Computational Geometric Performance Analysis of Limit Cycles in Timed Transition Systems

Marc Abrams

TR 93-30

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

September 20, 1993

Computational Geometric Performance Analysis of Limit Cycles in Timed Transition Systems

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

TR 93-30

September 10, 1993

Abstract

Certain parallel software performance evaluation problems are equivalent to computational geometric problems. Consider a timed transition system representing a parallel program: a set of variables, a set of states, an initial state, a transition function mapping a state to a set of successor states, and a description of the time between transitions. Program execution is represented by a sequence of states satisfying the transition function along with the times of state transitions, called a *timed execution sequence* (TES). For the program class considered, the TES may contain a suffix of repeated states, called a *limit cycle execution sequence* (LCES). Given a timed transition system, the paper solves four problems: (P1) State the necessary and sufficient conditions for a TES to contain a LCES. (P2) Given the initial starting time of each process, find a representation of the set of all possible TESs. (P3) Determine if there exists any initial process starting times that leads to a LCES in which no process ever blocks. (P4) Find the set of all possible LCESs. P1 to P4 are all embellishments of the ray shooting problem from computational geometry: Given a collection of line segments in a plane and a ray, find the first line segment that the ray intersects. This is demonstrated by defining *timed progress graphs* (TPG) and showing their equivalence to timed transition systems. A TPG maps the progress of each process to one Cartesian graph axis. Line segments represent interprocess synchronization. A directed, continuous path that does not cross a segment represents a TES. Although the solution is constrained to two processes that mutually exclusively share resources, the novel analysis method raises the question of whether other software performance problems are equivalent to computational geometric problems with known solutions.

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

General Terms: computational geometry, ray shooting, timed progress graphs, timed transition systems, software performance analysis

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

1 Introduction

Critical to the success of parallel and distributed programming are better methods to understand correctness and performance. Methods for understanding algorithms can be broadly categorized as formal versus empirical. Formal methods include proof systems and various analytic performance models (such as stochastic processes and queueing networks). Methods for empirical understanding include algorithm animation and performance visualization systems. Are these two worlds of formal and empirical methods disjoint, or can techniques from one assist the other? Roman and Cox [33] argue in the affirmative for correctness; this paper argues in the affirmative for performance.

The formal basis underling our work (and that of Roman and Cox) is to represent a program as a *transition system* and to represent an execution of a program as an *execution sequence*. A transition system consists of a set of variables, a set of states or interpretations of the variables, an initial state, and a transition function mapping a state to a set of successor states. Chandy and Misra's Unity [8] demonstrates that many and possibly all applications, programming languages, and paradigms can be represented by a transition system. Thus the ability to analyze transition systems yields the ability to analyze programs.

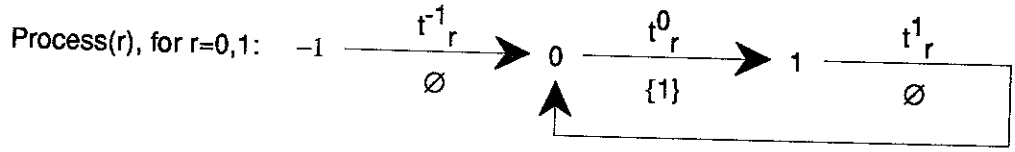
Transition systems are used to specify and formally reason about computations. Program execution is represented by a sequence of states satisfying the transition function, called an *execution sequence*. Augmenting a transition system by a description of the time between transitions yields a timed transition system (e.g., [16]), permitting proofs of properties about real time systems [17].

Roman and Cox illustrate that safety and liveness properties can be verified by defining a mapping of program states to a visual representation, and then observing that the sequence of visual images corresponding to an execution sequence meet some property. They discuss the ability "to render invariant properties of the program state as stable visual patterns and to render progress properties as evolving visual patterns." In this paper we show that the mapping of execution sequences to a visual image allows analysis of program performance properties. Our analysis uses known algorithms from computational geometry [31], which is a novel way to analyze performance. Although the algorithms in this paper analyze a limited class of software, the paper raises the question of whether performance analysis of other software classes is amenable to representation as computational geometric problems.

The precedence for mapping execution sequences to visual images to analyze software performance exists in visualization systems for software performance analysis (e.g., [2, 19, 25, 27]). In visualization systems the execution sequence is a trace of time stamped events obtained from instrumented source code [35]. One visualization of an execution sequence is an x-y plot of state as a function of time. Visualization systems lend insight into why a performance measure, such as program execution time or mean waiting time for a resource, has a particular value. The answer to "why" helps identify how to change the program to improve the measure.

Timed transition and visualization systems represent an execution sequence as a linear list of states and as a path in a geometric space, respectively. In the literature only the first has been used for formal analysis of timing properties. This paper demonstrates that the second can also be used, and in doing so shows that certain performance evaluation problems are equivalent to geometric problems.

Example: To motivate the problems to be solved, consider an example: the two process mutual exclusion problem. There are two processes that share a resource in a mutually exclusively (i.e., at most one process can use the resource at any time). A program solving the mutual exclusion problem must serialize access to the data. Figure 1 illustrates a solution using a timed transition diagram. The program consists of two instances of the diagram, representing the two processes that share the resource. (Timed transition diagrams are presented in §3.) Diagram vertices represent all possible code segments, and are labeled by consecutive integers starting at -1, called *locations*. Location -1 represents a process that is waiting to start execution. Diagram edges are labeled above by a *delay* (a real number) and below by a *condition* (a set of locations). Let $r \in \{0, 1\}$ denote a process. The intended meaning of an arc directed



Graph location	*Execution time required*	
	process $r = 0$	*process* $r = 1$
t_r^{-1}	0	0
t_r^0	1	2
t_r^1	4	1

Figure 1: Timed transition diagrams of a mutual exclusion program.

from locations i to $i+1$ in the diagram for process r that is labeled by delay t_r^i and condition c is that the process remains in a location i for t_r^i time units; then the process makes a transition from location i to location $i+1 \bmod 2$ at the *first* instant when the location of the other process is not in set c. In Fig. 1, location 0 represents a code segment in which a process is not accessing the shared resource, and location 1 is a code segment that uses the resource and hence can be executed by at most one process. Due to the label "{1}" on the edge out of location 0, a process blocks on transition out of location 0 if the other process is in location 1. The table in Fig. 1 illustrates one possible set of delays. Because $t_0^{-1} = t_1^{-1} = 0$, both processes simultaneously start execution.

