COLLEGE OF ARTS AND SCIENCES

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Blacksburg, Virginia 24061-0106

DEPARTMENT OF COMPUTER SCIENCE 562 McBRYDE HALL (703) 231-6931

Geometric Performance Analysis of Mutual Exclusion: The Model

By Marc Abrams

TR 90-58

Geometric Performance Analysis of Mutual Exclusion: The Model

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

TR 90-58

4 December 1990

Abstract

This paper is motivated by the need for techniques to better understand parallel program run-time behavior. The paper first formally describes a general model of program execution based on Dijkstra's progress graphs. The paper then defines a special case of the model representing two cyclic processes sharing mutually exclusive, reusable resources. Processes synchronize through semaphore operations that are not embedded in conditionally executed code segments. Model parameters are the times at which each semaphore operation completes and the independently derived, constant execution time required by each process between points of completion of semaphore operations.

Any program execution leads to either a nondeterministic state, a deadlock, an oscillating and nonblocking steady state execution sequence, or one of at most N oscillating and blocking steady state execution sequences, where N is twice the number of semaphores. Varying a parameter near a critical value suddenly changes the steady state execution sequence. Progress graphs illustrations lend insight into the synchronization structure of a program. An algorithm deriving the set of all blocking steady state execution sequences that arise for all initial process starting orders is stated and proved correct.

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, software performance analysis

Additional Key Words and Phrases: resource sharing, synchronization, parallel programs, Petri nets

Contents

1 Introduction

One approach to software performance analysis starts with a definition of state and program execution. *State* is an instantaneous description of some properties of interest of a program. *Program execution* is a mapping of a state to a successor state. Analyzing program performance then means deriving the sequence of states and state occupancy times that a program passes through during execution, given some initial program state. This state sequence is called the *program execution sequence,* and is formally defined later.

In general, for a given program there may exist an infinite number of possible execution sequences. For example, there may exist a different execution sequence for each initial program state. There may even exist multiple execution sequences for a single initial state, if program execution is nondeterministic.

Knowing the execution sequence of a program is desirable for several reasons. For example, the execution sequence may help in tuning a program, because it can be used to calculate a variety of performance metrics. For example, if the choice of program state distinguishes when a process in a parallel program is blocked from when it is running, then the fraction of time each process spends waiting may be derived. Finally, execution sequences reveal information about the transient behavior; such information is recognized to be particularly important in analyzing program behavior [18].

There are several ways to derive execution sequences for a program. One is to instrument a program, execute it for a finite time period, and observe a state sequence. Another is to use the definition of program execution, along with the initial state, to step through an initial subsequence of the execution sequence; this is simulation.

Both program execution and simulation have drawbacks. First, both techniques must be repeated for each initial condition. Second, if the program behavior is nondeterministic, the execution sequence observed represents only one possible behavior. Third, each technique is executed for a finite period, generating only an initial portion of an execution sequence. Therefore, any repetition of behavior that exists in an execution sequence may not be revealed.

This paper investigates the problem of efficiently finding the set of all possible execution sequences for a particular definition of state and for a class of two process programs that share mutually exclusive, reusable resources using semaphores. For this class, the paper:

1. defines transient and steady state execution sequences, and a notion of equivalent execution sequences,

2. shows that any program execution leads to either a nondeterministic state, a deadlock, an oscillating and nonblocking steady state execution sequence, or one of at most N oscillating and blocking steady state execution sequences, where N is twice the number of semaphores; and

3. presents and proves the correctness of an algorithm that outputs one member of each equivalence class of blocking steady state execution sequences that arise for all initial process starting orders.

The paper is organized as follows. The next section formally extends Dijkstra's progress graphs to analyzing program performance, and in doing so, establishes items (1) and (2). Section 3 then presents the algorithm of item (3) above, and section 4 contrasts the approach in this paper with related literature.

2 Performance Analysis with Progress Graphs

The first subsection states a set of postulates which form a model of computation. The second subsection defines a mapping from the computation model to progress graphs. Subsequent subsections identify and derive properties of a special class of programs using process graphs.

2.1 Computation Model

The computation model is based on six postulates. First, progress of processes is measured with respect to a clock external to the program. No processor has knowledge of this clock, and its value has no effect on the program. Second, the salient feature of a process that the model represents is whether a process is running or blocked.[1] Programs whose processes do not all simultaneously start execution are modeled by viewing each process as blocked at all time instances prior to when the process initially runs. Third, for the purpose of program performance analysis, the "state" of each process is represented by a single nonnegative real number equal to the duration of time the process has spent running. The state of a program is an ordered tuple whose components each represent the state of one process. Therefore the state space of a program is continuous, and each state is an instantaneous description of the program. Fourth, an execution of a program is represented by the sequence of program states, called an *execution sequence*, that a program passes through during that execution. Fifth, a *dead* program state is a state in which all processes are blocked. Finally, an execution sequence either does or does not have a final state; if a final state exists, the state is dead.

P1: A single clock external to the parallel program exists. References to time refer to values read from this clock.

P2: At any instant of time, each process is either *running* or *blocked*.

[1] Operating systems literature sometimes distinguishes blocked, ready, and running states. In this paper, ready states are included with blocked states.

P3: A *program state* (or *state*) is represented by an ordered tuple of nonnegative 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.

P4: The sequence of program states that a program passes through during a particular execution defines a *program execution sequence.*

P5: A program state is *dead* if and only if all processes are blocked.

P6: An execution sequence contains a dead state if and only if that state is the final state of the execution sequence.

The choice of state in Postulate P3 leads to a simple formulation of the computation model, but has several subtle implications that will become evident in the remainder of this section. For example, one cannot tell from a single state whether a process is blocked or running; this is only evident from sequences of states. Formally, by Postulates P2, P3, and P4, the subsequence $(x, y), \ldots, (x + \delta, y)$ of some execution sequence represents a period of time during which process zero runs for δ time units and process two is blocked, where x, y, and δ denote nonnegative real numbers. Therefore in the execution sequence (0,0), ..., (10,15), ..., (10,25), ..., (20,25), ..., at state (10,15) process zero is blocked for ten time units, after which process one blocks for ten time units.

Definition. *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 in two ways. The first classification partitions all program states into (1) those which either have no successor in any execution sequence or have a unique successor state in all execution sequences, and (2) the complement. Such states are called *deterministic* and *nondeterministic states*, respectively. The second classification partitions all program states into those in which (1) at least one process is blocked and (2) no processes are blocked. Such states are called *blocked* and *running* program states, respectively. The categories are defined below.

Definition. *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 blocked state is a state in which at least one process is blocked. All program states that are not blocked are running states.*

Lemma 1 *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.*

Proof: Follows from Postulates P2 and P3. □

2.2 Progress Graphs: The Geometric Consequence of the Computation Model

The general model is based on Dijkstra's *progress graphs* [7]. A progress graph is "a multidimensional, Cartesian graph in which the progress of each of a set of concurrent processes is measured along an independent time axis. Each point in the graph represents a set of process times." [6]

Papadimitriou, Yannakakis, Lipski, and Kung [14, 19, 25] 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 [6] prove liveness properties in systems with an arbitrary number of processes containing straight line sequences of Dijkstra's P and V operations.

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.

Interpreting the ordered tuple representing a program state as the Cartesian coordinates of a graph point implicitly defines a bijection 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. Corresponding to each program execution sequence is a continuous path in the graph called a *program execution trajectory*; the correspondence arises from the mapping from program states to graph points.

Definition. *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.*

Two process programs: In progress graphs of two process programs, subtrajectories of execution trajectories representing either a sequence of running states or a sequence of blocked states in which the same process remains blocked have a simple geometric manifestation: a ray. This paper uses the convention that a ray is closed at its initial point and open at its final point. The following lemma establishes that an execution trajectory consists of a sequence of rays possibly followed by a point.

Definition. *A* ray *is directed line segment of possibly infinite length.*

Lemma 2

2a. 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.

2b. 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.

2c. An execution trajectory is a continuous path in a progress graph consisting of either a sequence of zero or more rays followed by a point or a sequence of one or more rays. In either case, each ray has slope 0, 1, or infinity.

Proof: By P3 and P4, no component of any state in an execution sequence can be larger than the corresponding component of any subsequent state. Therefore any two unique states in an execution sequence obey the following property: Each component of the tuple representing the earlier state cannot be greater than the corresponding component of the tuple representing the other state.

2a and 2b: Follow from postulates P3 and P4.

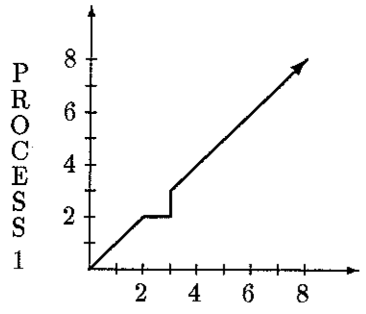
2c: By postulates P2, P3, and P4, an execution trajectory is a continuous path. By postulate P6 and Lemmas 2a and 2b, an execution trajectory consists of either a sequence of zero or more rays followed by a point or a sequence of one or more rays. All program states are either blocked or running; therefore by Lemmas 2a and 2b each ray has slope one or is parallel to an axis. □

Rays with slope one are referred to as *diagonal rays*; rays parallel to an axis are referred to as *nondiagonal rays*.

