An Example of Deriving Performance Properties from a Visual Representation of Program Execution

Marc Abrams*
Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, VA 24061-0106
abrams@cs.vt.edu

June 19, 1995

Abstract

This paper demonstrates that one can use a visual representation of the execution of a program to analyze performance properties. The method starts with a visualization of the progress of processes, and then applies geometric properties of the visualization to analyze program execution behavior. The visualization is based on progress graphs from the literature, which map the progress of each process to one Cartesian graph axis. Line segments represent interprocess synchronization. A directed, continuous path that does not cross a segment represents a particular execution of a program, and can be found using computational geometric algorithms. A special case is also considered in which programs display periodic behavior. The relation of the program class studied to Petri nets is also examined.

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

Index Terms – parallel computation, performance evaluation, computational geometry, ray shooting, timed progress graphs, mutual exclusion, Petri nets

1 Introduction

Systems to visualize the execution history collected from a program can help explain what happens during program execution (e.g., [4, 10, 13, 15]). Visualization systems can lend insight into why a

*Supported in part by National Science Foundation grant NCR-9211342.

performance measure, such as program execution time or mean waiting time for a resource, has a particular value. The answer to "why" helps identify how to change the program to improve the measure.

But visualization has a use beyond providing images on a computer monitor: one can formally deduce properties about program execution from a visualization. This was first demonstrated by Roman and Cox [19], who deduced *correctness* properties. Roman and Cox illustrate that safety properties (properties that hold in all computation states, such as invariants) and progress properties (properties that hold in a particular program state) can be verified by defining a mapping of program states to a visual representation, and then observing whether the sequence of visual images corresponding to an execution sequence satisfies a desired property. They discuss the ability "to render invariant properties of the program state as stable visual patterns and to render progress properties as evolving visual patterns."

This paper provides a second example, deducing the following *performance* properties from a visualization of a class of concurrent programs:

Q1: the sequence of synchronization points where the program blocks,

Q2: the blocking duration at each synchronization point, and

Q3: the duration of concurrent execution between synchronization points.

The class consists of program that meets the following assumptions: A program contains two processes (A1). A process executes on a dedicated processor (A2). A process only blocks at synchronization operations (A3). A synchronization operation in a process A is defined in terms of a code segment C in the other process B: when A reaches the synchronization operation it will block if and only if B is executing C. Each process is represented as a branchless directed graph in which vertices represent code segments and edges always represent precedence relations and may represent synchronization operations (A4). A process will optionally block when moving from one vertex to another if the edge corresponds to a synchronization operation. The execution time of the code segment corresponding to each vertex is an independent constant, exclusive of time spent blocked (A5). Each process has an initial vertex in which the process starts execution and, optionally, a final vertex in which the process terminates (A6). A process without a final vertex never terminates.

We argue next that the assumptions are reasonable. Consider the two process assumption (A1). The initial solutions to some classic parallel programming problems — such as shared memory mutual exclusion algorithms — were initially solved only for two processes. And one important performance evaluation tool — queueing networks — started only with the ability to solve just one kind of queue (M/M/1) in isolation. In principle the analysis presented here can be extended from two to an arbitrary number of processes; see the conclusions (§7) for a discussion. Regarding the constant time assumption (A5), Adve and Vernon [5] conclude based on seven parallel applications that "it appears reasonable to ignore the variability in execution times when estimating synchronization delays," and that an exponential task time assumption could actually lead to *more* severe errors than a constant-time assumption. As for relaxing assumption A2, multiprogramming has been modeled in past work that uses the visualization underlying our work, and can be incorporated into the model presented here. The branchless assumption (A4) is not as restrictive as it might first appear, because loops whose number of iterations is known can be unrolled to obtain a branchless graph. Our analysis includes non-terminating programs (A6), because certain long running programs repeatedly execute the same code, such as simulations and reactive programs (programs that react to external stimuli on an ongoing basis, such as operating systems).

Our solution method represents program execution by a Timed Progress Graph (TPG). To our knowledge, TPGs were originally used in Operations Research to find minimum length schedules for two jobs that share a set of machines [20, pp. 262-263]. However, a good and not necessarily minimal schedule was found "by eye" rather than by a formal method. Later, a simplified form of TPG, called an *untimed* progress graph (UTP), in which each delay in the timed transition diagram is unity, was used to analyze deadlocks.[1] Carson and Reynolds [7] define a UPG as "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." Kung, Lipski, Papadimitriou, Soisalon-Soininen, Yannakakis, and Wood [11, 17, 22, 24] used UPGs to detect deadlocks in lock-based transaction systems. More recently Carson and Reynolds [7] used UPGs to prove liveness properties in programs with an arbitrary number of processes containing P and V operations on semaphores that are unconditionally executed. Our use of TPGs to analyze program performance properties is novel.

TPGs map the progress of each process to one Cartesian graph axis. Line segments represent interprocess synchronization. A directed, continuous path that does not cross a segment represents a particular execution of a program. This path may be found by computational geometric algorithms to compute line segment intersections and for ray shooting.

The remainder of the paper is organized as follows. First we define TPGs and illustrate their use on sample programs in §2. The representation of an execution of a program is then formalized in §3. §4 presents an algorithm to solve the problems listed earlier. We then consider, in §5, a special class of programs that displays periodic behavior and present an algorithm to solve for the set of all possible periodic state sequences that can arise during any execution of a program. §6 relates TPGs to Petri Nets. Finally §7 contains conclusions.

2 Timed Progress Graphs

We first illustrate our approach to solving for quantities Q1 to Q3 from §1 on three programs.

[1]Apparently TPGs were reinvented in the Computer Science field and attributed to Dijkstra [8].

```
process producer
  X: integer;
  while (TRUE) do
    read X from disk;
    send X to consumer;
  do
```

```
process consumer
  Y: integer;
  while (TRUE) do
    receive X from consumer;
    write X to disk;
  od
```

Figure 1: Non-terminating producer/consumer program with one buffer.

2.1 Program 1: Non-terminating Producer/Consumer

Figure 1 contains a producer/consumer program that use one message buffer for interprocess communication. Thus when the producer sends a message it fills the single buffer and thus must wait until the consumer executes `receive` before performing the next `send`. Figure 2 contains an equivalent graph representation of the program and satisfies assumptions A1 to A6. The deterministic execution time of each graph vertex, excluding time spent blocked, is shown in square braces. Edge conditions B1 and B2 represent blocking that may occur during a `send` and `receive`, respectively. (For simplicity of presentation, we assume that a process blocks at the *end* of a send or receive operation; this condition is straightforward to relax.) Recall from A3 that specifying a synchronization operation requires identifying a code segment C for which a process may block. Therefore in Fig. 2 conditions B1 and B2 are the following: (B1) The producer never blocks when it performs the first `send`; furthermore the producer blocks when it performs the n-th `send` ($n > 1$) if the consumer has not yet executed the $n-1$-st `receive`. (B2) The consumer cannot execute the n-th ($n \geq 0$) `receive` until the producer has executed the n-th `send`.

Recall that a TPG maps the progress of each process to one Cartesian graph axis; the result for Fig. 1 is shown in Fig. 3. The sequence of graph vertices that each process passes through is mapped to a sequence of intervals (denoted by grey lines in Fig. 3(a)) along the corresponding axis. Interval widths correspond to execution time of vertices. Synchronization between processes is represented

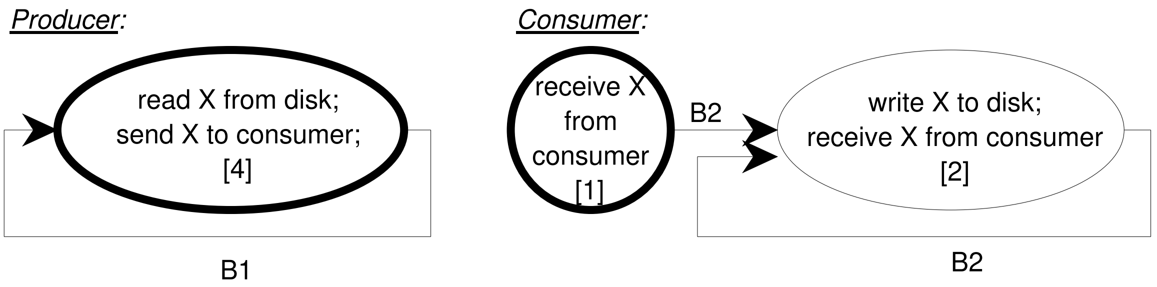

Figure 2: Graph model corresponding to Fig. 1. Numbers in square brackets refer to time spent in each code segment. Labels on each edges denote conditions under which blocking occurs. Thick circles represent initial vertices; there are no final vertices.

in Fig. 3(b) by horizontal and vertical line segments in the plane placed at appropriate boundaries of vertex intervals to prohibit a transition into a vertex. The line segments, called *constraint lines*, are closed at the end point closest to the origin and open at the other end point. Each line segment corresponds to a blocking condition in the graph. Consider B1 in Fig. 2. From Fig. 3(a), the producer makes the first transition from the `read...; send...;` vertex back to itself at time 4, the second at time 8, and the n-th at time $4n$ (in the absence of blocking). Similarly, the consumer completes its first `receive` at time 1, its second at time 3, its third at time 5, and its n-th at time $2n-1$. Therefore, by B1, the producer will block at time $4n$ (for $n > 1$) if the consumer has completed its $n-1$-st receive, corresponding to points that exceed coordinate $2(n-1)-1$ on the axis representing the consumer. Thus the producer will block in any point (x, y) meeting the following two conditions: $x = 4n$ and $y \in [0, 2n-3)$, for $n > 1$, corresponding to the vertical lines in Fig. 3(b). Similarly, by B2, the consumer will block when it tries to complete the n-th `receive` (at time $2n-1$) if the producer has not yet completed the n-th `send` (at time $4n$). Thus the consumer will block in any point (x, y) meeting the following two conditions: $y = 2n-1$ and $x \in [0, 4n)$ for $n \geq 1$, corresponding to the horizontal lines in Fig. 3(b).

