Geometric Performance Analysis of Semaphore Programs

Marc Abrams

TR 93-29

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

August 31, 1993

Geometric Performance Analysis of Semaphore Programs

Marc Abrams*
Computer Science Department, Virginia Tech, Blacksburg, VA 24061-0106
TR 93-29 abrams@cs.vt.edu

August 31, 1993

Abstract

A key problem in designing parallel programs that achieve a desired performance goal is the ability to exactly analyze program performance, given a specification of the process synchronization structure and the execution timings of all code segments. Given a definition of program *state*, an execution of a program can be represented by a *timed execution sequence* (TES). A TES is a sequence of states that the program passes through in an execution, along with the duration of time spent in each state. In some parallel programs, a representation of the set of all possible TESs that can arise in any execution contains a suffix that consists of the repetition of a finite sequence of states, excluding deadlocks and nondeterministic behavior. The sequence that is repeated is termed the *limit cycle execution sequence*. The problem solved in this paper is to derive, for all possible process starting times, the set of all possible limit cycle execution sequences in which a process blocks. The paper makes two contributions. First, it employs a novel analysis method that derives TESs from a geometric model of program execution, called *timed progress graphs*. A timed progress graph adds timing information to Dijkstra's (untimed) progress graph, which Carson and Reynolds define as "a multidimensional, Cartesian graph in which the progress of each of a set of concurrent processes is measured along an independent time axis" [10]. A timed progress graph represents process synchronization between processes by line segments, and a TES by a directed, continuous path that does not cross a segment. The second contribution of the paper is to solve the timed progress graph not by a computational geometric algorithm, as employed by most solutions in the literature to untimed progress graphs, but by an analytic solution. The analytic solution transforms the problem of deriving blocking limit cycles to that of finding the minimum value satisfying each of a set of Diophantine equations.

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, Diophantine equation, periodic behavior

1 Introduction

The ability to analyze program performance lies at the heart of integrating into the software development life cycle what Smith [35] terms *software performance engineering.* The requirements life cycle phase states performance goals. Ideally, we would like to establish whether a program under development

*This work was supported in part by National Science Foundation grant NCR-9211342.

meets the performance goals long before the integrated testing life cycle phase. Doing this requires techniques to verify whether a program specification and pieces of code that cannot yet be executed meet the performance goals. Unfortunately, few theoretical methods exist today to do this. (See §2.)

This paper considers a fundamental problem that underlies performance engineering of parallel software. First, we define the *state* of a program. (In this paper, state represents the control point of each process.) We represent an execution of a program by two sequences, one consisting of the states that a program passes through during an execution, and the second consisting of real numbers. The i-th value in the real number sequence represents the time duration that the program spent in the i-th state of the state sequence. The two sequences together are called the *timed execution sequence* (TES). For example, let the state of a two process program be represented by an ordered pair (x, y), where x and y each denote the control point of a different process, and let the control points be numbered (for reasons apparent later) -1, 0, 1, and 2. Then a TES for a program that never terminates[1] might be:

$$\begin{array}{ll} \text{state:} & (-1,-1),\ (0,-1),\ (1,-1),\ (2,-1),\ (0,0),\ \left\{ \begin{array}{lll} (1,0), & (2,1), & (0,2) \\ 3, & 1, & 1 \end{array} \right\}^* \\ \text{time:} & 0,\ \ 1,\ \ 3,\ \ 1,\ \ 1, & \end{array}$$

where "{}*" denotes an infinite number of repetitions of the enclosed sequence. The TES represents a program that starts in state (-1,-1), instantly moves to state (0,-1), remains in state (0,-1) for 1 time unit, then moves to state (1,-1) for 3 time units, and so on. In general, for a given program there may exist an infinite number of possible TESs. For example, there may exist a different TES for each initial program state. If program execution is nondeterministic, there may even exist multiple TESs for a single initial state. A TES has finite length iff the program terminates or reaches a deadlock.

Can we find a representation of the set of *all* possible TESs for a program? The ability to do so would allow computation of any performance measure required to evaluate a performance goal. Although it may be possible to compute performance measures in a simpler manner, the structure of the TESs themselves might give insights as to why a performance measure has a particular value. In fact, the idea of analyzing execution sequences is quite popular, as witnessed by recent work on software performance visualization systems (e.g., [16, 33]) that include the ability to display program event traces.

[1] Certain programs can be considered non-terminating for the purpose of analysis. One example is long running programs that execute the same code repeatedly, such as simulations. A second example is *reactive programs* [11], such as operating system algorithms, that react to external stimuli. In fact, Chandy and Misra [11] have suggest that all computations can be represented as non-terminating transition systems that reach a fixed point.

This paper examines a certain class of parallel programs, defined below. The set of all TESs for a program in the class contains a suffix that consists of the repetition of a finite sequence of states, excluding deadlocks and nondeterministic behavior (see Theorem 1 of Section 4.3). The sequence that is repeated is termed the *limit cycle execution sequence* (LCES). The TES listed above has as a LCES states (1,0), (2,1), and (0,2) for 3, 1, and 1 time unit, respectively.

Problem Statement: Let the program state be an enumeration of the control point of each process. We are given a specification of the process synchronization structure and the execution timings of all code segments, in the form of a set of timed transition diagrams (illustrated below). The problem solved here is to derive, for all possible process starting times, the set of all possible LCESs in which a process blocks.

LCESs and, more generally, periodic behavior, have been observed experimentally several times in the literature. Zhang, Shenker and Clark observed low frequency oscillations in queue length and packet traffic in a communication protocol [38]. Periodic behavior also occurs with periodic tasks, such as operating system daemons [9]. Our past studies [3, 5] using a commercial TCP/IP protocol and a dining philosophers program [13] have also exhibited LCESs. In fact, experience with our own visualization system [5] demonstrates that the TESs of many long running programs that spend the bulk of their time looping in small fractions of the code repeatedly exhibit certain state subsequences. If the program execution time is chiefly determined by a small set of repeated subsequences, then finding these subsequences given the code and timings is an effective way to diagnose performance problems.

Analysis presented later paper gives some insight into the circumstances under which LCESs arise. In addition, for a wide variety of initial process starting times, any TES of a program converges to the same LCES. If we imagine a knob whose setting determines the time required for a transition, and vary the knob, then the subsequence stays the same until we turn the knob past a critical point at which the LCES to which the program converges changes.

Assumptions: To obtain a tractable problem, we make the assumptions listed below. The assumptions are stated in terms of *binary semaphores* [14] and *mutually exclusive access to a set of resources.* A binary

semaphore is an abstract data type with initial value 0 or 1, with two operations: P and V. The process invoking a P operation blocks iff the initial semaphore value plus the number of prior V operations performed by all processes does not exceed the number of prior P operations performed by all processes. If a process is blocked at a P operation when a V operation is performed, then one blocked process unblocks. Associated with each resource is a set of one or more code segments in each process. Mutually exclusive resource access means that the control point of both processes cannot simultaneously be in a code segment associated with the resource.

A1: A program contains two processes.

A2: Each process meets the following assumptions:

A2.1. A process executes on a dedicated processor.

A2.2. A process executes a nonterminating loop.

A2.3. A process synchronizes with other processes through binary semaphores

A2.4. Binary semaphore operations are executed unconditionally.

A2.5. The execution time of each code segment within each process that either (1) starts at the initial statement of the loop body and continues to and includes the first semaphore operation, or (2) follows each semaphore operation and continues to and includes the next semaphore operation is an independent constant, exclusive of time spent blocked.

A3: Synchronization between processes is used only to achieve mutually exclusive access to each of a set of resources.

The assumptions, while restrictive, define a program class that always reaches a LCES (Theorem 1 in Section Theorem 1 of Section 4.3) if it does not deadlock, excluding nondeterministic behavior, and permits performance analysis through a novel approach: geometry. Therefore we lay the foundation for studying other parallel program classes with geometry.

We argue next that the assumptions are not unreasonable. Although the analysis is constrained to two processes (A1), a recent performance analysis of Lamport's mutual exclusion algorithm [7] using

Petri nets requires so much computation that its numeric solution is limited to only four processes. In addition, the initial solutions to other problems in parallel programming – such as shared memory mutual exclusion algorithms – were initially solved only for two processes. And one important tool used for parallel systems - queueing networks - started only with the ability to solve just one kind of queue (M/M/1) in isolation. Assumption A2.1 was addressed in the footnote earlier. Regarding A2.2, the assumption of constant timings is perhaps no more or less reasonable than the assumption of exponential timings required by the Markov chains underling some Petri net and queueing network models. In fact, one could view a (constant time) geometric model of the type used in this paper and an (exponential time) Markov process model as two extremes in modeling program behavior. A2.3, A2.4, and A3 could be relaxed by using the discussion in later sections on how to map a transition system to a geometric model as a guide to developing mappings for other classes of transition systems. For example, [1, pp. 41-44] illustrates how certain other synchronization constructs (CSP's [21] input and output commands) can be mapped to a geometric model.

Solution Method: The geometric solution method uses a program execution model called a *timed progress graphs* (TPG). A TPG adds timing information to Dijkstra's (untimed) progress graph (UPG), which Carson and Reynolds define as "a multidimensional, Cartesian graph in which the progress of each of a set of concurrent processes is measured along an independent time axis" [10]. A TPG represents synchronization between processes by lines, and a TES by a directed, continuous path that does not cross a line.

Dijkstra [12] devised UPGs. Later, Kung, Lipski, Papadimitriou, Soisalon-Soininen, Yannakakis, and Wood [24, 28, 36, 37] used UPGs to detect deadlocks in lock-based transaction systems. Carson and Reynolds [10] used UPGs to prove liveness properties in programs with an arbitrary number of processes containing P and V operations that are unconditionally executed. TPGs differ from UPGs in two ways: First, TPG's represent the time required for transitions, and can be used to derive performance properties of a program. UPG's represent timed transition diagram sets in which all times are equal. Second, UPG's are used to characterize deadlocks for any number of processes, any number of processors, and any scheduling discipline. In contrast, TPG's as defined here can be used to detect deadlock only

for two process programs executing on two dedicated processors. One final distinction between this paper and past work on UPGs is that we do not use a computational geometric algorithm to analyze the progress graph, as most solutions in the literature to UPGs employ, but rather we derive an analytic solution based on number theory. Combining Carson and Reynolds [10] work and this paper shows that one model can be used to verify liveness and analyze performance of semaphore programs.

Example: Dijkstra's dining philosophers problem with two philosophers exemplifies the class [13]. Two philosophers eating a meal share two chopsticks. A philosopher must wait upon attempting to acquire the chopsticks while the other is eating. Figure 1 shows a semaphore program solution along with the execution time required for each code segment in some time unit, excluding blocking. The `delay` statements cause Philosopher(1) to start execution 5 time units after Philosopher(0). Figure 2 represents the program by a set of two *timed transition diagrams.* (Timed transition diagrams are presented in Section 3.) Diagram vertices represent all possible code segments, and are labeled by consecutive integers starting at -1, for a reason that will become evident later, called *locations.* Location -1 represents a process that is waiting to start execution. Diagram edges are labeled above by a *time* t and below by a *condition* c, or a set of labels. The intended meaning is that the process remains in a location i for t time units; then the process makes a transition from location i to location $i+1 \bmod 2$ at the first instant when the location of the other process is not in set c. The program state is an ordered pair whose first (respectively, second) element denotes the location of Philosopher(0) (Philosopher(1)).