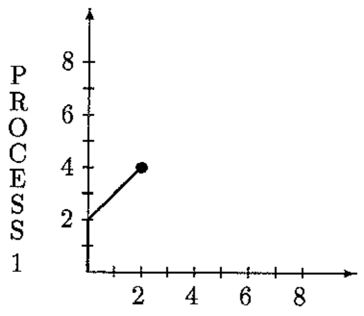
Definition. *Given two distinct states in an execution sequence, a* state transition *exists from the earlier to the later state. A* state transition vector *is an ordered pair corresponding to a ray in an execution trajectory representing the amount of time that each process has spent running during the transition from the state represented by the initial point of the ray to the state represented by the final point of the ray.*

Based on Lemma 2, Figure 1 illustrates several possible execution trajectories. P3 implies that the initial state of an execution sequence, representing the earliest time at which any process is running, corresponds to the origin.

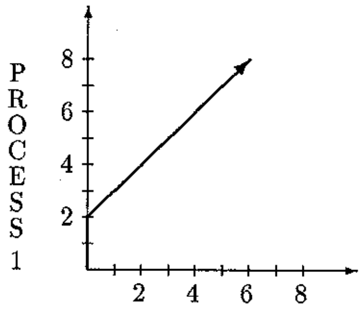
Figure 1(a) represents a program in which both processes run in parallel for two time units, followed by process one blocking for one time unit, followed by process zero blocking for one time unit, followed by both processes running in parallel. The arrowhead at point (8,8) indicates that only the initial portion of



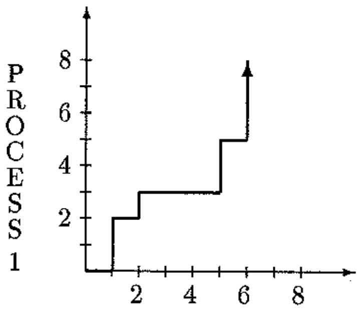
(a) Process one, then zero blocks



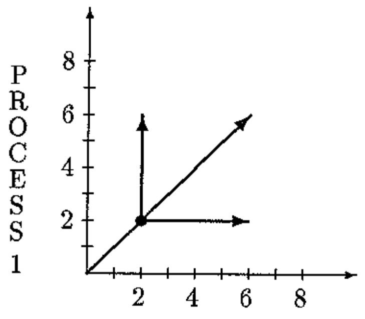
(b) Deadlock



(c) Process one starts first



(d) Multiprogramming



(e) Nondeterminism

Figure 1: Several illustrations of progress graphs in two dimensions.

the trajectory is illustrated. In contrast, the execution trajectory in Figure 1(b) is finite in length: by postulate P6, the final point, (2,4), is a dead state. Figure 1(c) illustrates a program execution in which process one alone executes for two time units, after which both processes run in parallel for at least six time units. Figure 1(d) illustrates a program whose processes are multiprogrammed: exactly one process is running at any time. Figure 1(e) illustrates the four possible outcomes of a nondeterministic program state in a program with two processes, according to Lemma 1: after both processes run for two time units, the program reaches a nondeterministic state represented by point $(2,2)$. In each execution, either process zero blocks or process one blocks or both processes block (therefore point $(2,2)$ represents a dead state) or both processes continue running.

2.3 Special Case: Cyclic Processes, Constant Timings

The remainder of the paper analyzes a special case of progress graphs, in which two processes synchronize through P and V operations on binary semaphores [9]. (The requirement of binary semaphores could be relaxed to permit general semaphores, but this would complicate the presentation.) Let σ denote a semaphore.

The special case models programs meeting the following conditions. First, a program consists of two processes. Second, each processor is uniprogrammed. Third, each process loops forever without termination. Fourth, certain degenerate uses of semaphores are ruled out. (The fourth condition is stronger than required, but will simplify the presentation). Fifth, processes only block due to P operations. Furthermore, whenever a process blocks, *blocking starts in the state following the program state representing the completion of a P operation.* Sixth, processes only unblock due to V operations. Furthermore, *a process only unblocks in the state following the program state representing the completion of a V operation.* Seventh, a process loop body consists of straight line sequences of semaphore operations. Eighth, the code segment that follows each semaphore operation, up to and including the next semaphore operation within a process, requires an independently derived, constant amount of execution time, exclusive of time spent blocked; furthermore this time must be greater than zero. Finally, both processes start running simultaneously (This is relaxed in section 2.6.). These properties are formalized in the conditions below.

C1: A program consists of two processes.

C2: A separate processor executes each process.

C3: The code that each process executes consists of a nonterminating loop.

C4: 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.

C5: 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.

C6: 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.

C7: No P or V operation is contained in a conditionally executed piece of code.

C8: 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 operation 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.

C9: The initial state of all execution sequences is a running state whose components are both zero.

The class of programs modeled consists of two non-terminating processes that share mutually exclusive, reusable resources. Furthermore, no explicit assumptions are made about the architecture (e.g., multiprocessor, network of workstations) on which programs are executed beyond the fact that semaphore semantics can be implemented.

Example 1 *The Dining Philosophers problem [8] meets the conditions stated above. In this problem, two philosophers eating a meal share 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 below.*

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

The time required for each code segment referred to in C8, excluding blocking, appears in Table 1.

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 Example 1 referred to in C8, excluding blocking.

Reasoning about timings of semaphore operations requires a notation to represent the time at which a process completes each P or V operation in a particular execution sequence. This is defined below. Without loss of generality, assume a process uses each semaphore exactly once on each iteration of the outermost loop in its body.

Definition. *For each semaphore σ, $\underline{(p_0(\sigma), p_1(\sigma)}$ (respectively, $\underline{(v_0(\sigma), v_1(\sigma))}$) denotes the components of the final state of the subsequence of states corresponding to the first execution of a $P(\sigma)$ (respectively, $V(\sigma)$).*

Let $r \in \{0, 1\}$ denote a process. C7 implies that $p_r(\sigma)$ and $v_r(\sigma)$ have the same value in all execution sequences.

Another way to interpret the preceding definition of $p_r(\sigma)$ (respectively, $v_r(\sigma)$) is with respect to a clock for process r that is enabled whenever process r switches from blocked to running and is disabled whenever process r switches from running to blocked. (By postulate P3, the state of a process is the value of this clock.) Then $p_r(\sigma)$ and $v_r(\sigma)$ represent the value of this clock the moment process r completes the first $P(\sigma)$ or $V(\sigma)$ operation in any execution sequence.

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. For example, in Table 1, summing the elements of the last two columns of the last two rows yields $c_0 = 5$ and $c_1 = 3$.

Example 2 *In Example 1, $p_r(a)$ is the time at which process r completes the $P(a)$ operation in the first iteration of its loop body, which is the time required to execute "Think; P(a);". From Table 1, $p_0(a) = p_1(a) = 1$. Similarly, $v_r(a)$ is the time required to execute "Think; P(a); Eat; V(a);". From Table 1, $v_0(a) = 4$ and $v_1(a) = 2$.*

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., $P^0 = (P^0_0, P^0_1)$). Let $\bar{r} = 1 - r$.

Definition. *Two points P^0 and P^1 are <u>congruent</u> (denoted as $P^0 \equiv P^1$) if and only if $\forall r, P^0_r \bmod c_r = P^1_r \bmod c_r$.*

Lemma 3 *Process r is blocked in the state represented by a point P if and only if there exists a semaphore σ such that*

$$P_r \bmod c_r = p_r(\sigma) \text{ and}$$

$$P_{\bar{r}} \bmod c_{\bar{r}} \in [p_{\bar{r}}(\sigma),\, v_{\bar{r}}(\sigma)).$$

Proof: Follows from C4, C1 through C9, and the definition of a semaphore. □

Example 3 *By Lemma 3, the following statements hold.*

1. *Process 0 will block in any state represented by a point P meeting the following two conditions:*

   $$P_0 \bmod 5 = 1 \text{ and}$$

   $$P_1 \bmod 3 \in [1, 2).$$

2. *Similarly, process 1 will block in any state represented by a point P meeting the following two conditions:*

   $$P_1 \bmod 3 = 2 \text{ and}$$

   $$P_0 \bmod 5 \in [1, 4).$$

Figure 2 illustrates a progress graph for the Dining Philosophers program. Vertical and horizontal line segments in the figure are the set of points in the plane described by statements 1 and 2, respectively, in Example 3. By Lemma 3 each constraint line is closed at its initial point and open at its final point, as Figure 2 illustrates. These line segments are called *constraint lines*, because no execution trajectory crosses them; this is demonstrated later in Lemma 5.

The constraint lines in Figure 2 belong to one of two congruence classes. A progress graph is defined in terms of one element in each class, namely the line closest to the origin. Each such line is called a *constraint line generator*, because it may be used to generate the coordinates of all line segments in the congruence class to which it belongs. The coordinates of the end points of a constraint line generator are determined as follows. Let $\overline{[P, P')}$ denote a line segment with closed end point P and open end point P'.