Given an initial point, execution of a program is represented by a point or a directed path in the

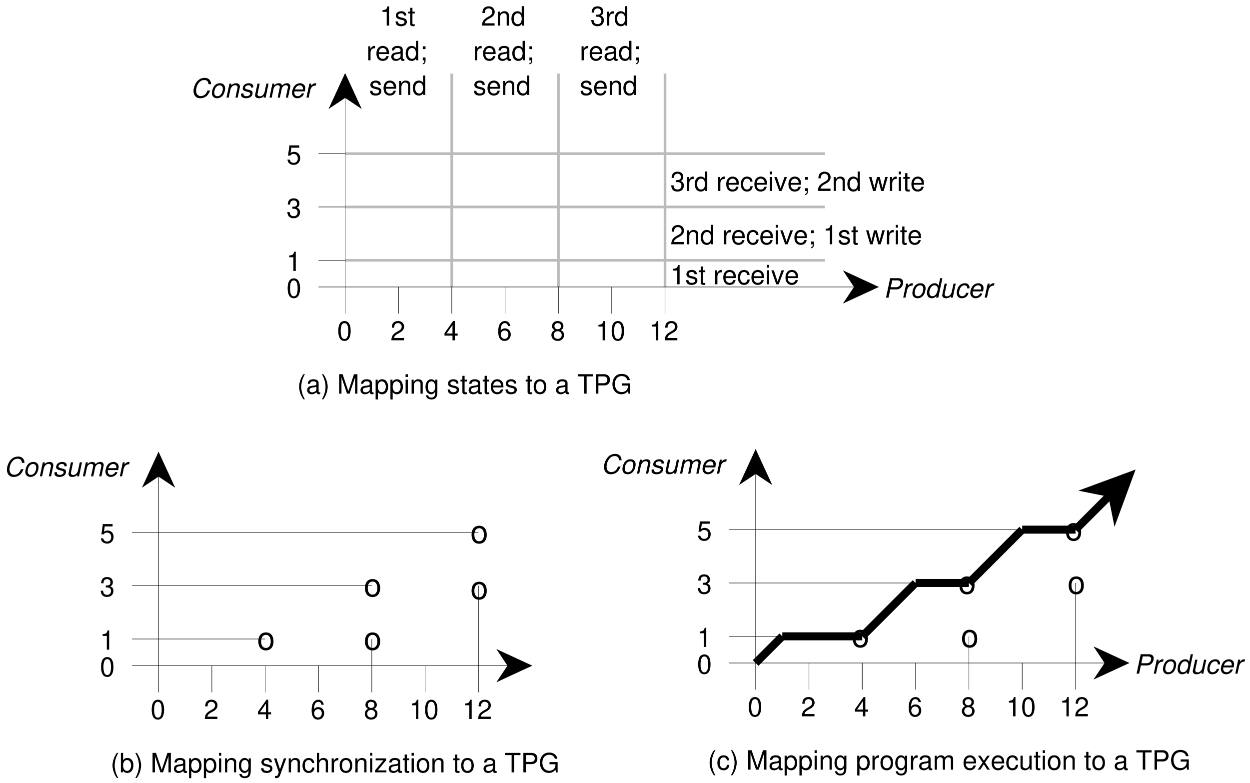

Figure 3: Representation of one-buffer producer/consumer program (Fig. 2) by a TPG.

plane, called a *timed execution trajectory* (TET). The initial point is (0,0) if both processes start simultaneously. A TET is a point only if the initial point represents a deadlocked state. Otherwise the TET is a path and consists of a possibly infinite sequence of rays that have slope 0, 1, or ∞. The ray has slope 0 (respectively, ∞) when the process corresponding to the vertical (respectively, horizontal) axis is blocked, and slope 1 when both processes are running concurrently. Because constraint lines represent forbidden state transitions, a TET cannot cross a constraint line. A finite portion of the TET for the one buffer producer/consumer problem is the thick directed path in Fig. 3(c). The general method of constructing a TET is given below.

TET construction rule: Given a TPG, a TET is constructed recursively as follows. Let upper case letters with optional superscripts (i.e., G, G', G^I) denote graph points. Let a point G denote the ordered pair of coordinates (G_0, G_1). For any two points G and G', $(G < G') \stackrel{def}{=} ((G_0 + G_1) < (G'_0 + G'_1))$. $\overline{[G', G'')}$ denotes a line segment that is closed at G', open at G'', and satisfies $G' < G''$.

Rule I: If G lies on some constraint line $\overline{[G', G'')}$, then there either (1) will or (2) will not exist a point G^I distinct from G' at which $\overline{[G', G'')}$ intersects another constraint line instance such that $G \leq G^I$. In case (1), the TET rooted at G is a ray with initial point G and final point G^I. In case (2), the TET rooted at G is a ray with initial point G and final point G'', followed by a TET rooted at G''.

Rule II: If G lies off a constraint line, then a slope one ray rooted at G either (1) will or (2) will not intersect a constraint line. In case (1), the TET rooted at G is a slope one ray with initial point G and final point G', where G' is the only point on the ray that lies on a constraint line, followed by the set of TETs rooted at G'. In case (2), the TET rooted at G is an infinite length, slope one ray rooted at G.

Example 1 Consider the TET portion in Fig. 3(c). Point $G = (0, 0)$ lies off a constraint line, and follows Rule II, case (1). Thus the TET is ray $\overline{[(0,0),(1,1))}$ followed by the TET rooted at (1,1). Next, $G = (1, 1)$ lies on horizontal constraint line $\overline{[(0,1),(4,1))}$. By Rule I case (2), the second TET ray must be $\overline{[(1,1),(4,1))}$. Next Rule II case (1) again applies and the third TET ray must be $\overline{[G, G')}$, where $G = (4, 1)$ and $G' = (6, 3)$. This process is continued forever to yield an infinite length TET. $\square$

The TET in Fig. 3(c) yields the three performance properties Q1 to Q3 sought in the opening paragraphs of this paper. The sequence of synchronization points, Q1, corresponds to the sequence of horizontal and vertical rays arising in a TET. The TET of Fig. 3(c) contains only horizontal rays, and thus the program never blocks when the producer executes a `send`. The blocking duration at each `receive`, Q2, is the length of each horizontal ray in the TET in Fig. 3(c): 3 time units for the first `receive` (because the first horizontal ray is $\overline{[(1,1),(4,1))}$), and 2 time units for each subsequence `receive`. Finally, the duration of concurrent execution between synchronization points, Q3, is the length of the perpendicular projection of each diagonal ray in the TET on either axis in Fig. 3(c): 1 time unit from when the program starts until a process first blocks (because the first diagonal ray is $\overline{[(0,1),(1,1))}$), and 2 time units after each subsequence `receive`.

2.2 Program 2: Non-terminating Mutual Exclusion

Figure 4 contains a different form of synchronization than the last program: two database transactions update the same record (a serially reusable resource) in mutually exclusive fashion using two semaphores. (Only one is needed, but we use two to illustrate several concepts.) The equivalent graph model is shown in Fig. 5. Blocking can only occur on the edges out of a vertex representing a code segment in which a P semaphore operation is performed. The blocking condition is that a process cannot complete a P operation until the other process is not in the code segment between completion of a P operation and a V operation on the same semaphore. Such code segments label edges in Fig. 5 and correspond to C in A4 (§1),

The corresponding TPG in Fig. 6 shows in heavy lines the set of TETs that could arise from two initial points. Suppose that process 1 runs for 5.5 time units before process 0 starts; this corresponds to initial point (0,5.5). The TET consists of a single slope one ray of infinite length by Rule II, case (2). Therefore the processes forever execute without blocking. Now suppose that process 0 starts execution 2 time units before process 1; there are two possible TETs, both rooted at initial point (2,0). Both TETs contain as their first ray $\overline{([(2,0),(3,1)))}$, representing concurrent execution by both processes for one time unit. The final point of this ray, (3,1), represents the program state when both processes simultaneous attempt to perform the `P(B)` semaphore

```
A,B: semaphore=1;

process 0           process 1
  while (TRUE) do     while (TRUE) do
    input;              input;
    P(A); P(B);         P(B); P(A);
    update;             update;
    V(B); V(A);         V(B); V(A);
    output;             output;
  do                  od
```

Figure 4: Non-terminating data-base transactions using semaphores and a serially reusable resource.

operation. There are two possible outcomes, corresponding to which process first completes `P(B)`. Thus point (3,1) represents a non-deterministic program state. If process 1 first completes `P(B)`, then process 0 blocks and the second TET ray is vertical: $\overline{([(3,1),(3,3)))}$. The TET has a final point, which is (3,3), representing a deadlock. The deadlock arises because process 1 then attempts `P(A)` and blocks because process 0 already holds semaphore A. The alternate TET with initial point (2,0) has as its second ray $\overline{([(3,1),(6,1)))}$ (in which process 1 blocks for 3 time units). This is followed by a third ray, with slope one, initial point (6,1), and infinite length. Thus when process 1 starts 2 time units after process 0, the program either reaches a deadlock or process 0 blocks for 3 time units after which both processes run forever without blocking. Because UPGs have been used extensively for analysis of deadlocks [7, 11, 17, 22, 24], deadlocks are not considered further in this paper.