Figure 3 lists all possible states that the timed transition diagram set in Fig. 2 can reach. Any infinite length path through the graph, starting with initial state (-1,-1), represents a possible TES. The example TES listed earlier is one such path. All such paths in Fig. 3 contain the LCES listed earlier: states (1,0), (2,1), and (0,2) for 3, 1, and 1 time unit, respectively.

The LCES can be derived from the TPG illustrated in Fig. 4. The figure illustrates a finite portion of the TPG. (In all TPG illustrations, the grey grid is not part of the TPG and is present only to help the reader to determine graph coordinates.) The figure contains two directed paths that have in common the ray with endpoints (0,0) and (1,1). The paths illustrate all possible TESs for Fig. 2. Each path is called a *timed execution trajectory* (TET). Each coordinate of a TET point may be interpreted as a

```
Philosopher(i) {
  L:  Think;
      P(a);        /* acquire chopsticks; wait if not available */
      Eat;
      V(a);        /* release chopsticks */
      goto L;
}

semaphore a=1;
parbegin
  begin delay 0; Philosopher(0); end
  begin delay 5; Philosopher(1); end
parend
```

Code segment	*Graph location*	*Execution time required*	
		Philosopher(0)	*Philosopher(1)*
`delay ...`	-1	0	5
`Think; P(a);`	0	1	1
`Eat; V(a);`	1	3	1
`goto L;`	2	1	1

Figure 1: One semaphore dining philosophers program.

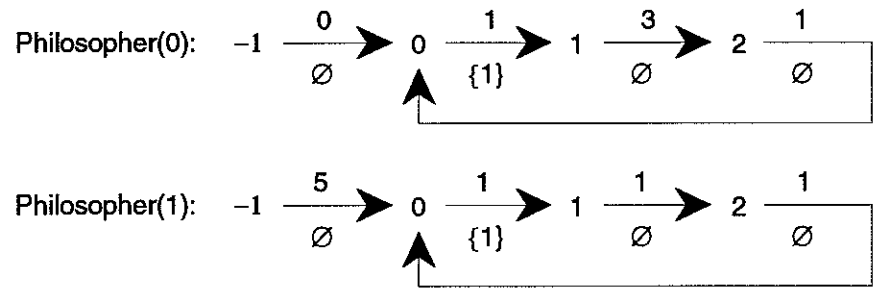


Figure 2: Timed transition diagrams corresponding to Figure 1.

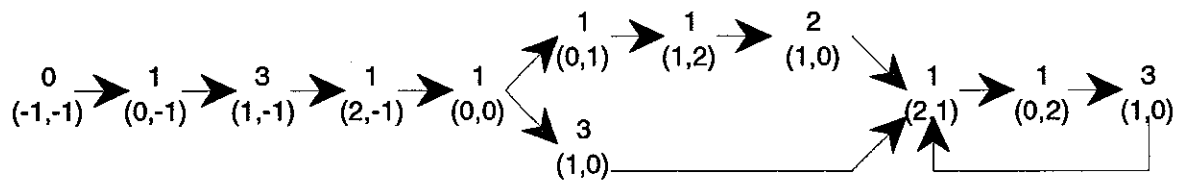


Figure 3: All possible states that can be reached in the timed transition diagram set (Fig. 2) solving the dining philosophers problem. The number above each state represents the time spent in each state.

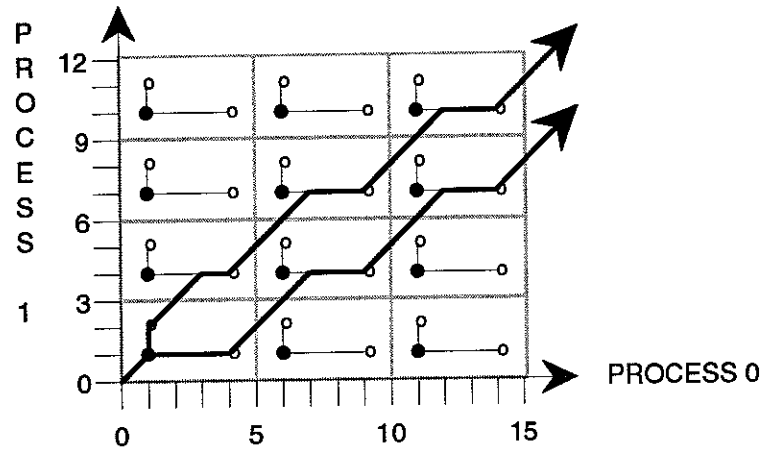


Figure 4: TPG corresponding to Fig. 2. Open (filled) circles represent open (closed) end points. Thick lines represent all possible TETs.

clock associated with the corresponding process that initially is enabled and has value zero, is disabled whenever the process to which it corresponds blocks, and is re-enabled whenever the process unblocks. For example, point (8,4) denotes that process 0 has run for 8 time units, and process 1 for 4 time unit. Thus process 1 has blocked for 8-4=4 time units more than process 0.

The TPG represents the suffix of a TES in which both processes are running. In Fig. 2, process 0 runs for 5 time units before process 1 starts. However when process 1 starts, both processes simultaneously enter location zero; hence the initial point of the TESs is (0,0).

To understand the mapping from transition diagram set to TPG, first consider what would happen if the P and V semaphore operations in Fig. 1 are deleted, so that the two processes never synchronize. Then process 0 is in location 0 during time interval [0,1), location 1 during [1,4), location 2 during [4,5), location 0 during [5,6), and so on. Therefore to find the location of process 0 corresponding to graph point (x, y), we calculate x mod t, where t is the sum of the delay in each location except -1 in the timed transition diagram of process 0; in this case $t = 1 + 3 + 1$. Therefore the location of process 0 is 0,1, or 2 depending on whether x mod 5 lies in interval [0,1), [1,4), or [4,5), respectively. A point that lies in an interval represents partial execution of the location: point (x, y) where $x = 7$ represents the case where

process 0 must remain in location 1 for 2 more time units, because $4 - (7 \bmod 5) = 2 \wedge 2 \in [1, 4)$. The TET representing the TES will be a diagonal ray with slope one, rooted at point (0,0).

Next consider the program as shown in Fig. 1, with semaphore operations. A P operation potentially blocks one process; therefore the TET portion representing a TES portion in which one process is blocked is a horizontal or vertical ray, depending on which process is blocked. The "L" shaped lines in the plane, called *constraint lines*, represent all possible situations in which one process is blocked at a P operation, waiting for another process to perform a V operation. An intuitive justification for the generator locations for the dining philosophers follows. Process 0 blocks iff it is about to enter location 1 and the current location of process 1 is 1. Process 0 unblocks when process 1 moves from location 1 to 2. Therefore process 0 blocks iff the TPG point representing its current state lies on a line congruent to $\overline{[(1,1),(1,2))}$. Similarly, process 1 blocks iff the point representing its current state lies on a line congruent to $\overline{[(1,1),(4,1))}$. A TET cannot cross a constraint line. A point that is the initial point of two constraint lines, such as (1,1), represents the race condition when both processes simultaneously attempt to perform a P operation.

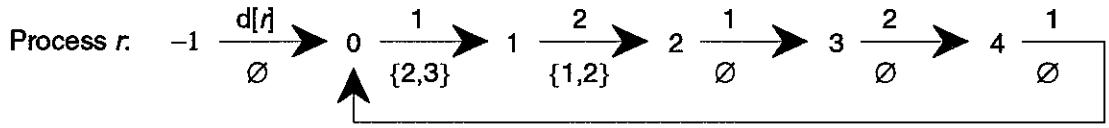
Figure 5 illustrates deadlock. The dining philosophers solution in Fig. 5(a) uses two semaphores; each process acquires the semaphores in the opposite order, which can cause deadlock for certain process starting times. In a TPG, a point that lies on two constraint lines, but is not the initial point of the two lines, represents a program deadlock. The point (3,3) in the TPG of Fig. 5(b) represents a deadlock. Figure 5(c) shows that certain TETs lead to deadlock the first time that the processes perform the P operations, while others do not. The different initial points of the four TETs in Fig. 5(c) illustrate four different relative starting times of the two processes. The points on line segments $\overline{((3,3),(4,3))}$ and $\overline{(3,3),(3,4)}$ do not lie on any TET, and hence are unreachable. Because UPGs have been used extensively for analysis of deadlocks [10, 24, 28, 36, 37], deadlocks are not considered further in this paper.

The paper is organized as follows. The following section compares the use of TPGs to find limit cycles to other techniques in the literature. Section 3 presents timed transition diagrams. Section 4 formally defines TPGs, TETs, and related terms. Sections 5 to 8 develop through stepwise refinement an algorithm that maps the geometric representation to a Diophantine equation, whose solution yields a

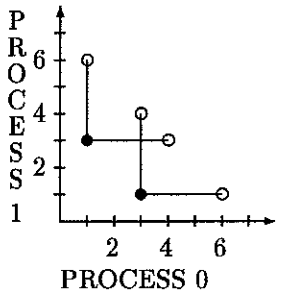
```
Philosopher(0) { L:Think; P(a); P(b); Eat; V(a); V(b); goto L; }
Philosopher(1) { L:Think; P(b); P(a); Eat; V(b); V(a); goto L; }

semaphore a=1,b=1;
integer array d[0..1];
parbegin
  begin delay d[0]; Philosopher(0); end
  begin delay d[1]; Philosopher(1); end
parend
```

Code segment	*Graph location*	*Execution time required (either philosopher)*
`Think; first P;`	0	1
`second P;`	1	2
`Eat; first V;`	2	1
`second V;`	3	2
`goto L;`	4	1

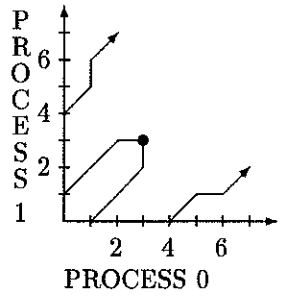
(a) Two semaphore dining philosophers program.



(b) Timed transition diagram for each process in (a)



(c) Deadlocking constraints



(d) Sample PETs from (c) with four values of G^C

Figure 5: Geometry of deadlock.

representation of the set of all blocking limit cycle execution sequences in a TPG.

2 Related Work

Performance goals are usually stated using measures, such as execution time, delay, and throughput. Measures are computing from more primitive quantities, for example the process synchronization structure and code segment timings used in this paper. The literature contains many approaches to deriving software performance measures. The approaches differ in their assumption about primitive quantities, such as code segment timings; they may be specified as constants, random variables, or lower and upper bounds. The approaches may compute exact, approximate (if the analysis requires one or more simplifying assumptions to make computation of the measure tractable), or lower and upper bounds on the measures. Deriving measures from code before execution can be done by analysis, simulation, or logic.

Analysis: Analytic methods include complexity analysis (e.g., [6]), micro-analysis using difference equations (e.g.,[20]), Petri nets (e.g.,[7, 22]), stochastic processes (e.g.,[15]), stochastic automata [29], queueing networks (e.g.,[17, 34]), and analysis of graphs whose nodes represent code segments (e.g.,[35]).