Consider now execution of the program in Fig. 1. Let $\bar{r} = 1 - r$. (If $r = 0$, then $\bar{r} = 1$ and vice versa.) We use an ordered four-tuple $(\pi_0, \rho_0, \pi_1, \rho_1)$ as the program state: π_r is the location of process r and ρ_r is the minimum time that process r must remain in its current location. Thus the initial state is $(-1, 0, -1, 0)$. Two transitions out of a state $(\pi_0, \rho_0, \pi_1, \rho_1)$ are defined. The first, called a *state transition*, takes time 0 and advances the location of one process. The location of process r will advance from π_r to $\pi_r + 1 \bmod 2$ iff the remaining delay of process r (i.e., ρ_r) is zero and the condition labeling the edge out of location π_r in Fig. 1 does not name the location of the other process (i.e, $\pi_{\bar{r}} \notin c_r^{\pi_r}$). The minimum time ρ_r is set to the delay of the newly entered state, $t_r^{\rho_r}$. To illustrate, initial state (-1,0,-1,0) has two possible successors by a state transition, because the remaining delay of both processes is zero and the condition labeling the edge out of location -1 is empty: (0,1,-1,0) and (-1,0,0,2). In each of these states, the remaining delay of one process is zero; hence a state transition exists out of each state. Both transitions result in the same state: (0,1,0,2). The second transition advances time, and takes time t, where $0 < t \leq \min(\rho_0, \rho_1)$, by subtracting t from each non-zero remaining delay. Returning to our example, we left off at state $(0, 1, 0, 2)$. After 1 time unit elapses, the system will reach state (0,0,0,1) by a time transition. Now process 0 must block for 1 time unit due to label {1} on the edge out of location 0. Thus the system reaches state (0,0,0,0) after 1 time unit by a time transition. Figure 2 continues this process to illustrate all states reachable from the initial state. Because the state space is continuous, we use the notation $(\pi_0, \rho_0..z_0, \pi_1, \rho_1..z_1)$ in the figure to denote the set of states $\{(\pi_0, \hat{\rho}_0, \pi_1, \hat{\rho}_1) \mid \langle \forall r, \exists t : r \in \{0,1\} \wedge 0 \leq t \leq \min(\rho_0 - z_0, \rho_1 - z_1) :: \hat{\rho}_r = \rho_r - t \rangle\}$[1]. (Fig. 2 contains letters, such as H, points labeling each state. These will be explained when used later, in Example 2 in §4.)

For each state $(\pi_0, \rho_0, \pi_1, \rho_1)$, consider an ordered pair (π_0, π_1); this pair denotes the location of each

[1] We use the notation from [8] ⟨ <quantified variable> : <domain of quantification> :: <quantified formula> ⟩, and omit <domain of quantification> when it is obvious from context.

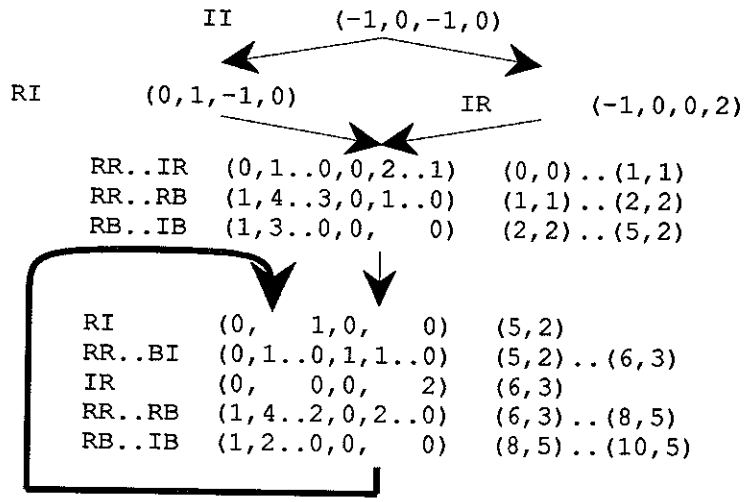


Figure 2: The set of all states reachable in the mutual exclusion program of Fig. 1.

process. From Fig. 2, there is only one possible location sequence for *any* possible execution of the program:

$$\begin{array}{lllll} \text{process locations:} & (-1,-1), & (0,-1), & (0,0), & \left\{ \begin{array}{cccc} (1,0), & (0,0), & (0,1), & (0,0) \\ 4, & 0 & 1 & 0 \end{array} \right\}^* \\ \text{time in location:} & 0, & 0, & 1, & \end{array}$$

where $\{...\}^*$ denotes an infinite repetition of a subsequence. Informally, the above sequence is called the *timed execution sequence* (TES) of the program, and the repeated subsequence the *limit cycle execution sequence* (LCES), due to its qualitative resemblance to limit cycles in a dynamic system. We will see that for a wide variety of initial process starting times (i.e., t_0^{-1} and t_1^{-1}), the program converges to the same location pairs of a LCES. If we imagine a knob whose setting determines one delay t_r^i, and vary the knob, then the location pairs of the LCES remain the same until we turn the knob past a critical point, at which point the location pairs of the LCES to which the program converges suddenly change.

Problem statement: Given a timed transition diagram and a set of timings (e.g., Fig. 1) the following problems are solved:

P1: State the necessary and sufficient conditions for a TES to contain a LCES.

P2: Given the initial starting time of each process, find the set of all possible TESs.

P3: Determine if there exists any process starting times that lead to a LCES in which no process ever waits for any resource. Furthermore, if there exists such times, then output an example.

P4: Find the set of all possible LCESs in a timed transition diagram that are reachable from some initial process starting times (i.e., any values of t_0^{-1} and t_1^{-1}).

The solution to P1 is used in solving P2 thorough P4. P2 is important because any performance measure of interest about a program can be computed from a TES, and the TES structure may give insight into how the program behavior governs measure values. Solution of P3 identifies cases in which no process in a parallel program ever blocks.

The analysis of LCESs (P4) is valuable because limit cycles represent periodic behavior. Periodic behavior is not limited to programs as academic as the mutual exclusion problem. For example, low frequency oscillations in queue length and packet traffic has been observed experimentally in communication protocols [2, 37]. Periodic behavior occurs with periodic tasks, such as operating system daemons [6]. In fact, experience with our own visualization system [2] demonstrates that the TESs of many long running programs that spend the bulk of their time looping in small fractions of the code repeatedly exhibit certain state subsequences. If the program execution time is chiefly determined by a small set of repeated subsequences, then finding these subsequences given the code and timings is an effective way to diagnose performance problems.

Assumptions: To obtain a tractable problem, we make the following assumptions. They are stated in terms of *resource operations* on serially reusable resources, which are code segments in which a process acquires a resource (and possibly blocks, while another process holds the resource) or releases a resource. A resource operation could be implemented by a binary semaphore or a spin lock (e.g., see [20, pp. 102,113]).

A1: A program contains two processes.

A2: Each process meets the following assumptions:

A2.1: A process executes a nonterminating loop.

A2.2: The execution time of each code segment within each process that either:

- starts at the initial statement of the loop body and continues to and includes the first resource operation, or
- follows each resource operation and continues to and includes the next resource operation

is an independent constant, exclusive of time spent blocked.

A2.3: A process executes on a dedicated processor.

