Geometric Performance Analysis of Periodic Behavior in Detail

Marc Abrams*

Computer Science Department, Virginia Tech, Blacksburg, VA 24061-0106

abrams@cs.vt.edu

September 5, 1995

Abstract

A fundamental 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. This paper presents a novel performance analysis method based on geometry. 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, all TESs that can arise in any execution contain a suffix that consists of the repetition of a finite sequence of states, excluding deadlocks and nondeterministic behavior. The repeated sequence is termed the *limit cycle execution sequence.* This paper derives, for all possible process starting times, a representation of 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 to derive TESs from a geometric program execution model, using timed progress graphs (TPGs). TPGs represent the progress of each process by an axis in a Cartesian graph; process synchronization by line segments; and a TES by a directed, continuous path that does not cross a segment. Second, it solves for TESs in TPGSs not by a computational geometric algorithm, as employed by most solutions in the literature to (untimed) progress graphs, but by an analytic solution.

1 Introduction

The ability to analyze program performance lies at the heart of integrating into the software development life cycle what Smith [36] terms *software performance engineering.* The requirements life cycle phase states performance goals. Ideally, we would like to establish whether a program under development 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.

Even when one can calculate a program performance measure of interest (e.g., waiting time for a resource), the method used to derive the measure may not give insight into why the measure has a particular value. It may also not give insight into what happens if some timing in the program is varied – such as the relative starting time of processes or the time for which a process holds a resource.

We consider one class of programs for which there is a simple way to derive performance measures and gain insight. The analysis method represents the progress of processes as a geometric problem. Relationships between synchronization structure, code timings, starting times of processes, and execution behavior are made tangible as relationships between lines and paths in a plane. The chief assumption is that the program displays periodic behavior: processes loop forever repeating the same code segments. Periodic behavior has been observed experimentally several times in the literature: in queue length and packet traffic in a communication protocol [39]; in periodic tasks, such as operating system daemons [10]; in a commercial TCP/IP protocol [5]; and in dining philosophers programs [3, 5].

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

1.1 Related Work

Three methods of deriving performance measures from programs given a representation of process synchronization structure and code segment timings are simulation, logic, and analysis. Given an initial state, simulation steps through an initial subsequence of one possible execution sequence. Uses of logic include Henzinger, Manna, and Pnueli's [20] proof system for real time systems using lower and upper bounds on transition times, Shaw's [34] extension of Hoare logic to reason about time in software; and Ramshaw's [31] generation of recurrence equations about program timings using Hoare style axioms. Analytic methods include complexity analysis (e.g., [7]), micro-analysis using difference equations (e.g., [21]), Petri nets (e.g., [8, 22]), stochastic processes (e.g., [16]), stochastic automata [29], queueing networks (e.g., [19]), and analysis of graphs whose nodes represent code segments (e.g., [17, 32]).

Of these, consistent Petri nets (i.e., nets that return to their initial marking) in which a deterministic firing time is associated with each transition are most related to this paper. 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 here, of processes synchronizing only to achieve mutual exclusion, cannot be represented by decision-free or persistent Petri nets. Instead, Holliday and Vernon [22] use Petri nets with frequency expressions (i.e., probabilities) to resolve deterministically which transition fires when a token enables two or more transitions simultaneously.

1.2 Problem Statement

Let the *state* of a program be the control point of each process. We represent an execution of a program by two sequences, one consisting of the states through which the program passes 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 non-terminating program might consist of state sequence (-1,-1), (0,0), $\{$ (1,0), (2,1), (0,2) $\}^*$ and time sequence 0, 1, $\{$ 3, 1, 1 $\}^*$, where "$\{\}^*$" denotes an infinite number of repetitions of the enclosed subsequence. The TES represents a program that starts in state (-1,-1), instantly moves to state (0,0), remains in state (0,0) for 1 time unit, then moves to state (1,0) for 3 time units, and so on. The repeated sequence (i.e., (1,0), (2,1), and (0,2) for 3, 1, and 1 time unit, respectively) is termed the *limit cycle execution sequence* (LCES). In general, for a given program there may exist an infinite number of possible TESs. 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 if and only if the program terminates or reaches a deadlock.

Can we find a finite 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., [18, 35]) that include the ability to display program event traces.

We address the following problem: *Given a specification of the process synchronization structure and the execution timings of all code segments, derive a representation of the set of all possible LCESs in which a process blocks, for all possible process starting times.*

1.3 Our Solution Approach

Assumptions: To obtain a tractable problem, we make the following assumptions:

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 [15]. (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: Process synchronization is used only for mutually exclusive access to each of a set of resources.

The assumptions give sufficient (but not necessary) conditions for a program to always reaches a LCES if it does not deadlock, excluding nondeterministic behavior. The assumptions are sufficient but not necessary conditions because they can be relaxed. For example, [2, Fig. 1] shows a program that uses message passing with one buffer for inter-process communication (violating A2.3 and A3) that reaches a LCES. The assumptions are, however, necessary and sufficient to use the geometric performance modeling approach and the analytic solution of this paper.

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 [8] 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. As for A2.2, certain programs can be considered non-terminating for the purpose of analysis: long running programs that execute the same code repeatedly, such as simulations, and *reactive programs* [12], such as operating system algorithms, that react to external stimuli. Chandy and Misra [12] view all computations as non-terminating transition systems that reach a fixed point. Regarding A2.5, Adve and Vernon [6] conclude based on seven parallel applications that "it appears reasonable to ignore the variability in execution times when estimating synchronization delays." Furthermore, an exponential task time assumption could actually lead to more severe errors than a constant-time assumption. Because we develop our solution method by starting with a program as a transition system, we lay the foundation for relaxing the remaining assumptions to study other parallel program classes with geometry (see [2] for other programs).

Timed Progress Graphs: We propose a novel way to analyze program performance: using *timed progress graphs* (TPG). A TPG adds timing information to an 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" [11].

To our knowledge, TPGs were first used in operations research to find schedules that minimize the last job completion time in a job shop with two jobs and an arbitrary number of machines [33, pp. 262-263]. UPGs were later used by Dijkstra [13] to illustrate multiprogramming deadlock; Kung, Lipski, Papadimitriou, Soisalon-Soininen, Yannakakis, and Wood [24, 28, 37, 38] to detect deadlocks in lock-based transaction systems; and Carson and Reynolds [11] to prove liveness properties in programs with an arbitrary number of processes under assumptions A2.3 and 2.4. TPGs differ from UPGs in two ways: First, TPGs represent the time required for transitions, which permits performance analysis.

UPGs represent timed transition diagram sets in which all times are equal. Second, UPGs are used to characterize deadlocks for any number of processes, any number of processors, and any scheduling discipline. In contrast, TPGs as defined here analyze only two process programs executing on two dedicated processors. One final distinction is that we do not use a computational geometric algorithm to analyze progress graphs, as do most past solutions, but rather derive an analytic solution based on number theory. Combining Carson and Reynold's [11] work with this paper shows that one model can be used to verify both liveness and analyze performance of semaphore programs.

Results: The use of TPGs is informally illustrated in §2. §3 formalizes timed transition diagrams. §4 formally defines TPGs as well as a transition function to construction of the geometric equivalent of TESs. A key theorem is that a TES with a finite number of nondeterministic states and no deadlocked states always contains a LCES. TPGs are then generalized to analyze simultaneously all possible initial starting times of processes. §5 to §8 develop by stepwise refinement an algorithm that derives a finite representation of the set of non-blocking LCESs. The problem is reduced to finding cycles in a graph whose vertices correspond to synchronization constraints in a program and whose edges are derived by an analytic method that expresses the timing of processes between synchronization points as a set of Diophantine equations. The correctness and time and space complexity of the algorithm are proved. §9 illustrates that TPGs can be used to calculate TESs for programs with more than two processes; however the algorithm proposed here to find all blocking LCESs can no longer be used.

2 Illustration of Solution Method

We illustrate our approach using Dijkstra's dining philosophers problem with two philosophers [14]. 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 program solution in which P and V operations on a semaphore correspond to acquiring and releasing chopsticks, respectively. The execution time required for each code segment in some time unit, excluding blocking, is also shown. The `delay` statements specifies how long the parent process waits before starting each philosopher; the value is 0 here but non-zero values are considered later. In this case both philosophers start simultaneously. Figure 2 represents the program by a set of two *timed transition diagrams.* Consider one diagram, with n edges. The 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. 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 $(i+1) \bmod (n-1)$ 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; one was listed in §1.2. All possible TESs contain the LCES (1,0), (2,1), and (0,2) for 3, 1, and 1 time unit, respectively.

