Geometric Performance Analysis of Mutual Exclusion: The Model Solution

By Marc Abrams and Ashok Agrawala

TR 90-59

Geometric Performance Analysis of Mutual Exclusion: The Model Solution*

TR 90-59
Marc Abrams
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106

Ashok K. Agrawala
Department of Computer Science
University of Maryland
College Park, MD 20742

4 December 1990

Abstract

This paper presents an analytic solution to progress graphs used for performance analysis. It derives the exact sequence of blocking and running times experienced by two processes sharing mutually exclusive, reusable resources. A novel application of Dijkstra's progress graphs yields the complex relationship between the waiting times at each synchronization point. The problem of solving progress graphs is formulated in terms of finding the minimum solution of each of a set of Diophantine equations. An algorithm is presented to find all steady state behaviors involving blocking that emerge from any initial condition.

Categories and Subject Descriptors: D.2.8 [**Software**]: Metrics — *Performance measures;* D.4.8 [**Operating Systems**]: Performance — *Modeling and prediction;* C.4 [**Performance of Systems**]: Modeling techniques

General Terms: progress graphs, parallel programs, resource sharing, software performance analysis, synchronization

Additional Key Words and Phrases: resource sharing, Diophantine equation

*This paper is a massive revision of *Exact Performance Analysis of Two Distributed Processes with Multiple Synchronization Points*, Dept. of Computer Science, Univ. of MD. TR-1845.

Contents

1 Introduction

The state of many continuous physical systems (e.g., an electrical circuit) may reach a fixed value as time grows to infinity. Alternately, the system may reach a limit cycle behavior, in which the stable system state varies periodically in time. This paper presents a method to predict the dynamics of a class of parallel programs that displays similar behavior.

The model used here to analyze program behavior is similar to Dijkstra's *progress graphs*, which were used to characterize deadlocks in multiprocessor systems [5]. Progress graphs are also similar to two-dimensional diagrams used in verification of parallel programs and communication protocols to illustrate interleaving of operations. Papadimitriou, Yannakakis, Lipski, and Kung [8, 9, 11] use progress graphs to detect deadlocks in locked data-base transaction systems, the equivalent of two process programs containing straight line sequences of binary semaphores. Carson and Reynolds [4] prove liveness properties in systems with an arbitrary number of processes containing straight line sequences of P and V operations on general semaphores.

Previous formulations of progress graph models are concerned with the *order* of events. The formulation in this paper represents the *timings* of events. Adding timings allows analysis of performance properties of a program.

Progress Graph Model of Program Execution: A companion paper [1] formally defines and proves properties about a progress graph model of program execution suitable for performance analysis. This model is summarized below using excerpts from the companion paper [1].

The model is based on six postulates. First, a single clock external to the parallel program exists. References to time refer to values read from this clock. Second, at any instant of time, each process is either *running* or *blocked*. Third, a *program state* (or *state*) is represented by an ordered tuple of non-negative real numbers, each of whose elements corresponds to one process. Each component of the program state at a particular time equals the total duration of time that the corresponding process has spent in a running condition. Fourth, the sequence of program states that a program passes through during a particular execution defines a *program execution sequence*. Fifth, a program state is *dead* if and only if all processes are blocked. Sixth, an execution sequence contains a dead state if and only if that state is the final state of the execution sequence.

Let t and δ represent real numbers. The *predecessor* to the program state at time t ($t > 0$) in an execution sequence is the program state at time $\lim_{\delta \to 0} t - \delta$ in that sequence. If the program state at time t ($t \geq 0$) in a particular execution sequence is not a dead state, then its *successor* is the program state at time $\lim_{\delta \to 0} t + \delta$, provided that $S(t)$ is not a dead state.

Program states may be classified as *deterministic* or *nondeterministic*. A deterministic state is a state that, in all execution sequences containing the

state, has either no successor or the same successor. All program states that are not deterministic are nondeterministic states.

A program state is nondeterministic if and only if the state of a process changes between running and blocked in the transition from the state to the successor in some but not all execution sequences containing the state.

Program states may also be classified as *blocked* or *running*. A blocked state is a state in which at least one process is blocked. All program states that are not blocked are running states.

Interpreting the ordered tuple representing a program state as the Cartesian coordinates of a graph point implicitly defines a one-to-one and onto mapping from program states to Cartesian graph points. In general, some graph points may represent states that do not arise in any execution sequence. Each graph axis corresponds to one process. A *program execution trajectory* is a directed, continuous path in a progress graph corresponding to the set of states in an execution sequence; furthermore the path is rooted at the point representing the initial state of the execution sequence.

Now consider progress graphs representing programs consisting of two processes. A *ray* is a directed line segment of possibly infinite length. A ray is rooted at its *initial point*. An execution trajectory is a continuous path in a progress graph consisting of either a possibly empty sequence of rays followed by a point or a sequence of one or more rays. Furthermore, all points in a continuous path in an execution trajectory defined by two end points represent running states if and only if the path is a ray with slope one rooted at the end point closest to the origin. All points in a continuous path in an execution trajectory defined by two end points represent blocked states in which the same process is blocked if and only if the path is a ray parallel to the axis representing the running process and rooted at whichever end point is closest to the origin.