A2.4: Resource operations are executed unconditionally.

A3: Processes synchronize only to achieve mutually exclusive access to each of a set of resources.

Although our assumptions constrain the class of programs analyzed in this paper, we lay the foundation for studying other parallel program classes by relating them to equivalent problems in computational geometry. Consider the two process assumption in A2. The initial solutions to some classic parallel programming problems – such as shared memory mutual exclusion algorithms – were initially solved only for two processes. And one important performance evaluation tool - queueing networks - started only with the ability to solve just one kind of queue (M/M/1) in isolation. In principle the analysis presented here can be extended from two to an arbitrary number of processes; see the conclusions (§7) for a discussion.

Consider next the remaining assumptions. Although the analysis is constrained to two processes (A1), a recent performance analysis of Lamport's mutual exclusion algorithm [5] using Petri nets requires so much computation that its numeric solution is limited to only four processes. Regarding A2.1, certain programs can be considered non-terminating for the purpose of analysis. Examples include long running programs that execute the same code repeatedly, such as simulations, and *reactive programs* [8], such as operating system algorithms, that react to external stimuli. In fact, Chandy and Misra [8] suggest that all computations can be represented as non-terminating transition systems that reach a fixed point. Regarding A2.2, the assumption of constant timings is perhaps no more or less reasonable than the assumption of exponential timings required by the Markov chains underling some Petri net and queueing network models. In fact, one could view a (constant time) geometric model of the type used in this paper and an (exponential time) Markov process model as two extremes in modeling program behavior.

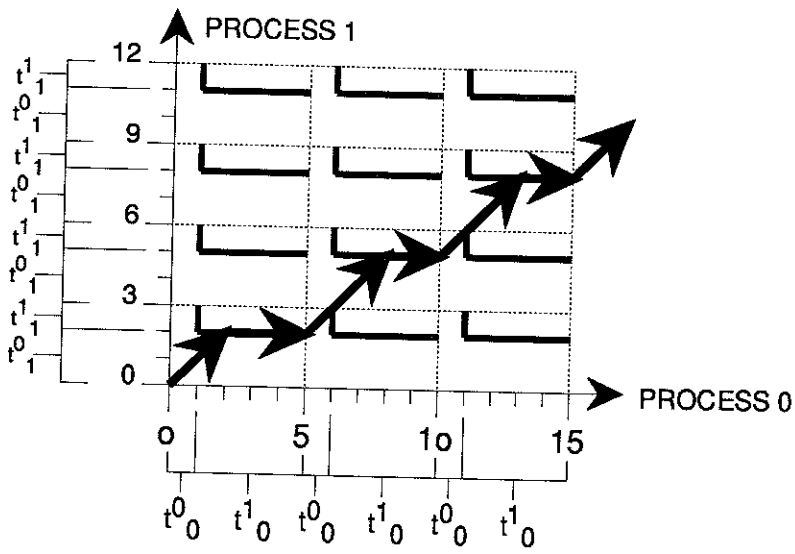


Figure 3: Timed progress graph for Fig. 1

A2.3, A2.4, and A3 could be relaxed by using the discussion in later sections on how to map a transition system to a geometric model as a guide to developing mappings for other classes of transition systems. For example, [1, pp. 41-44] illustrates how certain other synchronization constructs (CSP's [10] input and output commands) can be mapped to a geometric model.

Method of Solution: The geometric representation of timed transition diagrams used here is a *timed progress graph* (TPG). TPGs are based on *untimed* progress graphs (UPG) used in the literature. Carson and Reynolds [7] define an UPG as a "a multidimensional, Cartesian graph in which the progress of each of a set of concurrent processes is measured along an independent time axis. Each point in the graph represents a set of process times."

Figure 3 illustrates a finite portion of the TPG corresponding to Fig. 1. The dotted lines are not part of the TPG – they are a grid provided to help the reader determine point coordinates. A TPG illustrates program execution after both processes have made a transition out of location -1 and thus started execution, which starts with state $(0, 1, 0, 2)$ in Fig. 2. The lines with arrows in Fig. 3 form a directed path, called a *timed execution trajectory* (TET), that represents the single possible TES. The coordinates annotating Fig. 2 are the TET points corresponding to the *first* occurrence of each state. Each coordinate of a TET point may be interpreted as a clock associated with the corresponding process that initially is enabled and has value zero, is disabled whenever the process to which it corresponds waits for a resource, and is re-enabled whenever the process acquires a resource for which it was waiting. For example, point $(15, 8)$ denotes that process 0 (respectively, process 1) has run for exactly 15 (8) time units. Process 1 has waited 15-8=7 time units longer than process 1 in all resource requests it has made so far.

To understand the mapping from transition diagram set to TPG, first consider what would happen if the resource operations in Fig. 1 are deleted, so that the two processes never synchronize. Then process 0 is in location 0 during time interval $[0, t_0^0) = [0, 1)$, location 1 during $[t_0^0, t_0^0 + t_0^1) = [1, 5)$, location 1 during $[t_0^0 + t_0^1, t_0^0 + t_0^1 + t_0^0) = [5, 6)$, and so on. Process 0 repeats the cycle of locations 0,1 every $t_0^0 + t_0^1 = 5$ time units. Therefore to find the location of process 0 corresponding to graph point (x_0, x_1), we calculate

$x_0 \bmod 5$ and see if it lies in interval [0,1) or [1,5). Similarly, to find the location of process 1 we see if $x_1 \bmod 3$ lies in interval $[0, t_1^0) = [0, 2)$ or $[t_1^0, t_1^0 + t_1^1) = [2, 3)$. This mapping from point (x_0, x_1) to state implies that the TET representing the TES will be a diagonal ray with slope one, rooted at point (0,0).

Next consider the program as shown in Fig. 1, with resource operations. An acquire resource operation potentially blocks one process; therefore the TET portion representing a TES portion in which one process is blocked is a horizontal or vertical ray, depending on which process is blocked. The "L" shaped lines in the plane in medium thickness lines, called *constraint lines*, represent all possible situations in which one process is blocked acquiring a resource because the other process holds the resource. A TET cannot cross a constraint line. An intuitive justification for the generator locations in Figure 3 follows. Process 0 blocks iff it is about to enter location 1 and the current location of process 1 is 1. Therefore process 0 blocks iff the point (x_0, x_1) representing its current state satisfies

$$x_0 \bmod t_0^0 + t_0^1 = t_0^0 \text{ and}$$

$$x_1 \bmod t_1^0 + t_1^1 \in [t_1^0, t_1^1),$$

or $x_0 \bmod 5 = 1$ and $x_1 \bmod 3 \in [2, 3)$. Thus if point $(x_0 \bmod 5, x_1 \bmod 3)$ representing the current state lies on vertical line $\overline{[(1, 2), (1, 3))}$ then process 0 is blocked; this line is congruent to all medium thickness vertical lines in Fig. 3. Similarly, process 1 blocks iff the point equivalent to its current state lies on a horizontal line congruent to $\overline{[(1, 2), (5, 2))}$.