The TPG representing Fig. 1 will be developed in a set of steps. First consider what happens if philosopher 0 runs alone and philosopher 1 does *not* run. Philosopher 0 will never block at a P operation. In fact, as illustrated in the Gannt chart of Fig. 4, philosopher 0 enters code segment `Think; P(a)` (i.e., location 0) at time 0, leaves location 0 and enters code segment `Eat; V(a)` (i.e., location 1) at time 1, makes a transition from location 1 to location 2 at time 4, makes a transition from location 2 back to 0 at time 5, and so on. The *cycle time*, or the time required for a process to complete one iteration of its loop, equals 5 for philosopher 1 because it first returns to location 0 at time 5. If we ask at any time $x \geq 0$, "In which location is philosopher 0?", the answer is location 0 if $x \bmod 5 \in [0, 1)$, location 1 if $x \bmod 5 \in [1, 4)$, and location 2 if $x \bmod 5 \in [4, 5)$. This establishes a correspondence between program states and graph points (e.g., values of x). If we ask at time x, "How much longer will philosopher 1 remain in its current state?", we simply subtract $x \bmod 5$ from the end of the interval that contains the quantity. For example, for $x = 7$, $x \bmod 5 = 2$ and $2 \in [1, 4)$; therefore at time 7 philosopher 1 will

```
Philosopher(i) { L:Think; P(a); Eat; V(a); goto L; }
semaphore a=1;
parbegin
  begin delay 0; Philosopher(0); end
  begin delay 0; Philosopher(1); end
parend
```

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

Figure 1: One semaphore dining philosophers program.

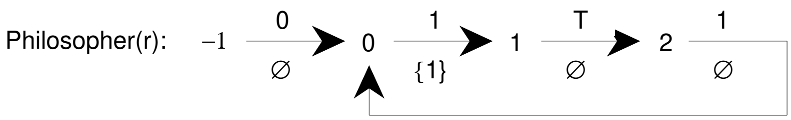


Figure 2: Timed transition diagrams corresponding to Fig. 1. For process $r = 0$, T=3; otherwise T=1.

remain its present location for $4 - 2 = 2$ more time units. Thus a point within an interval represents partial execution of a code segment. We could draw a similar diagram for philosopher 1: the intervals for each location would be [0,1), [1,2), and [2,3); the cycle time would be 1+1+1=3.

Next consider the joint progress of both philosophers if semaphore a had initial value 2. Neither process would ever block. Each process can be mapped independently to the non-negative real line (e.g., Fig. 4). Thus two processes can be mapped to the Cartesian product of non-negative reals, represented as a two dimensional graph (Fig 5(a)). Although we illustrate a finite portion of the plane, the plane is really semi-infinite. The intervals along each axis corresponding to each location partition the plane into a set of rectangles, each representing an occurrence of a global program state. For example, the rectangle with opposite vertices (6,4) and (9,5) represents an occurrence of state (1,1) (i.e., both philosophers are in location 1). This is because for a point (x, y), philosopher 0 is in location 1 when $x \bmod 5 \in [1, 4)$ and philosopher 1 is in location 1 when $y \bmod 3 \in [1, 2)$, by the formulas given earlier for Fig. 4. The correspondence of graph points and states implies that the TES must correspond to a directed path in the plane; in particular, the PES corresponds to a diagonal line with slope 1. The path corresponding to a TES is called a *timed execution trajectory* (TET). The TET is the diagonal slope one ray in Fig. 5(a).

In Fig. 1 semaphore a really has initial value 1; how does this alter the TET? A process may block when it attempts to *enter* location 1 according to Fig. 2. Furthermore, a process unblocks only when the other process *exits* location 1. Thus state (1,1), in which both processes are performing `Eat; V(a)`, can never arise in a TES. The shaded rectangles in Fig. 5(a) represent all occurrences of (1,1). A TET representing a TES can therefore not pass through a shaded region. Fig. 5(b) shows a TPG. It represents

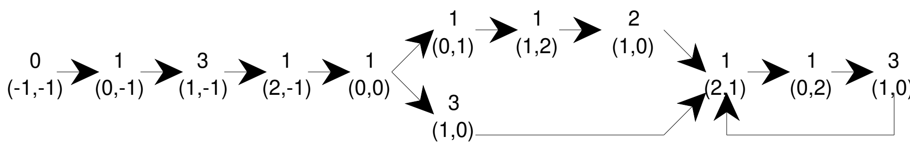
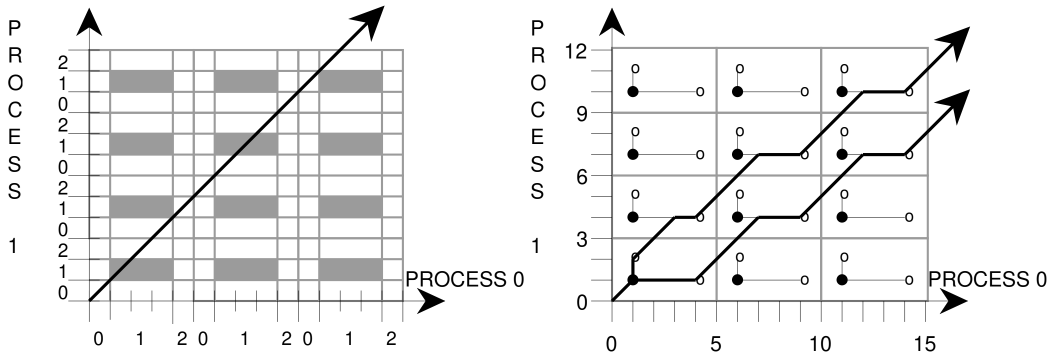


Figure 3: All possible states that can be reached in the timed transition diagram set (Fig. 2) solving the dining philosophers problem. The number above each state represents the time spent in each state.

Figure 4: Mapping of philosopher 0 running alone to the non-negative real axis. Numbers in parentheses denote locations in Fig.2.



(a) Mapping of states to Cartesian graph. Shaded regions represent illegal states. Axis labels denote locations.

(b) Timed progress graph. Open (filled) circles represent open (closed) end points. Thick lines represent all possible TETs (axis labels denote times).

Figure 5: Representations of Fig. 2.

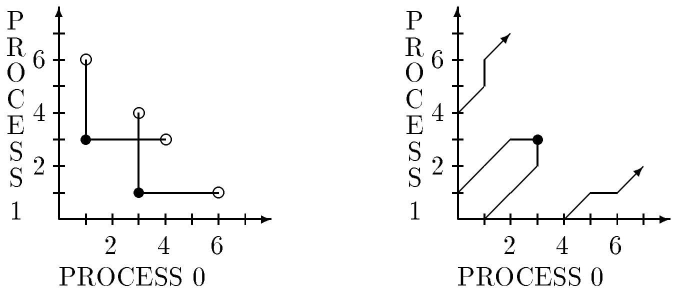
each illegal state by the left and lower sides of the corresponding rectangle; these "L" shaped lines are called *constraint lines.* Constraint lines mark forbidden transitions. In particular, philosopher 0 will block in any state represented by a point (x, y) meeting the following two conditions: $x \bmod 5 = 1$ (i.e., philosopher 0 wants to enter location 1) and $y \bmod 3 \in [1, 2)$ (i.e., philosopher 1 is in location 1). The set of points (x, y) in the plane satisfying these two equations are the vertical constraint lines in Fig. 5(b). A similar statement applies to the horizontal lines. The TET is constructed as follows: A TET cannot cross a constraint line. Starting at the origin, the TET for a program in which both processes simultaneously start execution can be constructed by moving on a diagonal line with slope one until a constraint line is intersected, then moving along the constraint line until its endpoint furthest from the origin is reached, then moving diagonally again, and so on. The heaviest lines in Fig. 5(b) denote the one possible TET.

If constraint lines overlap as shown in Fig. 6(a), then the program may deadlock. Point (3,3) is called a *dead* point, because it represents a state in which both processes are blocked at a P operation waiting for the other process to perform a V. Point (3,3) must be the final point of any TET that contains (3,3).

TET construction rule: Given a TPG, a TET is constructed recursively as follows. Let G denote a point; initially G is (0,0) if both processes start simultaneously. A line segment with open end point G and closed end point G' is denoted by $L = \overline{[G, G')}$. If both end points are open, $L = \overline{(G, G')}$.

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

Rule II: If G lies off a constraint line, then a slope one ray rooted at G either (1) will or (2) will not intersect a constraint line. In case (1), the TET rooted at G is a slope one ray with initial point G and



(a) Two semaphore TPG

(b) Sample TETs from (a) resulting from four initial points

Figure 6: Geometry of deadlock.

final point G', where G' is the only point on the ray that lies on a constraint line, followed by the set of TETs rooted at G'. In case (2), the TET rooted at G is an infinite length, slope one ray rooted at G.

