analysis as in [10], we can generalize the ideas of the previous section in order to handle more precisely what can happen when we consider languages with values.

Instead of considering regular interval languages to abstract the forbidden region, we can use irregular description of infinite forbidden regions. If we carry on with the PV example (with this time values associated to objects, and conditional statements upon these values), we can choose as underlying lattice (or dioid) of the one which contains local times of the form $(Pu_1)^{k_1}.(Vu_2)^{k_2}.\ldots.(Pu_l)^{k_l}.\ldots$ with u_1, u_2,...,u_l objects and k_1, k_2, ..., k_l integers, together with a value k' in an abstract numerical lattice, like M. Karr's [27] linear equations lattice for instance describing the relations between the integers k_1, k_2, ..., k_l. This is left for future work.

8.2. **Other synchronizers.** Let us consider another kind of synchronization which is very much used in shared memory concurrent languages, and which also models rendez-vous message passing (i.e., blocking sends and receives).

Syntactically, we add new actions to processes, that we write $B(P, Q, ...)$ with any number of processes in P, Q etc. as arguments. A process P executing the action $B(P, Q, ...)$ must wait for the processes Q etc. to have reached the same synchronisation barrier $B(P, Q, ...)$. When all processes have reached the same point, they can resume execution. This is a form of global synchronisation for a subset of the set of processes. The semantics of these actions can be described in a simple manner. Suppose (as for the second semantics of our PV language) that a process is a string of actions, some of which being of the form $B(P_{i_1}, \ldots, P_{i_k})$ $(1 \leq i_1 < i_2 < \ldots < i_k \leq n)$. Given n such processes P_1, ..., P_n, of respective lengths (as strings) l_1 to l_n we can describe the forbidden region associated to a given synchronisation barrier $B(P_{i_1}, \ldots, P_{i_k})$ as follows.

Suppose processes P_{i_1}, ..., P_{i_k} can execute $B(P_{i_1}, \ldots, P_{i_k})$ at "local time" (i.e. the position in the string here) X_{i_1}, respectively ..., X_{i_k} (with the convention $X_{i_k} = 0$ if P_{i_k} cannot execute the barrier). Then the corresponding forbidden hyperrectangle is R whose ith projection X_i is,

- if $i = i_j$ $(j = 1, \ldots, k)$ then $X_i = [X_{i_j}, l_i]$,
- otherwise $X_i = [0, l_i]$.

An example in dimension two is given in Figure 17.

The algorithms we have developped in this report can still be applied.

The full description of other synchronisation primitives is left for future work.

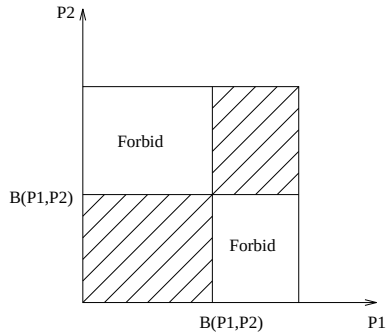


FIGURE 17. A synchronisation barrier

8.3. **Safety properties.** A first natural idea is (following in particular [19]) to prove more complex properties of a system than deadlock freedom, for instance like properties on paths of executions (like the ones which can be expressed with a suitable temporal logics), by composing the system with a new process (which we can call a tester) and relating the deadlocks of the new composed system with properties of the traces of the old one.

This could be done by slighly extending the automata theoretic framework to deal with Higher-Dimensional Automata. Unfortunately this can only be developped for the first deadlock algorithm. The criterion looks like something dealing with a "suspension" of the shape representing our system. As a matter of fact, as already proposed in [22] and [24] we are trying to prove properties about the possible scheduling of actions, which then correspond to properties of the paths of executions modulo a form of homotopy. This idea will be expanded in a forthcoming report.

If we want to use the second deadlock algorithm, we can only relate a very specific class of properties of traces with the deadlock problem.

Among them are "Serializability" properties; in concurrent database theory an execution (i.e., a dipath if we represent the database transactions like process graphs) is serial if it is "equivalent" to an execution of all processes one after another in some order. Geometrically this has been shown (see [28]) to be equivalent to proving that the forbidden region is connected. Good algorithms for proving that a union of hyperrectangles is connected should be studied in the future. A first possibility is to check that each new hyperrectangle we are adding to the semantics of a system has a non-empty intersection with some other hyperrectangle already in the list (or in the interval tree) containing the forbidden region.