The TET subpath rooted at $(3, 2)$ consists of an infinite number of repetitions of the following subpath: a horizontal ray of length 2 and a diagonal ray whose projected length on either axis is 3. This path is called the *limit cycle execution trajectory* (LCET), and represents a LCES. Note that the LCES given earlier and the LCET both have equal length: 5 time units. In the LCET, process 1 waits for 2 time units and process 1 does not wait. The TET subpath from (0,0) to (3,2) is a transient period, and (3,2) is the initial point of the first repetition of the LCET.

Additional features of TPGs are illustrated in Fig. 4. The figure shows, in the lines with arrows, the case of an initial condition (point (2,0)) that leads to two TETs. Both TETS contain a point that is the initial point of two constraint lines (point (3,1)), representing a nondeterministic state – one that would have two successors in a reachability graph such as Fig. 2. The nondeterministic state represents the situation when both processes simultaneously attempt to acquire the same resource (i.e., a race condition). One TET contains point (3,3), which lies on two constraint lines but is not the initial point of either line, and represents program deadlock. Finally, the graph illustrates through shading the set of all TETs rooted at points on line segments $\overline{[(0, 0), (0, 11)]}$ and $\overline{[(0, 0), (11, 0)]}$. All TETs either lead to a deadlock (at points (3,3), (3,14), or (14,3)) or to a TET of infinite length that extends off the diagram and never intersects a constraint line.

The paper is organized as follows. The following section compares the use of TPGs to find limit cycles to other techniques in the literature. §3 defines timed transition diagrams. §4 formally defines timed transition systems, to make precise terms describing program execution, such as "dead state" and "TES." §5 formally defines TPGs and TET and defines when a TET in a TPG represents a TES in a corresponding timed transition system. Sections 4 and 5 serve the secondary purpose of illustrating how other program classes, and their corresponding transition systems, could be mapped to other forms of timed progress graphs. §6 presents a computational geometric algorithm to solve problems P1 to P4. §7 discusses extending the algorithms in §6 to more than two dimensions, and lists open problems.

2 Related Work

The problem of finding limit cycles in parallel programs that synchronize only to achieve mutual exclusion and execute for constant time between synchronization operations is related to three areas of the literature: UPGs, Petri nets, and resource scheduling algorithms.

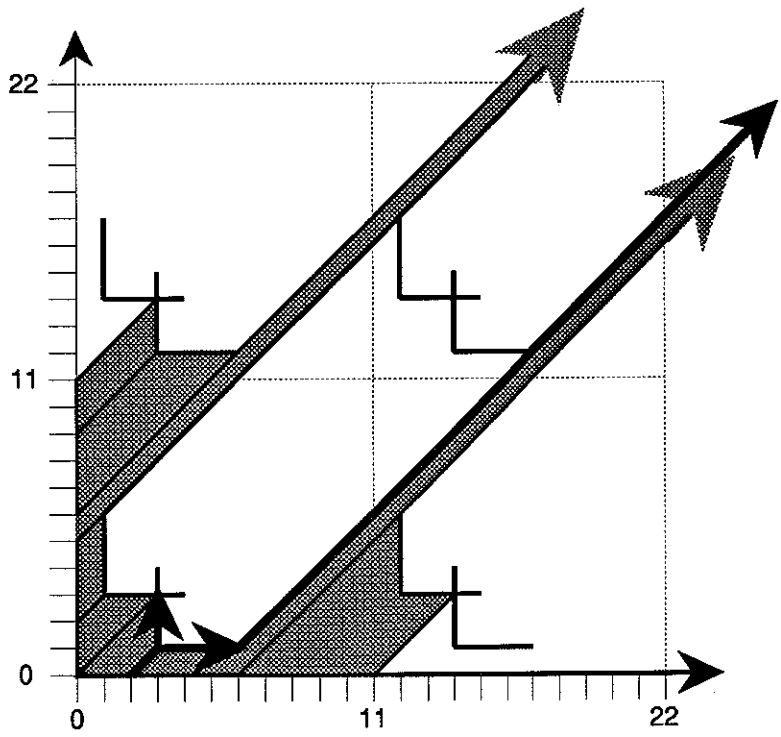


Figure 4: Geometry of deadlock and nondeterminism.

Related work on UPGs: Dijkstra [9] devised UPGs. Later, Papadimitriou, Yannakakis, Lipski, and Kung [21, 28, 36] used UPGs to detect deadlocks in lock-based transaction systems. Recently Carson and Reynolds [7] used UPGs to prove liveness properties in programs with an arbitrary number of processes containing P and V operations on semaphores that are unconditionally executed. TPGs differ from UPGs in two ways. First, TPGs represent the time required for transitions, and can be used to derive performance properties of a program. 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 can be used to detect deadlock only for two process programs executing on two dedicated processors.

Related work on Petri nets: The class of programs analyzed in this paper may be studied using Petri nets [18], queueing networks (e.g., [14, 15]), stochastic processes (e.g., [12, 29]), and stochastic automata ([30]). A survey of these approaches is contained in [1], Chapter 1. Of these, the most closely related work has been done using consistent Petri nets (i.e., nets that return to their initial marking) in which a deterministic firing time is associated with each transition.

Ramamoorthy and Ho [32] address minimum cycle time (MCT) calculation, or the minimum time required for the program to return to its initial state (corresponding to an initial marking of the Petri net). The Ramamoorthy and Ho method takes exponential time and works for both decision-free and persistent Petri nets, in which a token never enables two or more transitions simultaneously. Methods to compute bounds on the MCT of conservative, general Petri nets are given; finding the exact value is

proved NP-complete.

Magott [22] 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 [23] 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 [24] extends his earlier paper [23] 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 communicating through buffers.

The problem considered in this paper, of processes sharing reusable resources, cannot be represented by decision-free or persistent Petri nets. One mutual exclusion problem, the dining philosophers problem [11], has been analyzed by Holliday and Vernon [18] assuming deterministic as well as geometrically distributed local state occupancy times. Their Petri net model uses frequency expressions to resolve deterministically which transition fires when a token enables two or more transitions simultaneously. In their model of the dining philosophers problem, this expression takes the form of a probability. The dining philosophers problem has also been modeled using colored Petri nets by Gorton [13].

Compared to a Petri net approach, TPGs have two advantages:

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

2. TPGs give the TESs for *all* possible Petri net markings in a single solution, while existing Petri net solutions require repeatedly solving of the net for *each* marking.

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

3 Timed Transition Diagrams

Our model of a program is based on Henzinger, Manna, and Pnueli's timed transition diagrams [16].[2] Time is represented by nonnegative real numbers. Let R, R^+, Z, and Z^+ denote, respectively, the set of nonnegative reals, positive reals, nonnegative integers, and positive integers. We assume that $\langle \forall n : n \in Z :: n \leq \infty \rangle$ to simplify our notation. Let $\langle \forall r :: n_r \rangle$ denote finite, positive integers.