Example 1 Consider the two TETs in Fig. 5(b). Point $G = (0,0)$ lies off a constraint line, and follows rule II, case (1). Thus the TET is ray $\overline{[(0,0),(1,1))}$ followed by the TETs rooted at (1,1). Next, $G = (1,1)$ lies on two constraint lines. Considering the vertical line first, rule I case (2) applies and one TET rooted at (1,1) is the ray $\overline{[(1,1),(1,2))}$ followed by the TET rooted at (1,2). Considering the horizontal line, case (2) again applies, and the other TET rooted at (1,1) consists of ray $\overline{[(1,1),(4,1))}$ followed by the TET rooted at (4,1). This process is continued. As a second example, in Figure 6(b) certain TETs lead to deadlock while others do not. The different initial points of the four TETs in Fig. 6(b) illustrate four different relative starting times of the two processes, which result when the value for a `delay` statement in the code is non-zero. In addition, 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 [11, 24, 28, 37, 38], deadlocks are not considered further. □

3 Timed Transition Diagrams

Our model of a program is based on Henzinger, Manna, and Pnueli's timed transition diagrams [20]. 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 $\langle \forall n : n \in Z :: n \leq \infty \rangle$[1] to simplify our notation. Let $r \in \{0,1\}$ denote one of two processes, $\bar{r} = 1 - r$ denote the other process (i.e., 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. 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 $\langle \exists r :: t_r^{-1} = 0 \wedge t_{\bar{r}}^{-1} \in R \rangle \wedge \langle \forall t_r^i : 0 \leq i < n_r :: t_r^i \in R^+ \rangle$. Conditions in the graph of process r name labels of process $\bar{r}$ satisfying $c_r^{-1} = \emptyset \wedge \langle \forall v : v \in c_r^i \wedge 0 \leq i < n_r :: v \in \{0, 1, \ldots, n_{\bar{r}} - 1\} \rangle$. 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

[1] We use the notation from [12] $\langle$ <quantified variable> : <domain of quantification> :: <quantified formula> $\rangle$, and omit <domain of quantification> when it is obvious from context.

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. Figures 2 illustrates a set of two timed transition diagrams.

Timed transition diagrams represent semaphore programs as follows. Let n_s denote the number of semaphores used in a program, and let $sem_0, sem_1, \ldots, sem_{n_s-1}$ denote the semaphores themselves. Each graph edge out of vertex $i \geq 0$ in a timed transition diagram 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 $\hat{\imath}, \hat{\imath}+1, \ldots, \hat{\imath}+k$ if edge $\hat{\imath}-1$ corresponds to a $P(sem_j)$ operation, edge $\hat{\imath}+k$ corresponds to a $V(sem_j)$ operation, and none of the edges in $\{\hat{\imath}, \hat{\imath}+1, \ldots, \hat{\imath}+k-1\}$ 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 $\{\hat{\imath}, \hat{\imath}+1, \ldots, \hat{\imath}+k\} \subseteq S_r^{sem_j}$. 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 2 For Fig. 2, $\Psi_0 = \langle\, 3,\ \langle 0,1,3,1 \rangle,\ \langle \{1\}, \emptyset, \emptyset \rangle\, \rangle$ and $\Psi_1 = \langle\, 3,\ \langle 0,1,1,1 \rangle,\ \langle \{1\}, \emptyset, \emptyset \rangle\, \rangle$. □

4 Timed Process Graphs

4.1 Definition of a Timed Progress Graph

Let upper case 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., $G^0 = (G_0^0, G_1^0)$), where, unless otherwise noted, $\langle \forall r :: G_r^0 \in R \ \wedge\ G_r^0 \neq \infty \rangle$. A ray, which is a directed line segment, uses the same notation. For any point G, $mod(G) \stackrel{def}{=} (G_0 \bmod \phi_0, G_1 \bmod \phi_1)$. Two points G and G' are *congruent*, denoted $G \equiv G'$, iff $mod(G) = mod(G')$. Two line segments are congruent if their end points are congruent. For any line or ray $L = \overline{[G, G')}$, $mod(L)$ denotes $\overline{[mod(G), mod(G'))}$. The *cycle time* of process r with diagram Ψ_r, denoted ϕ_r, is the time required for process r to pass through locations 0 to $n_r - 1$ once, ignoring the time spent blocked. Formally, $\langle \forall r :: \phi_r \stackrel{def}{=} \sum_{0 \leq i < n_r} t_r^i \rangle$.

A TPG is uniquely specified by enumerating the cycle time of each process, the positions of the constraint lines, and the initial point corresponding to the initial program state.

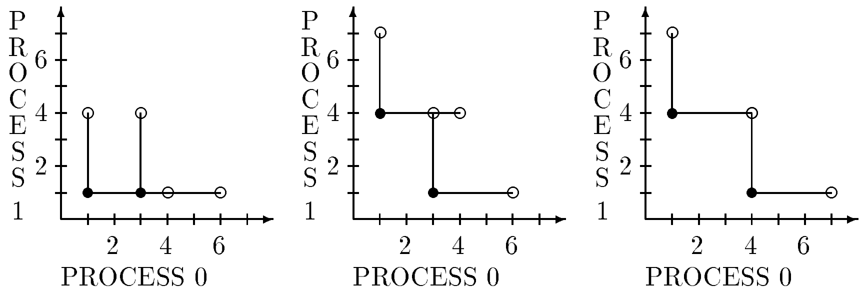
Definition. *A* <u>TPG</u> *for timed transition diagram set* $D = \langle \Psi_0, \Psi_1 \rangle$ *is* $\Gamma_D = \langle (\phi_0, \phi_1), \Lambda, G^C \rangle$*, where:*

Λ *is a set of* <u>constraint line generators</u>*, 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 non-empty condition* c_r^k *in* Ψ_r*, if* $\{\hat{\imath}, \hat{\imath}+1, \ldots, \hat{\imath}+z\} \subseteq c_r^k$*, for some* $\hat{\imath}, z \in Z$*, we write* $\langle \hat{\imath}, \hat{\imath}+z \rangle \subseteq c_r^k$*. Formally, generator* $\overline{[W, X)} \in \Lambda$ *iff*

$$\langle\, \exists r \exists k \exists i \exists i' : 0 \leq k < n_r, :: \ \langle i, i' \rangle \subseteq c_r^k \ \wedge$$
$$W_r = X_r = \sum_{0 \leq j \leq k} t_r^j \ \wedge\ W_{\bar{r}} = \sum_{0 \leq j < i} t_{\bar{r}}^j \ \wedge\ X_{\bar{r}} = \sum_{0 \leq j \leq i'} t_{\bar{r}}^j \,\rangle.$$

The <u>initial</u> *and* <u>final</u> *points of generator* $\overline{[W, X)}$ *are* W *and* X*, respectively. The* <u>instances</u> *of a generator are defined to be all lines in the* R^2 *plane congruent to the generator; formally, all instances of generators in* Λ *is* $\Lambda^* \stackrel{def}{=} \{\overline{[W, X)} \mid W \in R^2 \ \wedge\ X \in R^2 \ \wedge\ mod(\overline{[W, X)}) \in \Lambda\}$.

G^C *is an* <u>initial point</u> *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 execution. Formally,* G^C *satisfies* $\langle \forall r :: G_r^C = t_r^{-1} \bmod \phi_r \rangle$.



(a) Process one does simultaneous P ops

(b) Process one does simultaneous P, V

(c) Both processes do simultaneous P, V

Figure 7: Illegal constraint line geometries.

Example 3 The TPG of Fig. 5(b) is $\Gamma = \langle(\phi_0 = 5, \phi_1 = 3), \Lambda, (0,0)\rangle$, where $\Lambda = \{\ \overline{[(1,1),(1,2))}, \overline{[(1,1),(4,1))}\ \}$. There are twelve instances of each constraint line generator. Generator $\overline{[(1,1),(1,2))} \in \Lambda$ because process 0 performs $P(a)$ and process 1 holds a in location 1, $c_0^0 = \{1\}$ and $\langle 1,1\rangle \subseteq c_0^0$. 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 $X_1 = \sum_{0\leq j\leq 1+0} t_1^j = 2$. □

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 any two constraint lines may have at most one point in common (violated in Fig. 7(a)), the final point of one constraint line can never lie on another constraint line (violated in Fig. 7(b)), and the final point of all constraint lines must be distinct (violated in Fig. 7(c)).

4.2 Construction of a TET and Transition Function f

The following notation is required to describe TETs. For any continuous path γ and any point G on γ, we write $G \in \gamma$. For any directed, continuous path γ, we write $\gamma.i$ (respectively, $\gamma.f$) to denote the initial (final) point. We assume that $\gamma.i \neq \gamma.f$. For any two continuous paths γ and γ' in R^2, $\gamma \cap \gamma'$ denotes $\{G \mid G \in \gamma \wedge G \in \gamma'\}$. The relation $G \in L$ holds iff G lies on line or ray L. Line or ray $\overline{[G,(\infty,\infty))}$ has slope one and infinite length. For any two points G and G', $(G < G') \stackrel{def}{=} ((G_0 + G_1) < (G'_0 + G'_1))$.