Of these works, the closest is 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 [30] consider 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 [25] 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 [26] 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 [27] extends his earlier paper [26] by showing 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. The problem considered in this paper, of processes synchronizing only to achieve mutual exclusion, cannot be represented by decision-free or persistent Petri nets. The dining philosophers problem has been analyzed by Holliday and Vernon [22] 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, TPGs have two advantages. First, TPGs yield the *exact* limit cycle 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. Second, TPGs 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. A disadvantage of the TPG solution presented here is that it is limited to two processes, while Petri nets solutions have no such limitation.

Simulation: Simulation explicitly generates a TES by simulation uses the definition of program execution, along with the initial state, to step through an initial subsequence of the execution sequence. Simulation has three drawbacks. First, it 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 a TES, such as a LCES, may not be revealed.

Logic: Certain methods for formally reasoning about programs may be applied to reason about performance measures. Henzinger, Manna, and Pnueli [19] propose a proof system for real time systems uses lower and upper bounds on the times of transitions in a timed transition system. The proof system allows proof of properties about TESs, without explicitly deriving them, as the method used in this paper does. Shaw [32] extends Hoare logic to reason about time in software. Ramshaw [31] generates

recurrence equations about program timings using Hoare style axioms.

3 Timed Transition Diagrams

Our model of a program is based on Henzinger, Manna, and Pnueli's timed transition diagrams [18].[2] Time is represented by nonnegative real numbers. Let R, R^+, and Z denote, respectively, the set of nonnegative reals, positive reals, and nonnegative integers. We assume that $\langle \forall n : n \in Z :: n \leq \infty \rangle$[3] to simplify our notation. Let $r \in \{0,1\}$ denote one of two processes, $\bar{r} = 1 - r$ (if $r = 0$ then $\bar{r} = 1$ and vice versa), and $\langle \forall r :: n_r \rangle$ denote finite, positive integers.

Each process r is represented by a finite, connected, directed graph containing $n_r + 1$ vertices, each labeled by an integer in $\{-1, 0, 1, \ldots, n_r - 1\}$. Each integer label is called a *location*. Each vertex has exactly one outgoing edge; therefore the graph contains one cycle. The program uses variables $\{\pi_0, \pi_1\}$, which are shared by all processes; π_r denotes the control point of process r. Each edge with initial vertex i, for $-1 \leq i < n_r$, is labeled by a *delay* t_r^i and a *condition* c_r^i. Delays satisfy $\langle \exists r :: t_r^{-1} = 0 \wedge t_{\bar{r}}^{-1} \in R \rangle \wedge \langle \forall t_r^i : i \geq 0 :: t_r^i \in R^+ \rangle$. Conditions in the graph of process r, name labels of process $\bar{r}$, satisfying $c_r^{-1} = \emptyset \wedge \langle \forall v : v \in c_r^i :: v \in \{0, 1, \ldots, n_{\bar{r}} - 1\} \rangle$. Given two labels i and i' of process r, $i \oplus i'$ (respectively, $i \ominus i'$) denotes addition (subtraction) modulo n_r. The intended operational meaning of edge $-1 \xrightarrow[\emptyset?]{t_r^{-1}} 0$ is that process r waits for t_r^{-1} time units before it starts execution. The intended operational meaning of edge $i \xrightarrow[c_r^i?]{t_r^i} i \oplus 1$, for $i \geq 0$, is that if, for exactly t_r^i time units, control of process r has resided at vertex i, then control will move to vertex $i \oplus 1$ at the earliest time at which the control point of process $\bar{r}$ is not a label in c_r^i. Process r is *blocked* whenever it has remained in its current location, i, for longer than t_r^i time units. After process r starts execution, it is *running* whenever it is not blocked. Figure 2 and Fig. 5(b) each illustrate a set of two timed transition diagrams.

Timed transition diagrams represent semaphore programs as follows. Let $sem_0, sem_1, \ldots, sem_{n_s-1}$

[2] Our diagrams fundamentally differ from those of Henzinger, Manna, and Pnueli (HMP) in that the condition labeling an edge need not hold until the control point of the process has resided at the initial vertex of the edge for the time labeling the edge, whereas in the HMP diagrams, the condition must be continuously true for the time labeling the edge. Otherwise our diagrams are a special case of HMP diagrams because we allow no program variables, we limit edge conditions to a test for set membership, we require the minimal and maximal delays labeling an edge to be equal, and we require all but one graph nodes to be contained in a single cycle.

[3] We use the notation from [11] ⟨ <quantified variable> : <domain of quantification> :: <quantified formula> ⟩, and omit <domain of quantification> when it is obvious from context.

denote the semaphores used by a program. Each graph edge out of vertex $i \geq 0$ corresponds to at most one P or V operation. We say that a process *holds* semaphore sem_j (where $0 \leq j < n_s$) in locations $\hat{i}, \hat{i}+1, \ldots, \hat{i}+k$ if edge $\hat{i}-1$ corresponds to a $P(sem_j)$ operation, edge $\hat{i}+k$ corresponds to a $V(sem_j)$ operation, and none of the edges in $\{\hat{i}, \hat{i}+1, \ldots, \hat{i}+k-1\}$ correspond to a semaphore operation on sem_j. Let $S_r^{sem_j}$ denote the set of locations in process r that hold semaphore sem_j; formally $\{\hat{i}, \hat{i}+1, \ldots, \hat{i}+k\} \subseteq S_r^{sem_j}$. If the edge out of vertex i in process r corresponds to operation $P(sem_j)$, then c_r^i is the set of all locations in the diagram for process $\bar{r}$ in which process $\bar{r}$ holds semaphore sem_j. Formally, if the edge out of a location i corresponds to operation $P(sem_j)$, then $c_r^i = S_{\bar{r}}^{sem_j}$; otherwise $c_r^i = \emptyset$.

The set of timed transition diagrams representing a parallel program are denoted by enumerating, for each diagram, the number of vertices and the times and conditions labeling each edge. Formally, a timed transition diagram set $D = \langle \Psi_0, \Psi_1 \rangle$, where $\langle \forall r :: \Psi_r = \langle\, n_r,\, \langle t_r^{-1}, t_r^0, t_r^1, \ldots, t_r^{n_r-1} \rangle,\, \langle c_r^0, c_r^1, \ldots, c_r^{n_r-1} \rangle \,\rangle \rangle$.

Example 1 For Fig. 2, $\psi_0 = \langle\, 3,\, \langle 0,1,3,1 \rangle,\, \langle \{1\}, \emptyset, \emptyset \rangle \,\rangle$ and $\psi_1 = \langle\, 3,\, \langle 5,1,1,1 \rangle,\, \langle \{1\}, \emptyset, \emptyset \rangle \,\rangle$. □

4 Timed Process Graphs

4.1 Notation

The following notation is used throughout the paper. Let uppercase letters with optional superscripts denote graph points (e.g., G^0). Let the subscripts 0 and 1 denote the components of a point (e.g., $G^0 = (G_0^0, G_1^0)$). Unless otherwise noted, every point $G \in R^2 - \{(\infty, \infty)\}$. We assume that $\gamma.i \neq \gamma.f$. For any two points G and G', $G < G' \stackrel{def}{=} G_0 + G_1 < G_0' + G_1'$. 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. For any two continuous paths γ and γ' in R^2, $\gamma \cap \gamma'$ denotes $\{G \mid G \in \gamma \wedge G \in \gamma'\}$. A line segment with open end point G and closed endpoint G' is denoted either by $L = \overline{[G, G')}$. If both end points are open, $L = \overline{(G, G')}$. A ray, which is a directed line segment uses the same notation. 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.

We define the *cycle time* of process r, denoted ϕ_r, as the time required for process r to pass through each location once, ignoring the time spent blocked. Formally, $\langle \forall r :: \phi_r \stackrel{def}{=} \sum_{0 \leq i < n_r} t_r^i \rangle$. For any point

$G \in R^2$, $mod(G) \stackrel{def}{=} (G_0 \bmod \phi_0, G_1 \bmod \phi_1)$. Two points G and G' are *congruent*, denoted $G \equiv G'$, iff $mod(G) = mod(G')$. Two continuous paths γ and γ' are *congruent*, denoted $\gamma \equiv \gamma'$, iff there exists a one-to-one correspondence between their points such that corresponding points are congruent. For any line or ray $L = \overline{[G, G')}$, $mod(L)$ denotes $\overline{[mod(G), mod(G'))}$.

4.2 Definitions

A *TPG* $\Gamma_D = \langle \Phi, \Lambda, G^C, f \rangle$, corresponding to a timed transition diagram set $D = \langle \psi_0, \psi_1 \rangle$, consists of:

Φ: a set containing cycle times ϕ_0 and ϕ_1.

Λ: a set of *constraint line generators*, which are a set of line segments that lie in the R^2 plane in the rectangle with opposite vertices (0,0) and (ϕ_0, ϕ_1), each corresponding to one edge in one transition diagram labeled by a non-empty condition. For any condition $c_r^k = \{\hat{i}, \hat{i}+1, \ldots, \hat{i}+z\}$, for some $i', z \in Z$, we write $\langle \hat{i}, \hat{i}+z \rangle \in c_r^k$. Formally, generator $\overline{[W, X)} \in \Lambda$ iff

$$\langle \exists r \exists k \exists i \exists i' : 0 \le k < n_r, :: \langle i, i' \rangle \in c_r^k \wedge$$

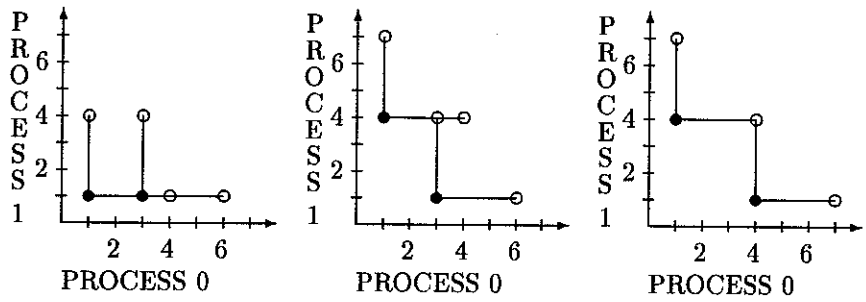
$$W_r = X_r = \sum_{0 \le j \le k} t_r^j \wedge W_{\bar{r}} = \sum_{0 \le j < i} t_{\bar{r}}^j \wedge X_{\bar{r}} = \sum_{0 \le j \le i'} t_{\bar{r}}^j \rangle.$$

The *instances* of a constraint line generator are defined to be all lines in the R^2 plane congruent to the generator. The set of all instances of all constraint lines in Λ is $\Lambda^* \stackrel{def}{=} \{\overline{[W, X)} \mid W \in R^2 \wedge X \in R^2 \wedge mod(\overline{[W, X)}) \in \Lambda\}$.

G^C: an *initial point* representing the earliest instance at which both processes have started execution. The initial point lies on an axis within one cycle time of the origin and represents the times at which each process starts exection. Formally, G^C satisfies $\langle \exists r : t_r^{-1} = 0 :: G_r^C = t_r^{-1} \bmod \phi_r \wedge G_{\bar{r}}^C = 0 \rangle$.

f: a *transition function*, which maps each point $G \in R^2 - \{(\infty, \infty)\}$ to a (possibly empty) set of successors $f(G) \subseteq R^2$, where $||f(G)|| \le 2$. The definition of f is stated in terms of two more primitive functions, f_1 and f_2: $f(G) \stackrel{def}{=} f_1(G) \cup f_2(G)$. Informally,

$\hat{G} \in f_1(G)$ iff G lies on a constraint line instance L and $\hat{G}$ is the smallest point on line L, excluding $L.i$, at which L intersects another constraint line instance and $\hat{G} \neq L.f$, or, if there is no such intersection point, $\hat{G} = L.f$; and



(a) Process one does simultaneous P ops

(b) Process one does simultaneous P, V

(c) Both processes do simultaneous P, V

Figure 6: Illegal constraint line geometries.