Let j denote a nonnegative integer. The class of programs analyzed here meets the following conditions. First, a program consists of two processes. Second, a separate processor executes each process. Third, the code that each process executes consists of a nonterminating loop. Fourth, for each semaphore σ in all execution sequences, the initial semaphore value is one and the j-th $V(\sigma)$ (respectively, $P(\sigma)$) operation executed by a process is preceded by execution of exactly j $P(\sigma)$ operations (respectively, if $j > 0$, $j - 1$ V operations) by that process. Fifth, if a process is blocked in some state in an execution sequence and was running in the predecessor state, then the process was executing a P operation in the predecessor state and is not executing a P operation in the current state. Sixth, if a process is running in some state in an execution sequence and was blocked in the predecessor state, then the other process was executing a V operation in the predecessor state and is not executing a V operation in the current state. Seventh, no P or V operation is contained in a conditionally executed piece of code. Finally, each code segment within each process that either starts at the initial statement of the loop body and continues to and includes the first semaphore operation, or follows each semaphore oper-

ation and continues to and includes the next semaphore operation requires an independently derived, constant, finite, and nonzero amount of execution time, exclusive of time spent blocked. Note that no assumptions are made about the architecture (e.g., multiprocessor, network of workstations) on which executes the program beyond the fact that semaphore semantics can be implemented.

Let $r \in \{0, 1\}$. The *cycle time* of a process r, denoted c_r, is the time required to execute the outermost loop of a process once, excluding blocking time.

Let uppercase letters with optional superscripts denote graph points (e.g., P^0). Let the subscripts 0 and 1 denote the components of a point (e.g., $P^0 = (P_0^0, P_1^0)$). Two points P^0 and P^1 are *congruent* (denoted as $P^0 \equiv P^1$) if and only if $\forall r, P_r^0 \bmod c_r = P_r^1 \bmod c_r$. Let $\overline{[P, P')}$ denote a line segment with closed end point P and open end point P'. For each semaphore σ, $(p_0(\sigma), p_1(\sigma))$ (respectively, $(v_0(\sigma), v_1(\sigma))$) denotes the final state of the subsequence of states corresponding to the first execution of a $P(\sigma)$ (respectively, $V(\sigma)$).

Let $\bar{r} = 1 - r$ denote processes. For each semaphore σ and for each process r, there exists a *constraint line generator*, which is an ordered pair (W, X) such that $X_r = W_r = p_r(\sigma)$, $X_{\bar{r}} = v_{\bar{r}}(\sigma)$, and $W_{\bar{r}} = p_{\bar{r}}(\sigma)$. Point W (respectively, X) is called the *initial* (respectively, *final*) point of the constraint line generator. Each element of $\{\overline{[W + (i_0 c_0, i_1 c_1), X + (i_0 c_0, i_1 c_1))} \mid \forall r, i_r \in \{0, 1, \ldots\}\}$ is a *constraint line*. The end point of a constraint line congruent to W (respectively, X) is called the *initial* (respectively, *final*) point of the constraint line.

Constraint lines have the following significance. A point represents a dead state if and only if it is the point of intersection of two constraint lines. A point represents a blocked state if and only if it lies on a constraint line. A point represents a nondeterministic state if and only if it is the initial point of a constraint line. The last two statements imply that all nondeterministic states are blocked states.

The execution trajectory rooted at a point P is recursively defined below.

- If P lies on some constraint line $\overline{[P', P'')}$, then there either (1) will or (2) will not exist a point P^D on the line $\overline{[P', P'')}$ representing a dead state such that $P \leq P^D$. In case (1), the execution trajectory is a ray with initial point P and final point P^D. In case (2), the execution trajectory is a ray with initial point P and final point P'', followed by an execution trajectory rooted at P''.

- If P does not lie on a constraint line, then a slope one ray rooted at P either (1) will or (2) will not intersect a constraint line. In case (1), the trajectory is an infinite length, slope one ray rooted at P. In case (2), the trajectory is a slope one ray with initial point P and final point P', where P' is the only point on the ray that lies on a constraint line, followed by an execution trajectory rooted at P'.

A deterministic and blocked state may be further classified as *dying* or *live*. A dying state is either a deterministic and dead state or a deterministic and

blocked state in which, in all execution sequences containing the state, one process is blocked and remains blocked in all subsequent states and the final state is dead. A live state is any deterministic and blocked state that is not dying.

A running state may be further classified as *free* or *restricted.* A free state is a state in which all processes are running and every subsequent state in any execution sequence containing the state is a running state. A restricted state is any running state that is not free.

A point P on a constraint line represents a dying state if P is not the initial constraint line point and there exists a point P^I at which the line intersects another constraint line such that $P \leq P^I$. A point P on a constraint line represents a live state if there does not exist a point P^I at which the line intersects another constraint line such that $P \leq P^I$. A point off a constraint line represents a free state if a diagonal ray rooted at the point does not intersect a constraint line; otherwise the point represents a restricted state.

The *concurrent portion* of an execution sequence is obtained by deleting the longest initial subsequence of the execution sequence consisting of blocked states. The concurrent portion of any execution sequence is represented in a progress graph by an execution trajectory whose initial point lies on either line $\overline{[(0,0),(c_0,0))}$ or $\overline{[(0,0),(0,c_1))}$, provided that the initial state of the concurrent portion is deterministic.

Consider any execution trajectories that contain only deterministic states. A *steady state execution trajectory* is any subtrajectory of an execution trajectory that contains exactly two congruent and distinct points, namely the initial and final points. The *transient execution trajectory* is the portion of an execution trajectory consisting of all points that do not lie on a steady state execution trajectory.

Two line trajectories are *equivalent* if and only if there exists a one-to-one correspondence between their points such that corresponding points are congruent.

Consider any progress graph representing a finite number of semaphores and any execution sequence containing only deterministic states. The corresponding execution trajectory can be partitioned into a possibly empty transient execution trajectory followed by an infinite number of equivalent steady state execution trajectories if and only if the sequence contains no dead state.

Example. *The Dining Philosophers problem [6] meets the conditions stated above. In this problem, each philosopher requires two chopsticks to eat. We consider here two philosophers sharing two chopsticks. If one philosopher attempts to acquire the chopsticks while the second is eating, the first must wait. The code for each process is shown in Figure 1.*

The time required for each code segment, excluding blocking, appears in Table 1. Summing the elements of the last two columns of the last two rows in Table 1 yields the cycle time of each process in the absence of blocking: $c_0 = 5$

```
/* Identifier a is a semaphore with initial value one*/
L:  Think;
    P(a);          /* acquire chopsticks */
    Eat;
    V(a);          /* release chopsticks */
    goto L;
```

Figure 1: Code for Dining Philosophers program.

and $c_1 = 3$.