Transition Function f: A TET is a point or a sequence of one or more rays whose end points are determined by a *transition function* f. Function f maps each point $G \in R^2 - \{(\infty,\infty)\}$ to a (possibly empty) set of successors $f(G) \subseteq R^2$, where $||f(G)|| \leq 2$. f is defined in terms of two more primitive functions, f_o and f_d, that formalize rules I and II in §2, respectively: $f(G) \stackrel{def}{=} f_o(G) \cup f_d(G)$. (The subscript "o" in f_o denotes a ray that is orthogonal to the axes, and the "d" in f_d denotes a ray that is diagonal [with slope one].) Informally,

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

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

Definition. *Transition function* $\underline{f}$ *is defined as follows. First, for a point* G *on a constraint line,* $\underline{S_o(G,L)} \stackrel{def}{=} \min\left(\{L.f\} \cup \{G' \mid G' > G \wedge \langle \exists L' : L' \in \Lambda^* - \{L\} :: G' \in (\overline{(L.i, L.f)} \cap L')\rangle\}\right)$. *For* G *off a constraint line,* $\underline{S_d(G)} \stackrel{def}{=} \min\left(\{(\infty,\infty)\} \cup \{G' \mid \langle \exists L' : L' \in \Lambda^* :: G' \in \overline{[G,(\infty,\infty))} \cap L'\rangle\right)$. *Second,* $f(G) \stackrel{def}{=} f_o(G) \cup f_d(G)$, *where* f_o *is defined to be a function satisfying* $\hat{G} \in f_o(G)$ *iff* $\langle \exists L : L \in \Lambda^* \wedge G \in L :: \hat{G} = S_o(G,L)\rangle$ *and* f_d *satisfies* $\hat{G} \in f_d(G)$ *iff* $\langle \nexists L : L \in \Lambda^* \wedge G \in L\rangle \wedge \hat{G} = S_d(G)$.

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

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 $f(G) = \emptyset$.*

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

Definition. *A* timed execution trajectory *(TET) 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 $\langle \forall i : 0 \leq i < n :: G^{i+1} \in f(G^i) \rangle$. In case (2), n is finite iff $f(G^n) = \emptyset$.*

Example 4 The heavy lines in Fig. 5(b) denote all possible TETs rooted at point (0,0) for the single philosopher dining philosophers example. They are constructed as follows. The initial point, G^0, is (0,0). The first line segment is $\overline{[G^0, G^1)}$, where $G^1 = (1, 1)$ because $f((0, 0)) = f_d((0, 0)) = \{(1, 1)\}$ (i.e., (0,0) does not lie on a constraint line instance and a slope one ray rooted at point (0,0) first intersects a constraint line instance at point (1,1)). Next, $f((1, 1)) = f_o((1, 1)) = \{(4, 1), (1, 2)\}$. Thus point (1,1) is nondeterministic. Thus there are two possible TETs rooted at (1,1). One TET has $G^2 = (4, 1)$, and the other has $G^2 = (1, 2)$. Continuing in this manner yields $\overline{[(0, 0), (1, 1))}$, $\overline{[(1, 1), (4, 1))}$, $\overline{[(4, 1), (7, 4))}$, $\overline{[(7, 4), (9, 4))}$, ... as one TET, and $\overline{[(0, 0), (1, 1))}$, $\overline{[(1, 1), (1, 2))}$, $\overline{[(1, 2), (3, 4))}$, $\overline{[(3, 4), (4, 4))}$, $\overline{[(4, 4), (7, 7))}$, ... as the second TET. □

Example 5 As a second example, Fig. 6(b) illustrates four TETs corresponding to four different choices of process starting times: $G^C \in \{(4, 0), (1, 0), (0, 1), (0, 4)\}$. Let d_r ($r \in \{0, 1\}$) denote the delay of process r before running. For $d_0 = 0$ and $d_1 = 4$, process 0 starts first and runs for 4 time units before process 1 starts. The initial ray of the corresponding TET has length 4 and lies on the process 0 axis. Therefore point G^C, representing the earliest time at which both processes are running, is (4,0). Whereas all TETs in Fig. 5(b) are of infinite length, a TET with initial point (0,1) in Fig. 6(b) has finite length because $f_d(0, 1) = \{(2, 3)\}$, $f_o(2, 3) = \{(3, 3)\}$, and $f(3, 3) = \emptyset$. □

4.3 Congruent TETs

Definition. *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.*

In Fig. 5(b), 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 $(3, 4)$. Adding vector (ϕ_0, ϕ_1) to the trajectory rooted at $(3, 1)$ yields the trajectory rooted at $(8, 4)$. Lemma 1 shows that TETs containing only deterministic points and rooted at congruent points are congruent.

Lemma 1 *Consider any two TETs γ and γ' of a TPG 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 Appendix B. □

4.4 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 6 Figure 5(b) contains two TETs. The TET containing point $(4,1)$ contains a transient execution trajectory with initial and final points $(0,0)$ and $(3,1)$, 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, demonstrates that any TET 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 $||\{G|G \in \gamma \wedge C_N(G)\}|| < \infty \ \wedge \ \langle \nexists G : G \in \gamma :: C_D(G) \rangle$.*

Proof outline: (See Appendix C for a formal proof.) *If part:* Let G be the largest nondeterministic point in γ; delete the initial subtrajectory of γ that has G as its final point. The remaining trajectory, denoted $\hat{\gamma}$, contains only deterministic points. Either $\hat{\gamma}$ intersects (a) a finite (possibly zero) number of constraint lines or (b) the complement of (a). In case (a), $\hat{\gamma}$ consists of a finite number of rays, and the final ray has infinite length (e.g., the ray rooted at (15,3) in Fig. 8). Choose any points $G', G' + (\phi_0, \phi_1), G' + (2\phi_0, 2\phi_1), \ldots$ on the final ray; the points are all congruent. In case (b), $\hat{\gamma}$ must contain an infinite number of rays. Because there are only a finite number of constraint lines (see Lemma 2), $\hat{\gamma}$ must intersect an instance of some constraint line L an infinite number of times. By hypothesis, $\hat{\gamma}$ contains no dead points; thus each time it intersects an instance of L the TET must pass through a point congruent to the final point of L. In either case (a) or (b), $\hat{\gamma}$ passes through a congruent point an infinite number of times, and, by Lemma 1, γ must contain an infinite number of congruent LCETs. The formal proof in Appendix C uses induction on the length of $\hat{\gamma}$ to make this argument rigorous.

Only if part: γ ends in an infinite number of repetitions of a LCET, and the transition function f applied to any point in any LCET cannot yield the empty set. Thus by definition γ contains no dead points. Furthermore, by definition the LCET contains deterministic points, and the portion of a TET preceding an LCET has finite length; thus γ contains a finite number of nondeterministic points. □

4.5 Timed Progress Graph Sets

Subsequent sections develop a solution for the problem of finding a finite representation for the set of blocking 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 §3). Therefore we define a timed progress graph *set* (TPGS), which lacks starting point G^C of a TPG. A TPGS with cycle lengths ϕ_0 and ϕ_1 and constraint line generator set Λ is denoted $\langle(\phi_0, \phi_1), \Lambda\rangle$. A point or a directed, continuous path γ is a TET in $\langle(\phi_0, \phi_1), \Lambda\rangle$ iff $\langle \ \exists r, \exists G^C : G^C_r = 0 \ \wedge \ 0 \leq G^C_{\bar{r}} < \phi_{\bar{r}} :: \gamma$ is a TET in $\langle(\phi_0, \phi_1), \Lambda, G^C\rangle \ \rangle$. Γ^+ denotes the TPGS that subsumes Γ.

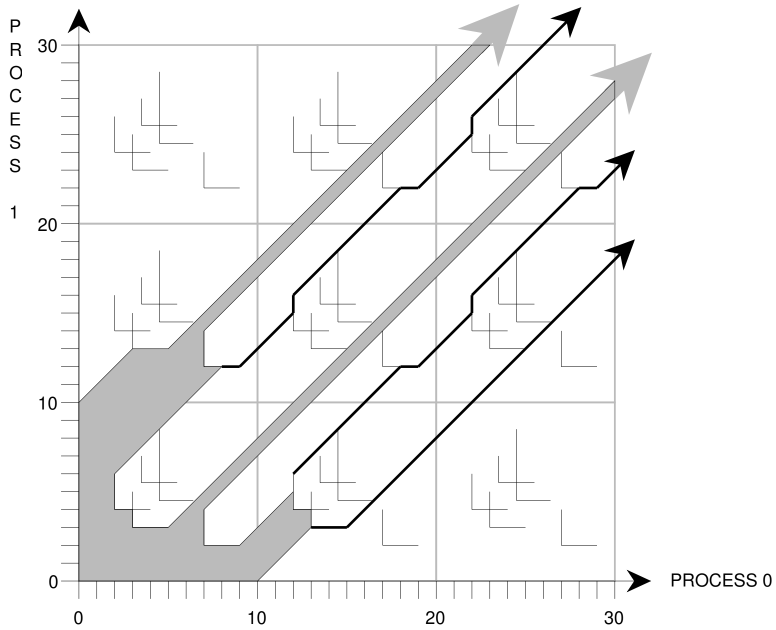


Figure 8: Illustration of a TPGS.

Example 7 Figure 8 shows a TPGS $\Gamma^+ = \langle(\phi_0, \phi_1), \Lambda\rangle$ 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. 8 may reach a nondeterministic state, a dead state, or a limit cycle execution sequence as indicated in Table 1. 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 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 LCETs are congruent. Therefore the problem of representing all possible LCETs reduces to finding one member of each congruence class of LCETs.