Each process r is represented by a finite, connected, directed graph containing $n_r + 1$ vertices, each labeled by an integer in $\{-1, 0, 1, \ldots, n_r - 1\}$. Each integer label is called a *location*. Each vertex has exactly one outgoing edge; therefore the graph contains one cycle. The program uses variables $\{\pi_0, \pi_1\}$, which are shared by all processes; π_r denotes the control point of process r. Each edge with initial vertex i, for $-1 \leq i < n_r$, is labeled by a *delay* t_r^i and a *condition* c_r^i. Delays satisfy $\langle \exists r :: t_r^{-1} = 0 \wedge t_{\bar{r}}^{-1} \in R \rangle \wedge \langle \forall t_r^i : 0 \leq i < n_r :: t_r^i \in R^+ \rangle$. Conditions in the graph of process r name labels of process $\bar{r}$: $c_r^{-1} = \emptyset \wedge \langle \forall v : v \in c_r^i :: v \in \{0, 1, \ldots, n_{\bar{r}} - 1\} \rangle$. Given two labels i and i' of process r, $i \oplus i'$ (respectively, $i \ominus i'$) denotes addition (subtraction) modulo n_r. The intended operational meaning of edge $-1 \xrightarrow[\emptyset?]{t_r^{-1}} 0$

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

is that process r waits for t_r^{-1} time units before it starts execution. The intended operational meaning of edge $i \xrightarrow[c_r^i?]{t_r^i} i \oplus 1$, for $i \geq 0$, is that if, for exactly t_r^i time units, control of process r has resided at vertex i, then control will move to vertex $i \oplus 1$ at the earliest time at which the control point of process $\bar{r}$ is not a label in c_r^i. Process r is *blocked* whenever it has remained in its current location, i, for longer than t_r^i time units. After process r starts execution, it is *running* whenever it is not blocked. Figure 1 illustrates a set of two timed transition diagrams.

Timed transition diagrams represent mutual exclusion in programs as follows. Let $\mathcal{R}_0, \mathcal{R}_1, \ldots, \mathcal{R}_{n_s-1}$ denote the resources used by a program. Each graph edge out of vertex i, where $0 \leq i < n_r$, corresponds to at most one resource operation. We say that a process *holds* resource $\mathcal{R}_j$ (where $0 \leq j < n_s$) in locations $\hat{i}, \hat{i}+1, \ldots, \hat{i}+k$ if edge $\hat{i}-1$ corresponds to an acquire $\mathcal{R}_j$ operation, edge $\hat{i}+k$ corresponds to a release $\mathcal{R}_j$ operation, and none of the edges in $\{\hat{i}, \hat{i}+1, \ldots, \hat{i}+k-1\}$ correspond to a resource operation on $\mathcal{R}_j$. Let $\alpha_r^{\mathcal{R}_j}$ denote the set of locations in process r that hold resource $\mathcal{R}_j$; formally $\{\hat{i}, \hat{i}+1, \ldots, \hat{i}+k\} \subseteq \alpha_r^{\mathcal{R}_j}$. If the edge out of vertex i in process r corresponds to operation acquire $\mathcal{R}_j$, then c_r^i is the set of all locations in the diagram for process $\bar{r}$ in which process $\bar{r}$ holds resource $\mathcal{R}_j$. Formally, if the edge out of a location i corresponds to operation acquire $\mathcal{R}_j$, then $c_r^i = \alpha_{\bar{r}}^{\mathcal{R}_j}$; otherwise $c_r^i = \emptyset$. For any condition $c_r^k = \{\hat{i}, \hat{i}+1, \ldots, \hat{i}+z\}$, for some $z \in Z$, we write $\langle \hat{i}, \hat{i}+z \rangle \in c_r^k$.

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

Example 1 For Fig. 1, $\psi_0 = \langle\, 2,\, \langle 0,1,4 \rangle,\, \langle \{1\}, \emptyset \rangle \,\rangle$ and $\psi_1 = \langle\, 2,\, \langle 0,2,1 \rangle,\, \langle \{1\}, \emptyset \rangle \,\rangle$. □

4 Timed Transition Systems

To make precise the terms PES, LCES, and dead and nondeterministic state, we formalize the timed transition system introduced in the introduction. A *timed transition system* $\mathcal{S}_D = \langle V, \Sigma, \Theta, \tau \rangle$ corresponding to a timed transition diagram $D = \langle \psi_0, \psi_1 \rangle$ consists of four components:

V: a finite set of four variables, denoted $\pi_0, \pi_1, \rho_0, \rho_1$, satisfying $\langle \forall r :: \pi_r \in \{i \mid -1 \leq i < n_r\} \wedge \rho_r \in R \rangle$. (Variables π_r and ρ_r were informally described in the introduction.)

Σ: a set of *states*. Every state $\sigma \in \Sigma$ is an interpretation of V; that is, it assigns to every variable $v \in V$ a value $\sigma(v)$ in its domain. Every state $\sigma \in \Sigma$ satisfies $\langle \forall r :: -1 \leq \sigma(\pi_r) < n_r \wedge 0 \leq \sigma(\rho_r) \leq t_r^{\sigma(\pi_r)} \rangle$. For a particular state σ, a process r may be in one of three mutually exclusive, exhaustive categories:

running(r, σ): Process r has not yet spent an amount of time in its current location equal to the delay associated with the location. Formally, *running*$(r, \sigma) \stackrel{def}{=} \sigma(\rho_r) > 0$.

blocked(r, σ): Process r has been in its current location for a time period at least equal to the delay associated with the location, but cannot advance its location because the current location of process $\bar{r}$ is an element of the condition labeling process r's outgoing edge. Formally, *blocked*$(r, \sigma) \stackrel{def}{=} \sigma(\rho_r) = 0 \;\wedge\; \sigma(\pi_r) = i \;\wedge\; \sigma(\pi_{\bar{r}}) \in c_r^i$.

intransit(r, σ): Process r is ready to advance its location because it has been in its current location for a time period at least equal to the delay associated with the location, and the current location of process $\bar{r}$ is not an element of the condition labeling the outgoing edge of the vertex corresponding to process r's current location. Formally, *intransit*$(r, \sigma) \stackrel{def}{=} \sigma(\rho_r) = 0 \wedge \sigma(\pi_r) = i \wedge \sigma(\pi_{\bar{r}}) \notin c_r^i$.

Θ: an *initial state*, satisfying $\Theta \in \Sigma$ and $\langle \exists r :: \Theta(\rho_r) = t_r^{-1} \wedge \Theta(\rho_{\bar{r}}) = 0 \wedge \Theta(\pi_r) = \Theta(\pi_{\bar{r}}) = -1 \rangle$. (The location of both processes is initially -1, and process r starts execution t_r^{-1} time units after process $\bar{r}$.)