2.3 Program 3: A Terminating Program

Fig. 7, unlike the previous two programs, contains a terminating program. A producer process reads a disk file whose first record specifies the number of successive records, and sends the records to the consumer, which then writes the records to another disk file. (To simplify presentation,

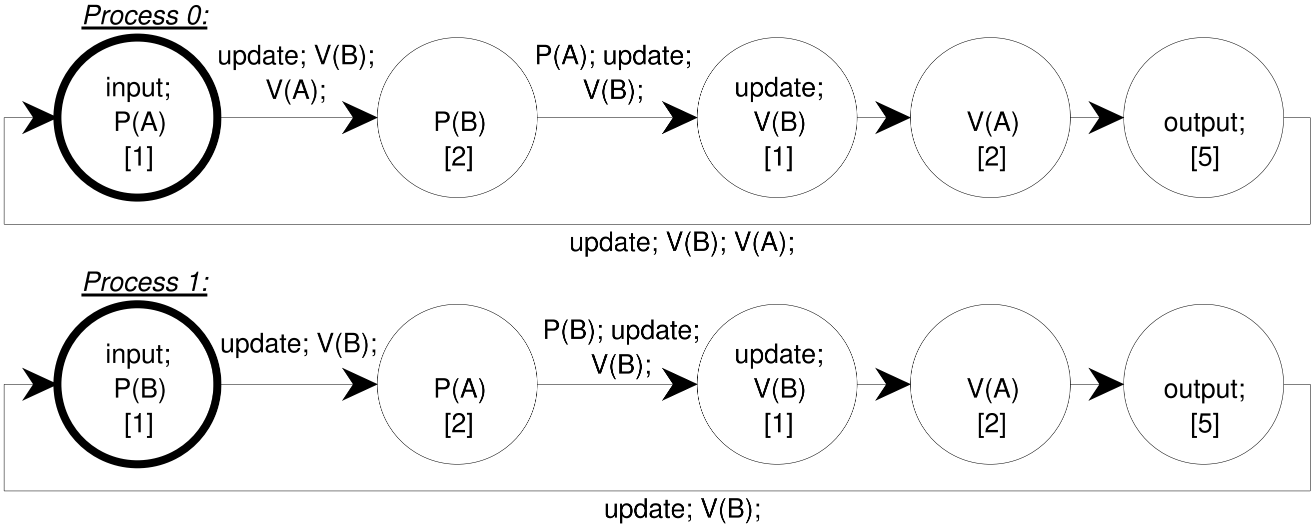

Figure 5: Graph model corresponding to Fig. 4.

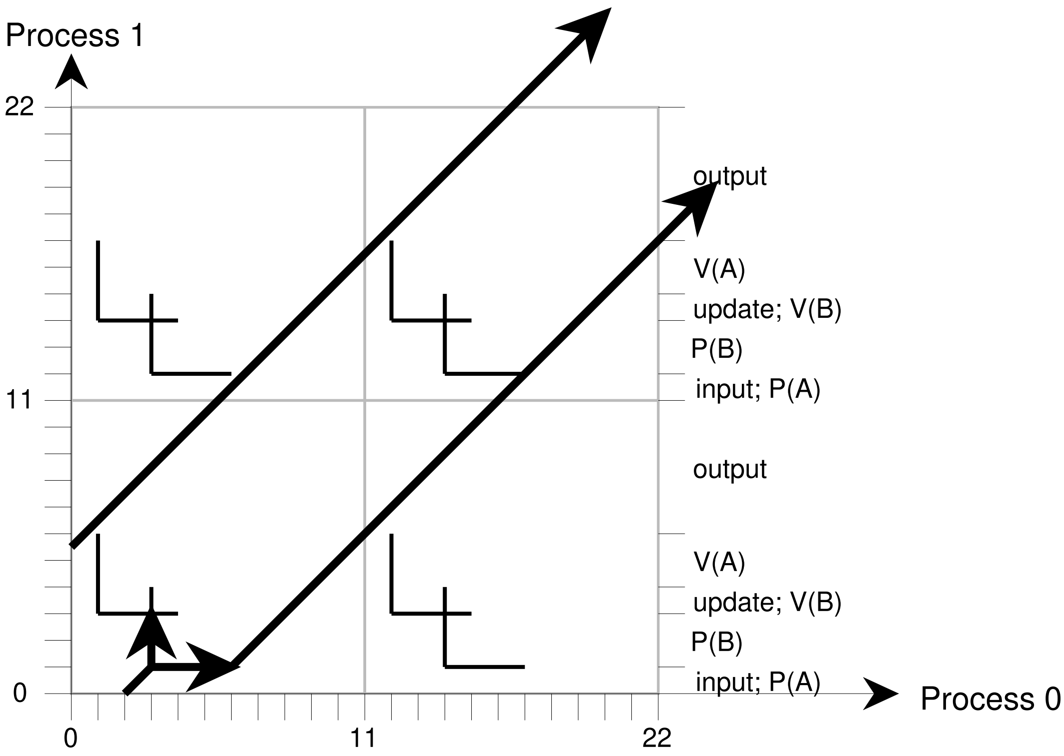

Figure 6: Timed progress graph corresponding to Fig. 5.

```
constant Size=2;    A:    integer[0..Size];
ME: semaphore=1;    Empty: semaphore=Size;    Full: semaphore=0;

process producer                        process consumer
  J: integer=0;                           K: integer=0;
  I, NumRecords: integer;

  read NumRecords from disk;              while (TRUE) do
  for I=1 to NumRecords do                  Y: character;
    X: character;                           P(Full); P(ME);
    read X from disk;                       Y=A[K];
    P(Empty); P(ME);                        K=K+1 mod Size;
    A[J]=X;                                 V(ME); V(Empty);
    J=J+1 mod Size;                         write Y;
    V(ME); V(Full);                       od
  od
```

Figure 7: Terminating producer/consumer code with shared memory interprocess communication.

the consumer is non-terminating.) We assume that interprocess communication is implemented through mutually exclusive access to shared memory using an array with `Size` elements. The corresponding graph (Fig. 8) is obtained by unrolling the loops in the two processes.

The corresponding TPG (Fig. 9, for `NumRecords=6`), unlike the preceding TPGs, is bounded on *four* sides, not just two (i.e., the left and bottom by the axes). The two additional bounds, called the *right* and *top bounding lines*, with equations $x = 25$ and $y = 31$, represent termination of the producer and consumer, respectively, at times 25 and 31.

Because Fig. 7 combines the synchronization of the preceding two examples, its TPG (Fig. 9) combines the two forms of constraint lines in the preceding TPG figures. The TETs shown represent all possible executions when both producer and consumer start simultaneously. Note that all TETs are of finite length and have a final point (at point (25,31)), because the producer terminates after sending six records and the consumer blocks forever at `P(Full)` after receiving six records.