Because each semaphore corresponds to two constraint line generators, requires the following constraint line generators are required: $((1,1),(4,1))$ *and* $((1,1),(1,2))$.

Illustrated in Figure 2 is an initial portion of the execution trajectory with initial point $(0,0)$. *Because* $(0,0)$ *does not lie on a constraint line, the initial ray of the execution trajectory must be diagonal, and the final point of this ray must lie on a constraint line; thus the ray is* $\overline{[(0,0),(1,1))}$. *Point* $(1,1)$ *lies on two constraint lines. The state represented by point* $(1,1)$ *has two possible successors. No matter which successor occurs in a particular execution sequence, the second ray in the execution trajectory has initial point* $(1,1)$ *and has as its final point the final point of a constraint line. Therefore the second execution trajectory ray is either or* $\overline{[(1,1),(4,1))}$ $\overline{[(1,1),(1,2))}$. *The remaining rays of each execution trajectory are constructed similarly.*

Nondeterministic states are represented by all points congruent to $(1,1)$ *in Figure 2; all other points that lie on constraint lines represent live states. No dying states are represented in the figure. All points lying off constraint lines in the figure represent restricted states; this fact is not obvious from the figure, but can be proven using Theorem 2 from section 4.*

The concurrent portion of all possible execution sequences are represented in the progress graph of Figure 3, which represents through shading all *execution trajectories whose initial point lies on line* $\overline{[(0,0),(5,0))}$ *or* $\overline{(0,0),(0,3)}$. *In general, a shaded region in a progress graph may have infinite extent. From Figure 3, all initial conditions result in a steady state execution trajectory equivalent to the execution trajectory rooted at* $(9,7)$ *with state transition vectors* $(3,3)$ *followed by* $(2,0)$.

2 Solving Progress Graphs: Algorithm A0

The companion paper [1] establishes that each execution trajectory in a progress graph representing a finite number of semaphores reaches either a nondeterministic state, a dead state, a non-blocking steady state, or a blocking steady state.

Code segment	Process	
	0	1
Think; P(a);	1	1
Eat; V(a);	3	1
goto L; Think; P(a);	2	2

Table 1: Time required for each code segment of the example, excluding blocking.

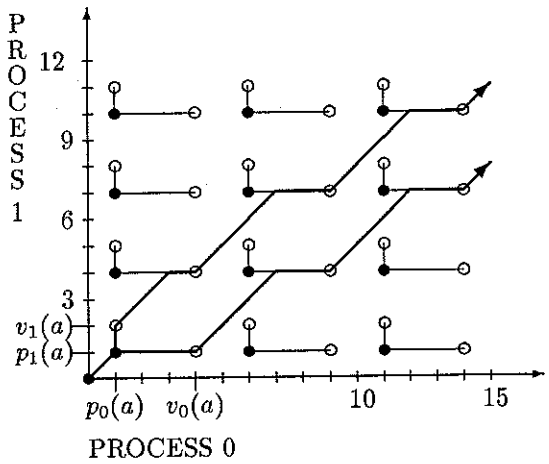

Figure 2: Synchronization constraints for the Dining Philosophers in the Cartesian graph. Open (filled) circles represent open (closed) end points. Thick lines represent two possible execution trajectories.

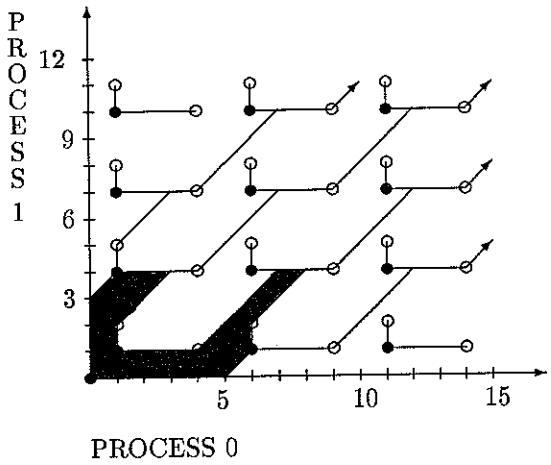

Figure 3: Representation of the concurrent portion of all execution sequences for the progress graph of Figure 2.

Furthermore, a "solution" to a progress graph:

1. reports whether any initial condition can lead to a nondeterministic state, a dead state, or a non-blocking steady state, and
2. reports one member of each equivalence class of blocking steady state execution trajectories that may be reached by any initial condition.

The companion paper presents algorithm A0 (Figure 4) and proves that it correctly solves problem 2 above. Algorithm A0 assumes that the number of semaphores is finite; let this number be $N/2$. Hence there are N equivalence classes of constraint lines in a progress graph. Let γ denote the set of N constraint line generators. A0 requires function f, defined below.