An LCET that contains at least one nondiagonal ray is characterized as *blocking*, and a *non-blocking* 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 6 and Fig. 5(b), philosopher 1 waits two time units (e.g., because each 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 0 never blocks at a P operation. In contrast, the explicit form of a non-blocking 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 Lemma 5 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,[2] and (2) reports one member of each congruence class of blocking LCETs. Presented in

[2]Carson and Reynolds' deadlock detection algorithm [11] cannot directly be applied, because a consequence of the

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. 8 TPGS.

	$f^{2m}(X)$	
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. 5(b).

algorithm A0 below is a solution to (2); (1) is addressed in part in [2].

The algorithm uses the following notation. For any point G, if $||f(G)||=1 \wedge f(G)=\{G'\}$, we drop the set notation and write $f(G)=G'$. 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 $||f(G)||=1$ then $\Delta f(G) \stackrel{def}{=} f(G)-G$; thus $\Delta f(G)$ is a vector representing the transition from the state represented by G to the state represented by $f(G)$.

5.1 Algorithm A0

Let ξ denote the set of final points of all constraint line generators; formally $\xi := \{\, X \mid \overline{[W,X)} \in \Lambda \,\}$. Let $N=||\xi||$. Algorithm A0 (Fig. 9) examines the N points in ξ. A0 uses a set, S, to store each point already examined that is either in ξ or is congruent to an element of ξ. Each **for each** loop iteration 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 $f^{2n}(X) \equiv X$ for each $X \in \xi$, if such an n exists. If n exists, then X lies 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 is addressed in §6 and §7.

Example 8 The TPGS for Fig. 5(b) contains two constraint line generators, as stated in Example 3. Thus $\xi=\{(4,1),(1,2)\}$. From Table 2, for $X=(4,1)$, $f^2(X) \equiv X$. However, for $X=(1,2)$, $\langle \nexists m : m \in \{1,2,\ldots,N\} :: f^{2m}(X) \equiv X \rangle$, because the TET rooted at (1,2) never intersects an instance of a constraint line with final point (1,2). Hence a single trajectory, congruent to those in Example 6, is output: point $X=(4,1)$ and vector sequence $(3,3)$, $(2,0)$. □

assumption in §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: $\langle(\phi_0, \phi_1), \Lambda\rangle$; Output: list of LCETs }
    declare S: set of points := $\emptyset$; { Set of points examined already for membership in a LCET }
    for each X **in** ξ **do**
        if $\langle \nexists G : G \in S :: G \equiv X\rangle \ \wedge \ ||f^0(X)|| = ||f^1(X)|| = \ldots = ||f^{2N}(X)|| = 1 \ \wedge$
                $\langle \exists m : m \in \{1, 2, \ldots, N\} :: f^{2m}(X) \equiv X\rangle$
        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 (less than N) 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 9: Algorithm A0, which outputs one member of each congruence class of blocking LCETs.

5.2 Correctness of Algorithm A0

Lemmas 4 through 7 establish the correctness of algorithm A0; see Appendix A for the proofs. The following lemma establishes that N is finite.

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 $\langle i, i'\rangle \subseteq c_r^i$. □

Lemma 3 *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. □

Lemma 4 *Every trajectory output by algorithm A0 is a blocking LCET.*

Lemma 5 establishes that A0 outputs *all* congruence classes of blocking LCETs that exist, which implies that examining *just* the N points in set ξ is sufficient. The intuitive justification follows. Two TETs, rooted at G' and G'', are said to *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 subtrajectories are congruent by Lemma 1. For example, in Fig. 8 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.

Lemma 5 *Any blocking LCET is congruent to one of the trajectories output by algorithm A0.*

Corollary to Lemma 5 *There are at most $||\Lambda||$ congruence classes of blocking LCETs in a TPGS.*

Proof: By Lemma 5 the number of congruence classes of blocking LCETs cannot exceed the number of constraint line generators, or, $||\Lambda||$. □

Lemma 6 *None of the trajectories output by algorithm A0 are congruent.*

Lemma 7 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.

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 $\Delta f(X^{k_1})$, $\Delta f(f(X^{k_1}))$, $\Delta f(X^{k_2})$, $\Delta f(f(X^{k_2}))$, ..., $\Delta f(X^{k_{M-1}})$, $\Delta f(f(X^{k_{M-1}}))$.

Figure 10: Algorithm A1, a refinement of algorithm A0.

Lemma 7 *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

Questions unanswered in Algorithm A0 are: (1) When are $\langle \forall X : X \in \xi :: f^0(X), f^1(X), \ldots, f^{2N}(X)\rangle$ defined? (2) How can $\langle \forall X : X \in \xi :: f^0(X), f^1(X), \ldots, f^{2N}(X)\rangle$ 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. 10), 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 §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 directed graph of N nodes in which each node has out-degree of zero or one, which reduces the worst case number of evaluations of f from $O(N^2)$ to $O(N)$. (This will be proved as a property of the second refinement, A2, in §8.) In algorithm A1, the N constraint line generators are denoted $\overline{[W^0, X^0)}$, $\overline{[W^1, X^1)}$, ..., $\overline{[W^{N-1}, X^{N-1})}$; also k and k' denote integers in interval $[0, N)$. Answering question (2) appears to be non-trivial, and is the subject of §7 and the second refinement, algorithm A2.

Proceeding with the first question, the terms *live* and *restricted* point are first defined. Informally, a point G in a TPGS is live, denoted $C_L(G)$, if G represents a deterministic and blocked 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, $C_L(G) \stackrel{def}{=} \langle \exists L : L \in \Lambda^* :: G \in L \wedge S_o(G, L) = L.f\rangle$. A point G is restricted, denoted $C_R(G)$, if G represents a deterministic state in which all processes are running, and in all TESs containing the state, in some subsequent state some process is blocked. (A diagonal ray rooted at a restricted point intersects a constraint line and thus cannot have infinite length; hence the name "restricted.") Formally, $C_R(G) \stackrel{def}{=} \langle \nexists L : L \in \Lambda^* :: G \in L\rangle \wedge S_d(G) \neq (\infty, \infty)$.

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

Lemma 8 $f(G) \neq \emptyset \wedge f(G) \neq (\infty, \infty)$ iff $C_L(G) \vee C_R(G)$.

Proof: Follows from the definition of f. □

Example 9 Let the two constraint line generators in Fig. 5(b) be denoted $\overline{[W^0, X^0)} = ((1,1),(4,1))$ 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 8. □

7 Formulas for $\mathbf{C_R(X)}$, $\mathbf{C_L(f(X))}$, $\mathbf{g(X)}$, $\mathbf{g(f(X))}$

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 $C_R(X) \wedge C_L(f(X))$. For

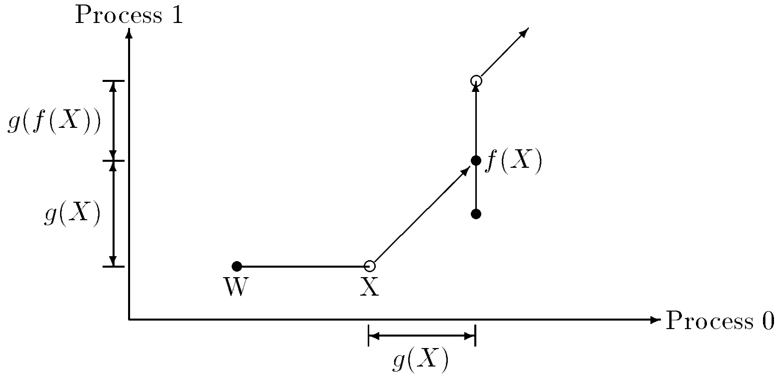


Figure 11: Illustration of four unknowns in Algorithm A1: $C_R(X)$, $C_L(f(X))$ if $C_R(X)$, $g(X)$ if $C_R(X)$, and $g(f(X))$ if $C_R(X)$ and $C_L(f(X))$.

simplicity, we will reason about scalars rather than vectors, by defining $g(G)$ to be a nonzero 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 nonzero and equal.)

The notation, illustrated in Fig. 11, is interpreted geometrically and programmatically as follows.

$C_R(X)$: 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?)

$g(f(X))$: 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 TPGS satisfying $C_R(G) \wedge C_L(f(G))$. This is because restricting G to a point in ξ 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 $\mathbf{C_L(G)}$

By definition, evaluating $C_L(G)$ for point G on line $L \in \Lambda^*$ requires evaluating $S_o(G, L)$. However, $S_o(G, L)$ 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 2 *$C_L(G)$ is true in TPGS $\langle(\phi_0, \phi_1), \Lambda\rangle$ 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: By the definition of Λ^* and C_L, $C_L(G) = \neg C_N(G) \wedge \langle \exists L : L \in \Lambda \wedge mod(G) \in L :: S_o(G, L) = \emptyset\rangle$. By the definition of Λ and Λ^*, $\langle \forall L, \forall L' : L \in \Lambda \wedge L' \in (\Lambda^* - \Lambda) :: L \cap L' = \emptyset\rangle$. By last two deductions and definition of S_o, $C_L(G) = \neg C_N(G) \wedge \langle \exists L : L \in \Lambda \wedge mod(G) \in L :: \{G' | G' \in L \cap \Lambda - \{L\} \wedge G' > mod(G)\} = \emptyset$. 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 (e.g., [9]) 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 2.