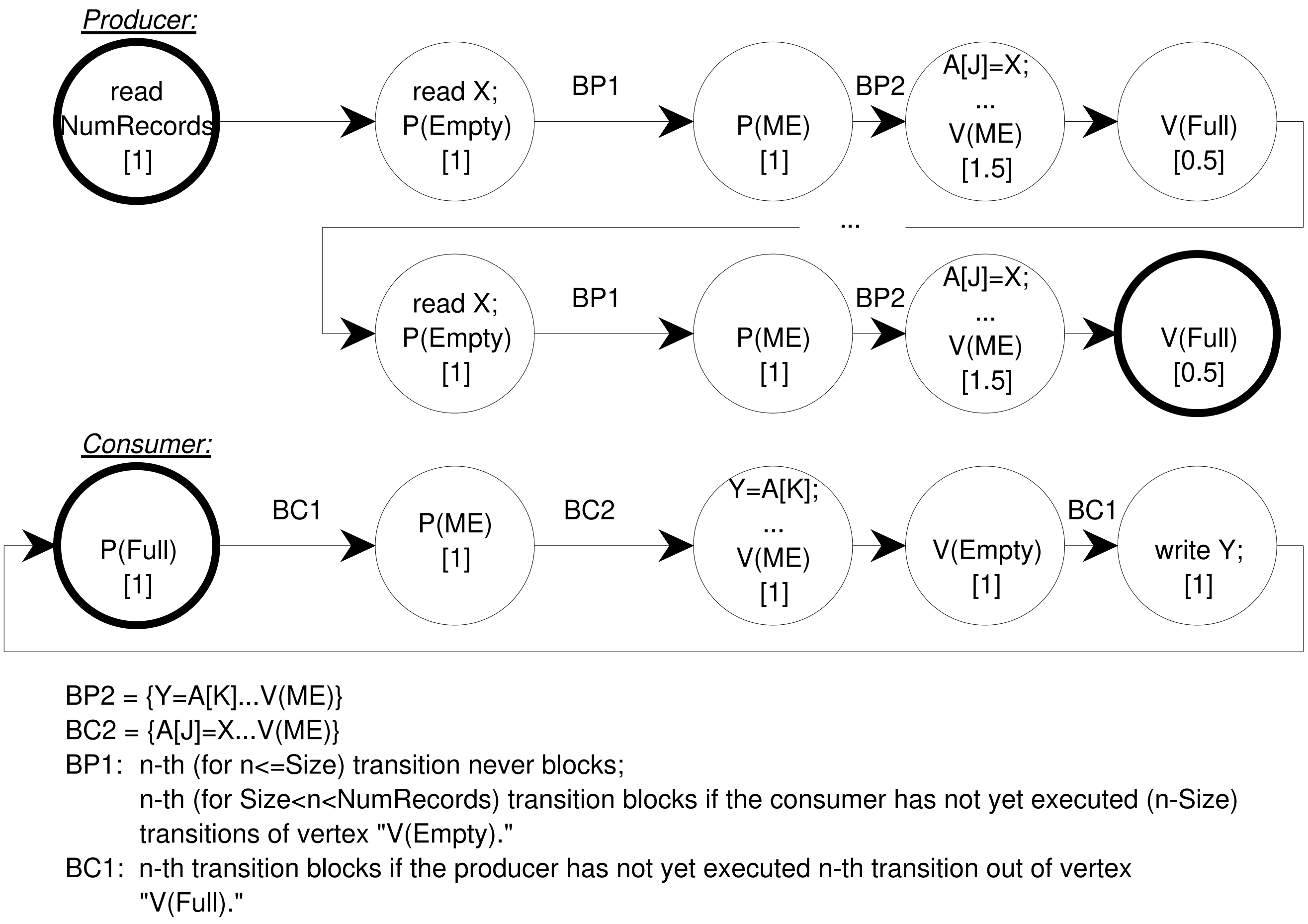

Figure 8: Graph model corresponding to Fig. 7. Thick circles denote initial and final vertices.

We learn from the TPG that the first synchronization point encountered is when the consumer performs `P(Full)` (point (1,1)). The next synchronization point occurs when the producer and consumer simultaneously attempt, respectively, the third and second access to the shared buffer pool (point (11,7)). At this time a race condition occurs. If the producer obtains semaphore ME first (and thus the TET contains ray $\overline{[(11,7),(12.5,7)))}$, then for the remainder of execution the consumer never blocks and the producer will repeatedly synchronize briefly at its `P(Empty)` operation. On the other hand, if the consumer obtains semaphore ME first (and thus the TET contains ray $\overline{[(11,7),(11,8)))}$, then the processes will again encounter a race condition followed

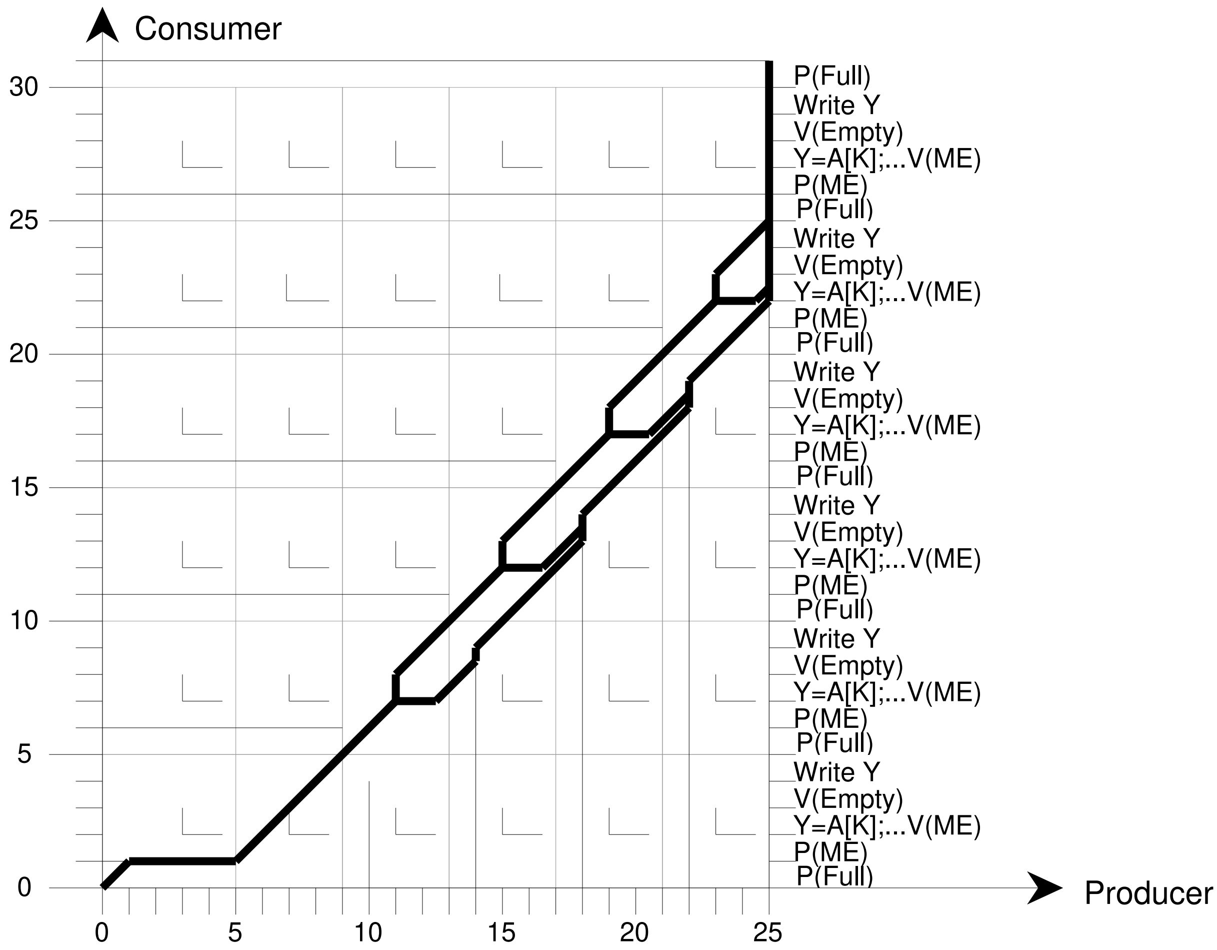

Figure 9: Timed progress graph corresponding to Fig. 8 with `NumRecords`=6.

by the same two possible outcomes. All TETs contain as part of their final ray $\overline{[(25,25),(25,31)]}$, which represents the consumer removing the final buffer after the producer has terminated.

2.4 Problem Statement

The preceding examples demonstrate that quantities Q1 to Q3 from §1 can be computed by solving problem P1 below. P2 is added to illustrate state reachability analysis with TPGs.

P1: Given a TPG, find the set of all possible TETs rooted at some initial point.

P2: Given a TPG, determine if there exists any process starting times that correspond to a point leading to exactly one TET in which no process ever blocks (e.g., the TET rooted at (0,5.5) in Fig. 6). Furthermore, if there exists such times, then output an example.

3 Construction of a TET

Before presenting algorithms solving P1 and P2, we formally define a TPG and the rule to construct all possible TETs. Let R and Z denote, respectively, the non-negative reals and integers.

3.1 Formal Definition of TPG

Definition. *A* TPG *for a graph representation of a program is an ordered pair* $\langle \Lambda, G^C \rangle$*, where* Λ *is a non-empty set of* constraint line segments *in the positive* R^2 *plane, and* G^C *is an* initial point *in* R^2 *representing the earliest instance at which both processes have started execution.*

For example, consider the TPG in Fig. 3(b). Let Λ_V denote the vertical lines: $\{(8+4i,0),(8+4i,1+2j) \mid i,j \in Z\}$. The horizontal lines are $\Lambda_H = \{(0,1+2j),(4+4i,1+2j) \mid i,j \in Z\}$. The TPG is $\langle \Lambda_V \cup \Lambda_H, (0,0) \rangle$. For terminating programs, Λ also contains the right and top bounding lines (e.g., $x = 25$ and $y = 31$ in Fig. 9).

3.2 Transition Function f

To formalize Rules I and II from §2.1 for a TPG $\langle \Lambda, G^C \rangle$, we define below two functions, f_o and f_d, respectively. (The subscript "o" in f_o denotes a ray that is orthogonal to the axes, and the "d" in f_d denotes a ray that is diagonal [with slope one].) Defining f_o and f_d requires some notation. For any continuous path γ and any point G on γ, we write $G \in \gamma$. For any directed, continuous path γ, we write $\gamma.i$ (respectively, $\gamma.f$) to denote the initial (final) point. We assume that $\gamma.i \neq \gamma.f$. For any two continuous paths γ and γ' in R^2, $\gamma \cap \gamma'$ denotes $\{G \mid G \in \gamma \wedge G \in \gamma'\}$. The relation $G \in L$ holds iff G lies on line or ray L. Line or ray $\overline{[G,(\infty,\infty))}$ has slope one and infinite length.