Definition. *For each semaphore σ and for each process r, there exists a* constraint line generator, *which is an ordered pair (W, X) such that*

$$X_r = W_r = p_r(\sigma),$$

$$X_{\bar{r}} = v_{\bar{r}}(\sigma), \text{ and}$$

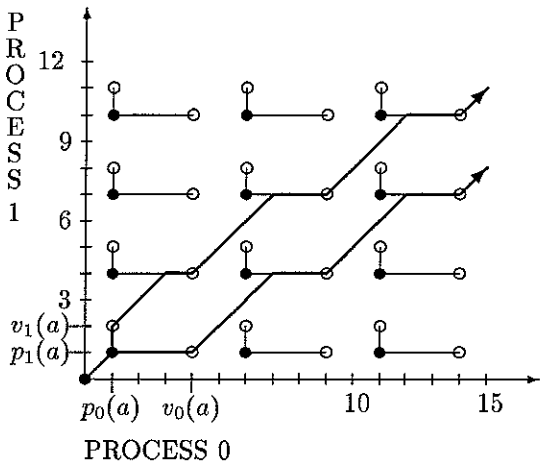


Figure 2: Synchronization constraints for the Dining Philosophers in the Cartesian graph. Open (filled) circles represent open (closed) end points. Thick lines represent two possible execution trajectories implied by Lemma 5.

$$W_{\bar{r}} = p_{\bar{r}}(\sigma).$$

W (respectively, X) is called the initial *(respectively,* final*) point of the constraint line generator. Each element of* $\{[W + (i_0c_0, i_1c_1), X + (i_0c_0, i_1c_1)) \mid \forall r,\ i_r \in \{0, 1, \ldots\}\}$ *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.*

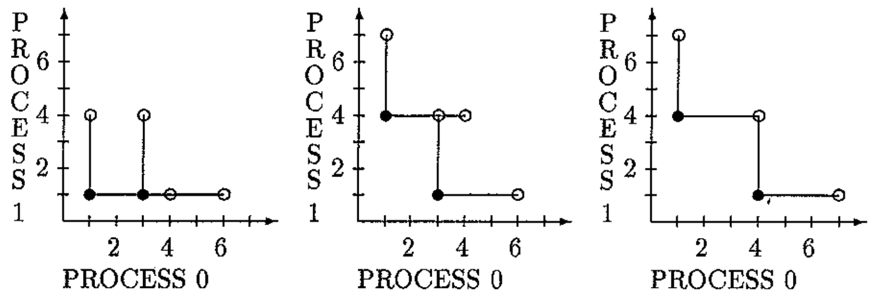
C8 implies that no constraint lines may overlap (see Figure 3(a)), that the final point of one constraint line never lies on another constraint line (see Figure 3(b)), and that the final point of all constraint lines must be distinct (see Figure 3(c)).

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

The following lemma establishes the geometric manifestations of dead, blocked, running, nondeterministic, and deterministic states.

Lemma 4

4a. A point represents a dead state if and only if it is the point of intersection of two constraint lines.



(a) Process one does simultaneous P ops

(b) Process one does simultaneous P, V

(c) Both processes do simultaneous P, V

Figure 3: Constraint line geometries precluded because each code segment in C8 requires time greater than zero.

4b. A point represents a blocked state if and only if it lies on a constraint line.

4c. A point represents a nondeterministic state if and only if it is the initial point of a constraint line.

4d. No running states are nondeterministic.

Proof:

4a. Follows from P5 and Lemma 4b.

4b. Follows from Lemma 3.

4c. Let P denote the point.

If part: By the definition of a constraint line, $P \equiv (p_0(\sigma), p_1(\sigma))$. The conclusion follows from the definition of a semaphore.

Only if part: By C5, C6, and C8, the transition between running and blocked, and by Lemma 1 any nondeterministic state, can only correspond to completion of a single P or V operation within a process. Furthermore, by definition of a semaphore, completion of a V operation cannot correspond to a nondeterministic state because the process performing the V operation does not block and the other process either unblocks or it was already running in *all* execution sequences containing the state. Therefore a nondeterministic state can only correspond to a process completing a P operation. Therefore a nondeterministic state corresponds to both processes completing a P operation on the same semaphore simultaneously, implying $P \equiv (p_0(\sigma), p_1(\sigma))$.

4d. Follows from Lemmas 4b and 4c. □

The contrapositive of Lemma 4b will sometimes be useful: "A point does not lie on a constraint line if and only if it represents a running state."

Given the placement of constraint lines in a progress graph, the following lemma describes how to construct the set of all possible execution trajectories in that progress graph.

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

Lemma 5 *Consider any point P in an execution trajectory T.*

5a. 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''.

5b. 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'.

Proof:

5a: In both cases, the initial ray of the execution trajectory must be contained within a constraint line by Lemmas 2b, 2c, and 4b. Furthermore, in case (1), the execution trajectory consists of a single ray whose final point represents a dead state by Lemma 4a and postulate P6.

5b: Follows from Lemmas 2a, 2c, and 4b. □

By Lemma 5, no execution trajectory crosses a constraint line. By Lemmas 4c and 5, there are exactly two possible initial rays of an execution trajectory rooted at a point representing a nondeterministic state, corresponding to the two constraint lines on which the point lies.

Example 5 *Illustrated in Figure 2 is an initial portion of the execution trajectory with initial point $(0,0)$. The exeution trajectory is a consequence of Lemmas 2c and 5. Because $(0,0)$ does not lie on a constraint line, by Lemma 5b the initial ray of the execution trajectory must have slope one, and exactly one ray point (i.e., the final point) must lie on a constraint line; thus the ray is $\overline{[(0,0),(1,1))}$. Point $(1,1)$ lies on two constraint lines. By Lemma 5a the state represented by point $(1,1)$ has two possible successors. No matter*

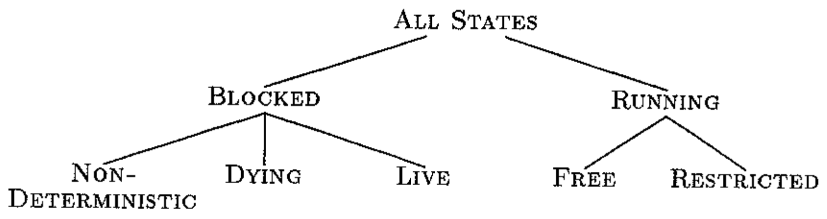


Figure 4: Mutually exclusive and exhaustive categorization of program states.

which successor occurs in a particular execution sequence, the ray with initial point $(1,1)$ *must have as its final point the final point of a constraint line, by Lemma 5a. Therefore the second execution trajectory ray is either* $\overline{[(1,1),(1,2))}$ *or* $\overline{[(1,1),(4,1))}$*. The remaining rays of each execution trajectory are constructed similarly.*

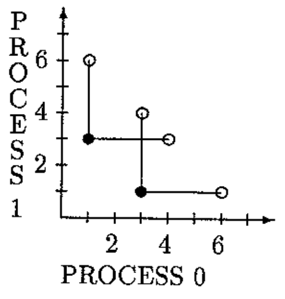
2.4 Characterizing Graph Points

Section 2.1 classifies program states as either blocked or running and as either deterministic or nondeterministic. Blocked states may be further partitioned into three mutually exclusive and exhaustive classes: (1) nondeterministic states, (2) *dying states*, which are deterministic states that are dead or lead to a dead state before the blocked process unblocks, and (3) *live states*, which are the complement of (1) and (2). Running states may be classified into those in which all processes will forever remain running and the complement; states belonging to each category are called *free* and *restricted*, respectively. Figure 4 illustrates the categorization.

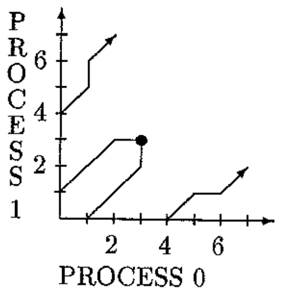
Definition. *A* dying state *is either a deterministic and dead state or a deterministic and blocked state in which, in all execution sequences containing the state, one process is blocked and remains blocked in all subsequent states and the final state is dead. A* live state *is any deterministic and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

Appendix A establishes the following. 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.

Let $\overline{(P,P')}$ denote a line segment with open end point P and open end point P'.



(a) Deadlocking constraints



(b) Sample execution trajectories from (a)

Figure 5: Geometry of deadlock. Each process tries to hold both resources simultaneously, and some execution trajectories reach a dead state. Point $(3,3)$ represents a dead state. Points $(1,3)$ and $(3,1)$ represent nondeterministic states.

Example 6 *Nondeterministic states are represented by points $(3,1)$ and $(1,3)$ in Figure 5(a). Dying states are represented by points on lines $\overline{((1,3),(3,3))}$ and $\overline{((3,1),(3,3))}$ in Figure 5. Live states are represented by points on line segments $\overline{((1,3),(1,6))}$, $\overline{((3,3),(4,3))}$, $\overline{((3,3),(3,4))}$, and $\overline{((3,1),(6,1))}$ in Figure 5(a). Free states will be illustrated later (in Figure 6). Restricted states are represented by all points off a constraint line in Figure 2.*[2]

2.5 Homomorphic Execution Trajectories

In Figure 2, the execution trajectories rooted at $(3,1)$, $(3,4)$, and $(8,4)$ are homomorphic. That is, adding the vector $(0,c_1)$ to all points on the trajectory rooted at $(3,1)$ yields the trajectory rooted at $(3,4)$. Furthermore, adding vector (c_0,c_1) to the trajectory rooted at $(3,1)$ yields the trajectory rooted at $(8,4)$.