7.2 Analytic Solution of $\mathbf{C_R(G)}$, $\mathbf{g(G)}$, and $\mathbf{g(f(G))}$

The solution method presented here assumes that constants ϕ_0 and ϕ_1 represent rational quantities. We use an equivalent assumption: *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 $\langle(\phi_0, \phi_1), \Lambda\rangle$, $\underline{C_R(G, W, X)}$ and $\underline{g(G, W, X)}$ are defined as $C_R(G)$ and $g(G)$, respectively, in TPGS $\langle(\phi_0, \phi_1), \{\overline{[W, X)}\}\rangle$.*

Therefore:

$$
\begin{aligned}
C_R(G) &= \bigvee_{\{\overline{[W,X)} \in \Lambda\}} C_R(G, W, X) \quad \text{and} &(1)\\
g(G) &= \min\{g(G, W, X) \mid \overline{[W, X)} \in \Lambda \wedge C_R(G, W, X)\} \text{ if } C_R(G). &(2)
\end{aligned}
$$

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 $C_R(G)$, $g(G)$, $g(f(G))$.

We henceforth assume without loss of generality that the generator in TPGS $\langle(\phi_0, \phi_1), \{\overline{[W, X)}\}\rangle$ is a horizontal line: $W_0 < X_0 \wedge W_1 = X_1$. (Interchange subscripts 0 and 1 below for a vertical line.)

7.2.1 Formula for $\mathbf{C_R(G)}$

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. 12.

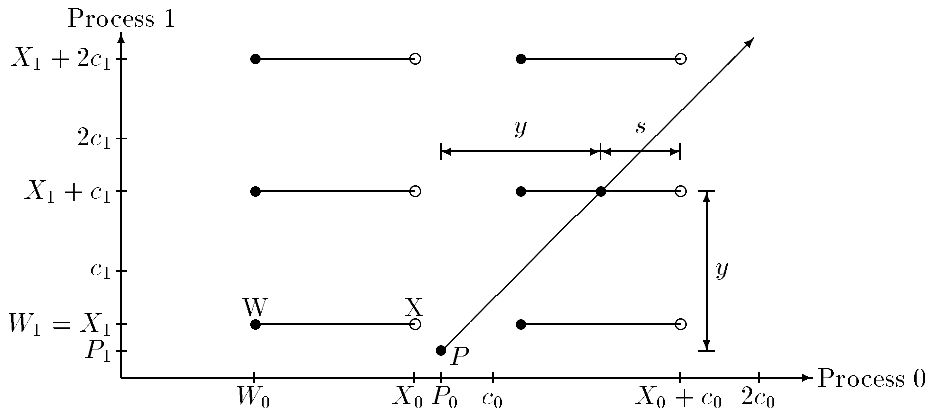


Figure 12: One possible relationship of G, y, and s in a TPGS with constraint lines generated by a single generator, $\overline{[W, X)}$. In general, there are either zero or an infinite number of points of intersection of a slope one diagonal ray with instances of a single constraint line generator, corresponding to zero or an infinite number of values of y, respectively. In the figure, $i_0 = i_1 = 1$ yields $g(G, W, X) = y$.

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

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

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):

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

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 3 *$C_R(G)$ is true in TPGS $\langle(\phi_0, \phi_1), \Lambda\rangle$ 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 $\mathbf{g(G)}$ and $\mathbf{g(f(G))}$ if $\mathbf{C_R(G) \wedge C_L(f(G))}$

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 4 *In TPGS $\langle(\phi_0, \phi_1), \Lambda\rangle$, 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 4 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 2 to 4 in algorithm A2 (Fig. 13). For TPGS $\langle(\phi_0, \phi_1), \Lambda\rangle$, A2 the edges of the graph that A1 generates in $N \times 1$ array E; for example E[0]=3 represents an edge from node 0 to node 3. The state transition vectors output by A1 are stored in $N \times 1$ arrays G0 and G1. The time required by A2 is dominated by evaluating the minimization in Theorem 4, as the following theorem establishes. Let $I_N(G,W,X)$ denote the number of integers in interval $I(G,W,X)$. Let $D = \max\{I_N(X^k, W^{k'}, X^{k'}) | k, k' \in \{0, 1, \ldots, N-1\}\}$.

Theorem 5 *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 A2.

Step A2.0: Step A2.0 requires $O(N)$ time due to the universal quantifier.

Step A2.1: The intersection may be computed using an algorithm that reports the points of intersection of horizontal and vertical line segments (e.g., algorithm 4.1 in [9]). 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 at worst time $O(N^2)$.

input: $\phi_0, \phi_1, N, \Lambda$; **output:** list of blocking LCETs

L,K,K': integer; T,LG,U,S': real;
E: **array** [0..N-1] **of** integer; {*Stores graph for step 1.2 in algorithm A1* }
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* $C_R(X^K) \wedge C_L(f(X^K))$}
MI: **array** [0..N-1,0..1] **of** real; {*Intersection point furthest from origin for each generator*}
M: **array** [0..N-1] **of** real; {M[K] = s^* *(from Theorem 4)*}

{*Step A2.0 (Initialization)*}
L:=l.c.d. of ϕ_0 and ϕ_1; LG:=g.c.d. of L$*\phi_0$ and L$*\phi_1$; ϕ_0:=ϕ_0*LG; ϕ_1:=ϕ_1*LG;
solve $\phi_1 * \mathrm{U} \equiv 1 \pmod{\phi_0}$ **for** U;
$\langle \forall \mathrm{K} : 0 \le \mathrm{K} < \mathrm{N} ::$ E[K]:=G0[K]:=∞; $\langle \forall \mathrm{r} :: \mathrm{W}^{\mathrm{K}}_{\mathrm{r}} := \mathrm{W}^{\mathrm{K}}_{\mathrm{r}}*\mathrm{LG}; \quad \mathrm{X}^{\mathrm{K}}_{\mathrm{r}} := \mathrm{X}^{\mathrm{K}}_{\mathrm{r}}*\mathrm{LG} \rangle \rangle$

{*Step A2.1 (Part of step A1.1 in Algorithm A1)*}
$\langle \forall \mathrm{K} : 0 \le \mathrm{K} < \mathrm{N} :: \mathrm{MI[K]} := \max\{\mathrm{G} | \mathrm{G} \in \overline{[\mathrm{W}^{\mathrm{K}}, \mathrm{X}^{\mathrm{K}})} \cap \Lambda - \{\overline{[\mathrm{W}^{\mathrm{K}}, \mathrm{X}^{\mathrm{K}})}\}\} \rangle$

{*Step A2.2 (Part of step A1.1 in Algorithm A1)*}
$\langle \forall \mathrm{K}, \mathrm{K}' : 0 \le \mathrm{K}, \mathrm{K}' < \mathrm{N} ::$ S':=$\lceil \mathrm{X}^{\mathrm{K}'}_1 - \mathrm{X}^{\mathrm{K}'}_0 - \mathrm{X}^{\mathrm{K}}_1 + \mathrm{X}^{\mathrm{K}}_0 \rceil$;
while S'$\le \lfloor X^{K'}_1 - X^K_1 + X^K_0 - W^{K'}_0 \rfloor$ **do begin**
T:=$(\phi_1 * \mathrm{U} * \mathrm{S}' + \mathrm{X}^{\mathrm{K}'}_1 - \mathrm{X}^{\mathrm{K}}_1) \bmod (\phi_0 * \phi_1)$;
if T<G0[K] **then begin** E[K]:=K'; G0[K]:=T; M[K]:=S' **end**
S':=S'+1;
end
if MI[K] < (X^K + G0[K]) **then** {$C_L(X^K)$ *holds*} G1[K]:=M[K]-IL(K,K') $\rangle$

{*Step A2.3 (Corresponds to step A1.2)*}
for each cycle $k_1, k_2, \ldots, k_M$ in graph in array E (where $k_1 = k_M$) **do begin**
output point ($\mathrm{X}^{k_1}_0$/LG, $\mathrm{X}^{k_1}_1$/LG); **for** K:=1 **to** M-1 **do output** GO[K]/LG, G1[K]/LG
end

Figure 13: Algorithm A2. Comments show implementation of each step in algorithm A1.

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

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 6 *Algorithm A2 requires $\Omega(N)$ storage, excluding the storage to required compute ι.*
Proof: Follows from the array dimensions in declaration portion of A2 (Fig. 13). □

Theorems 5 and 6 show that A2 requires space $\Omega(N)$ and time $O(N^2 D)$, where D is related to the precision with which measurements of program timings are desired. 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 §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 $D = 300 \cdot 10^3$.