$\hat{G} \in f_o(G)$ iff G lies on a constraint line instance L and $\hat{G}$ is the smallest of $L.f$ and those points greater than G at which L intersects another constraint line instance. Formally,

$$\hat{G} \in f_o(G) \text{ iff } \langle \exists L \,:\, L \in \Lambda \,\wedge\, G \in L \,::\, \hat{G} = \min\Big(\{L.f\} \,\cup\, \{G' \mid G' > G \,\wedge\, \langle \exists L' : L' \in \Lambda - \{L\} :: G' \in L' \cap \overline{(L.i, L.f)} \rangle\}\Big)\rangle. \quad (1)$$

$\hat{G} \in f_d(G)$ iff G lies off a constraint line instance and $\hat{G}$ is the smallest of (∞,∞) and the set of points at which a slope one ray rooted at G intersects a constraint line instance. Formally,

$$\hat{G} \in f_d(G) \text{ iff } \langle \nexists L \,:\, L \in \Lambda \,\wedge\, G \in L\rangle \,\wedge\, \hat{G} = \min\Big(\{(\infty,\infty)\} \,\cup\, \{G' \mid \langle \exists L' : L' \in \Lambda :: G' \in \overline{[G,(\infty,\infty))} \cap L' \rangle\}\Big). \quad (2)$$

Functions f_o and f_d together define *transition function* f: $f(G) \stackrel{def}{=} f_o(G) \cup f_d(G)$. Function f maps each point $G \in R^2 - \{(\infty,\infty)\}$ to a (possibly empty) set of successors $f(G) \subseteq R^2$.

A point is nondeterministic iff the transition out of the point is not unique (e.g., (3,1) in Fig. 6), for example representing states in which both processes simultaneously perform a P operation on the same semaphore. All other points are deterministic. A point is dead iff there is no transition out of the point (e.g., (3,3) in Fig. 6), representing states in which both processes are blocked. A dead point may represent either a program deadlock or program termination.

Definition. *A point G is* <u>nondeterministic</u> iff $||f(G)|| > 1$ and is <u>*dead*</u> *iff* $f(G) = \emptyset$.

3.3 Constructing a TET

A TET is a point or a directed continuous path consisting of a sequence of horizontal and diagonal rays. There may be multiple TETs rooted at the same initial point. The following definition of TET also provides a rule to construct a TET that formalizes the recursion in Rules I and II of §2.

Definition. *A* <u>timed execution trajectory</u> *(TET) of a TPG $\langle \Lambda, G^0 \rangle$ rooted at any point $G^0 \in R^2$ is either (1) a point G^0 or (2) a directed, continuous path rooted at G^0. Case (1) holds iff $f(G^0) = \emptyset$.*

Case (2) holds iff the path is a ray sequence $\overline{[G^0, G^1)}$, $\overline{[G^1, G^2)}$, $\ldots$, $\overline{[G^{n-1}, G^n)}$ *(where* n *may be infinite) satisfies* $\langle \forall i : 0 \le i < n :: G^{i+1} \in f(G^i) \rangle$*. In case (2),* n *is finite iff* $f(G^n) = \emptyset$*.*

Example 2 In Fig. 3(c), the first line segment in the TET rooted at initial point $G^0 = (0,0)$ is $\overline{[G^0, G^1)}$, where $G^1 = (1,1)$ because $f((0,0)) = f_d((0,0)) = \{(1,1)\}$. This is because (0,0) does not lie on a constraint line instance and a slope one ray rooted at point (0,0) first intersects a constraint line instance at point (1,1). Next, $f((1,1)) = f_o((1,1)) = \{(4,1)\}$. Thus, the second TET line segment is (G^1, G^2), where $G^2 = (4,1)$. Continuing in this manner yields $\overline{[(0,0),(1,1))}$, $\overline{[(1,1),(4,1))}$, $\overline{[(4,1),(6,3))}$, $\overline{[(6,3),(8,3))}, \ldots$ as the only possible TET.

As a second example, consider $G^0 = (2,0)$ in Fig. 6. Then $f(G^0) = \{(3,1)\}$ and $G^1 = (3,1)$. However, (3,1) is nondeterministic; thus $f(G^1) = \{(3,3),(6,1)\}$. Continuing like this yields two possible TETs: $\overline{[(2,0),(3,1))}$, $\overline{[(3,1),(3,3))}$ and $\overline{[(2,0),(3,1))}$, $\overline{[(3,1),(6,1))}$, $\overline{[(6,1),(\infty,\infty))}$. □

4 Case 1: Terminating Programs

This section contains algorithms to solve P1 and P2 from §2.4 for terminating programs.

4.1 Computing Function f

Essential to solving P1 and P2 is a method of computing transition function f for a TPG $\langle \Lambda, G^C \rangle$. By definition, computation of $f(G)$ for any point G in R^2 requires computing either $f_o(G)$, if G lies on a line in Λ, or $f_d(G)$ otherwise. Computation of f_o and f_d is discussed below.

Computation of $f_o(G)$: Computation of $f_o(G)$ for point G on line L is straightforward using relation (1) of §3.2. (1) requires computation of the set of all points of intersection of line $\overline{(L.i, L.f)}$ with other lines in Λ. We precompute $\{G' | \langle \exists L' : L' \in \Lambda - \{L\} :: G' \in L' \cap \overline{(L.i, L.f)} \rangle\}$ for all $L \in \Lambda$, which then reduces the computation of each evaluation of $f_o(G)$ to evaluating the minimization function in (1). Precomputing this set is equivalent to a well known computational geometric problem: computing all intersections of a collection of horizontal and vertical lines (e.g., see [21, Ch. 27]). Thus computation of f_o is not considered further.

Computation of $f_d(G)$: Recall that $f_d(G)$ is the smallest of (∞, ∞) and each point at which a slope one ray rooted at G intersects a constraint line instance. The well known problem of ray shooting with line segments [6, pp. 234-247] can be used to find $f_d(G)$: Given a point, a direction, and a finite set of line segments in a plane, find the first line segment intersected by a ray rooted at the point with the given direction. The typical ray shooting solution first stores the line segments in a data structure, so that subsequent queries consisting of a point and a direction can be answered in sublinear time. Let `ShootRay(point P, direction D, set of line L)` denote a ray shooting algorithm whose parameters are a point P; a direction D (either $+$ or $-$ for a ray directed away from or toward the axes, respectively); and a set L, containing a finite number of line segments. The return value is the point of intersection or (∞, ∞) if the ray does not intersect a line segment. Formally, $f_d(G) = \{\texttt{ShootRay}(G,+,\Lambda)\}$.

4.2 Problem P1: Finding All Possible TETs

Problem P1: *Given a TPG $\langle \Lambda, G \rangle$, output a representation of all TETs rooted at G.*

The definition of TET along with the aforementioned methods to compute f_o and f_d solve P1. However, there is one technical problem: $f(G)$ is a set that contains either one (for deterministic G) or two (for non-deterministic G) points. If $f(G)$ contains two points, then there are at least two possible TETs rooted at G. We say "at least two" because if a point on one of the TETs rooted at G contains a point distinct from G that is nondeterministic, there will be more than two TETs rooted at G. Therefore a solution to P1 requires calculating a *set* of TETs.

The proposed algorithm constructs a directed graph. If the set of all possible TETs rooted at G is simply the point G, then the graph contains one vertex, labeled G. Otherwise the graph contains one vertex for each ray end point in all possible TETs rooted at G. The graph is colored, with green representing unexplored vertices and the remaining vertices colored red.

FindAllTETs (for terminating programs): Initialize the graph to contain one vertex representing G, colored green. The following step is repeated until there are no more green vertices in the graph: select a green vertex G' in the graph; color G' red; for each vertex $G'' \in f(G')$, add an arc from G' to G'' and color G'' green.

Example 3 *For Fig. 9, the graph consists of a vertex for point (0,0) with an outgoing arc to a vertex for point (1,1), an arc from (1,1) to a vertex for (5,1), an arc from (5,1) to (11,7), two outgoing arcs from (11,7) to (12.5,7) and (11,8), and so on, with all graph paths leading to the vertex for point (25,31). Only the vertex for (25,31) has no outgoing arc.* □

4.3 Problem P2: Deciding Existence of Non-blocking TETs

We restate P2 using the following definition: a TET is *non-blocking* if the TET contains no points representing a state in which a process is blocked. (Geometrically, a diagonal ray rooted at a point that leads to a non-blocking TET never intersects a constraint line in Λ and its final point lies on the top or right bounding line in Λ.) Problem P2 is equivalent to: *Given a constraint line set Λ from a TPG, determine if there exists an initial point on the x or y axis that leads to exactly one TET such that the TET is non-blocking. If there is, output the TET.*

Algorithm FindFreePoints: Let $\mathcal{L}$ be a set containing the lines in Λ and *left* := $\overline{[(0,0),(0,\infty)]}$ and *bottom* := $\overline{[(0,0),(\infty,0)]}$. Set G to $\texttt{ShootRay}(L.f,-,\mathcal{L})$. If there exists a line L in Λ such that $G \in \{$ *left*, *bottom*$\}$ and G'=$\texttt{ShootRay}(L.f,+,\mathcal{L})$ lies on the top or right bounding line, then return $\overline{G,G'}$; otherwise report "no non-blocking TETs exist."