Definition. *Function $f(P)$ maps any point P representing a live or restricted state to the final point of the initial ray of the execution trajectory rooted at P. Furthermore, $f^0(P) \stackrel{\text{def}}{=} P$ and $f^n(P)$, for positive n, denotes the n-fold composition of f applied to P.*

3 First Refinement: Algorithm A1

Algorithm A0 leaves unanswered three questions:

1. When are $\forall X$, $(W, X) \in \gamma$, $f^0(X), f^1(X), \ldots, f^{2N}(X)$ defined?

declare S: set of points; { Set of points X that either lie on a steady state execution trajectory previously output or are known not to lie on a steady state execution trajectory }

$S := \emptyset$;
for each X **in** ξ **do**
  if
    $\nexists x,\ x \in S,\ x \equiv X \wedge$
      $f^0(X), f^1(X), \ldots, f^{2N}(X)$ are defined $\wedge$
      $\exists m, m \in \{1, 2, \ldots, N\}, f^{2m}(X) \equiv X$
  then begin
    output point X and state transition vector sequence
      $f^1(X) - f^0(X), f^2(X) - f^1(X), \ldots, f^{2n}(X) - f^{2n-1}(X)$,
      where n is the smallest natural satisfying $f^{2n}(X) \equiv X$;
    $S := S \cup \{f^{2i}(X) \mid i = 0, 1, \ldots, n-1\}$
    end
  else $S := S \cup \{X\}$

Figure 4: Algorithm A0, which outputs one member of each equivalence class of blocking steady state execution trajectories.

2. How can $\forall X$, $(W, X) \in \gamma$, $f^0(X), f^1(X), \ldots, f^{2N}(X)$ be computed?
3. Algorithm A0 executes the "for each" loop N times, evaluating f at worst $O(N)$ times on each iteration, thereby requiring $O(N^2)$ evaluations of f. Does a more efficient algorithm exist?

The first and third questions are addressed in algorithm A1, which is the first of two refinements of A0. Answering the second question appears to be non-trivial, and is the subject of section 4 and the second refinement, algorithm A2.

A1 reformulates the problem of finding solutions to $\forall X$, $f^{2i}(X) \equiv X$, for $i = \{0, 1, \ldots, N-1\}$ as a problem of finding cycles in a graph of N nodes, which reduces the worst case number of evaluations of f to $O(N)$. (This will be proven as a property of the second refinement, A2, in section 5.)

Let $C_L(P)$ (respectively, $C_R(P)$) denote a predicate whose value is true if and only if P represents a live (respectively, restricted) state. Algorithm A1 reformulates question 1 above in terms of predicates $C_L(P)$ and $C_R(P)$. A method to efficiently decide when the predicates hold is given in section 4.

Let the N constraint line generators be denoted (W^0, X^0), (W^1, X^1), ..., (W^{N-1}, X^{N-1}). Let k and k' denote integers in interval $[0, N)$. It is convenient to define a function g as follows: $g(P) = f(P) - P$; $g(P)$ is a vector representing the transition from the state represented by P to the state represented by $f(P)$. Algorithm A1 appears in Figure 5.

Step A1.1: Construct a directed graph with N nodes labeled $0, 1, \ldots, N-1$ as follows. For each $k \in [0, N)$, if $C_R(X^k) \wedge C_L(f(X^k))$ then draw an arc from node k to node k' satisfying $X^{k'} \equiv f^2(X^k)$.

Step A1.2: If the graph contains no cycles, output "No blocking steady state execution trajectories exist." Otherwise, for each cycle $k_1, k_2, \ldots, k_M$ in the graph (where $k_1 = k_M$), output point X^{k_1} and state transition vector sequence $g(X^{k_1}), g(f(X^{k_1})), g(X^{k_2}), g(f(X^{k_2})), \ldots, g(X^{k_{n-1}}), g(f(X^{k_{n-1}}))$.

Figure 5: Algorithm A1, a refinement of algorithm A0.

Example. *Recall that the progress graph of Figure 2 contains two constraint line generators. Let* $(W^0, X^0) = ((1,1),(4,1))$ *and* $(W^1, X^1) = ((1,1),(1,2))$. *Step A1.1 of algorithm A1 yields a graph of two nodes, with an arc directed from both nodes zero and one directed towards node zero. Step A1.2 outputs point* $X^0 = (4,1)$ *and state transition vector sequence* $(3,3),(2,0)$, *which can be written as rays* $\overline{(4,1),(7,4)}$ *and* $\overline{(7,4),(9,4)}$.

4 Formulas for $C_R(X)$, $C_L(f(X))$, $g(X)$, $g(f(X))$

Let X represent the final point of a constraint line. Algorithm A1 contains four unknown quantities: $C_R(X)$, $C_L(f(X))$ if $C_R(X)$, $g(X)$ if $C_R(X)$, and $g(f(X))$ if $C_R(X) \wedge C_L(f(X))$. These are illustrated in Figure 6.

Geometrically, these represent in terms of a progress graph:

$C_R(X)$: Does a slope one ray rooted at X intersect a constraint line? (When the program enters the state represented by X, does it ever block again?)

$C_L(f(X))$: If $C_R(X)$, is the final point of the second ray in an execution trajectory rooted at X the final point of a constraint line? (If $C_R(X)$, when the program enters the blocked state represented by $f(X)$, does a blocked process eventually unblock?)

$g(X)$: If $C_R(X)$, what is the length of the perpendicular projection on either axis of a slope one ray rooted at X whose final point is P', such that P' is the only point on the ray that lies on a constraint line? (If $C_R(X)$, how long does the program run in parallel before blocking at the next semaphore?)

$g(f(X))$: If $C_R(X) \wedge C_L(f(X))$, what is the distance from $f(X)$ to the final point of the constraint line containing $f(X)$? (If $C_R(X) \wedge C_L(f(X))$, how long does the program block when it enters the state represented by $f(X)$?)

We solve the more general problem of how to compute $C_R(P)$, $C_L(f(P))$ if $C_R(P)$, $g(P)$ if $C_R(P)$, and $g(f(P))$ if $C_R(P) \wedge C_L(f(P))$ for any point P in a progress graph. This is because restricting P to be the final point of a constraint line does not appear to simplify the problem.

4.1 Solution Alternatives

Three methods of solution are an analytic method, computational geometry, and integer programming. An analytic method is used in this paper.[1] A purely computational geometric method is left as an open problem. Finding $g(P)$ is equivalent to the following integer programming problem: minimize the length of a slope one diagonal ray rooted at P subject to the constraint that the final ray point lies on some constraint line generated by an element of γ.