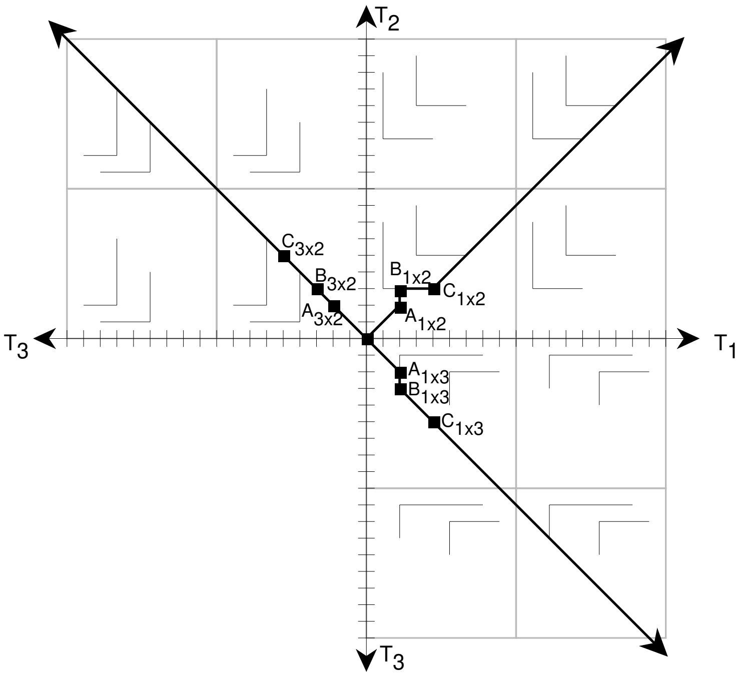


Figure 14: Projection of a three dimensional TPG into a set of three two dimensional TPGs.

9 More Than Two Processes

There is promise in using timed progress graphs with more than two processes. To illustrate, consider the three database transactions used by Lipski and Papadimitriou (Fig. 11 in [24]) and later by Carson and Reynolds (Appendix in [11]). A TET in three dimensions can be constructed by drawing one TPG for every pair of processes, as shown in Fig. 14. (See [24] for a true three dimensional sketch.) Each axis in each graph is labeled by one of three transaction numbers (T1, T2, or T3). We denote the graphs as 1×2 (for the graph with axes T1 and T2), 1×3, and 3×2.

To construct the TET with initial point (0,0,0), we calculate $g((0,0))$ in each graph. (Recall from §7 that $g(G)$ is a nonzero component of ray $\overline{[G, f(G))}$, representing the length of the perpendicular projection of the ray on an axis to which the ray is not perpendicular.) Now $g((0,0))$ is 3, 2, and ∞ in graphs 1×2, 1×3, and 3×2, respectively. Therefore the initial ray in the TET in each TPG i has initial point $(0,0)$ and length 2 and lies on $\overline{[(0,0), f(0,0))}$. Denote the final point of the first TET ray in TPG i as A_i. Because $A_{1\times 3}$ lies on a constraint line perpendicular to the T1 axis, process T1 will now block for $g(A_{1\times 3}) = 1$ time unit. We temporarily add a constraint line of length 1 with initial point $A_{1\times 2}$ perpendicular to the T1 axis in any other graph that contains an axis for T1, namely graph 1×2. Now we repeat the procedure we used to find the first TET ray: Find the minimum value of $g(A_i)$ in each graph i; it is 1 (from both graphs 1×2 and 1×3). Therefore the second TET ray in TPG i has initial point A_i and length 1 and lies on $\overline{[A_i, f(A_i))}$. Let B_i be the final point of this ray in TPG i. We repeat the procedure with each point B_i to find C_i as indicated in Fig. 14. Next we repeat the procedure to find the third TET ray: Find the minimum value of $g(B_i)$ in each graph i; it is 2 in graphs 1×2 and 3×2. Thus the third TET ray has initial point B_i and length 2 and lies on ray $\overline{[B_i, f(B_i))}$. Because C_i does not lie on a constraint line in any graph, repeating the procedure again shows that $g(C_i) = \infty$ in *all* graphs i. Thus the fourth and final TET ray in each graph is $\overline{[C_i, (\infty, \infty))}$. *Thus the program reaches a non-blocking LCES.*

Based on this exercise, one must modify the definition of f and TET from §4.2 for more than two processes; however the modification is straightforward because one can visualize the process by a set of two dimensional projections of the three dimensions. Therefore, one can construct the TET from TPGs in three and probably more dimensions. In fact, in k dimensions, the analysis complexity will not grow as $\binom{k}{2}$ for k processes if some of the two process projections are identical. For example, a program with k identical processes reduces to just one two-dimensional TPG. However, the problem addressed in this paper – finding a finite representation of the set of all blocking LCETs – cannot be solved by algorithm A0 (or its refinements A1 and A2). A0 examines just the final point of all constraint line generators, which is sufficient to discover all blocking LCETs, because every blocking LCET passes through at least one final point of at least one generator. The property does not hold in three dimensions, as Fig. 14 illustrates: points $C_{1\times 2} \neq C_{1\times 3} \neq C_{3\times 2}$. Thus if we consider a blocking LCET that passes through the final point of one constraint line, such as $C_{1\times 2}$, *we do not in general know to what point it corresponds in the other graphs.*

10 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 the program reaches.

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 in §1.1 provide *average* measures (i.e., mean cycle time with the Ramamoorthy/Ho and Magott solutions, and the long run fractions of time that each process spends in a state with the Holliday/Vernon solution.) 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.

Compared to TPGs, simulation output represents only one possible execution behavior for one initial condition, despite possible nondeterministic behavior. Also a simulation generates a finite length prefix of an execution sequence which may not be long enough to reveal a LCES.

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. 6, 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 differ 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 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].

Open problems include relaxing the assumptions in §1.3 to permit geometric modeling of a broader class of programs; characterizing the necessary and sufficient conditions for a program to reach a LCES; and model analysis for irrational cycle times ϕ_0 and ϕ_1 (Chaotic behavior may exist for these values.).

Acknowledgements: Part of this work was done under A. K. Agrawala in the authors' dissertation research. He and S. K. Tripathi provided many important discussions. Comments from L. Svobodova, D. Kafura, and anonymous referees on various manuscript drafts improved the final paper.

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. An example of deriving performance properties from a visual represention of program execution. Technical Report TR 95-09, Computer Sci. Dept., Virginia Tech, Blacksburg, VA 24061-0106, June 1995. Available from World Wide Web location http://www.cs.vt.edu/reports.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

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

[16] 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.

[17] R. Govindarajan and G. R. Gao. A novel framework for multi-rate scheduling in dsp applications. In *Proc. Inter. Conf. on Application-Specific Array Processors*, pages 77–88, Venice, Italy, Oct. 1993. IEEE Press.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[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] R. Reiter. Scheduling parallel computations. *JACM*, 15(4):590–599, Oct. 1968.

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

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

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

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

[37] 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.

[38] 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.

[39] 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_d

If $i > 0$ then $f^{2i}(X) \in \xi$
, by applying definition of f_o 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_o

$C_L(f^{2i+1}(X))$
, 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 4 *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 9 *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 5 *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_o

$\exists X \in \xi$, 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 9

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

, by F3 and last deduction

A0 outputs γ''

, by last deduction and Theorem 4 □

Lemma 10 *Given a point, all LCETs containing that point are congruent.*
Proof: Follows from Lemma 9. □

Theorem 6 *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 6.

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

, by Lemma 10

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 7 *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 9 and last two deductions □

B Proof of Lemma 1

The following lemma will simplify the proof of Lemma 1.

Lemma 11 *For any two congruent points G and G' in a TPG, if $C_D(G)$ then $f(G) = f(G')$.*

Proof:
$f(G) = \emptyset$
, by hypothesis and definition f
$\langle \exists L, \exists L' : L, L' \in \Lambda^* :: G \in L \cap L' - \{L.i\}\rangle$
, by hypothesis $C_D(G^1)$ and definition of f
$\langle \exists L, \exists L' : L, L' \in \Lambda^* :: G' \in L \cap L' - \{L.i\}\rangle$
, by the last deduction, the definition of Λ^*, and because $G \equiv G'$
$C_D(G')$
, by last deduction and definitions of f and C_D
$f(G^2) = \emptyset$
, by last deduction and definition of f
$f(G) = f(G')$
, by first and last deductions □

Lemma 1 *Consider any two TETs γ and γ' of a TPG 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: The TET γ is either a single point or a ray sequence. Consider first the single point case. Then $\gamma = G$ and G is dead. By Lemma 11, $\gamma = \gamma'$.