τ: a *transition function* τ, which maps each state $\sigma \in \Sigma$ to a (possibly empty) set of successors $\tau(\sigma) \subseteq \Sigma$, where $||\tau(\sigma)|| \leq 2$. Function τ advances exactly one of the following: time (e.g., by reducing ρ_0 and/or ρ_1) or the location of exactly one process (e.g., by increasing one of π_0 or π_1 from i to $i \oplus 1$). Let AT(σ) (respectively, AL(σ)) hold if $\tau(\sigma)$ advances time (location). A time advance occurs iff at least one process is running and no process is in transit. Formally, $\mathrm{AT}(\sigma) \stackrel{def}{=} \langle \exists r :: \mathit{running}(r, \sigma) \rangle \wedge \langle \forall r :: \neg \mathit{intransit}(r, \sigma) \rangle$. A location advance occurs iff some process is in transit. Formally, $\mathrm{AL}(\sigma) \stackrel{def}{=} \langle \exists r :: \mathit{intransit}(r, \sigma) \rangle$. State $\sigma' \in \tau(\sigma)$, also denoted $\sigma \rightarrow \sigma'$, iff

$$[\,\mathrm{AT}(\sigma) \wedge \langle \forall r :: \sigma'(\pi_r) = \sigma(\pi_r) \wedge \langle \exists t : 0 < t \leq \min\{\sigma(\rho_{\hat{r}}) \mid \hat{r} \in \{0,1\} \wedge \mathit{running}(\hat{r}, \sigma)\} :: \sigma'(\rho_r) = \begin{cases} \sigma(\rho_r) - t & \text{if } \mathit{running}(r,\sigma) \\ \sigma(\rho_r) & \text{otherwise} \end{cases} \rangle \rangle\,] \bigvee$$

$$[\,\mathrm{AL}(\sigma) \wedge \langle \exists r : \mathit{intransit}(r, \sigma) :: \sigma'(\pi_r) = \sigma(\pi_r) \oplus 1 \wedge \sigma'(\rho_r) = t_r^{i \oplus 1} \wedge \langle \forall v : v \in \{\pi_{\bar{r}}, \rho_{\bar{r}}\} :: \sigma'(v) = \sigma(v) \rangle \rangle\,].$$

A state σ is *dead*, denoted $\mathit{dead}(\sigma)$, iff all processes block. Formally, $\mathit{dead}(\sigma) \stackrel{def}{=} \langle \forall r :: \mathit{blocked}(r, \sigma) \rangle$. The definition of τ implies $\tau(\sigma) = \emptyset$ iff $\mathit{dead}(\sigma)$.

Example 2 Consider the timed transition system corresponding to Fig. 1 and its state space (Fig. 2). To restate precisely comments from the introduction, we denote each state $\sigma \in \Sigma$ by the four-tuple $(\sigma(\pi_0), \sigma(\rho_0), \sigma(\pi_1), \sigma(\rho_1))$. The initial state is $\Theta = (-1, 0, -1, 0)$. The notation $(x_0, y_0..z_0, x_1, y_1..z_1)$ for a state in Fig. 2 denotes the set of states $\{\sigma \mid \langle \forall r \exists t : 0 \leq t \leq y_0 - z_0 :: \sigma(\pi_r) = x_r \wedge \sigma(\rho_r) = y_r - t \rangle\}$. In the figure, states σ and σ' are written on successive lines, or an edge is drawn with initial state σ and final state σ' iff $\sigma \rightarrow \sigma'$. Annotating each state in the figure is a pair that classifies the state. The pairs use the symbols R, I, and B to denote the predicates *running*, *intransit*, and *blocked*, respectively. For example, "IR" next to σ = (-1,0,0,2) denotes $\mathit{intransit}(0, \sigma) \wedge \mathit{running}(1, \sigma)$. An analogous meaning applies to other pairs. Finally, the notation RR .. IR next to state $(0, 1..0, 0, 2..1)$ denotes $\mathit{running}(0, \sigma) \wedge \mathit{running}(1, \sigma)$ in state $\sigma = (0, 1, 0, 2)$ and $\mathit{intransit}(0, \sigma) \wedge \mathit{running}(1, \sigma)$ in state $\sigma = (0, 0, 0, 1)$. □

Σ is a continuous state space with infinite cardinality. We now define a discrete state space, denoted Σ^*, with finite cardinality that is embedded in Σ at the time instances when a process changes location. The *closure* function, mapping a state to a state, defines this discrete state space: $\Sigma^* = \{\, \sigma \mid \langle \exists \sigma' : \sigma' \in \Sigma :: \sigma = \mathit{closure}(\sigma') \rangle \,\}$. The closure of a state is obtained by first advancing time, if possible, and then advancing the location of as many processes as possible.

Definition. *The* closure of a state σ under τ, *denoted* $\mathit{closure}(\sigma)$, *satisfies* $\sigma' \in \mathit{closure}(\sigma)$ *iff there exist* $m \geq 2$ *unique states* $\sigma_0 = \sigma, \sigma_1, \ldots, \sigma_{m-1} = \sigma'$ *such that* $\sigma_0 \rightarrow \sigma_1 \cdots \rightarrow \sigma_{m-1}$ *satisfying*

$$[AT(\sigma_0) \wedge \langle \exists r :: \sigma_0(\rho_r) > 0 \wedge \sigma_1(\rho_r) = 0 \rangle \wedge \langle \forall i : 1 \leq i \leq m-2 :: AL(\sigma_i) \rangle \wedge AT(\sigma_{m-1})] \bigvee [AL(\sigma_0) \wedge \langle \forall i : 1 \leq i \leq m-2 :: AL(\sigma_i) \rangle \wedge AT(\sigma_{m-1})].$$

A state $\sigma \in \Sigma^*$ *is* nondeterministic *iff* $||\mathit{closure}(\sigma)|| > 1$.

Example 3 *Figure 5 shows the discrete embedding* Σ^* *for Fig. 2. The figure is obtained from Fig. 2 by eliminating any states in which a process is in transit, and for each state* $(x_0, y_0..z_0, x_1, y_1..z_1)$ *retaining only* (x_0, y_0, x_1, y_1). *States* σ *and* σ' *are written on successive lines, or an edge is drawn with initial state* σ *and final state* σ' *iff* $\sigma' \in \mathit{closure}(\sigma)$.

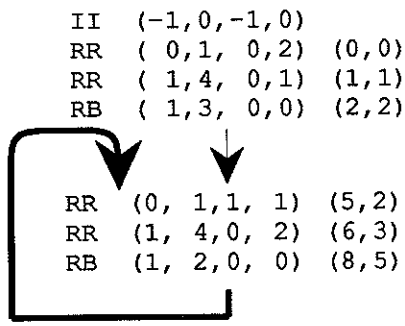


Figure 5: Embedded, discrete state space Σ^* corresponding to continuous space Σ of Fig. 2

The introduction informally defined a TES. We now formalize the term by defining it as a sequence of states in Σ^*. The sequence has finite length iff the final state is dead.