We rule out the use of integer programming based on two drawbacks. First, integer programming examines a potentially infinite search space, and only terminates if $C_R(P)$ holds. Second, integer programming wastes time searching infeasible points, because each constraint line generated by an element of γ may or may not contain a feasible solution.

4.2 Formula for $C_L(f(P))$ if $C_R(P)$

Definition. *For any two points P and P', $P < P'$ if and only if $P_0 \leq P'_0 \wedge P_1 \leq P'_1 \wedge P_0 + P_1 < P'_0 + P'_1$.*

Computation of $C_L(P)$, true if and only if point P represents a live state, follows directly from the definition of live state. Recall from section 1 that a live state is any deterministic and blocked state that is not dying. Also recall that a state is deterministic if it is not the initial point of a constraint line, which is a point of intersection of two constraint lines, and a state is blocked if it lies on a constraint line. By Lemma 9 of the companion paper [1], a deterministic point lying on a constraint line represents a dying state if and only if there exists no point P' at which the constraint line intersects another constraint line such that $P \leq P'$. Therefore to decide whether a point P represents a dying state requires computing, for one instance of each constraint line, the set of all other constraint lines which it intersects.

Definition.
For any constraint line generator $(W, X) \in \gamma$, $\iota(W, X)$ denotes the set of all points P such that line segment $\overline{[W, X)}$ intersects another constraint line at P.

Known computational geometric algorithms for reporting intersections of line segments may be used to compute $\iota(W, X)$; see for example Bentley and Ottmann [3].

[1] The method is not purely analytic because the formula for $C_L(f(X))$ is based on a list of constraint line intersections, obtained from a computational geometric algorithm.

The following theorem provides a means to compute $C_L(P)$, given $\iota(W,X)$ for all $(W,X) \in \gamma$.

Theorem 1 *$C_L(P)$ is true if and only if P lies on a constraint line generated by some $(W,X) \in \gamma$ and* $\max\{P' | P' \in \iota(W,X)\} < (P_0 \bmod c_0, P_1 \bmod c_1)$.

Proof: Follows from Lemma 9 in the companion paper [1]. □

4.3 Analytic Solution of $C_R(P)$, $g(P)$, and $g(f(P))$

The solution method presented here requires the following assumption.

Assumption. *Constants c_0 and c_1 represent rational quantities.*

An equivalent assumption, which is the one used in this section, follows.

Assumption. *Constants c_0 and c_1 represent relatively prime integers.*

To demonstrate the equivalence, let $L(c_0,c_1)$ denote the least common denominator of c_0 and c_1. Let $G(c_0,c_1)$ denote the greatest common divisor of $c_0 L(c_0,c_1)$ and $c_1 L(c_0,c_1)$. Multiplying rational c_0 and c_1 by $L(c_0,c_1)/G(c_0,c_1)$ yields relatively prime quantities, to which the solution method below is applied to calculate $C_R(P)$ and $g(P)$. The resulting (real) value of $g(P)$ multipled by $G(c_0,c_1)/L(c_0,c_1)$ corresponds to the solution for the original, rational c_0 and c_1.

The analytic solution for $C_R(P)$, $g(P)$, and $g(f(P))$ is based on a simplified version of a progress graph which has constraint lines generated by *only one* line segment (W,X) in generator set γ.

Definition. *$C_R(P,W,X)$ and $g(P,W,X)$ are defined as $C_R(P)$ and $g(P)$, respectively, in a progress graph with $\gamma = \{\overline{[W,X)}\}$.*

Therefore:

$$C_R(P) = \bigvee_{\{\overline{[W,X)} \in \gamma\}} C_R(P,W,X) \quad \text{and} \tag{1}$$

$$g(P) = \min\{g(P,W,X) \mid \overline{[W,X)} \in \gamma \wedge C_R(P,W,X)\} \text{ if } C_R(P). \tag{2}$$

The minimization in equation (2) arises because $P + (g(P), g(P))$ is the point closest to the origin at which a slope one ray rooted at P intersects a constraint line, if an intersection occurs. The subsequent discussion derives formulas for $C_R(P,W,X)$ and $g(P,W,X)$, to which equations (1) and (2) can be applied to obtain $C_R(P)$ and $g(P)$.

Notation conventions: The remainder of this section assumes without loss of generality that each generator $(W,X) \in \gamma$ generates horizontal constraint lines. (The case of vertical lines follows by interchanging subscripts 0 and 1 in the subsequent text.) The symbol Z (respectively, R) denotes the set of nonnegative integers (respectively, nonnegative reals).

4.3.1 Formula for $C_R(P)$

By Lemma 9 of the companion paper [1], point P represents a restricted state if and only if P lies off a constraint line and a slope one ray rooted at P intersects a constraint line generated by an element of set γ. Each point on the slope one diagonal ray rooted at P is $P + (y, y)$, for $y \in R$. Therefore $C_R(P, W, X)$ holds if and only if

$$\exists y, i_0, i_1;\ i_0 \in Z, i_1 \in Z, y \in R;$$
$$P + (y, y) \in \overline{[W + (i_0 c_0, i_1 c_1),\ X + (i_0 c_0, i_1 c_1))}.$$

The relationship above is rewritten below as two equations, each corresponding to one component of a point. This requires the binary relationship $\in$ to be transformed to an equality relationship by introducing a slack variable, denoted s. The resulting equations are illustrated in Figure 7.

$$P_0 + y + s = X_0 + i_0 c_0 \text{ and} \tag{3}$$

$$P_1 + y = X_1 + i_1 c_1, \tag{4}$$

where

$$s \in (0, X_0 - W_0]. \tag{5}$$

Letting $s' = s + X_1 - X_0 - P_1 + P_0$ and $I(P, W, X)$ denote the set of integers in the interval $(X_1 - X_0 - P_1 + P_0,\ X_1 - P_1 + P_0 - W_0]$, variable y may be eliminated from system (3) to (5):

$$i_1 c_1 - i_0 c_0 + s' = 0, \tag{6}$$

where s' must lie in the interval

$$s' \in I(P, W, X). \tag{7}$$

Therefore $C_R(P, W, X)$ holds if and only if there exists a solution to equation (6). Because c_0 and c_1 are relatively prime integers, s must be an integer by equation (6). Therefore equation (6) is a Diophantine equation.