Consider next the ray sequence case. The proof consists of demonstrating that the i-th ($i = 1, 2, \ldots$) noncollinear ray in the execution trajectories rooted at G and G' are congruent by induction on the value of i. Let G^1 and G^2 (respectively, $\hat{G}^1$ and $\hat{G}^2$) be the initial (respectively, final) points of the i-th noncollinear ray in the execution trajectories rooted at G and G', respectively. The formula to be proven by induction is

$$G^1 \equiv G^2 \Rightarrow \begin{cases} C_D(G^2) & \text{if } C_D(G') \\ f(G^1) = \{\hat{G}^1\} \wedge f(G^2) = \{\hat{G}^2\} \wedge \hat{G}^1 \equiv \hat{G}^2 & \text{otherwise} \end{cases}.$$

The proof for all i is the same. If $i = 1$, $G^1 \equiv G^2$ by hypothesis. If $i > 1$, $G^1 \equiv G^2$ by the inductive hypothesis that the $(i-1)$-th noncollinear rays are congruent. The proof considers the two cases implied by the inductive formula above ($C_D(G^1)$ and its negation), further dividing the second case on the basis of whether $f = f_o$ or $f = f_d$.

C1: $C_D(G^1)$: Follows from Lemma 11.

C2: $\neg C_D(G^1) \wedge f(G^1) = f_o(G^1)$
$f(G^2) = f_o(G^2)$
, by hypothesis $G^1 \equiv G^2$ and the definitions of f_o and Λ^*
There must exist lines $L_1, L_2 \in \Lambda$ such that $L_1 \equiv L_2 \wedge \langle \forall i : i \in \{1, 2\} :: G^i \in L_i\rangle$
, by hypothesis $f(G^1) = f_o(G^1)$, last deduction, and the definition of f
$S_o(G^1, L_1) \equiv S_o(G^2, L_2)$
, by the definition of S_o, hypothesis $G^1 \equiv G^2$, and the last deduction
There must exist $\hat{G}^1$ and $\hat{G}^2$ such that $\langle \forall i : i \in \{1, 2\} :: f_i(G^i) = \{\hat{G}^i\}\rangle$
, by hypothesis $\neg C_N(G^1) \wedge \neg C_N(G^2)$
$\hat{G}^1 \equiv \hat{G}^2$
, last two deductions

C3: $\neg C_D(G^1) \wedge f(G^1) = f_d(G^1)$

$f(G^2) = f_d(G^2)$
, by hypothesis $G^1 \equiv G^2$ and the definitions of f_d and Λ^*

$\langle \exists L : L \in \Lambda^* :: G^1 \in L \vee G^2 \in L \rangle$
, by hypothesis $f(G^1) = f_d(G^1)$, last deduction, and the definition of f_d

$\langle \forall L : L \in \Lambda^* :: \overline{[G^1, (\infty, \infty))} \cap L \neq \emptyset \Rightarrow \overline{[G^2, (\infty, \infty))} \cap L \neq \emptyset \rangle$
, by hypothesis $G^1 \equiv G^2$ and the definition of Λ^*

$S_d(G^1) \equiv S_d(G^2)$
, by last deduction and the definition of S_d

There must exist points $\hat{G}^1$ and $\hat{G}^2$ such that $\langle \forall i : i \in \{1, 2\} :: f_d(G^i) = \{\hat{G}^i\} \rangle$
, by last deduction (and observing that $\langle \forall G : G \in F^2 - (\infty, \infty) :: ||S_d(G)|| = 1 \rangle$)

$\hat{G}^1 \equiv \hat{G}^2$
, combine previous two deductions □

C Proof of Theorem 1

The following notation is used. For a continuous path γ and a set of continuous paths S, $\gamma \cap S$ is the set of points in γ that lie on some path in S: $\bigcup_{\gamma' \in S} \gamma \cap \gamma'$.

We first require one lemma. 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, $C_L(G) \stackrel{def}{=} \langle \exists L : L \in \Lambda^* :: G \in L \ \wedge \ S_o(G, L) = L.f \rangle$.

Lemma 12 *Consider any TET γ in a TPG. If $||\{G \,|\, G \in \gamma \wedge C_N(G)\}|| < \infty$ and $\langle \nexists G : G \in \gamma :: C_D(G) \rangle$ then γ contains a LCET.*

Proof: Consider two cases: (a) γ intersects a finite number of constraint lines and (b) the complement of (a). Formally, (a) is $\langle \exists G : G \in \gamma :: \langle \nexists L : L \in \Lambda^* :: G \in L \rangle \ \wedge \ S_d(G) = \{(\infty, \infty)\} \rangle$.

case (a):

There must exist a G such that $G \in \gamma \wedge \langle \nexists L : L \in \Lambda^* :: G \in L \rangle \wedge S_d(G) = \{(\infty, \infty)\}$
, by hypothesis

$f(G) = f_d(G) \wedge f_d(G) = \{(\infty, \infty)\}$
, by last deduction and definition of f

$G + (\phi_0 \phi_1, \phi_0 \phi_1) \in \gamma$
, by last deduction

$G \equiv G + (\phi_0 \phi_1, \phi_0 \phi_1)$
, by the definition of congruence

$\langle \nexists G' : G' \in \overline{[G, (\infty, \infty)} :: C_N(G') \rangle$
, by hypothesis and definition of C_N

γ contains a LCET
, by last deduction and definition of LCET

case (b): (Hypothesis is $\langle \exists L : L \in \Lambda^* :: G \in L \rangle \vee S_d(G) \neq \{(\infty, \infty)\}$.)

If $\langle \exists G : G \in \gamma :: C_N(G) \rangle$ then choose G^N to be $\max\{G \,|\, G \in \gamma \wedge C_N(G)\}$ and let $\hat{\gamma}$ be the subpath obtained by deleting $\gamma.i$ up to but excluding G^N; otherwise $\hat{\gamma} = \gamma$.

$\langle \forall G : G \in \hat{\gamma} :: ||f(G)|| = 1 \rangle$
, by the definitions of C_N and $\hat{\gamma}$

$\langle \forall G : G \in \hat{\gamma} \wedge \langle \exists L : L \in \Lambda^* :: G \in \Lambda^* \rangle :: C_L(G) \rangle$
, by hypothesis $\langle \forall G : G \in \gamma :: \neg C_D(G) \rangle$ and definitions of $\hat{\gamma}$, C_D, and C_L

$\langle \forall G : G \in \hat{\gamma} \wedge \langle \nexists L : L \in \Lambda^* :: G \in \Lambda^* \rangle :: \neg C_F(G) \rangle$
, by hypothesis $S_d(G) \neq \{(\infty, \infty)\}$ and definition of C_F and S_d

(†) $\langle \forall G : G \in \gamma :: \neg C_F(G) \vee C_L(G) \rangle$
, by last two deductions

$||\hat{\gamma} \cap \Lambda^*|| = \infty$
, by last deduction and hypothesis $\langle \forall G : G \in \gamma :: \neg C_D(G) \rangle$

$\langle \exists L : L \in \Lambda :: ||\gamma \cap \{L' \mid L' \in \Lambda^* \wedge L' \equiv L\}|| = \infty \rangle$
, by last deduction and Lemma 2

$\langle \forall G : G \in \hat{\gamma} \cap \Lambda^* :: C_L(G) \rangle$
, by deduction † and definitions of C_L, C_F

$\langle \forall L : L.i, L.f \in R^2 \wedge mod(L) \in \Lambda \wedge \hat{\gamma} \cap L \neq \emptyset :: L.f \in \hat{\gamma} \rangle$
, by last deduction and definition of f_o

$\langle \exists G, G' : G, G' \in \hat{\gamma} :: G \equiv G' \rangle$
, by last two deductions

$\hat{\gamma}$ contains a LCET
, by second and last deductions

γ contains a LCET
, by last deduction and because $\hat{\gamma}$ is a subtrajectory of γ □

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* $||\{G | G \in \gamma \wedge C_N(G)\}|| < \infty \ \wedge \ \langle \nexists G : G \in \gamma :: C_D(G) \rangle$.

Proof of Theorem 1:

Only if part:

$||\{G | G \in \gamma \wedge C_N(G)\}|| < \infty$
, by the hypothesis that γ ends in an infinite number of LCETs and the definition of a LCET as a set of deterministic points

$\langle \forall G : G \in \gamma :: f(G) \neq \emptyset \rangle$
, by hypothesis that γ ends in an infinite number of LCETs and definitions of f and LCET

$\langle \nexists G : G \in \gamma :: C_D(G) \rangle$
, by definition of C_D and last deduction

If part: The inductive proof below demonstrates that γ contains at least i congruent LCETs, for all $i > 0$. Lemma 12 establishes the base case ($i = 1$). The proof for $i > 1$ follows:

γ contains at least $i - 1$ LCETs
, by the inductive hypothesis

Let γ_{i-1} denote the $(i-1)$-th LCET; then $\gamma_{i-1}.i \equiv \gamma_{i-1}.f$
, by definition of LCET

TETs rooted at $\gamma_{i-1}.i$ and $\gamma_{i-1}.f$ are congruent
, by Lemma 1

$\langle \exists G'' : G'' \in$ TET rooted at $\gamma_{i-1}.f \wedge G'' > \gamma_{i-1}.f \ :: \ G'' \equiv \gamma_i.f \rangle$
, by last deduction

The subtrajectory of γ with initial point $\gamma_i.i$ and final point $\gamma_i.f$ is a LCET
, by last deduction and the definition of LCET

There exist at least i congruent LCETs in γ
, combine last deduction with inductive hypothesis □