Intuitively, due to the congruence of constraint line end points, shifting an execution trajectory by a vector whose components are multiples of the cycle lengths yields another execution trajectory. The following definition and lemma establish that execution trajectories rooted at congruent points are homomorphic. The lemma is used in many subsequent proofs.

Definition. *Two line segments, rays, or trajectories are equivalent if and only if there exists a one-to-one correspondence between their points such that corresponding points are congruent. The operator $\equiv$ denotes equivalence.*

Lemma 6 *Given any two execution sequences containing only deterministic states with initial states represented by points P and P', if $P \equiv P'$ then the execution trajectories rooted at P and P' are equivalent.*

[2]That all points off a constraint line in Figure 2 represent restricted states is not obvious from the figure, but follows from Theorem 2 in the companion paper.

Proof: See Appendix B. □

2.6 Initial Conditions

Condition C9 requires both processes to start execution simultaneously. This section relaxes C9 to permit one process to start execution before the other, as is illustrated in Figure 1(c).

This presents two technical problems, with respect to the way progress graphs have been defined. First, as discussed in section 2.2 (see Figure 1) there is exactly one initial state in a progress graph, represented by point $(0, 0)$. Second, by Lemma 4b a blocked state corresponds to a point on a constraint line, and by definition, constraint lines correspond to semaphores.

To relax C9 without altering the definition of progress graphs, the key idea is that given any execution sequence violating C9, *the execution sequence obtained by deleting the longest prefix of blocked states* will be represented by an execution trajectory in a progress graph.

Definition. *The* concurrent portion *of an execution sequence is obtained by deleting the longest initial subsequence of the execution sequence consisting of blocked states.*

For example, point $(0, 2)$ represents the initial state of the concurrent portion of the execution sequence represented in Figure 1(c).

Lemma 7 reduces the problem of finding the concurrent portion of *any* execution of a program to the problem of obtaining the set of all execution trajectories whose initial point is on the x or y axis and within one cycle length of the origin.

Lemma 7 *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.*

Proof: By P3, the initial state of the concurrent portion of any execution trajectory must lie on the x or y axis. Applying Lemma 6 completes the proof. □

The concurrent portion of all possible execution sequences of a program may be illustrated by drawing a progress graph along with *all* execution trajectories whose initial point lies on line $\overline{[(0,0),(c_0,0))}$ or $\overline{(0,0),(0,c_1)}$. This is exemplified in Figure 6, in which each process of a program performs five P and five V operations. The shaded areas in the graph represent an infinite number of execution trajectories and have infinite extent. The figure will be discussed further in the following section.

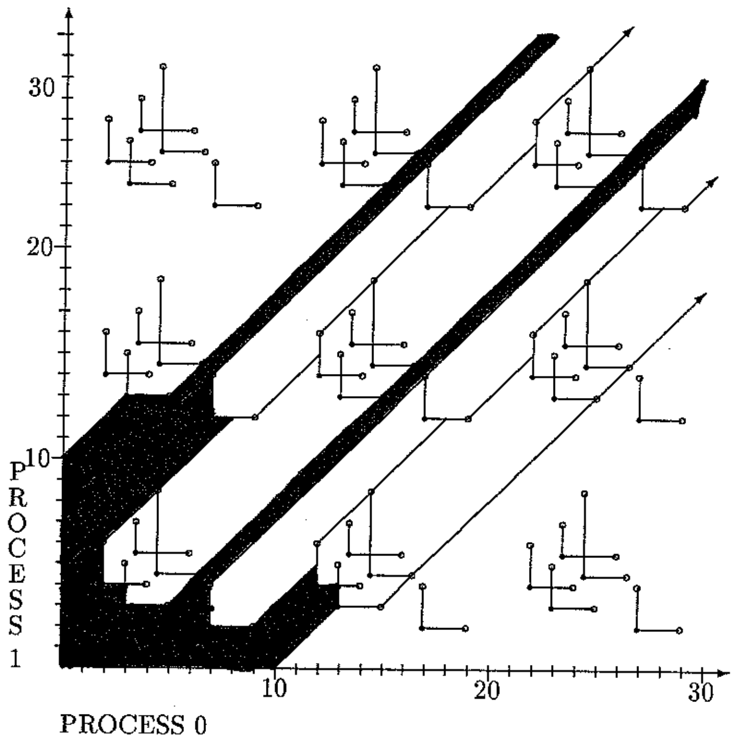


Figure 6: Illustration of a complex progress graph.

2.7 Transient and Steady State

In general, an execution sequence consists of a *transient portion* followed by an infinite number of repetitions of a *steady state cycle.* Either portion may be empty. The final state in the transient portion is the initial state of the first cycle of the steady state sequence. These concepts are formalized in the following definition and established in Theorem 1.

Definition. *The following definitions apply to 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 of the subtrajectory, which are called the* initial *and* final points of the steady state execution trajectory, *respectively. 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. The* initial point of a transient execution trajectory *is the initial point of the execution trajectory. The* final point of a transient execution trajectory *is the smallest point of the execution trajectory lying on any steady state execution trajectory, if the execution trajectory contains a steady state execution trajectory.*

Example 7 *Figure 2 contains two program execution trajectories; consider the one containing point* $(1,2)$. *This trajectory contains a transient execution trajectory with initial and final points* $(0,0)$ *and* $(3,4)$, *respectively, followed by an infinite number of repetitions of a steady state execution trajectory. One steady state execution trajectory has initial and final ponts* $(3,4)$ *and* $(8,7)$, *respectively. Another steady state execution trajectory has initial and final points* $(8,7)$ *and* $(13,10)$, *respectively.*

Excluding execution sequences containing nondeterministic states, all programs meeting conditions C1 to C8 either reach a dead state or enter steady state, as the following theorem establishes. Appendix C contains the proof of the theorem.

Theorem 1 *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 8 *Depending on the initial condition, the program whose progress graph is shown in Figure 6 may reach a nondeterministic state, a dead state, a non-blocking steady state, or a blocking steady state, as indicated in Table 2. The shaded regions containing lines* $\overline{[(22,30),(23,30)]}$ *and* $\overline{[(30,27),(30,28)]}$ *are of infinite extent.*

For initial point:	execution trajectory reaches:
on $\overline{((0,5),(0,10))}$	non-blocking steady state
$(0,5)$	nondeterministic state at $(7,12)$
on $\overline{((0,2),(0,5))}$	blocking steady state
$(0,2)$	nondeterministic state at $(2,4)$
on $\overline{((0,0),(0,2))}$	dead state $(3,4)$
$(0,0)$	nondeterministic state at $(3,3)$
on $\overline{((0,0),(5,0))}$	non-blocking steady state
$(5,0)$	nondeterministic state at $(7,2)$
on $\overline{((5,0),(8,0))}$	blocking steady state
$(8,0)$	nondeterministic state at $(12,4)$
on $\overline{((8,0),(10,0))}$	dead state $(13,4)$

Table 2: Behaviors present in Figure 6 progress graph.

To conclude this section, note that illustrations of progress graphs represent one more piece of information: convergence of execution trajectories. A set of execution trajectories *converge* if they all contain a common point. For example, in Figure 6 all execution trajectories with initial points on line $\overline{((0,2),(0,5))}$ converge to the same blocking steady state execution trajectory. This is represented by a shaded polygon in which one edge is the line $\overline{(2,4),(2,6)}$.

3 Solving Progress Graphs

Theorem 1 may be summarized as follows: 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. This section characterizes the set of all such steady state execution trajectories.

The example below illustrates that many steady state execution trajectories are equivalent. By exploiting the definition of equivalent trajectories, the problem of characterizing all possible steady state execution trajectories reduces to finding one member of each equivalence class of steady state execution trajectories.

Example 9 *The subtrajectory whose rays are $\overline{[(3,1),(4,1))}$, $\overline{[(4,1),(7,4))}$, and $\overline{[(7,4),(8,4))}$ in Figure 2 is a steady state execution trajectory. In addition, the subtrajectory consisting of rays $\overline{[(4,4),(7,7))}$, $\overline{[(7,7),(8,7))}$, and $\overline{[(8,7),(9,7))}$ is a steady state execution trajectory. The first, second, and third rays of the first subtrajectory are equivalent to the third, first, and second rays of the second* *subtrajectory, respectively; hence the two steady state execution trajectories are equivalent.*

Steady state execution trajectories may be characterized on the basis of whether they do or do not block. A blocking steady state execution trajectory contains one or more nondiagonal rays, while a nonblocking steady state execution trajectory consists of a single slope one ray. The explicit form of a blocking steady state execution trajectory reveals the sequence and duration of waiting times that each process encounters. This is of interest because some blocking steady state execution trajectories require a process to wait for longer periods than others. In contrast, the explicit form of a nonblocking steady state execution trajectory, representing only running states and hence no blocking, reveals little information. Furthermore, the number of equivalence classes of non-blocking steady states is at most infinite; and by the corollary to Theorem 3 below, the number of blocking steady states is at most twice the number of semaphores. Therefore 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,[3] and
2. reports one member of each equivalence class of blocking steady state execution trajectories that may be reached by any initial condition.