A necessary and sufficient condition for a solution to exist is that the greatest common divisor of c_1 and c_0 divides s, by Jones' Theorem 3.3 [7]. Therefore a solution exists if and only if interval $I(P, W, X)$ contains an integer value. Applying equation (1) establishes the following theorem.

Theorem 2 *$C_R(P)$ is true if and only if $\exists (W, X) \in \gamma$ such that $\lceil X_1 - X_0 - P_1 + P_0 \rceil \leq \lfloor X_1 - P_1 + P_0 - W_0 \rfloor$.*

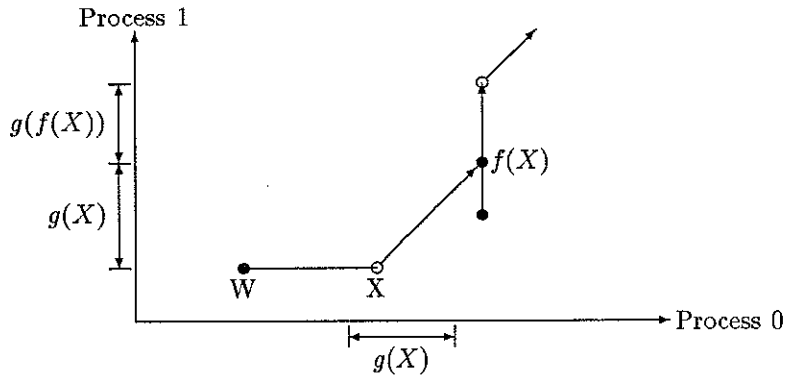

Figure 6: Illustration of four unknowns in Algorithm A1: $C_R(X)$, $C_L(f(X))$ if $C_R(X)$, $g(X)$ if $C_R(X)$, and $g(f(X))$ if $C_R(X)$ and $C_L(f(X))$.

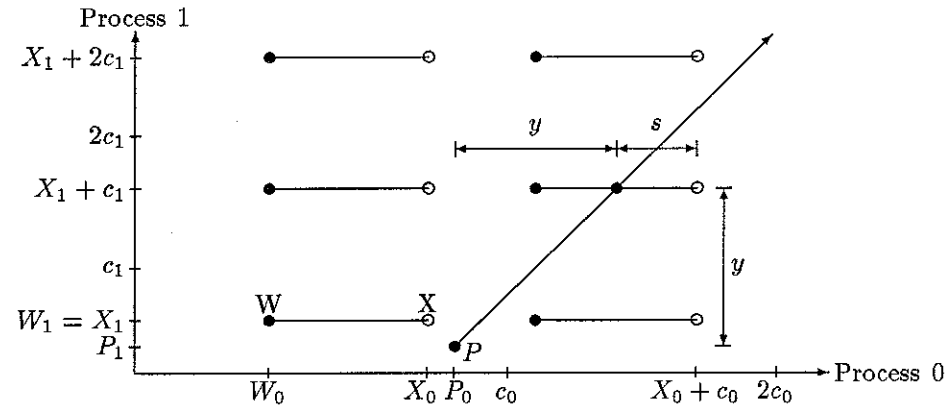

Figure 7: One possible relationship of P, y, and s in a progress graph with constraint lines generated by a single generator, (W, X). In general, there are either zero or an infinite number of points of intersection of a slope one diagonal ray with instances of a single constraint line generator, corresponding to zero or an infinite number of values of y, respectively. In the figure, $i_0 = i_1 = 1$ yields $g(P, W, X) = y$.

4.3.2 Formulas for g(P) if $C_R(P)$ and g(f(P)) if $C_R(P) \wedge C_L(f(P))$

By definition, $g(P, W, X)$ is the minimum nonnegative integer value of y satisfying either (3) or (4). Therefore, from (4),

$$g(P, W, X) = \min\{i_1 c_1 + X_1 - P_1 \mid i_1 \in Z \wedge i_1 c_1 + X_1 - P_0 \geq 0\}. \quad (8)$$

The right hand side of equation (8) requires the minimum element of a set containing an infinite number of elements. The right hand side will be reexpressed as a set containing a *finite* number of elements to permit an algorithm to compute the minimum by exhaustive search. The rewritting is done by expressing unknown i_1 in terms of unknown s and an integer parameter α by applying the solution technique for three variable Diophantine equations in [7], pp. 67-68. There are an infinite number of solutions, which parameter α expresses. The solution to (6) is

$$i_1 = us' + c_0\alpha, \quad (9)$$

where u is an integer satisfying $c_1 u \equiv 1 \pmod{c_0}$. Combining equations (8) and (9) yields

$$\begin{aligned} g(P, W, X) = \min\{&(us' + c_0\alpha)c_1 + X_1 - P_1 \mid \\ &\alpha \in Z \wedge s' \in I(P, W, X) \wedge (us' + c_0\alpha)c_1 + X_1 - P_1 \geq 0\}. \end{aligned} \quad (10)$$

The right hand side of equation (10) still requires the minimum value of an infinite set of elements. However, α can be rewritten in terms of s. If $C_R(P, W, X)$, then equation (10) has a solution. Solving $(us'+c_0\alpha)c_1+X_1-P_1 \geq 0$ for the value of α that yields, for a given value of s', the minimum, nonnegative value of $(us' + c_0\alpha)c_1 + X_1 - P_1$, we obtain

$$\alpha = -\left\lceil \frac{c_1 us' - P_1 + X_1}{c_1 c_0} \right\rceil. \quad (11)$$

Combining equations (2), (10), and (11) establishes the following theorem.