Definition. *A possibly infinite sequence of m states (where m may be infinite) $\Pi = \sigma_0, \sigma_1, \ldots, \sigma_{m-1}$ is a* TES *of the timed transition system $\mathcal{S} = \langle V, \Sigma, \Theta, \tau \rangle$ iff $\sigma_0 = \Theta$ and $\langle \forall i : 0 \leq i \leq m-2 :: \sigma_{i+1} = closure(\sigma_i) \rangle \wedge [m < \infty$ iff $\tau(\sigma_{m-1}) = \emptyset]$.*

Example 4 *From Fig. 5, there is one possible TES for the mutual exclusion program: $\Pi = (-1, 0, -1, 0)$, $(0, 1, 0, 2)$, $(1, 4, 0, 1)$, $(1, 3, 0, 0)$, $(0, 1, 1, 1)$, $(1, 4, 0, 2)$, $(1, 2, 0, 0)$, $(0, 1, 1, 1)$, ... This corresponds to the location sequence defined in the introduction.*

A TES contains some but not all states that a transition system reaches in a particular execution, because the TES contains only states in discrete space Σ^* and not continuous space Σ. The set of all states reached in an execution satisfying TES $\Pi = \sigma_0, \sigma_1, \ldots, \sigma_{m-1}$ (where m may be infinite) is denoted by Π^*. State $\sigma \in \Pi^*$ iff $\sigma = \sigma_0 \vee \sigma = \sigma_{m-1} \vee \langle \exists i : 0 \leq i \leq m-2 :: \sigma_i \rightarrow \ldots \rightarrow \sigma \rightarrow \ldots \sigma_{i+1} \rangle$.

Example 5 *(1,3.4,0,0) $\in \Pi^*$ because (1,4,0,1) $\rightarrow$ (1,3.4,0,0.4) $\rightarrow$ (1,3,0,0).*

Definition. *A TES can be partitioned into two subsequences, either of which may be empty. The first subsequence, the* transient execution sequence*, consists of states that occur exactly once in the TES. The second consists of an infinite number of repetitions of a subsequence containing states that occur exactly once in the subsequence. The repeated subsequence is called the* LCES*.*

Example 6 When both processes in Fig. 1 simultaneously start execution, the program always reaches one unique LCES, as evidenced by Fig. 5: (0,1,1,1), (1,4,0,2), (1,2,0,0). This is equivalent to the limit cycle of program locations listed in the introduction: (1,0) for 4 time units and (0,1) for 1 time unit. Figure 5 shows that there is only one possible transient execution sequence that leads to this LCES: (-1,0,-1,0), (0,1,0,2), (1,4,0,1), (1,3,0,0). □

Recall from the introduction that a TPG represents program execution only after both processes have stated execution. Defined next is a modified state transition system, called a *concurrent timed transition system*, which eliminates all states in which a process is in location -1 and hence has not started execution.

Definition. *The* concurrent timed transition system *of a timed transition system $\mathcal{S} = \langle V, \Sigma, \Theta, \tau \rangle$ is $\mathcal{S}^+ = \langle V, \Sigma, \sigma_C, \tau \rangle$, where σ_C is defined as follows: Choose any TES in $\mathcal{S}$. Delete the longest initial*

subsequence of the TES in which in each state σ, $\langle \exists r :: \sigma(\pi_r) = -1 \rangle$. Then σ_C is the initial state of the remaining subsequence. (All TESs of a given timed transition system have the same σ_C because a nondeterministic state can only arise when both processes are executing.)

Example 7 In Fig. 2, σ_C = (0,1,0,2). Thus, deleting state (-1,0,-1,0) from the TES in Example 4 yields the TES of the corresponding concurrent timed transition system. □

5 Timed Process Graphs

This section defines a TPG, and discusses its equivalence to a timed transition system.

5.1 Notation

The following notation is used throughout the paper. 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., $G^0 = (G^0_0, G^0_1)$). Unless otherwise noted, every point $G \in R^2 - \{(\infty, \infty)\}$. For any two points G and G', $G < G' \stackrel{def}{=} G_0 + G_1 < G'_0 + G'_1$. For any continuous path γ and any point G on γ, we write $G \in \gamma$. For any directed, continuous path γ, we write $\gamma.i$ (respectively, $\gamma.f$) to denote the initial (final) point. We assume that $\gamma.i \neq \gamma.f$. For any two continuous paths γ and γ' in R^2, $\gamma \cap \gamma'$ denotes $\{G \mid G \in \gamma \wedge G \in \gamma'\}$. A line segment with open end point G and closed endpoint G' is denoted $L = \overline{[G, G')}$; if both end points are open then $L = \overline{(G, G')}$. A ray, which is a directed line segment, uses the same notation. Line or ray $\overline{[G, (\infty, \infty))}$ has slope one and infinite length.

We define the *cycle time* of process r, denoted ϕ_r, as the time required for process r to pass through each location once, ignoring the time spent blocked. Formally, $\langle \forall r :: \phi_r \stackrel{def}{=} \sum_{0 \leq i < n_r} t^i_r \rangle$. For any point $G \in R^2$, $mod(G) \stackrel{def}{=} (G_0 \bmod \phi_0, G_1 \bmod \phi_1)$. Two points G and G' are congruent, denoted $G \equiv G'$, iff $mod(G) = mod(G')$. Two continuous paths γ and γ' are congruent, denoted $\gamma \equiv \gamma'$, iff there exists a one-to-one correspondence between their points such that corresponding points are congruent. For any line or ray $L = \overline{[G, G')}$, let $mod(L)$ denote $\overline{[mod(G), mod(G'))}$.

5.2 Definitions

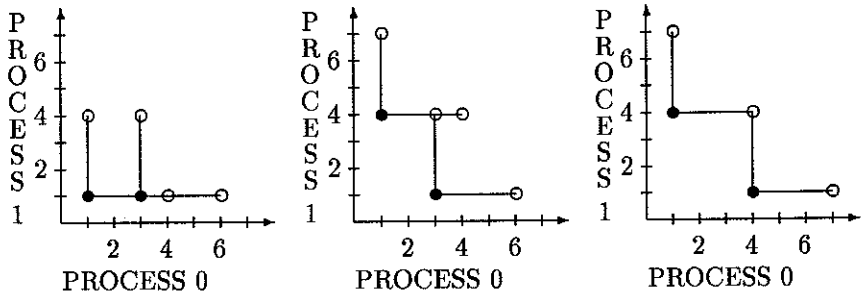
A *TPG* $\Gamma_D = \langle C, \Lambda, G^C, f \rangle$, corresponding to a timed transition diagram set $D = \langle \psi_0, \psi_1 \rangle$, consists of four components:

Φ: a set containing cycle times ϕ_0 and ϕ_1.