$\hat{G} \in f_2(G)$ iff G does not lie on a constraint line instance and $\hat{G}$ is the smallest point at which a slope one ray rooted at G intersects a constraint line instance, or, if there is no such intersection point, $\hat{G} = (\infty, \infty)$.

To formally define f_1 and f_2, let $S_1(G, L)$, where $G \in L$, be the smallest point in set $\{L.f\} \cup \{G' | G' \geq G \wedge \langle \exists L' : L' \in \Lambda^* \wedge L \neq L' :: G' \in (\overline{(L.i, L.f)} \cap L') \rangle\}$. Let $S_2(G)$ be the smallest point in set $\{(\infty, \infty)\} \cup \{G' \mid \langle \exists L' : L' \in \Lambda^* \wedge L \neq L' :: G' \in \overline{[G, (\infty, \infty))} \cap L'\rangle$. Then

$\hat{G} \in f_1(G)$ iff $\langle \exists L : L \in \Lambda^* \wedge G \in L :: \hat{G} = S_1(G, L) \wedge G \neq \hat{G} \rangle$, and

$\hat{G} \in f_2(G)$ iff $\langle \nexists L : L \in \Lambda^* \wedge G \in L \rangle \wedge \hat{G} = S_2(G)$.

Example 2 In Fig. 4, $\Gamma = \langle \{\phi_0 = 5, \phi_1 = 3\}, \Lambda, (0,0), f \rangle$, where $\Lambda = \{ \overline{[(1,1),(1,2))}, \overline{[(1,1),(4,1))} \}$. One illustration of the composition of set Λ is the following: Because process 0 performs $P(a)$ and process 1 holds a in location 1, $c_0^0 = \{1\}$ and $\langle 1, 1 \rangle \in c_0^0$. This in turn implies that $W_0 = X_0 = \sum_{0 \leq j \leq 0} t_0^j = 1$, $W_1 = \sum_{0 \leq j < 1} t_1^j = 1$, and $X_1 = \sum_{0 \leq j \leq 1+0} t_1^j = 2$. Thus generator $\overline{[(1,1),(1,2))} \in \Lambda$. The figure illustrates twelve instances of each constraint line generator. □

Recall that each timed transition diagram edge t_r^i (for $0 \leq i < n_r$) with a non-empty condition must have a positive delay and correspond to at most one P or V operation. Therefore no constraint lines may overlap (see Fig. 6(a)), the final point of one constraint line can never lie on another constraint line (see Fig. 6(b)), and the final point of all constraint lines must be distinct (see Fig. 6(c)).

A point is *nondeterministic* iff the transition out of the point is not unique. All other points are *deterministic*. A point is *dead* iff there is no transition (in the sense of function f) out of the point.

Definition. Consider a point $G \in R^2 - (\infty, \infty)$. G is nondeterministic, denoted $C_N(G)$, iff $||f(G)|| > 1$. G is *dead*, denoted $C_D(G)$, iff $f(G) = \emptyset$.

In Fig. 5, (3,3) is dead and points (1,3) and (3,1) are nondeterministic. Informally, nondeterministic points represent states in which both processes simultaneously perform a P operation on the same semaphore. Dead points represents states in which both processes are blocked

"Execution" of a program is represented by a timed execution trajectory (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.

Definition. *A timed execution trajectory of a TPG Γ 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 $\overline{[G^0, G^1)}$, $\overline{[G^1, G^2)}$, ..., $\overline{[G^{n-1}, G^n)}$ (where n may be infinite) satisfying $\langle \forall i : 0 \leq i < n :: G^{i+1} \in f(G^i) \rangle$.*

Example 3 The heavy lines in Fig. 4 denote all possible TETs rooted at point (0,0) for the single philosopher dining philosophers example: $\overline{[(0,0),(1,1))}$, $\overline{[(1,1),(4,1))}$, $\overline{[(4,1),(7,4))}$, $\overline{[(7,4),(9,4))}$, ..., and $\overline{[(0,0),(1,1))}$, $\overline{[(1,1),(1,2))}$, $\overline{[(1,2),(3,4))}$, $\overline{[(3,4),(4,4))}$, $\overline{[(4,4),(7,7))}$, ... The point (0.75, 0.75) represents a remaining occupancy time of 0.25 time units for both processes in location zero. Each line segment $\overline{[G, G')}$ satisfies $G' \in f(G)$. Point $(1,1) \in f_2(0,0)$ because (0,0) does not lie on a constraint line instance, and a slope one ray rooted at (0,0) first intersects a constraint line instance at point (1,1). As a second example, Fig. 5(b) illustrates four TETs corresponding to four different choices for `d[0]` and `d[1]`: 4,0; 1,0; 0,1; and 0,4. Consider `d[0]=0` and `d[1]=4`. Process 0 starts first, and runs for four time units before process 1 starts. The initial ray of the corresponding TET should have length 4 and lies on the process 0 axis. Therefore the point G^C, representing the earliest time at which both processes are running, is (4,0). In general, G^C = (`d[1]`, `d[0]`). Point (3,1) is dead. Whereas all TETs in Fig. 4 are of infinite length, a TET with initial point (0,1) in Fig. 5 has finite length because $f_2(0,1) = \{(2,3)\}$,

$f_1(2,3) = \{(3,3)\}$, and $f(3,3) = \emptyset$. □

In Fig. 4, the TETs rooted at $(3,1)$, $(3,4)$, and $(8,4)$ are congruent. That is, adding the vector $(0, \phi_1)$ to all points on the trajectory rooted at $(3,1)$ yields the trajectory rooted at $(3,4)$. Furthermore, adding vector (ϕ_0, ϕ_1) to the trajectory rooted at $(3,1)$ yields the trajectory rooted at $(8,4)$. In general, TETs rooted at congruent points are congruent, as the following Lemma establishes.

Lemma 1 *Consider any two TETs γ and γ' of a timed progress graph Γ with initial points G and G', respectively. If $G \equiv G' \wedge \langle \nexists \hat{G} : \hat{G} \in \gamma \vee \hat{G} \in \gamma' :: C_N(\hat{G}) \rangle$ then $\gamma \equiv \gamma'$.*

Proof: See [2]. □

4.3 Transient and Limit Cycle Execution Trajectories

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

Definition. *Consider a TET γ. A directed, continuous path $\hat{\gamma}$ is a* limit cycle execution trajectory *(LCET) of γ iff $\langle \forall G : G \in \hat{\gamma} :: G \in \gamma \wedge \neg C_N(G) \rangle \wedge \hat{\gamma}.i \equiv \hat{\gamma}.f$. $\hat{\gamma}.i$ and $\hat{\gamma}.f$ are called the* initial *and* final points of the LCET, *respectively. The* transient execution trajectory *is the portion of γ consisting of all points that do not lie on a LCET. The* initial point of the transient execution trajectory *is $\gamma.i$. The* final point of the transient execution trajectory *is the smallest point of γ lying on any LCET, if the TET contains a LCET.*

Example 4 Figure 4 contains two TETs. The TET containing point $(1,2)$ contains a transient execution trajectory with initial and final points $(0,0)$ and $(3,4)$, respectively, followed by an infinite number of congruent LCETs. One is $\overline{[(3,1),(4,1))}$, $\overline{[(4,1),(7,4))}$, $\overline{[(7,4),(8,4))}$; another, congruent LCET is $\overline{[(4,4),(7,7))}$, $\overline{[(7,7),(8,7))}$, $\overline{[(8,7),(9,7))}$. The two are congruent because the first, second, and third rays of the first subtrajectory are congruent to the third, first, and second rays of the second subtrajectory, respectively. □

The following theorem, along with the equivalence of transition systems and TPGs, implies that any TES that does not contain a dead point and contains a finite (possibly zero) number of nondeterministic points reaches a limit cycle.

Theorem 1 *A TET γ in a timed progress graph consists of a transient execution trajectory followed by an infinite number of congruent LCETs iff* $||\{G|G \in \gamma \wedge C_N(G)\}|| < \infty$.

Proof: See [2]. □

4.4 Timed Progress Graph Sets

The remainder of this paper solves for the set of LCETs in a TPG. However, the solution method is more powerful than solving a single TPG: it yields the the set of *all possible* limit cycle execution trajectories for *any* initial condition in the corresponding timed transition diagram set (e.g., any process starting times t_0^{-1} and t_1^{-1} satisfying the conditions stated in Section 3). Therefore we henceforth consider a *timed progress graph set* (TPGS), which has three components: $\langle \Phi, \Lambda, f \rangle$. A point or a directed, continuous path γ is a TET in $\langle \Phi, \Lambda, f \rangle$ iff $\langle \exists r, \exists G^C : G_r^C = 0 \wedge 0 \leq G_{\bar{r}}^C < \phi_{\bar{r}} :: \gamma$ is a TET in $\langle \Phi, \Lambda, G^C, f \rangle \rangle$. Given a timed progress graph Γ, the timed progress graph that subsumes it is denoted Γ^+.

Example 5 Figure 7 shows a TPGS $\Gamma^+ = \langle \Phi, \Lambda, f \rangle$ for a program where $C = \{\phi_0 = \phi_1 = 10\}$ and Λ represents five semaphores. If we consider any initial condition, the program corresponding to the TPG portion shown in Fig. 7 may reach a nondeterministic state, a dead state, or a limit cycle execution sequence as indicated in Table 1. (The terms "blocking" and "non-blocking" in Table 1 are defined in Section 5.) The set of TETs in Γ^+ is shown by heavy lines and, in regions containing an infinite number of TET rays, shading. Two portions of the shaded polygon are of infinite extent, indicated by the arrows at edges $\overline{[(22,30),(23,30)]}$, and $\overline{[(30,27),(30,28))]}$. □

5 Solving for TETs in TPG Sets

This section is the first of three sections that develops, through stepwise refinement, an algorithm that outputs a representation of the set of LCETs in a TPGS. The algorithm exploits the fact that many