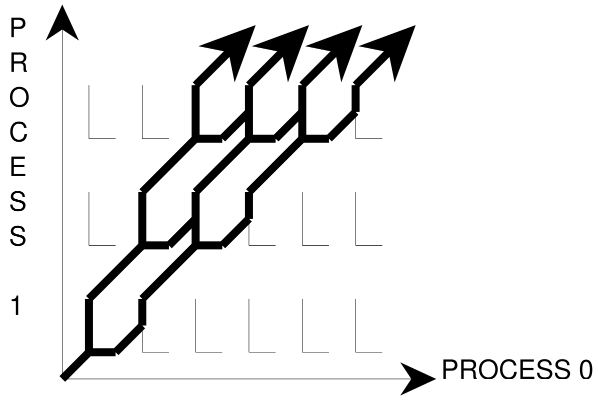

Figure 10: TPG containing an infinite number of TETs.

5 Case 2: Non-terminating Programs

The preceding algorithms cannot be used for non-terminating programs for two reasons. First, a TPG for terminating programs is a rectangle with four finite length sides, while a TPG for non-terminating programs extends to infinity in the Cartesian quadrant in which both axes are positive. The unbounded nature of the TPG means that a TET will have *infinite length*, unless the execution reaches a deadlock. Second, there may be an *infinite set* of TETs rooted at a point in the plane (Fig. 10). This occurs when the program timings are such that in any state represented by a point on some TET, the program will eventually reach another race condition.

Non-terminating programs whose synchronization is only for mutually exclusive resource access (e.g., §2.2) permit a simple characterization, because they exhibit periodic behavior. The grey lines in Fig. 6 partition the plane into a set of equal-size rectangles, in which the location of the rectangle sides in the plane correspond to initiation of new infinite loop iterations in a process. Each rectangle is called a *quadrant*. Formally defining a quadrant requires the notion of process *cycle time*. Let $r \in \{0, 1\}$ denote one of two processes, and $\bar{r} = 1 - r$ denote the other process (i.e., if $r = 0$ then $\bar{r} = 1$ and vice versa). The *cycle time* of process r, denoted ϕ_r, is the time required for process r to pass through each vertex in the infinite loop in its graph representation once, ignoring the time spent blocked. Thus in Fig. 5 $\phi_0 = \phi_1 = 1 + 2 + 1 + 2 + 5 = 11$.

Definition. *A* quadrant *is a region* $\{(G_0, G_1) \mid \forall r, r \in \{0, 1\}, \exists i_r, i_r \in Z, i_r\phi_r \le G_r < (i_r + 1)\phi_r\}$. *The* initial quadrant *is the quadrant containing the origin.*

Example 4 The initial quadrant in Fig. 6 has opposite vertices (0,0) and (11,11). Each quadrant has opposite vertices $(11i_0, 11i_1)$ and $(11(i_0 + 1), 11(i_1 + 1))$ for some i_0, $i_1 \in Z$. □

The significance of quadrants is that constraint lines in all quadrants are *congruent*:

Definition. *For any point* $G = (G_0, G_1)$, *mod*$(G) \stackrel{def}{=} (G_0 \bmod \phi_0, G_1 \bmod \phi_1)$. *Two points* G *and* G' *are* congruent, *denoted* $G \equiv G'$, *iff mod*(G) = *mod*(G'). *Two line segments are congruent if their end points are congruent.*

The fact that the placement of constraint lines in all quadrants are congruent has two implications. First, it is sufficient to analyze just one quadrant in the plane to derive the set of all possible TETs, and this modification of the algorithms from the preceding section follows. Second, a TET containing only deterministic points will consist of a transient portion followed by an infinite number of repetitions of congruent subtrajectories, representing periodic behavior. (See Theorem 1 in [3].) A repeated subtrajectory is called a *limit cycle execution trajectory* (LCET).

Example 5 In Fig. 11, $\phi_0 = 5, \phi_1 = 3$. The TET subpath rooted at point $(3, 2)$ in Fig. 11 consists of an infinite number of repetitions of the following LCET: a horizontal ray of length 2 and a diagonal ray whose projected length on either axis is 3. The transient TET portion is the subpath with initial point (0,0) and final point (3,2). All instances of the LCET in the TET are congruent. For example, two instances are $\overline{[(3, 2), (5, 2))}$, $\overline{[(5, 2), (8, 5))}$ and $\overline{[(13, 8), (15, 8))}$, $\overline{[(15, 8), (18, 10))}$. The two are congruent because the first and second rays of the first subtrajectory are congruent to the first and second rays of the second subtrajectory, respectively. □

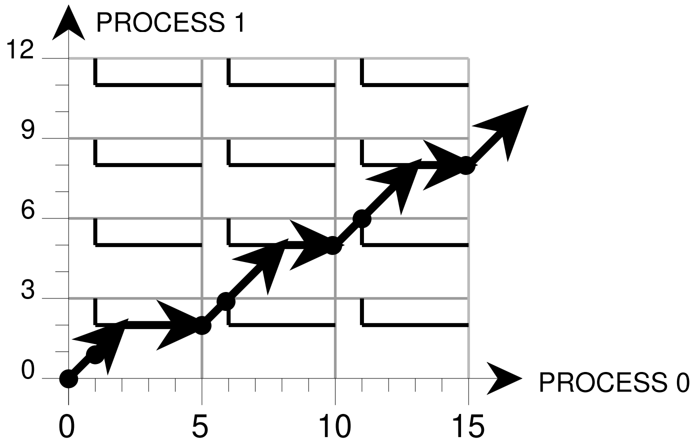

Figure 11: Timed progress graph of non-terminating program that uses one semaphore.

LCETs motivate one problem in addition to P1 and P2 (§2.4), solved in this section:

P3: Find the set of all possible LCETs in a TPG that are reachable from some initial point.

Special Case TPG Definition: A TPG for a non-terminating program synchronizing only for mutually exclusive resource access is an ordered triple $\langle \Phi, \Lambda, G^C \rangle$ specifying process cycle times, constraint lines in the initial quadrant only, and a point representing the initial program state:

Φ: an ordered pair of cycle times ϕ_0 and ϕ_1 satisfying $\phi_0, \phi_1 \in R^2 - \{\infty\}$.

Λ: a set of constraint line <u>*generators*</u>, or line segments $\overline{[W, X)}$ that lie in the initial quadrant, each corresponding to one edge in the graph model labeled by a non-empty condition. The <u>*initial*</u> and <u>*final*</u> points of generator $\overline{[W, X)}$ are W and X, respectively. The <u>*instances*</u> of a generator are defined to be all lines in the R^2 plane congruent to the generator; formally, all instances of generators in Λ is $\Lambda^* \stackrel{def}{=} \{\overline{[W, X)} \mid W \in R^2 \ \wedge \ X \in R^2 \ \wedge \ \overline{[mod(W), mod(X))}) \in \Lambda\}$.

G^C: an <u>*initial point*</u> satisfying $\langle \exists r : r \in 0, 1 :: G^C_r = 0 \ \wedge \ 0 \leq G^C_{\bar{r}} < \phi_{\bar{r}} \rangle$. Therefore G^C lies on either the x or y axis, within one cycle time of the origin.

The TET construction rules (I and II) and the definition of transition function f given earlier in §2.1 and §3.2, respectively, apply unaltered to a TPG $\langle \Phi, \Lambda, G^C \rangle$.

Example 6 Figure 11 illustrates a finite portion of the TPG $\langle \{\phi_0, \phi_1\}, \Lambda, G^C \rangle$, where $\phi_0 = 5$, $\phi_1 = 3$, $\Lambda = \{\overline{[(1,2),(1,3))}, \overline{[(1,2),(5,2))}\}$, and $G^C = (0,0)$. □

A practical consideration: We henceforth assume that the execution time of each vertex in a graph representing a program (e.g., the numbers in square brackets in Figs. 2, 5, and 8) is an integer rather than a real number. Otherwise the computational geometric algorithms to be presented will not work correctly with finite precision arithmetic (e.g., computation of the *mod* operation is subject to roundoff error). The assumption of integer delays is not unreasonable in practice for software performance evaluation. For example, in measurements from a computer with a microsecond period clock and all measured times are rational numbers of the form $\frac{x}{10^6}$, where $x \in Z$. Therefore scaling all measurements by the inverse of the clock period (e.g., 10^6) yields the integer quantities required by the proposed algorithms.