Presented in algorithm A0 below is a solution to (2); (1) is left as an open problem. Before the algorithm is given, a unique way to denote execution trajectories is chosen in the following section.

3.1 Denoting Execution Trajectories

Lemma 2c and Postulate P6 together imply that any execution trajectory that does not reach a dead state consists of a sequence of rays. Such an execution trajectory may be specified by an initial point and a sequence of state transition vectors corresponding to a sequence of rays comprising the execution trajectory.[4]

Example 10 *The execution trajectory with initial point* $(1, 2)$ *in the progress graph of Figure 2 consists of the state transition vector sequence* $(2, 2)$, $(1, 0)$, $(3, 3)$, $(2, 0)$, $(3, 0)$, $(2, 2)$, ...

[3]Note that existence of nondeterministic (respectively, dying) states in a progress graph is only a necessary condition for an execution sequence to exist that contains a nondeterministic (respectively, dead) state. Finding a sufficient condition is an open problem. Also note that Carson and Reynolds' deadlock detection algorithm [6] cannot directly be applied, because a consequence of condition C2 is that the set of all execution trajectories in Carson and Reynolds' progress graphs is a superset of all execution trajectories in progress graphs meeting conditions C4 to C8.

[4]Note that the ray sequence is not unique; for example in Example 9 the rays $\overline{[(7,7),(8,7))}$, and $\overline{[(8,7),(9,7))}$ are equivalent to the single ray $\overline{[(7,7),(9,7))}$.

An execution trajectory may be denoted by a state transition function, denoted f. Such a function could be defined in many ways. In the definition below, f maps a program state to the next state in which a process either completes a P operation that results in blocking or a V operation that results in unblocking.

Definition. *If all points in the execution trajectory rooted at P represent only live or restricted states, then function f is defined such that the set of all end points of non-collinear rays comprising the trajectory is $\{f^i(P) \mid i = 0, 1, \ldots\}$, where $f^0(P) = P$ and $f^n(P)$, for nonnegative n, denotes the n-fold composition of f applied to P.*

Example 11 *In Figure 2, $f^0((4,1)) = (4,1)$, $f^1((4,1)) = (7,4)$, and $f^2((4,1)) = f^1((7,4)) = (9,4)$.*

3.2 Algorithm A0

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 the set of final points of the N constraint line generators be denoted ξ. Figure 7 contains the algorithm. A0 essentially finds the smallest $i \in \{0, 1, \ldots, N-1\}$ satisfying $f^{2i}(X) \equiv X$ for each $X \in \xi$ where a solution exists. A0 does not explain how to compute $\forall X \in \xi$, $f(X)$; this topic is addressed in the companion paper [1].

Algorithm A0 examines exactly N points, namely the final point of each constraint line generator. Set S contains each point already examined that is either in set ξ or is congruent to an element of ξ. Each iteration of the **for each** loop in A0 selects an arbitrary point X in ξ that is not congruent to any point in S and determines if X lies on a steady state execution trajectory. If X does, then X and all other end points of non-collinear rays comprising the steady state trajectory are added to S; otherwise only X is added to S.

Theorem 3 below establishs that examining the N points in set ξ is sufficient to find all equivalence classes of blocking steady state execution trajectories. The intuitive justification is that all rays incident to any point representing a live state in any element of a particular equivalence class of constraint lines are "merged." That is, consider point P on any ray incident to any constraint line generated by some generator with final point X in a progress graph. Then X must be congruent to a point on the execution trajectory with initial point P. By Lemma 6, the execution trajectories of all such points P in the graph contain a subtrajectory equivalent to the execution trajectory with initial point X. Therefore all steady state execution trajectories that block at a particular semaphore contain a point congruent to the final point of the corresponding constraint generator.