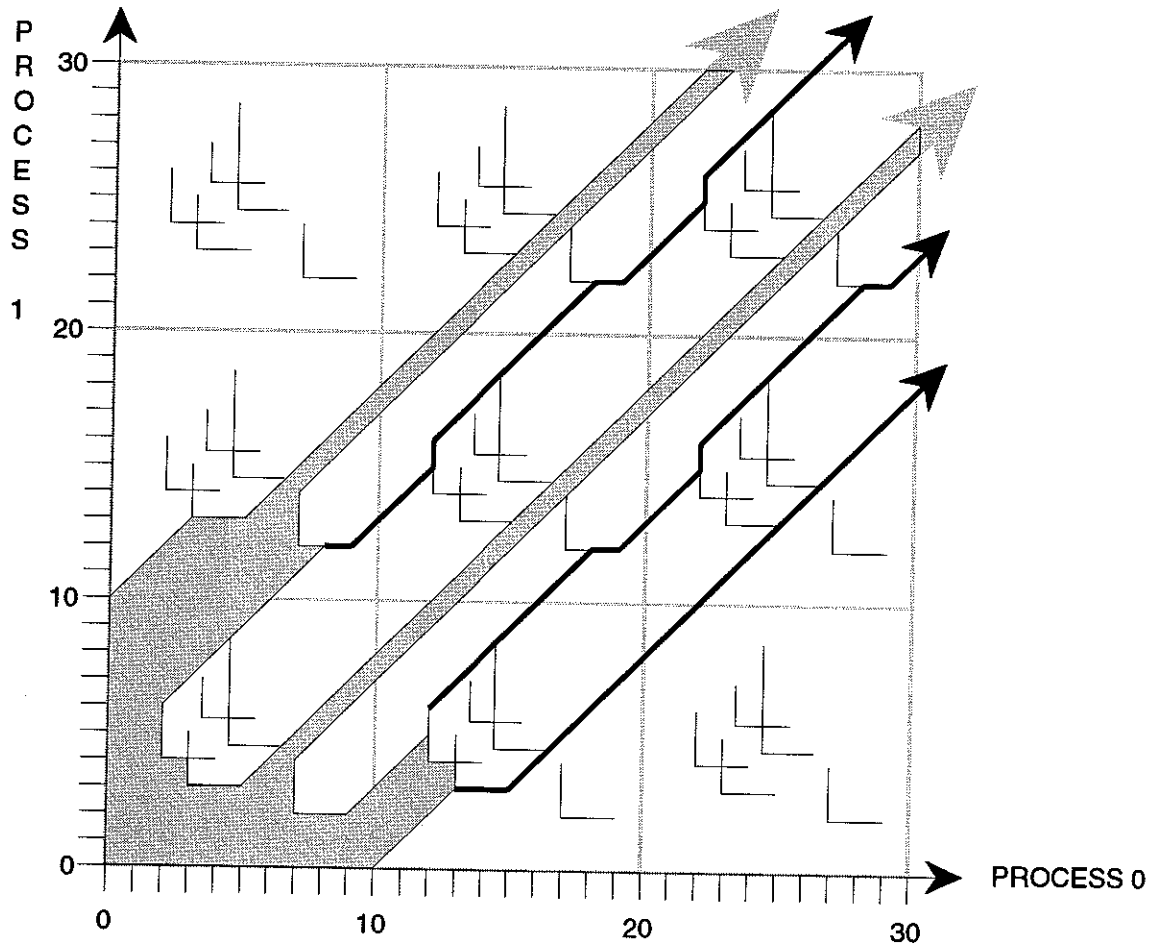


Figure 7: Illustration of a TPGS.

LCETs are congruent. Therefore the problem of representing all possible LCETs reduces to finding one member of each congruence class of LCETs.

LCETs may be characterized on the basis of whether they do or do not block. A *blocking* LCET contains at least one nondiagonal ray, while a *nonblocking* LCET consists of a single slope one ray. The explicit form of a blocking LCET reveals the sequence and duration of waiting times that each process encounters. This is of interest because some blocking LCETs require a process to wait for longer periods than others. For example, by Example 4 and Fig. 4, philosopher 0 waits two time units (e.g., because each

For initial point:	TET contains:
on $\overline{((0,5),(0,10))}$	non-blocking limit cycle
$(0,5)$	nondeterministic point at $(7,12)$
on $\overline{((0,2),(0,5))}$	blocking limit cycle
$(0,2)$	nondeterministic point at $(2,4)$
on $\overline{((0,0),(0,2))}$	dead state $(3,4)$
$(0,0)$	nondeterministic point at $(3,3)$
on $\overline{((0,0),(5,0))}$	non-blocking limit cycle
$(5,0)$	nondeterministic point at $(7,2)$
on $\overline{((5,0),(8,0))}$	blocking limit cycle
$(8,0)$	nondeterministic point at $(12,4)$
on $\overline{((8,0),(10,0))}$	dead point $(13,4)$

Table 1: Behaviors present in Fig. 7 TPGS.

LCET contains a line congruent to line $\overline{[(7,7),(9,7))}$, which is parallel to the process 0 axis) at each P operation, and philosopher 1 never blocks at a P operation. In contrast, the explicit form of a nonblocking LCET, representing only running states and hence no blocking, reveals little information. The number of congruence classes of non-blocking limit cycles is at most infinite; and by the corollary to Theorem 4 below, the number of blocking limit cycle execution sequences is at most twice the number of semaphores. Therefore a "solution" to a TPGS (1) reports whether any initial condition can lead to a TET containing a dead state, an infinite number of nondeterministic states, a blocking limit cycle execution sequence, or a non-blocking limit cycle sequence,[4] and (2) reports one member of each congruence class of blocking LCETs. Presented in algorithm A0 below is a solution to (1); (2) is left as an open problem.

The algorithm uses the following notation. For any point G, if $f(G) = G' \wedge ||f(G)|| = 1$, we use $f(G)$ to denote not only a function whose domain is a set (e.g., $\{G'\}$), but also a function whose range is a point: $f(G) = G'$. Further, if $||f(G)|| = ||f(f(G))|| = ||f(f(f(G)))|| = \ldots = 1$, then the i-fold (for $i \geq 0$) composition of function f is denoted $f^i(G)$, and $f^0(G) \stackrel{def}{=} G$. For convenience, define a function Δf as follows: if $||f(P)|| = 1$ then $\Delta f(G) \stackrel{def}{=} f(G) - G$; $\Delta f(G)$ is a vector representing the transition from the state represented by G to the state represented by $f(G)$.

[4] Carson and Reynolds' deadlock detection algorithm [10] cannot directly be applied, because a consequence of the assumption in Section 3 that each process executes on a dedicated processor is that the set of all execution trajectories in Carson and Reynolds' UPG is a superset of all execution trajectories in a TPGSs.

{ Input: $\langle \Phi, \Lambda, f \rangle$; Output: list of LCETs }
declare S: set of points; { Set of points examined already for membership in a LCET }
$S := \emptyset$; $N := ||\Lambda||$; $\xi := \{ X \mid \overline{[W, X)} \in \Lambda \}$;
for each X **in** ξ **do**
if
$\langle \nexists x : x \in S :: x \equiv X \rangle \ \wedge \ ||f^0(X)|| = ||f^1(X)|| = \ldots = ||f^{2N}(X)|| = 1 \ \wedge$
$\langle \exists m : m \in \{1, 2, \ldots, N\} :: f^{2m}(X) \equiv X \rangle$
then begin
output point X and state transition vector sequence
$\Delta f(f^0(X))$, $\Delta f(f^1(X))$, $\ldots$, $\Delta f(f^{2n-1}(X))$
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 8: Algorithm A0, which outputs one member of each congruence class of blocking LCETs.

5.1 Algorithm A0

Algorithm A0 (Fig. 8) examines N points, where N is the number of constraint line generators, namely the final point of each constraint line generator (set ξ). 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 LCET by finding n, the smallest integer in $\{0, 1, \ldots, N-1\}$ satisfying $f^{2i}(X) \equiv X$ for each $X \in \xi$, if such an n exists. If n exists, then X does lie on a LCET that is then output, and X and all other end points of non-collinear rays comprising the LCET rooted at X are added to S; otherwise only X is added to S. A0 does not explain how to compute $\langle \forall X : X \in \xi :: f(X) \rangle$; this topic is addressed in Sections 6 and 7.

Example 6 The TPGS of Fig. 4 contains two constraint line generators, as stated in Example 2. Thus $\xi = \{(4,1), (1,2)\}$. From Table 2, for $X = (4,1)$, $f^2(X) \equiv X$. However, for $X = (1,2)$, $\langle \nexists m : m \in \{1, 2, \ldots, N\} :: f^{2m}(X) \equiv X \rangle$, which is because the TET rooted at (1,2) blocks once in its transient execution trajectory at 0, but never again blocks at 0. Hence a single trajectory, congruent to those in Example 4, is output: point $X = (4,1)$ and vector sequence $(3,3)$, $(2,0)$. □

5.2 Correctness of Algorithm A0

Theorems 3 through 6 establish the correctness of algorithm A0. Appendix A contains omitted proofs.

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

Table 2: Quantities required by Algorithm A0 for Fig. 4.

Lemma 2 $||\Lambda|| < \infty$.

Proof: The definition of timed transition diagrams requires $\langle \forall r :: n_r < \infty \rangle$. Therefore $\langle \forall i \forall r : 0 \leq i < n_r :: ||c_r^i|| < \infty \rangle$. Thus there exist a finite number of integers i and i' satisfying $\langle i, i' \rangle \in c_r^i$. □

Theorem 2 *Algorithm A0 terminates if each evaluation of function f terminates.*

Proof: By Lemma 2, $||\xi||$ and hence N is finite. Hence all quantifications are over finite sets and the **for** loop iterates a finite number of times.

□

Theorem 3 *Every trajectory output by algorithm A0 is a blocking LCET.*

The following theorem establishes that examining the N points in set ξ is sufficient to find all congruence classes of blocking LCETs. The intuitive justification follows. Two TETs, rooted at G' and G'' *converge* iff the TETs contain a congruent point. Given that TETs with initial points G' and G'' converge, if we delete from each TET the subtrajectory with initial point equal to G' or G'' and a final, congruent point, then the remaining trajectories are congruent by Lemma 1. For example, in Fig. 7 all execution trajectories with initial points on line $\overline{((0,2),(0,5))}$ converge because they all contain point (8,12). Deleting the shaded polygon with vertices (0,2), (2,4), (2,6), (8,12), (7,12), and (0,5) yields a single trajectory common to all these TETs rooted at (8,12). An implication of convergence is that all TETs in a TPGS that contain the final point of an instance of a constraint line generator must converge. In addition, by definition, a blocking LCET must contain the final point of a constraint line. Therefore examining just the final points of all generators in Λ is sufficient to find all blocking LCETs.

Theorem 4 *Any blocking LCET is congruent to one of the trajectories output by algorithm A0.*

Corollary to Theorem 4 *There are at most* $||\Lambda||$ *congruence classes of blocking LCETs in a TPGS.*

Proof: By Theorem 4 the number of congruence classes of blocking LCETs cannot exceed the number of constraint line generators, or, $||\Lambda||$. □

Theorem 5 *None of the trajectories output by algorithm A0 are congruent.*

The following theorem shows that for each trajectory output by A0, either that trajectory or some congruent trajectory is reachable, meaning it occurs in some TET that exists in a TPGS.

Theorem 6 *For each trajectory output by A0, either that trajectory or some congruent trajectory is contained in some TET rooted at a point either on line* $\overline{[(0,0),(\phi_0,0))}$ *or on line* $\overline{[(0,0),(0,\phi_1))}$.

6 First Refinement: Algorithm A1

Algorithm A0 leaves unanswered three questions:

1. When are $\langle \forall X, : \overline{[W,X)} \in \Lambda :: f^0(X), f^1(X), \ldots, f^{2N}(X) \rangle$ defined?
2. How can $\langle \forall X : \overline{[W,X)} \in \Lambda :: f^0(X), f^1(X), \ldots, f^{2N}(X) \rangle$ be computed?
3. Algorithm A0 evaluates f at worst $O(N^2)$ times; does a more efficient algorithm exist?