5.1 Modified Algorithms to Compute $f(G)$

Computing $f_o(G)$: Recall from §3.2 that for a point G on a constraint line L, $f_o(G)$ is the smallest point $G' > G$ in the set containing the final point of L and the points of intersection of L with other constraint lines. Because Λ in $\langle \Phi, \Lambda, G^C \rangle$ contains only constraint lines in the initial quadrant, we map G to a congruent point in the initial quadrant (i.e., $mod(G)$), then compute $f_o(mod(G))$ as described in §4.1, and finally map each $G' \in f_o(mod(G))$ back to the quadrant containing G. Formally, we compute $f_o(G) = \{G'' \mid \forall G' \, \forall r,\, G' \in f_o(mod(G)) \wedge r \in \{0,1\},\, G''_r = G'_r + \lfloor \frac{G_r}{\phi_r} \rfloor \phi_r\}$.

Computing $f_d(G)$: The method to compute $f_d(G)$ given in §4.1 must be modified because a non-terminating TPG has an infinite number of constraint lines in the plane. Therefore, as we

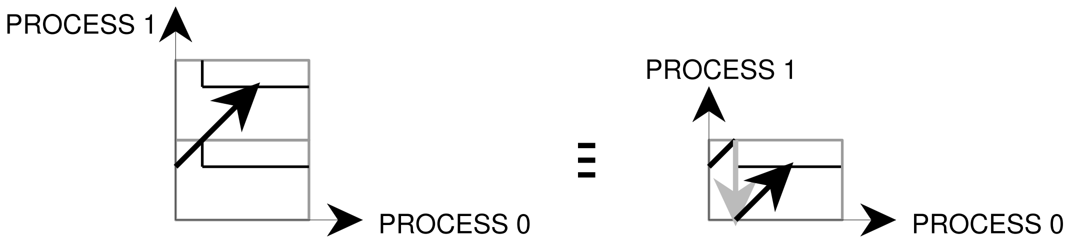

Figure 12: Illustration of using `ShootRay`, but only within initial quadrant.

did with $f_o(G)$, we compute f_d using only the initial quadrant. We use the observation that any diagonal ray γ in a TET can be partitioned into collinear rays $\gamma_1, \gamma_2, \ldots, \gamma_n$ (where $n \geq 1$) in which the initial points of $\gamma_2, \ldots, \gamma_n$ lie on a quadrant boundary. This fact provides an algorithm to compute $f_d(G) = \gamma$: for each ray γ_k ($1 \leq k < n$), translate $\gamma_k.i$ to the initial quadrant, compute $G' = \texttt{ShootRay}(\gamma_k.i, +, \Lambda \cup \{\text{top and right initial quadrant edges}\})$, and translate G' back to γ_k's quadrant to obtain $\gamma_{k+1}.i$. See [1] for the complete algorithm.

Example 7 *In Fig. 11, $f_d(5,2) = (8,5)$. Fig. 12 shows the two* `ShootRay` *operations required to compute $f_d(G)$ because ray $\overline{(5,2),(8,5)}$ lies in exactly two quadrants. Here, $n = 2$, $\gamma_1 = \overline{[(5,2),(6,3))}$, and $\gamma_2 = \overline{[(6,3),(8,5))}$.* □

5.2 P1: Finding All Possible TETs

Recall problem P1: *Given a TPG $\langle \Phi, \Lambda, G \rangle$, output a representation of all TETs rooted at G.* Before stating a solution, two implications of non-terminating programs must be considered: A TET may have infinite length, and there may be an infinite number of TETs (recall Fig. 10). For the former case (infinite length TET), the TET must consist of a transient subtrajectory followed by an infinite number of repetitions of an LCET. Therefore our algorithm will output a representation of the transient trajectory and the first LCET. For the later case (infinite number of TETs), we choose an integer value *maxNPaths* such that if our algorithm finds more than *maxNPaths* possible TETs, it assumes that there are an infinite number of TETs and terminates without further exploration. We now generalize algorithm **FindAllTETs** (§4.2) to solve P1 for non-terminating programs.

Algorithm FindAllTETs (for non-terminating programs):

1. Initialize a graph to contain one green vertex labeled by G. Set *nPaths*=1.
2. Set G' = some green vertex. Color G' red. Increment *nPaths* by $||f(G')|| - 1$.
3. For each point G'' in $f(G')$, create a vertex labeled by G'', and add a directed edge from G' to G''. If G'' is congruent to some point P labeling a vertex in the graph path from G to G'', then add an arc from G'' to P and color vertex G'' red; otherwise color it green.
4. If the graph contains a green vertex and *nPaths*$\leq$ *maxNPaths*, then go to step 2. Otherwise output each graph path rooted at G. Label paths containing a cycle as a transient followed by an LCET; label the remaining paths as a transient only.

Example 8 The graph constructed by **FindAllTETs** for the TPG of Fig. 11 consists of an edge from the graph vertex labeled (0,0) to the vertex labeled (2,2), an edge from (2,2) to (5,2), from (5,2) to (8,5), and from (8,5) back to (5,2). The fact that the graph contains only one path with one cycle means that if the both processes simultaneously start execution (i.e., the program starts in the state represented by (0,0)), then it must reach a periodic state sequence in which process 0 blocks for two time units and then both processes run concurrently for three time units. □

5.3 Problem P2: Deciding Existence of Non-blocking TETs

Recall problem P2: *Given Φ and Λ from a TPG, determine if there exists an initial point on the x or y axis that leads to exactly one TET such that the TET is non-blocking. If there is, output the TET.*

Program termination makes only one minor difference from the algorithm given earlier (**Find-FreePoints** in § 4.3): Change "lies on the top or right bounding line" to "equals (∞, ∞)", and "return $\overline{G, G'}$" to "return $\overline{G, (\infty, \infty)}$".

5.4 Problem P3: Find All Possible LCETs

We restate P3 by categorizing LCETs as *blocking* or *non-blocking.* A blocking LCET contains a point representing a state in which some process is blocked. Geometrically, a blocking LCET contains a horizontal or vertical ray, and a non-blocking LCET consists of a single diagonal ray. Two TETs are *homotopic* if they are continuously transformable avoiding the constraint lines (term due to Lipski and Papadimitriou [11] for paths in UPGs). For example, the TETs rooted at all points in $\{G \,|\, 5 \leq G_0 \leq 6 \wedge G_1 = 0\}$ in Fig. 6 all consist of one diagonal ray and are homotopic.

Problem P3: *Given Φ and Λ from a TPG, output one element of each equivalence class of blocking LCETs, and one element of each set of homotopic LCETs.*

Algorithm FindNonBlockingLCETs (finds all *non-blocking* LCETs)**:** Initialize $\mathcal{L}$ as in **Find-FreePoints** (§4.3). Set $\mathcal{G} = \emptyset$. For each $L \in \Lambda$ do: if $\texttt{ShootRay}(L.f, +, \mathcal{L}) = (\infty, \infty)$, then $\mathcal{G} = \mathcal{G} \cup \{L.f\}$. If $\mathcal{G} = \emptyset$, output "no non-blocking LCETs"; otherwise for each point $G \in \mathcal{G}$, output "$\overline{[G,\ G + (\phi_0, \phi_1)]}$ is a non-blocking LCET."

Consider next finding all *blocking* LCETs. The key insight is that a set of TETs that contains rays that intersect a given constraint line instance, denoted L, also contain a common point that lies on L: either a dead point, in which case the point is the final point of all the TETs, or $L.f$. In the later case, this set must either have the same LCET or reach the same dead state that lies on another constraint line instance. If they reach a LCET, it is therefore only necessary to consider the TET rooted at $L.f$ and determine if it contains another point congruent to $L.f$. Lemma 1 makes this precise.

Lemma 1 *The set of all (possibly unreachable) blocking LCETs can be found by finding all* $L \in \Lambda$ *and all* $i \in Z$ *satisfying* $0 < i \leq 2||\Lambda|| \ \wedge \ ||f^i(L.f)|| = 1 \ \wedge \ G \in f^i(L.f) \ \wedge \ G \equiv L.f$.

Proof: See Theorems 3 and 4 in [2]. □

Let ζ denote the set of LCETs satisfying Lemma 1. We can determine which LCETs in ζ are reachable by calculating, for each line L in an LCET in ζ, `ShootRay`$(L.f, -, \mathcal{L})$, where $\mathcal{L}$ is the same line set as in algorithm **FindFreePoints**. If the return value of `ShootRay` lies on the bottom or left edge of the initial quadrant, then the LCET containing L is reachable. Therefore we propose as a solution to P3 the exhaustive testing of all L and i, evaluating $f^i(L.f)$ as described in §5.1.

6 Relation of TPGs to Petri Nets

Figures 13(a) and 13(b) are Petri net [16] representations of Figs. 1 and 4, respectively. The producer/consumer program in Figure 13(a) is in the Petri Net class called Deterministic Systems of Synchronizing Processes (DSSP) [18]; the shaded place denotes a buffer shared by the two processes. The data-base transaction program (Fig. 13(b)), however, is not in class DSSP, because it violates the rule of a place representing a buffer being the input to at most one process. In Fig. 13(b) the places representing semaphores as buffers are inputs to *both* processes. Fig. 13(b) is a simple (or asymmetric choice) net (i.e., all arc weights are one and if two places share an output transition then the set of output transitions of one place is either equal to or a subset of the output transitions of the other place [16, p. 554]). Therefore the class of programs meeting assumptions A1 through A6 is a DSSP restricted to two linear processes [23] but generalized to omit the private-buffer assumption (i.e., Definition 2.7(ii) in [18]).