COMPARISON AND PERSPECTIVES

Traditional deadlock detection algorithms rely on a clever exploration of the *state space*. The overall idea for both the algorithms proposed in this paper is to take advantage of its complement, the *forbidden region*, and to exploit *duality*. Compared to the state space, the forbidden region is easier to describe geometrically and combinatorially. Moreover, in realistic situations, its size (volume) will be considerably minor.

The first algorithm only uses the fact that the state space has a model (abstraction) as the complement of a subdigraph in a digraph. Thus it can be applied in situations that are much more general than the process graphs and PV languages described here. The range of applications encompasses at least HDAs with cubical complexes as their geometric model.

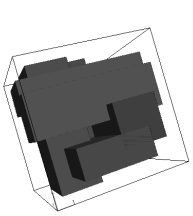


FIGURE 18

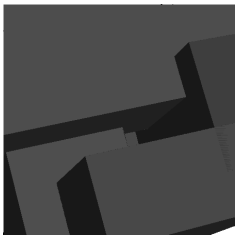


FIGURE 19. Close-up

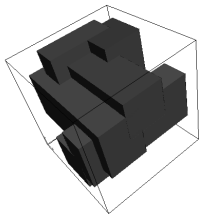


FIGURE 20. Turning around

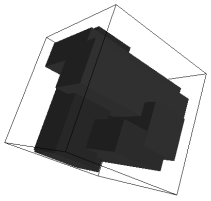


FIGURE 21. Exit

The second algorithm takes decisively advantage of the combinatorial geometry of the forbidden region as a subcomplex in a hyperrectangle. This is why it can have a much better performance for our PV-language and a range of extensions of it that we are about to explore. On the other hand, there might be situations with less geometrical information to which it cannot be generalised.

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 [15, 22, 24] 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/`

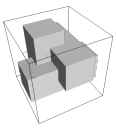
to make the 3D pictures of this article (in a fully automated way).

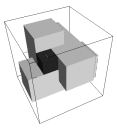
APPENDIX A. THE EXAMPLES DETAILED

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

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





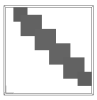
- This is example of Figure 7.

```
/* ``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 [24] and Figure 18 to Figure 21) which produces no deadlock.

```
/* ``l'' */
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
```

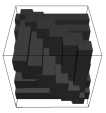
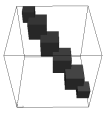
- This is a staircase (worst complexity case for the second algorithm).