Questions 1 and 3 are addressed in algorithm A1 (Fig. 9), which is the first of two refinements of A0. Algorithm A1 reformulates question 1 above in terms of two predicates, C_L and C_R, defined below. A method to efficiently decide when the predicates hold is given in Section 7. A1 transforms the problem of finding solutions to $\langle \forall X : i = \{1, 2, \ldots, N\} :: f^{2i}(X) \equiv X \rangle$ to a problem of finding cycles in a graph of N nodes, which reduces the worst case number of evaluations of f from $O(N^2)$ to $O(N)$. (This will be proven as a property of the second refinement, A2, in Section 8.) Answering question 3 appears to be non-trivial, and is the subject of Section 7 and the second refinement, algorithm A2.

We answer question 1 using the notion of a *live* and *restricted* point. Informally, a point G in a TPG is live, denoted $C_L(G)$, if G represents some deterministic and blocked state, called a live state, in which exactly one process is blocked (call it r), and in all TESs containing the state, in some subsequent state process r is running. (If in all subsequent states process r is blocked, then the TES would be of finite length and its final state would be dead; hence the name "live.") Formally, $C_L(G) \stackrel{def}{=} \langle \exists L :$

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

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

Figure 9: Algorithm A1, a refinement of algorithm A0.

$L \in \Lambda^* :: G \in L \wedge S_1(G, L) = L.f\rangle$. A point G is restricted, denoted $C_R(G)$, if G represents some deterministic state, called a restricted state, in which all processes are running, and in all TESs containing the state, in some subsequent state some process is block. (A diagonal ray rooted at a restricted point intersects a constraint line and thus cannot have infinite length; hence the name "restricted.") Formally, $C_R(G) \stackrel{def}{=} \langle \nexists L : L \in \Lambda^* :: G \in L\rangle \wedge S_2(G) \neq (\infty, \infty)$.

By the following lemma, question 1 is equivalent to $\langle \forall i : 0 \leq i < 2N :: C_L(f^i(X)) \vee C_R(f^i(X))\rangle$.

Lemma 3 $f(G) \neq \emptyset \wedge f(G) \neq (\infty, \infty)$ iff $C_L(G) \vee C_R(G)$.

Proof: Follows from the definition of f. □

Algorithm A1 uses the following notation. Let the N constraint line generators be denoted $\overline{[W^0, X^0)}$, $\overline{[W^1, X^1)}$, $\ldots$, $\overline{[W^{N-1}, X^{N-1})}$. Let k and k' denote integers in interval $[0, N)$.

Example 7 Recall that the TPG of Fig. 4 contains two constraint line generators. Let $\overline{[W^0, X^0)} = ((1,1),(4,1))$ and $\overline{[W^1, X^1)} = ((1,1),(1,2))$. Step A1.1 of algorithm A1 yields a graph of two nodes, with edges directed from both nodes zero and one towards node zero. Step A1.2 outputs point $X^0 = (4,1)$ and state transition vector sequence $(3,3),(2,0)$, which is identical to the result of algorithm A0 in Example 6. □

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

Let X represent the final point of a constraint line generator. Algorithm A1 contains four unknown quantities: $C_R(X)$, $C_L(f(X))$ if $C_R(X)$, $\Delta f(X)$ if $C_R(X)$, and $\Delta f(f(X))$ if $C_R(X) \wedge C_L(f(X))$. To simplify the presentation, we will reason about scalars rather than vectors, and hence define $g(G)$ to be

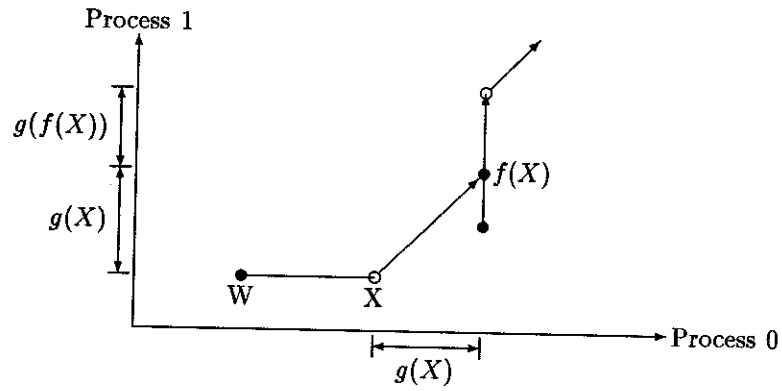


Figure 10: Illustration of four unknowns in Algorithm A1: $C_R(X)$, $C_L(f(X))$ if $C_R(X)$, $g(X)$ if $C_R(X)$, and $g(f(X))$ if $C_R(X)$ and $C_L(f(X))$.

a non-zero component of $\Delta f(G)$. ($\Delta f(G)$ corresponds to a horizontal, vertical, or diagonal ray. In the first two cases, exactly one vector component is nonzero; in the last case both components are non-zero and equal.) This notation is illustrated in Fig. 10.

The interpretation of the unknowns in a TPGS and a timed transition system are reviewed below:

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

$C_L(f(X))$: If $C_R(X)$, is the final point of the initial ray in a TET rooted at $f(X)$ the final point of a constraint line instance? (If $C_R(X)$, when the program enters a state represented by $f(X)$, in which some process is blocked, does a blocked process eventually unblock?)

$g(X)$: If $C_R(X)$, what is the length of the perpendicular projection on either axis of a slope one ray rooted at X whose final point is G', such that G' is the only point on the ray that lies on a constraint line instance? (If $C_R(X)$, how long do both processes run in parallel before some process blocks at the next semaphore?)

$g(f(X))$: If $C_R(X)$, then $f(X)$ lies on a constraint line instance. Let X' denote the final point of this constraint line instance. If $C_L(f(X))$, then the final point of the initial ray in a TET rooted at $f(X)$ is X'. What is the length of the perpendicular projection of the ray with initial point $f(X)$ and final point X' on the axis parallel to the ray? (If $C_R(X) \wedge C_L(f(X))$, how long does a process

block when the program enters the state represented by $f(X)$?)

We solve the more general problem of how to compute $C_R(G)$ and $C_L(G)$ for any point G in a TPGS, as well as how to compute $f(G)$ and $g(f(G))$ for any point G in a TPG set satisfying $C_R(G) \wedge C_L(f(G))$. This is because restricting G does not appear to simplify the problem.

Solution Alternatives: Three solution methods are an analytic method, computational geometry, and integer programming. We use here a predominately analytic method. The method is not purely analytic because the formula for $C_L(f(X))$ is based on a list of constraint line intersections, obtained from a computational geometric algorithm. A purely computational geometric method is given in [2], based on ray shooting. Finding $g(G)$ is equivalent to the following integer programming problem: minimize the length of a slope one diagonal ray rooted at G subject to the constraint that the final ray point lies on some constraint line generated by an element of Λ. We rule out the use of integer programming based on two drawbacks. First, integer programming examines a potentially infinite search space, and only terminates if $C_R(G)$ holds. Second, integer programming wastes time searching infeasible points, because each constraint line generated by an element of Λ may or may not contain a feasible solution.

7.1 Formula for $C_L(G)$

By definition, evaluating $C_L(G)$ for point G on line $L \in \Lambda^*$ requires evaluating $S_1(G, L)$. However, $S_1(G, L)$ is defined in terms of set Λ^*, which contains an infinite number of elements. The following theorem provides a way to compute $C_L(G)$ in terms of finite set Λ.

Theorem 7 *$C_L(G)$ is true in TPGS $\langle \Phi, \Lambda, f \rangle$ iff $\langle \exists L : L \in \Lambda :: mod(G) \in L \wedge G \not\equiv L.i \wedge \neg\langle \exists G', \exists L' : L' \in \Lambda \wedge L \neq L' :: G' \in L \cap L' \wedge G' > mod(G) \rangle\rangle$.*

Proof:

$C_L(G) = \neg C_N(G) \wedge \langle \exists L : L \in \Lambda \wedge mod(G) \in L :: S_1(G, L) = \emptyset \rangle$

, by the definition of Λ^* and C_L

$\langle \forall L, \forall L' : L \in \Lambda \wedge L' \in (\Lambda^* - \Lambda) :: L \cap L' = \emptyset \rangle$

, by the definition of Λ and Λ^*

$C_L(G) = \neg C_N(G) \wedge \langle \exists L : L \in \Lambda \wedge mod(G) \in L :: \{G' \mid G' \in L \cap \Lambda - \{L\} \wedge G' > mod(G)\} = \emptyset$

, by last two deductions and definition of S_1

Applying the definition of C_N and simplifying yields the expression for C_L in the theorem. □

Known computational geometric algorithms for reporting intersections of line segments may be used to compute $\langle \forall L, \forall L' : L \in \Lambda \wedge L \in \Lambda \wedge L \neq L' :: L \cap L' \rangle$, which is required in Theorem 7; see for example Bentley and Ottmann [8].

7.2 Analytic Solution of $C_R(G)$, $g(G)$, and $g(f(G))$

The solution method presented here requires the following assumption.

Assumption 1 *Constants ϕ_0 and ϕ_1 represent rational quantities.*

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

Assumption 2 *Constants ϕ_0 and ϕ_1 represent relatively prime integers.*

To demonstrate the equivalence, let $L(\phi_0, \phi_1)$ denote the least common denominator of ϕ_0 and ϕ_1. Let $G(\phi_0, \phi_1)$ denote the greatest common divisor of $\phi_0 L(\phi_0, \phi_1)$ and $\phi_1 L(\phi_0, \phi_1)$. Multiplying rational ϕ_0 and ϕ_1 by $L(\phi_0, \phi_1)/G(\phi_0, \phi_1)$ yields relatively prime quantities, to which the solution method below is applied to calculate $C_R(G)$ and $g(G)$. The resulting (real) value of $g(G)$ multipled by $G(\phi_0, \phi_1)/L(\phi_0, \phi_1)$ corresponds to the solution for the original, rational ϕ_0 and ϕ_1.

The analytic solution for $C_R(G)$, $g(G)$, and $g(f(G))$ is based on a simplified version of a TPGS which has constraint lines generated by *only one* generator in set Λ.

Definition. *Given $\langle \Phi, \Lambda, f \rangle$, $\underline{C_R(G, W, X)}$ and $\underline{g(G, W, X)}$ are defined as $C_R(G)$ and $g(G)$, respectively, in TPGS $\langle \Phi, \overline{\{[W, X)\}}, f \rangle$.*

Therefore:

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

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

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

Notation conventions: The remainder of this section assumes without loss of generality that the generator in TPGS $\langle \Phi, \{\overline{[W, X)}\}, f \rangle$, is a horizontal line; that is, $W_0 < X_0 \wedge W_1 = X_1$. (The case of vertical lines follows by interchanging subscripts 0 and 1 in the subsequent text.)