Magott [12] gives an $O(N)$ algorithm to compute minimum cycle time (MCT), or the minimum time required for a consistent Petri net to return to its initial marking, given deterministic firing times for nets consisting of a set of N cyclic processes that mutually exclusively share a single

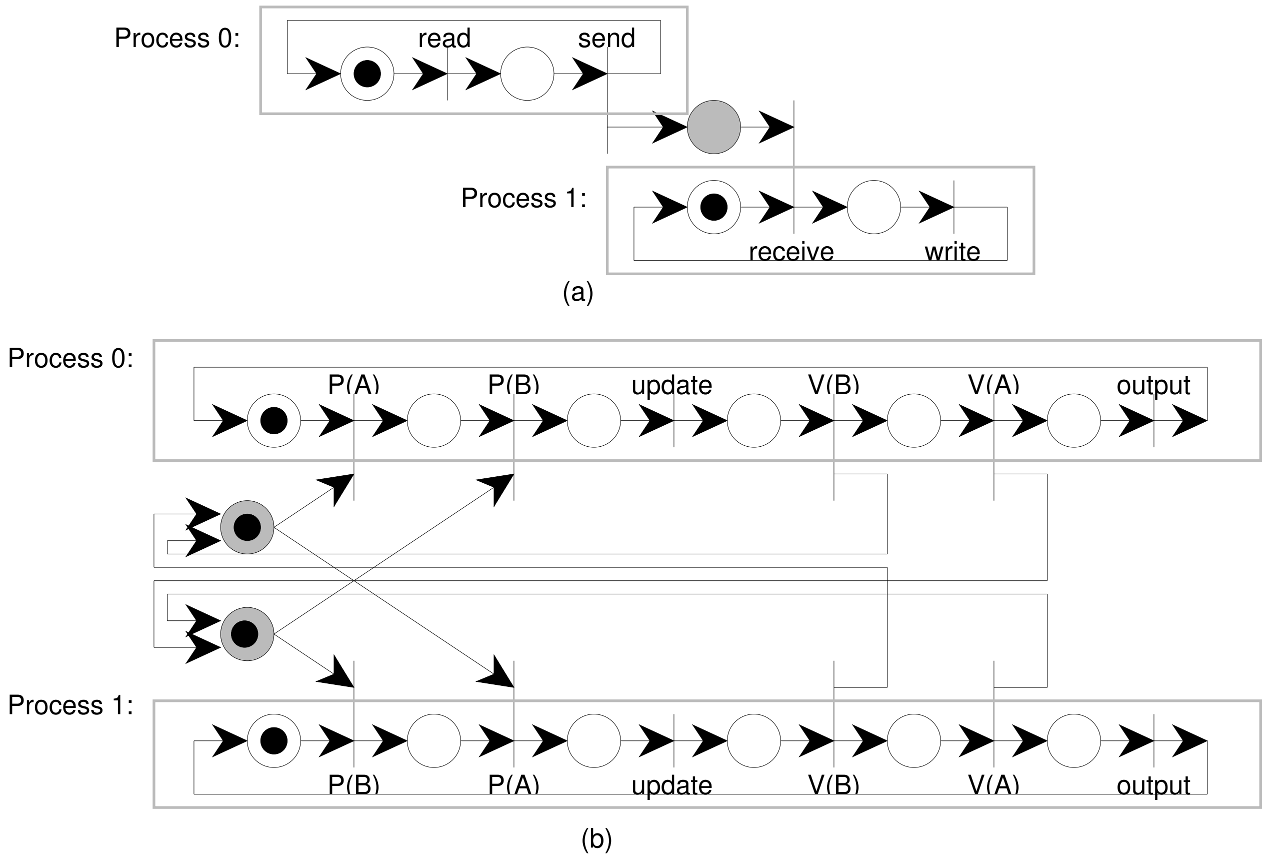

Figure 13: Petri net representations of Figs. 1 and 4.

resource, and shows that finding MCT in most nets with more complex resource sharing is NP-hard. Also proved are complexity results for systems of processes with communication by buffers. Finally, Holliday and Vernon [9] use Petri nets with frequency expressions (i.e., probabilities) to resolve deterministically which transition fires when a token enables two or more transitions simultaneously, and analyze a program similar to Fig. 4.

Our TPG solution provides a fourth analysis method of one Petri net class (in addition to coverability trees, matrix-equations, and decomposition techniques) for certain behavioral properties (i.e., P1 — enumerate all possible transition firing sequences, given an initial marking) and certain structural properties (i.e., P2 and P3).

7 Conclusions

We have analyzed two building blocks of interprocess synchronization: mutual exclusion and asynchronous communication with a finite number of buffers. This paper demonstrates that properties about the set of all possible executions of certain parallel programs can be exactly analyzed by solving an equivalent computational geometric problem.

The analysis is limited to two processes. Extension to d processes requires ray shooting in a d dimensional Cartesian graph with $d-1$ dimensional hyperplanes in a non-simple arrangement that are bounded in one dimension. To our knowledge, this is an open computational geometric problem, whose solution would allow solution of problems P1 and P2 for an arbitrary number of processes. The closest problem solved in d dimensions is ray shooting with unbounded hyperplanes that form a simple arrangement (e.g., see [14]). One limitation in d dimensions is that a progress graph represents contention by multiple processes for a resource as a nondeterministic choice of which process gets the resource, and does not represent different queueing disciplines.

The broader implication of this work is two open questions: Can other program visualizations be mapped to geometric problems? Can geometry be used to analyze other Petri net classes?

Acknowledgments

D. Allison, L. Heath, D. Kafura, A. Mathur, and S. Tripathi and anonymous referees made suggestions that improved the manuscript. C. Shaffer helped locate computational geometric algorithms.

References

[1] M. Abrams. Computational geometric performance analysis of limit cycles in timed transition systems. Technical Report TR 93-30, Computer Sci. Dept., Virginia Tech, Blacksburg, VA 24061-0106, Sept. 1993. Available from World Wide Web location http://info.cs.vt.edu/reports.

[2] M. Abrams. Geometric performance analysis of semaphore programs. Technical Report TR 93-29, Computer Sci. Dept., Virginia Tech, Blacksburg, VA 24061-0106, Aug. 1993. Available from World Wide Web location http://info.cs.vt.edu/reports.

[3] M. Abrams. Geometric performance analysis of periodic behavior. *J. Parallel and Distributed Computing*, to appear.

[4] M. Abrams, N. Doraswamy, and A. Mathur. Chitra: Visual analysis of parallel and distributed programs in the time, event, and frequency domain. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):672–685, Nov. 1992.

[5] V. S. Adve and M. K. Vernon. The influence of random delays on parallel execution times. In *Proc. SIGMETRICS*, pages 61–73, Santa Clara, CA, June 1993. ACM.

[6] P. K. Agarwal. *Intersection and Decomposition Algorithms for Planar Arrangements*. Cambridge Univ. Press, Cambridge, 1991.

[7] S. D. Carson and J. P. F. Reynolds. The geometry of semaphore programs. *ACM Trans. on Programming Languages and Systems*, 9(1):25–53, Jan. 1987.

[8] E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Comp. Surv.*, 3(2):70–71, June 1971.

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

[10] T. Lehr, Z. Segall, and et al. Visualizing performance debugging. *IEEE Computer*, 22(10):38–51, Oct. 1989.

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

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

[13] A. D. Malony. JED: Just an event display. In M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, pages 99–114. ACM Press, 1989.

[14] J. Matousek. On vertical ray shooting in hyperplanes. *Computational Geometry: Theory and Applications*, 2(5):279–286, Mar. 1993.

[15] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Trans. on Parallel and Distributed Systems*, 1(2):206–217, Apr. 1990.

[16] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

[17] C. H. Papadimitriou. Concurrency control by locking. *SIAM J. on Computing*, 12(2):215–226, May 1983.

[18] W. Reisig. Deterministic buffer synchronization of sequential systems. *Acta Informatica*, 18:117–134, 1982.

[19] G. Roman and K. Cox. A declarative approach to visualizing concurrent computations. *IEEE Computer*, 22(10):25–36, Oct. 1989.

[20] M. Sasieni, A. Yaspan, and L. Friedman. *Operations Research – Methods and Problems*. John Wiley, New York, 1959.

[21] R. Sedgewick. *Algorithms in C*. Addison Wesley, Reading, MA, 1990.

[22] E. Soisalon-Soininen and D. Wood. An optimal algorithm for testing for safety and detecting deadlocks in locked transaction system. In *Symp. on Principles of Database Systems*, pages 108–116, Los Angeles, Mar. 1982. ACM.

[23] Y. Souissi and N. Beldiceneau. Deterministic systems of sequential processes: Theory and tools. *Concurrency 88*, 335 Lecture Notes in Computer Science:380–400, 1988.

[24] M. Yannakakis, C. H. Papadimitriou, and H. T. Kung. Locking policies: Safety and freedom from deadlock. In *20th ACM Symp. on the Foundations of Computer Sci.*, pages 283–287, 1979.