Example 12 *The progress graph of Figure 2 contains two constraint line generators, as stated in Example 4. Thus $\xi = \{(4,1), (1,2)\}$. From Table 3, for*

```
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 $\xi$ 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 7: Algorithm A0, which outputs one member of each equivalence class of blocking steady state execution trajectories.

	$f^{2m}(X)$	
X	$m = 1$	$m = N = 2$
(4,1)	*(9,4)*	*unnecessary*
(1,2)	*(4,4)*	*(9,7)*

Table 3: Quantities required by Algorithm A0 for Figure 2.

$X = (4,1)$, $f^2(X) \equiv X$. *However, for* $X = (1,2)$, $\nexists m,\ m \in \{1,2,\ldots,N\}$, $f^{2m}(X) \equiv X$. *Hence a single trajectory is output: point* $X = (4,1)$ *and vector sequence* $(3,3)$, $(2,0)$. *This trajectory is equivalent to the steady state execution trajectories given in Example 9.*

3.3 Correctness of Algorithm A0

The correctness of algorithm A0 is established in Theorems 2 through 5. Appendix D contains the proofs. Note that algorithm A0 terminates if each evaluation of function f terminates because all quantifications are over finite sets.

Theorem 2 *Every trajectory output by algorithm A0 is a blocking steady state execution trajectory.*

Theorem 3 *Any blocking steady state execution trajectory that exists in a progress graph and is contained in an execution trajectory that represents only live or restricted states is equivalent to one of the trajectories output by algorithm A0.*

Corollary to Theorem 3 *The number of equivalence classes of steady state execution trajectories that block in a process graph is at most twice the number of semaphores.*

Proof: By Theorem 3 the number of equivalence classes of blocking steady state execution trajectories cannot exceed the number of constraint line generators. □

Theorem 4 *None of the trajectories output by algorithm A0 are equivalent.*

The following theorem shows that for each trajectory output by A0, either that trajectory or some equivalent trajectory is reachable, meaning it occurs in some execution trajectory that exists in a progress graph.

Theorem 5 *For each trajectory output by A0, either that trajectory or some equivalent trajectory is contained in some execution trajectory rooted at a point either on line* $\overline{[(0,0),(c_0,0))}$ *or on line* $\overline{[(0,0),(0,c_1))}$.

4 Related Work

Related work on progress graphs: As was discussed in section 2, this paper used a formulation of Dijkstra's progress graphs, which were previously used for characterization of deadlocks in multiprocessor systems. Progress graphs are also similar to two-dimensional diagrams used in verification of parallel programs and communication protocols to reason about interleaving of operations. Our application of progress graphs for performance analysis is novel.

Related work on Petri nets: The class of programs analyzed in this paper may be studied using Petri nets [24], queueing networks (e.g., [11, 12]), stochastic processes (e.g., [10, 20]), and stochastic automata ([21, 22]). A survey of these approaches is contained in [2], Chapter 1. Of these, the most closely related work has been done using consistent Petri nets (i.e., nets that return to their initial marking) in which a deterministic firing time is associated with each transition.

Ramamoorthy and Ho [23] address minimum cycle time (MCT) calculation, or the minimum time required for the program to return to its initial state (corresponding to an initial marking of the Petri net). The Ramamoorthy and Ho method takes exponential time and works for both decision-free and persistent

Petri nets, in which a token never enables two or more transitions simultaneously. Methods to compute bounds on the MCT of conservative, general Petri nets are given; finding the exact value is proved NP-complete.

Magott [15] formulates the MCT problem for decision-free and persistent Petri nets as a linear programming problem, and therefore solvable in polynomial time. He gives an improved lower bound and shows that it also applies to non-conservative general Petri nets. Magott [16] gives an $O(N)$ algorithm to compute MCT for nets consisting of a set of N cyclic processes that mutually exclusively share a single resource. Finally, Magott [17] extends his earlier paper [16] by showing that finding MCT in most nets with more complex resource sharing is NP-hard. Also proved are complexity results for systems of process with communication by buffers.

The problem considered in this paper, of processes sharing reusable resources, cannot be represented by decision-free or persistent Petri nets. The example used here, of the Dining Philosophers problem, has been analyzed by Holliday and Vernon [13] assuming deterministic as well as geometrically distributed local state occupancy times. Their Petri net model uses frequency expressions to resolve deterministically which transition fires when a token enables two or more transitions simultaneously. In their model of the Dining Philosophers problem, this expression takes the form of a probability.

Compared to a Petri net approach, solving progress graphs have two advantages:

1. Progress graphs yield the *exact* steady state program execution sequence that the program follows; in contrast the Petri net solutions listed above provide average measures. The Ramamoorthy/Ho and Magott solutions yield the mean cycle time, while the Holliday/Vernon solution yields the long run fractions of time that each process spends in a state.

2. Progress graphs give the execution sequence for *all* possible Petri net markings in a single solution, while existing Petri net solutions require resolving the net for *each* marking.

However, the progress graph solution presented here is limited to two process programs, while Petri net solutions have no such limitation.

Analysis of Resource Scheduling Algorithms: One final category of related work analyzes algorithms that schedule mutually exclusive, reusable resources. Barbosa and Gafni ([4, 5]) consider the average number of processes that may simultaneously use resources, denoted $\gamma_0(\omega)$, when scheduling by arc reversal is done. Each initial assignment of resources to processes (denoted by ω) results in a different value of $\gamma_0(\omega)$. The problem considered, finding which ω maximizes $\gamma_0(\omega)$, is proved to be NP-complete.

This problem corresponds to examining a *set* of multi-dimensional progress graphs and asking what point on a steady state execution trajectory in any of these graphs maximizes $\gamma_0(\omega)$. The particular set used is chosen to represent all values of ω. For example, for two processes there are two values of ω corresponding to two unique points in a single two-dimensional progress graph. The steady state execution trajectories obtained from algorithm A0 yield the ω that maximizes $\gamma_0(\omega)$.

5 Conclusions

The chief qualitative result of the progress graph analysis in this paper is that any execution trajectory of a program fitting our model that does not contain nondeterministic or dead staates reaches a steady-state behavior consisting of a repetition of equivalent steady state execution trajectories. There may be several equivalence classes of steady state execution trajectories; the initial program condition (i.e., the relative times at which processes start execution) determines which class which the program reaches.

There is some similarity between parallel programs and classical dynamic systems, such as electrical circuits with feedback. Dynamic systems may reach a limit cycle behavior, analogous to the repetitions of steady state execution trajectories studied here.

Furthermore, we have some experimental evidence of the similarity. In Chapter 6 of [2], we measured a Dining Philosophers algorithm with between four and 64 processors. For small numbers of processes, the global-state transition sequence is deterministic. Starting at about nine processors, small perturbations occur in the steady-state cycle for short instances of time, after which the program returns to steady state.

Any model requires assumptions that are usually not strictly met by the systems they model. For example, the code segments of condition C8 require an independently derived, constant, finite, and nonzero amount of execution time, exclusive of time spent blocked. Drift among processor clocks and contention for resources (e.g., a bus, a network, or a memory cell) prevent programs from strictly meeting C8. One may ask how accurately the progress graphs of this paper model programs meeting the remaining conditions (C1 to C7). Our experience so far indicates that the model is highly accurate [2, 3].

However, parallel programs are dissimilar from dynamic systems because of discontinuities. If we vary the length of a constraint line generator or the cycle time of a process, the blocking time will change linearly within a certain interval but the sequence of constrain lines intersected does remains constant. However, when variation is large enough to exceed a critical value, the blocking times and sequence of constraints intersected change discontinuously.

Discontinuities complicate design and tuning of a parallel program, because a programmer is unaware of the discontinuity locations. This causes counterintuitive behavior, such as when one process is speeded up, the overall program performance is degraded.

Acknowledgements
This work arose from questions raised by an anonymous referee for the companion paper [1]. The origin of this work was in the authors' dissertation research, which was done under Prof. Ashok Agrawala. Carson and Reynolds' work ([6]) influenced the definition of progress graphs in section 2.

References

[1] M. Abrams and A. K. Agrawala. *Geometric Performance Analysis Of Mutual Exclusion: The Model Solution.* Dept. of Computer Science, Virginia Tech, TR-90-59, Dec. 1990.

[2] M. Abrams. *Performance analysis of unconditionally synchronizing distributed computer programs using the geometric concurrency model.* Ph.D. Dissertation, Dept. of Computer Science, University of Maryland, TR-1696, Aug. 1986.

[3] M. Abrams and A. K. Agrawala. Automated measurement and prediction of unconditionally synchronizing distributed algorithms. *Proceedings of the 7th Inter. Conf. on Distributed Computer Systems.* Berlin, Sept. 1987.

[4] V. C. Barbosa and E. M. Gafni. Concurrency in heavily loaded neighborhood-constrained systems. *Proc. 7th Int. Conf. on Distributed Computer Systems.*, Berlin Sept. 1987.

[5] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods.* Englewood Cliffs: Prentice Hall. 1989. Section 8.3.

[6] 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.

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

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