7.2.1 Formula for $C_R(G)$

Recall that point G is restricted iff G lies off a constraint line and a slope one ray rooted at G intersects a constraint line generated by an element of set Λ. Each point on the slope one diagonal ray rooted at G is $G + (y, y)$, for $y \in R$. Therefore $C_R(G, W, X)$ holds iff

$$\langle\, \exists y, i_0, i_1 \,:\, y \in R, i_0 \in Z, i_1 \in Z \,::\, G + (y, y) \;\in\; \overline{[W + (i_0\phi_0, i_1\phi_1),\, X + (i_0\phi_0, i_1\phi_1))} \,\rangle.$$

The relationship above is rewritten below as two equations, each corresponding to one component of a point. This requires the binary relationship $\in$ to be transformed to an equality relationship by introducing a slack variable, denoted s. The resulting equations are illustrated in Fig. 11.

$$G_0 + y + s = X_0 + i_0\phi_0 \quad \text{and} \tag{3}$$

$$G_1 + y \;=\; X_1 + i_1\phi_1, \tag{4}$$

where

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

Letting $s' \;=\; s + X_1 - X_0 - G_1 + G_0$ and $I(G, W, X)$ denote the set of integers in the interval $(X_1 - X_0 - G_1 + G_0,\; X_1 - G_1 + G_0 - W_0]$, variable y may be eliminated from system (3) to (5):

$$i_1\phi_1 - i_0\phi_0 + s' = 0, \tag{6}$$

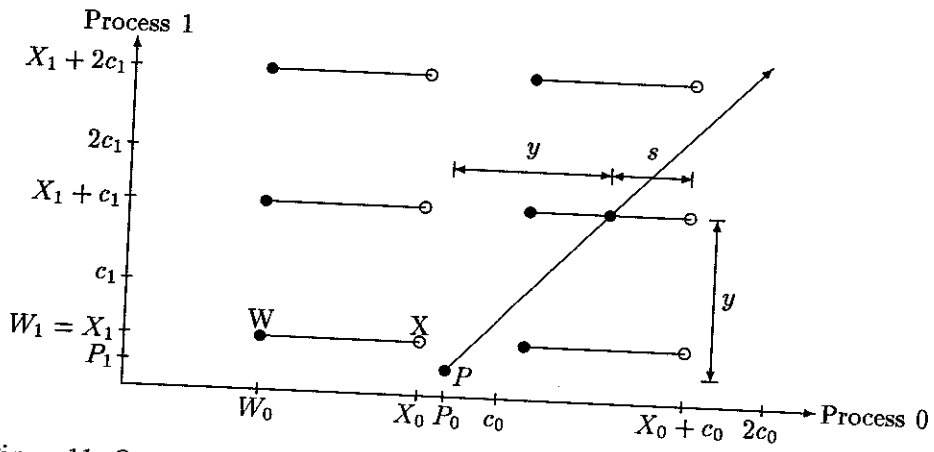


Figure 11: One possible relationship of G, y, and s in a TPGS with constraint lines generated by a single generator, $\overline{[W,X)}$. In general, there are either zero or an infinite number of points of intersection of a slope one diagonal ray with instances of a single constraint line generator, corresponding to zero or an infinite number of values of y, respectively. In the figure, $i_0 = i_1 = 1$ yields $g(G,W,X) = y$.

where s' must lie in the interval

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

Therefore $C_R(G,W,X)$ holds iff there exists a solution to equation (6). Because ϕ_0 and ϕ_1 are relatively prime integers, s' must be an integer by equation (6). Therefore equation (6) is a Diophantine equation. A necessary and sufficient condition for a solution to equation (6) to exist is that the greatest common divisor of ϕ_1 and ϕ_0 divides s', by Jones' Theorem 3.3 [23]. Therefore a solution exists iff interval $I(G,W,X)$ contains an integer value. Applying equation (1) establishes the following theorem.

Theorem 8 *$C_R(G)$ is true in TPGS $\langle \Phi, \Lambda, f \rangle$ iff $\langle \exists \overline{[W,X)} : \overline{[W,X)} \in \Lambda :: \lceil X_1 - X_0 - G_1 + G_0 \rceil \leq \lfloor X_1 - G_1 + G_0 - W_0 \rfloor \rangle$.*

7.2.2 Formulas for g(G) and g(f(G)) if $C_R(G) \wedge C_L(f(G))$

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

$$g(G,W,X) = \min\{i_1\phi_1 + X_1 - G_1 \mid i_1 \in Z \wedge i_1\phi_1 + X_1 - G_1 \geq 0\}. \tag{8}$$

The right hand side of equation (8) requires the minimum element of a set containing an infinite number of elements. The right hand side will be reexpressed as a set containing a *finite* number of

elements to permit an algorithm to compute the minimum by exhaustive search. The rewritting is done by expressing unknown i_1 in terms of unknown s' and an integer parameter α by applying the solution technique for three variable Diophantine equations in [23], pp. 67-68. There are an infinite number of solutions, which parameter α expresses. The solution to (6) is

$$i_1 = us' + \phi_0\alpha, \tag{9}$$

where u is an integer satisfying $\phi_1 u \equiv 1 \pmod{\phi_0}$. Combining equations (8) and (9) yields

$$\begin{aligned} g(G,W,X) = \min\{&(us' + \phi_0\alpha)\phi_1 + X_1 - G_1 \mid \\ &\alpha \in Z \wedge s' \in I(G,W,X) \wedge (us' + \phi_0\alpha)\phi_1 + X_1 - G_1 \geq 0\}. \end{aligned} \tag{10}$$

The right hand side of equation (10) still requires the minimum value of an infinite set of elements. However, α can be rewritten in terms of s'. Solving $(us' + \phi_0\alpha)\phi_1 + X_1 - G_1 \geq 0$ for the value of α that yields, for a given value of s', the minimum, nonnegative value of $(us' + \phi_0\alpha)\phi_1 + X_1 - G_1$, we obtain

$$\alpha = -\left\lceil \frac{\phi_1 us' - G_1 + X_1}{\phi_1\phi_0} \right\rceil. \tag{11}$$

Theorem 9 *In TPGS $\langle \Phi, \Lambda, f \rangle$, where ϕ_0 and ϕ_1 are relatively prime, given a point G such that $C_R(G) \wedge C_L(f(G))$,*

$$g(G) = \min\{\ (\phi_1 us' + X_1 - G_1) \bmod \phi_0\phi_1 \mid \overline{[W,X)} \in \Lambda \wedge C_R(G,W,X) \wedge s' \in I(G,W,X)\ \} \tag{12}$$

where u satisfies $\phi_1 u \equiv 1 \pmod{\phi_0}$. Furthermore, $g(f(G)) = s^* - (X_1 - X_0 - G_1 + G_0)$, where s^* is the value of s' that yields the minimum $g(G)$ in equation (12).

Proof: The expression for $g(G)$ follows by combining equations (2), (10), and (11). The expression for $g(f(G))$ follows because, by definition, $g(f(G,W,X))$ is the value of s satisfying equation (3) when $y = g(G,W,X)$ and from equation (11). □

The fact that Theorem 9 requires the minimum of a set containing a finite number of elements permits computation of $g(G)$ and $g(f(G))$ by exhaustively examining all set elements. The second refinement of algorithm A0, presented in the following section, exploits this fact.

8 Second Refinement: Algorithm A2

This section refines algorithm A1 by incorporating the formulas for $C_L(G)$, $C_R(G)$, and $g(G)$ given in Theorems 7 to 9. The result is algorithm A2 (Fig. 12). For TPGS $\langle \Phi, \Lambda, f \rangle$, algorithm A2 stores the cycle times, ϕ_0 and ϕ_1, as the two elements of array C and the constraint line generator set, Λ, as an $N \times 2$ array of initial points (W) and an $N \times 2$ array of final points (X). For example, the initial point W^{N-1} of constraint line generator (W^{N-1}, X^{N-1}) is stored in (W[N-1,0],W[N-1,1]). The edges of the graph algorithm A1 generates are stored in $N \times 1$ array E; if E[0]=3 then an edge exists from node 0 to node 3. The state transition vectors output by algorithm A1 are stored in $N \times 1$ arrays G0 and G1.

We next consider the time and space Requirements of algorithm A2. The time required by algorithm A2 is dominated by the time required to evaluate the minimization in Theorem 9, as the following theorem establishes. Let $I_N(G, W, X)$ denote the number of integers in interval $I(G, W, X)$. Let $D = \max\{I_N(X^k, W^{k'}, X^{k'}) \mid k, k' \in \{0, 1, \ldots, N-1\}\}$.

Theorem 10 *Algorithm A2 requires at worst time $O(N^2 D)$, excluding the time to compute the l.c.d. and g.c.d. of two rational numbers (in function LG), and the time to solve the congruence for u.*

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

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

Step A2.1: The intersection may be computed using an algorithm that reports the points of intersection of horizontal and vertical line segments, such as Bentley and Ottman's algorithm 4.1 in [8]. Rather than reporting all intersections, the algorithm is modified to store in ML[K] the intersection point furthest from the origin found so far for the instance of constraint line K that the algorithm considers. The algorithm requires time $O(N \log N + K)$, where K is the number of intersecting pairs. In our case, half the constraint lines in Λ are vertical and half are horizontal. At worst, every horizontal line intersects every vertical line, and $K = O(N^2)$. Thus step A2.1 requires time $O(N^2)$.

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

```
var
  N,K,K': integer;
  T,LG,U,S': real;
  C:    array [0..1] of rational;   {C[0]=φ0; C[1]=φ1;}
  W,X:  array [0..N-1,0..1] of real; {(W[K],X[K])=(W^k, X^k)}
  E:    array [0..N-1] of integer;  {E[K]=∞, if f^2(X^k) exists, else E[K]=k', where X^k' ≡ f^2(X^k)}
  G:    array [0..1] of real;       {a point}
  G0:   array [0..N-1] of real;     {G0[K]=g(X^K) if C_R(X^K)}
  G1:   array [0..N-1] of real;     {G1[K]=g(f(X^K)) if C_R(X^K) ∧ C_L(f(X^K))}
  MI:   array [0..N-1,0..1] of real; {MI[K]=max{G|G ∈ closure([W[K],X[K]]) ∩ Λ − {[W[K],X[K]]}}}
  M:    array [0..N-1] of real;     {M[K] = s* (from Theorem 9)}

{Interval I(X^K', W^K, X^K) is represented as [ IL(K,K'), IH(K,K') ) using the following two functions.}
  function IL(K,K':integer):real  begin return X[K',1]-X[K',0]-X[K,1]+X[K,0] end
  function IH(K,K':integer):real  begin return X[K',1]-X[K,1]+X[K,0]-W[K',0] end

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

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

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

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

{Step A2.3; corresponds to step A1.2}
  for each cycle k1, k2, ..., kM in graph in array E (where k1 = kM) do begin
    output point (X[k1, 0]/LG, X[k1, 1]/LG); for K:=1 to M-1 do output GO[K]/LG, G1[K]/LG
end end
```

Figure 12: Algorithm A2

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

Theorem 11 *Algorithm A2 requires $\Omega(N)$ storage, excluding the storage to required compute ι.*

Proof: Follows from the array dimensions in declaration portion of algorithm A2 (Fig. 12). □

Theorems 10 and 11 show that algorithm A2 requires space $\Omega(N)$ and time $O(N^2D)$, where D is related to the precision with which measurements of program timings are desired. Consider the time requirement. In practice, the N^2 term is not prohibitive, because two-process programs usually use a small number of semaphores. The D term, however, may force us to make approximations. As discussed in Section 7.2, we map any rational cycle time ϕ_0 or ϕ_1 to relatively prime integers by multiplying both cycle times by LG, where LG is the least common denominator of ϕ_0 and ϕ_1 divided by the greatest common divisor of a multiple of ϕ_0 and ϕ_1. Thus, D will grow with the product $LG \cdot \phi_0$ or $LG \cdot \phi_1$, which is the ratio of the cycle time to the resolution of the measurement clock. For example, if we measure an algorithm to microsecond resolution, LG is at most 10^6. If $\phi_0 = \phi_1 = 100$ seconds, then $D = 300 \cdot 10^6$. However, we may be willing to trade accuracy for computation time by approximating measurements by milliseconds, so that $D = 300 \cdot 10^3$.