Theorem 3 *Given relatively prime c_0 and c_1, and given P such that $C_R(P) \wedge C_L(f(P))$,*

$$\begin{aligned} g(P) = \min\{&(c_1 us' + X_1 - P_1) \bmod c_0 c_1 \mid \\ &(W, X) \in \gamma \wedge C_R(P, W, X) \wedge s' \in I(P, W, X)\} \end{aligned} \quad (12)$$

where u satisfies $c_1 u \equiv 1 \pmod{c_0}$. Furthermore, $g(f(P)) = s^* - (X_1 - X_0 - P_1 + P_0)$, where s^* is the value of s' that yields the minimum $g(P)$ in equation (12).

The fact that Theorem 3 requires the minimum of a set containing a finite number of elements permits computation of $g(P)$ and $g(f(P))$ by exhaustively examining all set elements. The second refinement of algorithm A0, presented in the following section, exploits this fact.

5 Second Refinement: Algorithm A2

This section refines algorithm A1 by incorporating the formulas for $C_L(P)$, $C_R(P)$, and $g(P)$ given in Theorems 1 to 3. The result is algorithm A2. The data and function declarations of A2 are shown in Figure 8, and the executable code is shown in Figure 9. Algorithm A2 is written in a Pascal-like notation, which simplifies analysis of the time and space required during execution.

The input to algorithm A2 is the constraint line generator set, γ, and cycle times, c_0 and c_1. Set γ is stored as an $N \times 2$ array of initial points (W) and an $N \times 2$ array of final points (X). For example, the initial point W^{N-1} of constraint line generator (W^{N-1}, X^{N-1}) is stored in (W[N-1,0],W[N-1,1]). The cycle times are stored in array C.

The edges of the graph algorithm A1 generates are stored in $N \times 1$ array E; if E[0]=3 then an arc exists from node 0 to node 3. The state transition vectors output by algorithm A1 are stored in $N \times 1$ arrays G0 and G1.

5.1 Time and Space Requirements of Analytic Solution

The time required by algorithm A2 is dominated by the time required to evaluate the minimization in Theorem 3, as the following theorem establishes. Let $I_N(P, W, X)$ denote the number of integers in interval $I(P, W, X)$. Let $D = \max\{I_N(X^k, W^{k'}, X^{k'}) \mid k, k' \in \{0, 1, \ldots, N-1\}\}$.

Theorem 4 *Algorithm A2 requires at worst time $O(N^2 D)$, excluding the time to compute the l.c.d. and g.c.d. of two rational numbers (in function LG), and the time to solve the congruence for u.*

Proof: Consider the time required by each step of algorithm A2.

Step A2.0: The for loop iterates N times; therefore step A2.0 requires $O(N)$ time.

Step A2.1: Set $\iota(W^k, X^k)$ may be computed using an algorithm that reports the points of intersection of line segments. In particular, the constraint line instances in the rectangle whose opposite vertices are the origin and point (c_0, c_1) may be used.[2] Shamos and Hoey[10] have shown that a lower bound on the time required to report intersections is $O(N log N + k)$, where k is the number of intersecting pairs. Thus step A2.1 requires time $O(N^2)$ at worst.

Step A2.2: The inner loop of step A2.2 is iterated at most $O(N^2 D)$ times. At worst the if test in the inner loop is true in each iteration; this implies the inner loop requires constant time in each of $O(N^2 D)$ iterations.

[2]Such an algorithm is modified so that, rather than reporting all intersections, it stores in ML[K] the intersection point furthest from the origin found so far for the instance of constraint line K that the algorithm considers.

```
var
  N,K,K': integer;
  T,LG,U,S': real;
  C:    array [0..1] of rational;     {C[0]=c_0; C[1]=c_1;}
  W,X: array [0..N-1,0..1] of real; {(W[K],X[K])=(W^k, X^k)}
  E:    array [0..N-1] of integer;    {E[K]=∞, if f^2(X^k) exists,
                                       else E[K]=k', where X^k' ≡ f^2(X^k)}
  G0:   array [0..N-1] of real;       {GO[K]=g(X^K) if C_R(X^K)}
  G1:   array [0..N-1] of real;       {G1[K]=g(f(X^K)) if C_R(X^K) ∧ C_L(f(X^K))}
  MI:   array [0..N-1,0..1] of real; {MI[K]=max{P|P ∈ ι(X^K, W^K)}}
  M:    array [0..N-1] of real;       {M[K] = s' minimizing g(P) in Theorem 3}
  P:    array [0..1] of real;         {a point}

{Interval I(X^K', W^K, X^K) is represented as [ IL(K,K'), IH(K,K') )
using the following two functions. }

function IL(K,K':integer):real
begin return X[K',1]-X[K',0]-X[K,1]+X[K,0] end

function IH(K,K':integer):real
begin return X[K',1]-X[K,1]+X[K,0]-W[K',0] end

function LG(Y0,Y1:rational):integer
var L: integer;
begin
  L := l.c.d. of Y0 and Y1;
  return g.c.d. of L*Y0 and L*Y1
end
```