```
/* ``s2'' */
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 "s3"'), there is no 3-deadlock.

```
/* ``s3'' */
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

1. 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** (1991), no. 11, 1204–1222.
2. B. Boigelot and P. Godefroid, *Model checking in practice: An analysis of the access.bus protocol using spin*, Proceedings of Formal Methods Europe'96, vol. 1051, Springer-Verlag, Lecture Notes in Computer Science, March 1996, pp. 465–478.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, *Symbolic model checking:* 10^{20} *states and beyond*, Proc. of the Fifth Annual IEEE Symposium on Logic and Computer Science, IEEE Press, 1990, pp. 428–439.
4. S.D. Carson and P.F. Reynolds, *The geometry of semaphore programs*, ACM TOPLAS **9** (1987), no. 1, 25–53.
5. A. T. Chamillard, L. A. Clarke, and G. S. Avrunin, *An empirical comparison of static concurrency analysis techniques*, Tech. Report 96-84, Department of Computer Science, University of Massachusetts, August 1996.
6. J. C. Corbett, *Evaluating deadlock detection methods for concurrent software*, IEEE Transactions on Software Engineering **22** (1996), no. 3.
7. 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 (1977), 238–252.
8. P. Cousot and R. Cousot, *Abstract interpretation frameworks*, Journal of Logic and Computation **2** (1992), no. 4, 511–547.
9. R. Cridlig, *Semantic analysis of shared-memory concurrent languages using abstract model-checking*, Proc. of PEPM'95 (La Jolla), ACM Press, June 1995.
10. A. Deutsch, *Interprocedural may-alias analysis for pointers: Beyond k-limiting*, Proc. of PLDI'94, 1994, pp. 230–241.
11. E.W. Dijkstra, *Co-operating sequential processes*, Programming Languages (F. Genuys, ed.), Academic Press, New York, 1968, pp. 43–110.
12. M. B. Dwyer and L. A. Clarke, *Data flow analysis for verifying properties of concurrent programs*, Proc. of the Second Symposium on Foundations of Software Engineering, December 1994, pp. 62–75.
13. H. Edelsbrunner, *Dynamic data structures for orthogonal intersection queries*, Tech. Report F59, Tech. Univ. Graz, Institute fur Informationsverarbeitung, 1980.
14. S. Eilenberg, *Automata, languages and machines*, Academic Press, 1974.
15. L. Fajstrup and M. Raußen, *Some remarks concerning monotopy of increasing paths*, unpublished manuscript, Aalborg University, 1996.
16. H. Garavel, M. Jorgensen, R. Mateescu, Ch. Pecheur, M. Sighireanu, and B. Vivien, *Cadp'97 – status, applications and perspectives*, Tech. report, Inria Alpes, 1997.
17. P. Godefroid, G. J. Holzmann, and D. Pirottin, *State-space caching revisited*, Formal Methods and System Design, vol. 7, Kluwer Academic Publishers, November 1995, pp. 1–15.
18. 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** (1996), no. 7, 496–507.

19. P. Godefroid and P. Wolper, *Using partial orders for the efficient verification of deadlock freedom and safety properties*, Proc. of the Third Workshop on Computer Aided Verification, vol. 575, Springer-Verlag, Lecture Notes in Computer Science, July 1991, pp. 417–428.
20. M. Gondran and M. Minoux, *Graphes et algorithmes*, Collections de la Direction des Etudes et Recherches d'Electricité de France, 1995.
21. E. Goubault, *The geometry of concurrency*, Ph.D. thesis, Ecole Normale Supérieure, 1995, to be published, 1998, also available at http://www.dmi.ens.fr/~goubault.
22. ———, *Schedulers as abstract interpretations of HDA*, Proc. of PEPM'95 (La Jolla), ACM Press, also available at http://www.dmi.ens.fr/~goubault, June 1995.
23. E. Goubault and T. P. Jensen, *Homology of higher-dimensional automata*, Proc. of CONCUR'92 (Stonybrook, New York), Springer-Verlag, August 1992.
24. J. Gunawardena, *Homotopy and concurrency*, Bulletin of the EATCS **54** (1994), 184–193.
25. M. Herlihy and N. Shavit, *The topological structure of asynchronous computability*, Tech. report, Brown University, Providence, RI, January 1996.
26. M. Herlihy and S.Rajsbaum, *Algebraic Topology and Distributed Computing. A Primer*, Lecture Notes in Computer Science, vol. 1000, Springer-Verlag, 1995.
27. M. Karr, *Affine relationships between variables of a program*, Acta Informatica (1976), no. 6, 133–151.
28. W. Lipski and C.H. Papadimitriou, *A fast algorithm for testing for safety and detecting deadlocks in locked transaction systems*, Journal of Algorithms **2** (1981), 211–226.
29. S. Melzer and S. Roemer, *Deadlock checking using net unfoldings*, Proc. of Computer Aided Verification, Springer-Verlag, 1997.
30. V. Pratt, *Modeling concurrency with geometry*, Proc. of the 18th ACM Symposium on Principles of Programming Languages, ACM Press, 1991.
31. F. P. Preparata and M. I. Shamos, *Computational geometry, an introduction*, Springer-Verlag, 1993.
32. W. H. Six and D. Wood, *Counting and reporting intersections of d-ranges*, IEEE Transactions on Computers **31** (1982), no. 3, 181–187.
33. A. Valmari, *Eliminating redundant interleavings during concurrent program verification*, Proc. of PARLE, vol. 366, Springer-Verlag, Lecture Notes in Computer Science, 1989, pp. 89–103.
34. ———, *A stubborn attack on state explosion*, Proc. of Computer Aided Verification, no. 3, AMS DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1991, pp. 25–41.
35. R. van Glabbeek, *Bisimulation semantics for higher dimensional automata*, Tech. report, Stanford University, Manuscript available on the web as http://theory.stanford.edu/~rvg/hda, 1991.
36. W. J. Yeh and M. Young, *Compositional reachability analysis using process algebras*, Proc. of the symposium on Testing, Analysis and Verification, ACM Press, October 1991, pp. 178–187.

Dept of Mathematics, Aalborg University, Fredrik Bajersvej 7E, DK-9220 Aalborg Øst, and LETI (CEA - Technologies Avancées), DEIN-SLA, CEA F91191 Gif-sur-Yvette Cedex

E-mail address: {fajstrup,raussen}@math.auc.dk, goubault@aigle.saclay.cea.fr