[9] E. W. Dijkstra. *Cooperating sequential processes.* In *Programming Languages,* F. Genuys, Ed. Academic Press, New York, 1968, 67-68.

[10] E. Gelenbe, A. Lichnewsky, and A. Staphylopatis. Experience with the parallel solution of partial differential equations on a distributed computing system. *IEEE Trans. Comput. C-31,* 12, (Dec. 1982), 1157-1164.

[11] P. Heidelberger and K. S. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *IEEE Trans. Comp. C-31,* 11, (Nov. 1982), 1099-1109.

[12] P. Heidelberger and K. S. Trivedi. Analytic queueing models for programs with internal concurrency. *IEEE Trans. Comp. C-32,* 1, (Jan. 1983), 73-82.

[13] M. A. Holliday and M. K. Vernon. A generalized timed petri net model for performance analysis. *Proc. Int. Workshop on Timed Petri Nets.* July 1985.

[14] 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.

[15] J. Magott. Performance evaluation of concurrent systems using Petri nets. *Information Processing Letters 18,* Jan. 1984, 7-13.

[16] J. Magott. Performance evaluation of systems of cyclic sequential processes with mutual exclusion using Petri nets. *Information Processing Letters 21,* Nov. 1985, 229-232.

[17] J. Magott. Performance Evaluation of systems of cyclic sequential processes with mutual exclusion and communication by buffers using timed Petri nets. *Proc. Workshop on Timed Petri Nets.* Madison Wisconsin, IEEE Press. 1987, 146-153.

[18] A. D. Malony and D. A. Reed, Visualizing parallel computer system performance. In M. Simmons, R. Koskela, and I. Bucher, eds., *Instrumentation for Future Parallel Computer Systems,* Addison-Wesley. 1989, pp. 59-90.

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

[20] B. Plateau and A. Staphylopatis. Modeling of the parallel resolution of a numerical problem on a locally distributed computing system. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Performance Evaluation Review 11,* 4, (Winter 1982), 108-117.

[21] B. Plateau. De l'évaluation du parallélisme and de la synchronisation. Thése d'état, Universitéde Paris-Sud, 91405 Orsay, France, Nov. 1985.

[22] B. Plateau. On the stochastic structure of parallelism and synchronization models for distributed algorithms. In *Proceedings of the SIGMETRICS 1985,* Austin, Aug. 1985, 147-154.

[23] C. V. Ramamoorthy, and G. S. Ho. Performance evaluation of asynchronous concurrent systems using petri nets. *IEEE Trans. on Software Eng. SE-6,* 5, Sept. 1980, 440-448.

[24] M. Vernon, J. Zahorjan, and E. D. Lazowska. *A comparison of performance petri nets and queueing network models.* TR-669, University of Wisconsin, Sept. 1986.

[25] 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.

A Characterizing Graph Points

The following lemma formally establishes the claims about the geometric manifestation of dying, live, free, and restricted states made in section 2.4.

For any point P in a progress graph, let $C_N(P)$, $C_D(P)$, $C_L(P)$, $C_F(P)$, and $C_R(P)$ each denote a logical predicate whose value is true if and only if P represents a nondeterministic, dying, live, free, and restricted state, respectively. The following lemma gives a rule to decide which of these five mutually exclusive and exhaustive predicates holds for an arbitrary point in a progress graph.

Lemma 8

8a. $C_N(P)$ *if and only if* P *is the initial point of a constraint line.*

8b. $C_D(P)$ *is true if and only if* $\neg C_N(P)$ *and* P *lies on a constraint line and there exists a point* P^I *at which the constraint line intersects another constraint line such that* $P \leq P^I$.

8c. $C_L(P)$ *if and only if* P *lies on a constraint line and* $\neg C_N(P) \wedge \neg C_D(P)$.

8d. $C_F(P)$ *if and only if* P *does not lie on a constraint line and a diagonal ray rooted at* P *does not intersect a constraint line.*

8e. $C_R(P)$ *if and only if* P *does not lie on a constraint line and* $\neg C_F(P)$.

Proof:

8a: Equivalent to Lemma 4c.

8b:

P is a blocked state
, by definition of dying state

P lies on a constraint line
, by Lemma 4b

Execution sequence with initial state P contains a dead state; let P^D denote the point representing the dead state
, by definition dying state

Execution trajectory rooted at P is a nondiagonal ray with final point P^D
, by last deduction, P6, and Lemma 2b

Execution trajectory rooted at P is contained in constraint line containing P
, Lemma 5a

$P^D = P^I$
, Lemma 4a

8c: Follows from Lemma 4b and definition of live state.

8d:

Execution sequence with initial state represented by P contains no blocked states
, by definition free state

No point on execution trajectory rooted at P lies on a constraint line
, by last deduction and Lemma 4b

Execution trajectory rooted at P is a diagonal ray that never intersects a constraint line
, Lemma 5b

8e: Follows from Lemmas 4b and definition of restricted state. □

B Proof of Lemma 6

The proof of Lemma 6 is simplified by the following two lemmas. The first establishes three intuitive properties: that any two congruent points in a progress graph either (1) both lie on or lie off a constraint line, (2) either both do or do not lie at the intersection of two constraint lines, and (3) that given any parallelogram in a progress graph with one vertex on a constraint line and two other congruent vertices, the remaining vertex lies on a constraint line. The second lemma demonstrates that whichever of the predicates introduced in Appendix A holds for a point P also holds for any point congruent to P.

Lemma 9

9a. If $P^1 \equiv P^2$ and P^1 lies on a constraint line, then P^2 lies on an equivalent constraint line.

9b. Given two congruent points P^1 and P^2, P^1 is a point of intersection of two constraint lines if and only if P^2 is a point of intersection of two constraint lines.

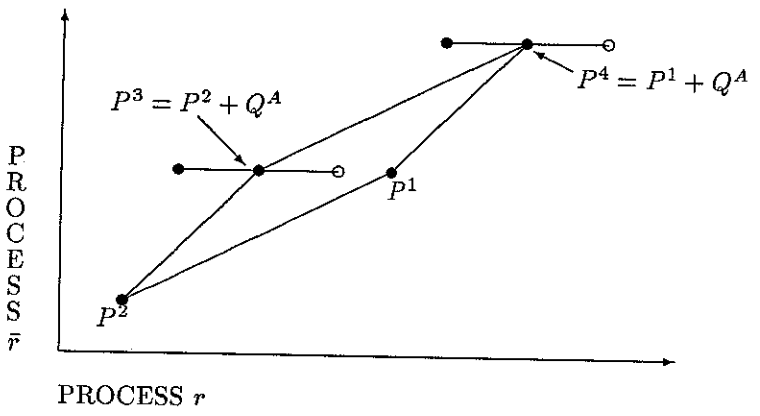


Figure 8: Illustration of Lemma 9c, containing two equivalent horizontal constraint lines.

9c. For any parallelogram $P^1P^2P^3P^4$, if $P^1 \equiv P^2$ and P^3 lies on a constraint line, then P^4 lies on an equivalent constraint line.

Proof:

9a and 9b: Follows from the definition of a constraint line.

9c:

$P^3 = P^2 + Q^A$ and $P^4 = P^1 + Q^A$ for some vector Q^A (see Figure 8)
, by hypothesis that $P^1P^2P^3P^4$ is a parallelogram

$P^3 \equiv P^4$
, by hypothesis $P^1 \equiv P^2$ and previous deduction

P^4 lies on a constraint line
, by Lemma 9a. □

Lemma 10

10a. If $P^1 \equiv P^2 \wedge C_N(P^1)$ then $C_N(P^2)$.

10b. If $P^1 \equiv P^2 \wedge C_D(P^1)$ then $C_D(P^2)$.

10c. If $P^1 \equiv P^2 \wedge C_L(P^1)$ then $C_L(P^2)$.

10d. If $P^1 \equiv P^2 \wedge C_R(P^1)$ then $C_R(P^2)$.

10e. If $P^1 \equiv P^2 \wedge C_F(P^1)$ then $C_F(P^2)$.

Proof:

10a:

P^1 lies on some constraint line $\overline{[P^1, X)}$
, by Lemma 8a

There exists a constraint line equivalent to $\overline{[P^1, X)}$ with initial point P^2.
, by definition of constraint line and hypothesis $P^1 \equiv P^2$

10b:

P^1 lies on some constraint line; let L_1 denote the line
, by Lemma 8b

There exists a point P^I at which L_1 intersects another constraint line such that $P^1 \leq P^I$
, by definition of dying state

P^2 also lies on a constraint line, denoted by L_2, such that $L_2 \equiv L_1$
, by Lemma 9a

There exists a point $\hat{P}^I$ at which L_2 intersects another constraint line such that $P^I \equiv \hat{P}^I$
, by the definition of a constraint line

$P^2 \leq \hat{P}^I$
, by last two deductions

$C_D(P^2)$
, by applying Lemma 8b

10c:

$\neg C_N(P^2) \wedge \neg C_D(P^2)$ and P^2 lies on a constraint line
, by Lemmas 8c, 9a, 10a, and 10b

$C_L(P^2)$
, Lemma 8c

10d:

A diagonal ray rooted at P^1 must intersect a constraint line; let $P^1 + Q^A$ be the intersection point
, by hypothesis $C_R(P^1)$

$P^2 + Q^A$ must lie on a constraint line (see Figure 8)
, by Lemma 9c

$C_R(P^2)$
, because $\overline{P^2, P^2 + Q^A}$ has slope one

10e:

P^2 does not lie on a constraint line and $\neg C_R(P^2)$
, by Lemmas 8d, 9a, and 10d

$C_F(P^2)$

, the definition of a free state. □

Lemma 6 [from section 2.5] *Given any two execution sequences containing only deterministic states with initial states represented by points P and P', if $P \equiv P'$ then the execution trajectories rooted at P and P' are equivalent.*

Proof: The proof consists of demonstrating that the i-th ($i = 1, 2, \ldots$) noncollinear ray in the execution trajectories rooted at P and P' are equivalent by induction on the value of i.

The proof for all i is the same. Let P^1 and P^2 be the initial points of the i-th noncollinear ray in the execution trajectories rooted at P and P', respectively. If $i = 1$, $P^1 \equiv P^2$ by hypothesis . If $i > 1$, $P^1 \equiv P^2$ by the inductive hypothesis that the $(i-1)$-th noncollinear rays are equivalent. By hypothesis, $\neg C_N(P^1)$; therefore there are four mutually exclusive and exhaustive cases:

$C_D(P^1)$:

$C_D(P^2)$

, because $C_D(P^1)$ and by Lemma 10b

Final point of initial ray in execution trajectory rooted at P^1 (respectively, P^2) is some point P^{I_1} (respectively, P^{I_2}) such that $\overline{[P^1P^{I_1})}$ (respectively, $\overline{[P^2P^{I_2})}$) lies on a constraint line, does not cross another constraint line, and P^{I_1} (respectively, P^{I_2}) is a point of intersection with another constraint line

, by definition of dying state and Lemma 5a

$P^{I_1} \equiv P^{I_2}$

, by Lemma 9b

i-th noncollinear rays in execution trajectories rooted at P and P' are equivalent

, combine last two deductions

$C_L(P)$:

$C_L(P^2)$

, because $C_L(P^1)$ and by Lemma 10c

Final point of initial ray in execution trajectory rooted at P^1 (respectively, P^2) is final point X^1 (respectively, X^2) of some constraint line, such that $\overline{[P^1X^1)}$ (respectively, $\overline{[P^2X^2)}$) lies on a constraint line.

, by definition of dying state and Lemma 5a

$X^1 \equiv X^2$

, by Lemma 9a

i-th noncollinear rays in execution trajectories rooted at P and P' are equivalent

, combine last two deductions

$C_R(P)$:

$C_R(P^2)$

, because $C_R(P^1)$ and by Lemma 10d

Final point of initial ray in execution trajectory rooted at P^1 (respectively, P^2) lies on first constraint line intersected by diagonal ray rooted at P^1 (respectively, P^2); let $\hat{P}^1$ (respectively, $\hat{P}^2$) denote the intersection point

, by definition of restricted state

Rays $\overline{[P^1, \hat{P}^1)}$ and $\overline{[P^2, \hat{P}^2)}$ have equal length

, because assuming unequal lengths and applying Lemma 9a to $\hat{P}^1$ and $\hat{P}^2$ contradicts the assertion that $\hat{P}^1$ and $\hat{P}^2$ represent first constraint lines intersected

$P^1 \equiv P^2$

, by Lemma 9c using parallelogram $P^1\hat{P}^1\hat{P}^2P^2$

i-th noncollinear rays in execution trajectories rooted at P and P' are equivalent

, combine second and fourth deductions

$C_F(P)$: Follows from Lemmas 10e and 5b. □