Figure 8: Algorithm A2: data and function declarations

```
begin
{Step A2.0; initialization}
  input C[0], C[1], N;
  LG:=LG(C[0],C[1]);  C[0]:=C[0]*LG;  C[1]:=C[1]*LG;
  solve C[1] * U ≡ 1  (mod C[0]) for U;
  for K:=0 to N-1 do begin
    E[K]:=G0[K]:=∞;
    input W[K,0], W[K,1], X[K,0], X[K,1];
    W[K,0]:=W[K,0]*LG;  X[K,0]:=X[K,0]*LG;
    W[K,1]:=W[K,1]*LG;  X[K,1]:=X[K,1]*LG
    end

{Step A2.1 (Steps A2.1 and A2.2 together correspond to step A1.1.)}
  for K:=0 to N-1 do MI[K]:= max{P|P ∈ ι(W[K], X[K])};

{Step A2.2}
  for K:=0 to N-1 do begin
    for K':=0 to N-1 do
      if ⌈IL(K, K')⌉ ≤ ⌊IH(K, K')⌋ then begin {C_R(X^K) holds; compute G0[K]}
        for S':=⌈IL(K, K')⌉ to ⌊ IH(K,K') ⌋ do begin
          T:=(C[1] * U * S' + X[K', 1] − X[K, 1]) mod C[0] * C[1]);
          if T<G0[K] then begin E[K]:=K'; G0[K]:=T; M[K]:=S' end
          end
        if MI[K] < (X^K + G0[K]) then {C_L(X^K) holds; compute G1[K]}
          G1[K]:=M[K]-IL(K,K')
      end
    end

{Step A2.3; corresponds to step A1.2}
  for each cycle k_1, k_2, ..., k_M in graph in array E (where k_1 = k_M) do begin
    output point (X[k_1, 0]/LG, X[k_1, 1]/LG);
    for K:=1 to M-1 do output GO[K]/LG, G1[K]/LG
    end
end
```

Figure 9: Algorithm A2: executable code

Step A2.3: The graph represented by array S has at most one outgoing arc from each node. Hence the graph has at most N edges, and all cycles can be detected in time $O(N)$. □

Theorem 5 *Algorithm A2 requires $O(N)$ storage, excluding the storage to required compute ι.*
Proof: Follows from the array dimensions in declaration portion of algorithm A2 (Figure 8). □

6 Conclusion

A novel analysis method to derive the exact sequence and duration of blocking and running experienced by programs consisting of two non-terminating processes sharing mutually exclusive, reusable resources has been presented. Proved elsewhere is that any execution sequence of a program under consideration that does not contain a nondeterministic or dead state reaches a cyclic steady state behavior [1]. The paper develops through successive refinements algorithm A2, which reports one member of each equivalence class of blocking steady state execution trajectories reachable by any initial condition. A2 finds the minimum solution to each of a set of Diophantine equations. The algorithm is based on a mapping of the program to a progress graph.

Recall from section 5 that A2 requires space $O(N)$ and time $O(N^2D)$, where N is twice the number of resources shared and D is related to the precision with which measurements of program timings are desired. Consider the time requirement. In practice, the N^2 term is not prohibitive, because two process algorithms usually have a fairly small number of synchronization constraints. The D term, however, may force us to make approximations. As discussed in section 4, we map any rational cycle time c_0 or c_1 to relatively prime integers by multiplying both cycle times by LG, where LG is the least common denominator of c_0 and c_1 divided by the greatest common divisor of a multiple of c_0 and c_1. Thus, D will grow with the product $LG \cdot c_0$ or $LG \cdot c_1$, which is the ratio of the cycle time to the resolution of the measurement clock. For example, if we measure an algorithm to microsecond resolution, LG is at most 10^6. If $c_0 = c_1 = 100$ seconds, then $D = 300 \cdot 10^6$. However, we may be willing to trade accuracy for computation time by approximating measurements by milliseconds, so that $D = 300 \cdot 10^3$.

Several open problems remain:

- solution of the model for an arbitrary number of processes (Empirical evidence confirms the existence of steady state modes for at least 64 processes [2].);
- refinement of algorithm A1 using a computational geometric algorithm as an alternative to the analytic algorithm A2;

- analysis of the model for irrational cycle times c_0 and c_1 (Chaotic behavior may exist for these values.); and
- analysis of reachability, deadlock, and nondeterministic states.

7 Acknowledgements

The authors wish to thank S. Tripathi for discussions during the course of this work, and L. Svobodova and the referee for suggestions on an earlier form of the manuscript.

References

[1] M. Abrams. *Geometric Performance Analysis of Mutual Exclusion: The Model.* Dept. of Computer Science, Virginia Tech, TR-90-58, Dec. 1990.

[2] M. Abrams and A. K. Agrawala. Performance study of distributed resource sharing algorithms. *Distributed Processing Technical Committee Newsletter 7,* IEEE Technical Committee on Distributed Processing, 3, (Nov. 1985), 18-26.

[3] J. L. Bentley and T. A. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Trans. on Computers C-28.*, 9, (Sept. 1978), 643-647.

[4] S. D. Carson and P. F. Reynolds, Jr. The geometry of semaphore programs. *ACM Trans. on Programming Languages and Systems 9.* 1, Jan. 1987, 25-53.

[5] E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Comp. Surv. 3,* June 1971, 70-71.

[6] E. W. Dijkstra. *Cooperating sequential processes.* Tech. Rep. EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[7] B. W. Jones. *The theory of numbers.* Rinehart: New York, 1955.

[8] W. Lipski and C. H. Papadimitriou. A fast algorithm for testing for safety and detecting deadlocks in locked transaction systems. *J. Alg. 2,* 3, Sept. 1981, 211-226.

[9] C. H. Papadimitriou. Concurrency control by locking. *SIAM J. Comput. 12,* 2, May 1983, 215-226.

[10] M. I. Shamos and D. J. Hoey, Geometric intersection problems. In *Proc. 17th Annual Conference on Foundations of Computer Science,* Oct. 1976, pp. 208-215.

[11] M. Yannakakis, C. H. Papadimitriou, and H. T. Kung. Locking policies: safety and freedom from deadlock. In *Proc. of the 20th ACM Symposium on the Foundations of Computer Science,* 1979, pp. 283-287.