9 Conclusions

The TPG analysis in this paper shows that any execution sequence of a program fitting our model that does not contain a dead state or an infinite number of nondeterministic states converges on a behavior consisting of a repetition of states, termed the *limit cycle execution sequence.* There may be several congruence classes of limit cycle execution sequences; 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 limit cycle execution trajectories studied here. Furthermore, we have some experimental evidence of the similarity from two sets of experiments. The first [5] uses a dining philosophers program similar to Fig. 5, except that spinlocks rather than binary semaphores are used, and the order of acquiring

and releasing spinlocks differs from the order of semaphores shown in the figure. The program was executed on a shared memory multiprocessor (a Sequent Symmetry), and produces a LCES with up to nine philosophers. The second [1, Ch. 6] uses a dining philosophers algorithm in which each resource is controlled by a separate monitor process that runs on a dedicated processor. Its execution with between four and 64 processors on a multicomputer with interprocess communication done by message passing shows that for small numbers of processes, the global-state transition sequence is deterministic. Starting at about nine processors, small perturbations occur in the limit cycle for short instances of time, after which the program returns to a limit cycle.

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 constraint lines intersected does remains constant. However, when the 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.

Any model requires assumptions that are usually not strictly met by the systems they model. For example, the delay of each edge in a timed transition diagram must be an independent constant. Drift among processor clocks and contention for resources (e.g., a bus, a network, or a memory cell) prevent programs from strictly meeting this assumption. One may ask how accurately the TPGs of this paper model programs that are otherwise representable as timed transition diagrams. Our experience so far indicates that the model is highly accurate [1, 4].

Several open problems remain:

- relaxing the assumptions in §1 to permit geometric modeling of a broader class of programs;
- solution of the model for an arbitrary number of processes (Application of the analytic method proposed here requires solution of a multidimensional Diophantine equation that we are unable to solve. In addition, if two or more processes are identical (i.e., they are represented by identical timed transition diagrams), then the multidimensional TPG displays symmetries, which can be exploited

to avoid a naive generalization of the solution presented here to more than two dimensions.); and

- model analysis for irrational cycle times ϕ_0 and ϕ_1. (Chaotic behavior may exist for these values.)

Acknowledgements

Ashok K. Agrawala provided many important discussions and comments during the development of this work. Satish K. Tripathi also provided many suggestions. Comments from Liba Svobodova and anonymous referees on an earlier manuscript draft improved the final paper. Dennis Kafura provided several suggestions that improved the manuscript readability.

References

[1] M. Abrams. *Performance Analysis of Unconditionally Synchronizing Distributed Computer Programs Using The Geometric Concurrency Model.* PhD thesis, Computer Sci. Dept., Univ. of MD, Aug. 1986. TR-1696.

[2] M. Abrams. Geometric analysis of limit cycles in timed transition systems. Technical Report TR 92-30, Computer Sci. Dept., Virginia Tech, Blacksburg, VA 24061-0106, Sept. 1993.

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

[4] M. Abrams and A. K. Agrawala. Automated measurement and prediction of unconditionally synchronizing distributed algorithms. In *Proc. of the 7th International Conf. on Distributed Computing Systems*, pages 498–505, Berlin, Sept. 1987.

[5] 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.

[6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading, MA, 1974.

[7] G. Balbo, G. Chiola, and S. C. Bruell. An example of model and evaluation of a concurrent program using colored stochastic Petri nets: Lamport's fast mutual exclusion algorithm. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):221–240, Mar. 1992.

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

[9] R. F. Berry and J. L. Hellerstein. Characterizing and interpreting periodic behavior in computer systems. In *Proc. SIGMETRICS*, pages 241–242, Newport, RI, June 1992. ACM.

[10] 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.

[11] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, Reading, MA, 1988.

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

[13] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

[14] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 67–68. Academic Press, New York, 1968.

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

[16] G. Haring and G. Kotsis, editors. *Performance Measurement and Visualization of Parallel Systems*, volume 7 of *Advances in Parallel Computing*. North-Holland, Moravany, Czechoslovakia, 1993. Proc. of the Workshop on Performance Measurement and Visualization, Oct. 1992.

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

[18] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. Technical Report TR 93-1263, Dept. of Comp. Sci., Cornell Univ., Jan. 1992.

[19] T. A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. Technical Report TR 93-1330, Dept. of Comp. Sci., Cornell Univ., Mar. 1993.

[20] T. J. Hickey, J. Cohen, H. Hotta, and T. Petitjean. Computer-assisted microanalysis of parallel programs. *ACM Trans. on Programming Languages and Systems*, 14(1):54–106, Jan. 1992.

[21] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1984.

[22] 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.

[23] B. W. Jones. *The Theory of Numbers.* Rinehart, New York, 1955.

[24] 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.

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

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

[27] J. Magott. Performance evaluation of systems of cyclic sequential processes with mutual exclusion and communication by buffers using timed Petri nets. In *Proc. Int. Workshop on Timed Petri Nets*, pages 146–153, Madison WI, 1987. IEEE Press.

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

[29] B. Plateau and K. Atif. Stochastic automata network for modeling parallel systems. *IEEE Trans. on Software Engineering*, 1991(10):1093–1109, Oct. 1991.

[30] C. V. Ramamoorthy and G. S. Ho. Performance evaluation of asynchronous concurrent systems using Petri nets. *IEEE Trans. on Software Engineering*, SE-6(5):440–448, Sept. 1980.

[31] L. H. Ramshaw. *Formalizing the Analysis of Algorithms.* PhD thesis, Computer Sci. Dept., Stanford Univ., June 1979. STAN-CS-79-741.

[32] A. J. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. on Software Engineering,* 15(7):875–889, July 1989.

[33] M. Simmons and R. Koskela, editors. *Performance Instrumentation and Visualization.* ACM, New York, 1990. Based on the Workshop on Parallel Computer Systems: Instrumentation and Visualization, May 1989.

[34] C. Smith and J. C. Browne. Aspects of software design analysis: Concurrency and blocking. In *Proc. Performance 80,* pages 245–253, summer 1990. in *Performance Evaluation Review 9,* 2.

[35] C. J. Smith. *Performance Engineering of Software Systems.* Addison Wesley, Reading, MA, 1990.

[36] 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.

[37] 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.

[38] L. Zhang, S. Shenker, and D. D. Clark. Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic. In *Proc. SIGCOMM,* pages 133–147, Zurich, Sept. 1991. ACM.

A Proofs of Correctness of Algorithm A0

All TETs in this appendix are presumed to consist of either live or restricted points. (This excludes dead, nondeterministic, and free points, because they never arise in a blocking LCET.) We first prove three properties about function f. Recall from Section 5.1 that ξ denotes the set of final points of all generators, and set ξ contains N elements. Let X denote an element of ξ and i and j denote integers.

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

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

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 points are restricted, and all blocked points 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 inductive hypothesis $C_L(f^{2i-1}(X))$ and definition of f_2

If $i > 0$ then $f^{2i}(X) \in \xi$

, by applying definition of f_1 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 last deduction and the definition of f_1

$C_L(f^{2i+1}(X))$

, by last deduction and assumption that all points in a TET are live or restricted. □

Proof of F2:

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

$\exists j, j \geq 0, G = f^{2j}(X)$, 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 □

Theorem 3 *Every trajectory output by algorithm A0 is a blocking LCET.*

Proof: By definition, a blocking LCET must: (1) contain a point that lies on a constraint line, (2) be a subtrajectory of some TET in the TPG, 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 point 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 G and final point G' that is a LCET

, by last deduction

$X \leq G < G' \leq f^{2n}(X)$

, by last deduction

$G' = f^{2n}(X)$

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

$G = X$

, by F2

Every trajectory output by A0 contains exactly two congruent points

, because $G = X$, $G' = f^{2n}(X)$

□

Lemma 4 *Consider a LCET S representing only deterministic points. There exists a LCET rooted at any point congruent to a point on S.*

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

LCET rooted at Y exists; denote its final point by Y'

, by Theorem 1 and Lemma 1

Subtrajectory with initial and final points G and Y is congruent to subtrajectory with initial and final points G' and Y', respectively

, because $G \equiv G'$, $Y \equiv Y'$, and by Lemma 1

S is congruent to LCET rooted at Y

, by last deduction and because trajectory with initial point Y and final point G' is a subtrajectory of both S and the trajectory rooted at Y

S is congruent to LCET rooted at any point congruent to Y

, by last deduction and Lemma 1

□

Theorem 4 *Any blocking LCET is congruent to one of the trajectories output by algorithm A0.*

Proof: Consider some blocking LCET, denoted γ.

γ contains ray with its initial point on a constraint line

, because LCET blocks

Final point of ray in last deduction is final point of some constraint line

, because LCET does not contain dead points, and by the definition of f_1

$\exists X \in \xi$, where X is congruent to final point in last deduction

, by definition of constraint line

LCET with initial point X exists and is congruent to γ; denote it by γ''

, by Lemma 4

$\exists m, m \in \{1, 2, \ldots, N\}, f^{2m}(X) \equiv X$

, by F3 and last deduction

A0 outputs γ''

, by last deduction and Theorem 3

□

Lemma 5 *Given a point, all LCETs containing that point are congruent.*

Proof: Follows from Lemma 4.

□

Theorem 5 *None of the trajectories output by algorithm A0 are congruent.*

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 5.

$f^{2n}(X^2)$ must be end point of two non-collinear rays in any LCET on which it lies

, by Lemma 5

X^2 must be end point of two non-collinear rays in any LCET 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 6 *For each trajectory output by A0, either that trajectory or some congruent trajectory is contained in some TET rooted at a point either on line* $\overline{[(0,0),(\phi_0,0))}$ *or on line* $\overline{[(0,0),(0,\phi_1))}$.

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

$f^{2n}(X) \in \gamma$

, by definition of f

$\exists i_0, \exists i_1, f^{2n}(X) = X + (i_0\phi_0, i_1\phi_1)$

, because $f^{2n}(X^2) \equiv X$, where $n > 0$

$\exists G, G \in \gamma,\ \exists r, r \in \{0,1\},\ X \leq G < f^{2n}(X)\ \wedge\ G_r \bmod \phi_r = 0$

, by last deduction and because $X \in S \wedge f^{2n}(X) \in S$

$G \equiv (G_0 \bmod \phi_0, G_1 \bmod \phi_1)$

, by definition of congruent points

Execution trajectory rooted at $(G_0 \bmod \phi_0, G_1 \bmod \phi_1)$ contains a point congruent to $f^{2n}(X)$

, by Lemma 1 and last deduction

Execution trajectory rooted at $(G_0 \bmod \phi_0, G_1 \bmod \phi_1)$ contains a trajectory congruent to γ

, by Lemma 4 and last two deductions

□