C Proof of Theorem 1

Lemma 11 will simplify the proof of Theorem 1.

Lemma 11 *Consider any progress graph representing a finite number of semaphores and any execution sequence containing only deterministic states. If the sequence contains no dead state then the corresponding execution trajectory contains a steady state execution trajectory.*

Proof: Let T denote the execution trajectory. Each point in T represents a free, restricted, or live state, by the hypothesis that execution trajectory contains no dead or nondeterministic state. Consider two cases: (a) T intersects a finite number of constraint lines and (b) the complement. These correspond to (a) $\exists P$, $P \in T$, $C_F(P)$ and (b) $\nexists P$, $P \in T$, $C_F(P)$.

case (a): Let P' be the smallest point on the execution trajectory satisfying $C_F(P')$.

Execution trajectory contains $P' + (c_0c_1, c_0c_1)$

, by Lemma 2a

$P' \equiv P' + (c_0c_1, c_0c_1)$

, definition of congruence

Execution trajectory contains steady state execution trajectory

, by last deduction

case (b):

All points in execution trajectory represent restricted or live states
, by second deduction and the definition of case (b)

Execution trajectory intersects infinite number of constraint lines
, by last deduction

Execution trajectory intersects an infinite number of constraint lines belonging to the same equivalence class
, by last deduction and because number of semaphores is finite

Execution trajectory contains the final point of each constraint line it intersects
, since all constraint line points represent live states and by Lemma 5a

There exist two congruent points in the execution trajectory
, by last two deductions

Execution trajectory contains steady state execution trajectory
, by last deduction □

Theorem 1 *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.*

Proof of Theorem 1:

Only if part:

Execution trajectory contains an infinite number of equivalent steady state execution trajectories; let P and P' denote the initial and final points of one such trajectory
, by hypothesis

Transition from state represented by initial to final point of any steady state execution trajectory corresponds to displacement of $P' - P$ in the Cartesian plane
, using last deduction

Execution trajectory has no final point
, by last deduction and $P' - P > 0$

Execution sequence contains no dead state
, by P6

If part: The inductive proof below demonstrates that the execution trajectory contains at least i equivalent steady state execution trajectories, for all $i > 0$. The base case ($i = 0$) is established by Lemma 11. The case of $i > 1$ follows:

The execution trajectory contains at least $i-1$ steady state execution trajectories
    , statement of the inductive hypothesis

Let P and P' denote the initial and final points of the $(i-1)$-th steady state execution trajectory; then $P \equiv P'$
    , by definition of steady state execution trajectory

Execution trajectories rooted at P and P' are equivalent
    , by Lemma 6

There exists a point P'' on the execution trajectory such that $P'' > P' \wedge P'' \equiv P$
    , by last deduction

There exist at least i equivalent steady state execution trajectories
    , combine last deduction with inductive hypothesis □

D Proofs of Correctness of Algorithm A0

All execution trajectories considered in this appendix are presumed to consist of points representing either live or restricted states. (This excludes dead, nondeterministic, and free states, which never arise in a blocking steady state execution trajectory.)

D.1 Properties About f

Stated and proved below are several properties about function f. Recall that ξ denotes the set of final points of all constraint line generators, and that set ξ contains N elements. Let X denote an element of ξ. Let i and j denote integers.

F1: If $\forall j, j \leq i, f^{2j}(X)$ and $f^{2j+1}(X)$ are defined, then $f^{2i}(X) \in \xi$ and $C_L(f^{2i+1}(X))$.

F2: Let P be any point congruent to X that lies on the execution trajectory rooted at X. If $\exists i, i > 0, X \equiv f^{2i}(X)$, then $\exists j, j \geq 0, P = f^{2j}(X)$.

F3: $\exists i, i > 0, X \equiv f^{2i}(X) \Rightarrow \exists j, j \in \{1, 2, \ldots, N\}, f^{2j}(X) \equiv X$.

Proof of F1: Follows by induction on i. Note that all running states are restricted, and all blocked states are live.

If $i = 0$ then $f^{2i}(X) \in \xi$
    , from premise that $X \in \xi$

If $i > 0$ then $f^{2i-1}(X)$ lies on a constraint line
    , by Lemma 4b applied to the inductive hypothesis $C_L(f^{2i-1}(X))$

If $i > 0$ then $f^{2i}(X) \in \xi$
    , by Lemma 5a applied to last deduction

$\forall i, i \geq 0, f^{2i}(X) \in \xi$
, combine first and last deductions

$f^{2i+1}(X)$ lies on a constraint line
, by contrapositive of Lemma 4b, Lemma 5a, and last deduction

$C_L(f^{2i+1}(X))$
, by Lemma 4b □

Proof of F2:

$\exists j, j \geq 0, P = f^j(X)$, by Theorem 1

$\exists j, j \geq 0, P = f^{2j}(X)$, combine last deduction with F1 □

Proof of F3:

$\forall i, i \geq 0, f^{2i}(X) \in \xi$
, by F1

$\exists i, i > 0, X \equiv f^{2i}(X)$
, hypothesis

$\exists i, i \in \{1, 2, \ldots, N\}, f^{2i}(X) \equiv X$
, by last two deductions and because ξ contains N elements □

D.2 Proofs of Theorems

Theorem 2 *Every trajectory output by algorithm A0 is a blocking steady state execution trajectory.*

Proof: By definition, a blocking steady state execution trajectory must:

1. contain a point that lies on a constraint line,
2. be a subtrajectory of some execution trajectory in the progress graph, and
3. contain exactly two congruent points, namely the initial and final points.

Let X denote the initial point of a trajectory output by A0.

Proof of 1:

$f(X)$ is defined
, because A0 outputs a trajectory with initial point X

$C_L(f(X))$
, by F1 and last deduction

$f(X)$ lies on a constraint line
, by definition of live state and last deduction

Proof of 2: Follows from definition of f.

Proof of 3:

Initial, final points of trajectories output are congruent
, because $X \equiv f^{2n}(X)$ in algorithm A0

Trajectory contains a subtrajectory with initial point P and final point P' that is a steady state execution trajectory
, by last deduction

$X \leq P < P' \leq f^{2n}(X)$
, by last deduction

$P' = f^{2n}(X)$
, by Theorem 1 and since n is smallest natural satisfying $f^{2n}(X) \equiv X$

$P = X$
, by F2

Every trajectory output by A0 contains exactly two congruent points
, because $P = X$, $P' = f^{2n}(X)$ □

Lemma 12 *Consider a steady state execution trajectory S representing only deterministic states. There exists a steady state execution trajectory rooted at any point congruent to a point on S.*

Proof: Let the initial and final points of S be P and P', respectively. Let Y be any point on S. The proof first demonstrates that there exists a steady state execution trajectory rooted at Y.

Steady state execution trajectory rooted at Y exists; denote its final point by Y'
, by Theorem 1 and Lemma 6

Subtrajectory with initial and final points P and Y is equivalent to subtrajectory with initial and final points P' and Y', respectively
, because $P \equiv P'$, $Y \equiv Y'$, and by Lemma 6

S is equivalent to steady state execution trajectory rooted at Y
, by last deduction and because trajectory with initial point Y and final point P' is a subtrajectory of both S and the trajectory rooted at Y

S is equivalent to steady state execution trajectory rooted at any point congruent to Y
, by last deduction and Lemma 6

□

Theorem 3 *Any blocking steady state execution trajectory that exists in a progress graph and is contained in an execution trajectory that represents only live or restricted states is equivalent to one of the trajectories output by algorithm A0.*

Proof: Consider some blocking steady state execution trajectory, denoted S.

S contains ray with its initial point on a constraint line
, because steady state execution trajectory blocks

Final point of ray in last deduction is final point of some constraint line
, because steady state execution trajectory does not contain dead points, and by Lemma 5a

$\exists X \in \xi$, where X is congruent to final point in last deduction
, by definition of constraint line

Steady state execution trajectory with initial point X exists and is equivalent to S; denote it by S''
, by Lemma 12

$\exists m, m \in \{1, 2, \ldots, N\}, f^{2m}(X) \equiv X$
, by F3 and last deduction

A0 outputs S''
, by last deduction and Theorem 2 □

Lemma 13 *Given a point, all steady state execution trajectories containing that point are equivalent.*

Proof: Follows from Lemma 12. □

Theorem 4 *None of the trajectories output by algorithm A0 are equivalent.*

Proof: Consider any two trajectories output by A0. Let X^1 (respectively, X^2) be the initial point of the first (respectively, second) of these trajectories. Let S^1 denote the set of all end points of non-collinear rays comprising the first of these trajectories. Let n be the smallest natural satisfying $f^{2n}(X^2) \equiv X^2$. Proving that there exists a single point (namely, X^2) on the second trajectory that is not congruent to any point on the first trajectory is sufficient to establish Theorem 4.

$f^{2n}(X^2)$ must be end point of two non-collinear rays in any steady state execution trajectory on which it lies
, by Lemma 13

X^2 must be end point of two non-collinear rays in any steady state execution trajectory on which it lies
, because $X^2 \equiv f^{2n}(X^2)$

$S^1 \subseteq S$ when A0 applies if test to X^2
, S is set to $S \cup S^1$ after point X^1 is output in A0

$\nexists x, x \in S, x \equiv X^2$
, if X^2 is output then if test in A0 was true

$\nexists x, x \in S^1, x \equiv X^2$
, combine last two deductions

Theorem 5 *For each trajectory output by A0, either that trajectory or some equivalent trajectory is contained in some execution trajectory rooted at a point either on line* $\overline{[(0,0),(c_0,0))}$ *or on line* $\overline{[(0,0),(0,c_1))}$.

Proof: Let S denote some trajectory output by A0. Let X denote the initial point of S. Let n be the smallest natural satisfying $f^{2n}(X) \equiv X$. Let i_0 and i_1 denote naturals.

$f^{2n}(X)$ lies on S
, by definition of f

$\exists i_0, \exists i_1, f^{2n}(X) = X + (i_0 c_0, i_1 c_1)$
, because $f^{2n}(X^2) \equiv X$, where $n > 0$

$\exists P, P \in S,\ \exists r, r \in \{0,1\},\ X \leq P < f^{2n}(X)\ \wedge\ P_r \bmod c_r = 0$
, by last deduction and because $X \in S \wedge f^{2n}(X) \in S$

$P \equiv (P_0 \bmod c_0, P_1 \bmod c_1)$
, by definition of congruent points

Execution trajectory rooted at $(P_0 \bmod c_0, P_1 \bmod c_1)$ contains a point congruent to $f^{2n}(X)$
, by Lemma 6 and last deduction

Execution trajectory rooted at $(P_0 \bmod c_0, P_1 \bmod c_1)$ contains a trajectory equivalent to S
, by Lemma 12 and last two deductions $\square$