Λ: a set of *constraint line generators*, which are a set of line segments that lie in the R^2 plane in the rectangle with opposite vertices (0,0) and (ϕ_0, ϕ_1), each corresponding to one edge in one transition diagram labeled by a non-empty condition. Formally, generator $\overline{[W, X)} \in \Lambda$ iff

$$\langle \exists r \exists k \exists i \exists i' : 0 \leq k < n_r, :: \langle i, i' \rangle \in c^k_r \wedge W_r = X_r = \sum_{0 \leq j \leq k} t^j_r \wedge W_{\bar{r}} = \sum_{0 \leq j < i} t^j_{\bar{r}} \wedge X_{\bar{r}} = \sum_{0 \leq j \leq i'} t^j_{\bar{r}} \rangle.$$

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



(a) Process one does simultaneous P ops

(b) Process one does simultaneous P, V

(c) Both processes do simultaneous P, V

Figure 6: Illegal constraint line geometries.

G^C: an *initial point* satisfying $\langle \exists r : t_r^{-1} = 0 :: G_r^C = 0 \wedge G_r^C = t_r^{-1} \bmod \phi_r \rangle$. Therefore G^C lies on either the x or y axis, within one cycle time of the origin.

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

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

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

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

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

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

Example 8 *Figure 3 contains TPG* $\Gamma = \langle \{\phi_0 = 5, \phi_1 = 3\}, \Lambda, (0,0), f \rangle$, *where* $\Lambda = \{ \overline{[(1,2),(1,3))}, \overline{[(1,2),(5,2))} \}$. *To illustrate set* Λ , $\overline{[W,X)} = \overline{[(1,2),(1,3))}$ *because* $\langle 1,1 \rangle \in c_0^0$, $W_0 = X_0 = \sum_{0 \leq j \leq 0} t_0^j = 1$, $W_1 = \sum_{0 \leq j < 1} t_1^j = 2$, *and* $X_1 = \sum_{0 \leq j \leq 1+0} t_1^j = 3$. *Figure 3 illustrates twelve instances of each constraint line generator.* □

Because 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 resource operation, no constraint lines may overlap (see Fig. 6(a)), the final point of one constraint line can never lie on another constraint line (see Fig. 6(b)), and the final point of all constraint lines must be distinct (see Fig. 6(c)).

Recall from Fig. 4 that (3,3) is dead and points (1,3) and (3,1) are nondeterministic. A point is *nondeterministic* iff the transition out of the point is not unique. All other points are *deterministic*. A point is *dead* iff there is no transition (in the sense of transition function f) out of the point. Informally, dead points represents states in which both processes are blocked

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

"Execution" of a program is represented by a TET. A TET is either a dead point or a directed continuous path consisting of a sequence of horizontal and diagonal rays. Just as there may be multiple PESs given an initial state, there may exist multiple TETs rooted at a given point.

Definition. *A timed execution trajectory of a TPG Γ rooted at any point $G^0 \in R^2 - \{(\infty, \infty)\}$ is either (1) a point G^0 or (2) a directed, continuous path rooted at G^0. Case (1) holds iff $f(G^0) = \emptyset$. Case (2) holds iff the path is a ray sequence $\overline{[G^0, G^1)}$, $\overline{[G^1, G^2)}$, ..., $\overline{[G^{n-1}, G^n)}$ (where n may be infinite) satisfying $\langle \forall i : 0 \leq i < n :: G^{i+1} \in f(G^i) \rangle$.*

Example 9 Consider the TET in Fig. 3: $\overline{(0,0),(2,2)}$, $\overline{(2,2),(5,2)}$, $\overline{(5,2),(8,5)}$, $\overline{(8,5),(10,5)}$, ... Each line segment $\overline{[G, G')}$ satisfies $G' \in f(G)$. For example, $(2,2) \in f_2(0,0)$ because (0,0) does not lie on a constraint line instance and a slope one ray rooted at (0,0) first intersects a constraint line instance at point (2,2); hence $S_1((0,0)) = (2,2)$. In Fig. 4, there are two TETs rooted at (2,0): $\overline{[(2,0),(3,1))}$, $\overline{[(3,1),(3,3))}$ and $\overline{[(2,0),(3,1))}$, $\overline{[(3,1),(6,1))}$, $\overline{[(6,1),(\infty,\infty))}$. Let L_1 (respectively, L_2) be the vertical (horizontal) constraint line containing point (3,1). The two TETs arise at point (3,1) because $f(3,1) = \{S_1((3,1), L_1), S_1((3,1), L_2)\}$ = { (3,3), (6,1) }. □

In Fig. 3, the TETs rooted at $(5,2)$ and $(10,5)$ are congruent. That is, adding the vector (ϕ_0, ϕ_1) to all points on the trajectory rooted at $(5,2)$ yields the trajectory rooted at $(10,5)$. In general, TETs rooted at congruent points containing only deterministic points are congruent, as the following Lemma establishes.

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

Proof: See Appendix A. (The proof establishes that the algebraic form of f preserves a property that is apparent from illustrations of TPGs.) □

5.3 TPGs that Represent Timed Transition Systems

The continuous state space Σ of a timed transition system can be interpreted as a four dimensional Cartesian product, in which two dimensions are integer (i.e., representing current locations π_0, π_1) and two real (i.e., representing minimum times ρ_0, ρ_1). A state is a four dimensional coordinate in this space. A TPG is a two dimensional graph, which represents a projection of the four dimensions of a timed transition system. The projection is defined by function h. Function h maps a state $\sigma \in \Sigma$ in a timed transition system to an infinite set of congruent points $h(\sigma) \subseteq R^2$ in a TPG. Note that multiple states may map to the same congruence class of TPG points.

Definition. *Consider a point G of TPG $\Gamma = \langle C, \Lambda, G^C, f \rangle$ and a state σ of concurrent timed transition system $\langle V, \Sigma, \sigma_C, \tau \rangle$. Then $h(\sigma) \stackrel{def}{=} \{G \mid G \in R^2 \wedge \langle \forall r, \exists i_r :: \sigma(\pi_r) = i_r \wedge G_r \equiv \sum_{0 \leq k \leq i_r} t_r^k - \sigma(\rho_r) \rangle\}$. Point G represents state σ if $G \in h(\sigma)$.*

Example 10 The correspondence of states and points in Fig. 2 exemplifies function h; $h((0,0,0,2)) = h((1,4,0,0)) = \{G \mid G \equiv (6,3)\}$. Note that in Fig. 2 $h(\sigma)$ is undefined for states (-1,0,-1,0) up to but excluding state (0,1,0,2) because these states do not arise in a concurrent timed transition system (See Example 7.).) □

The correspondence between states and points, defined by function h, allows definition of when a TET in a TPG represents a TES in a corresponding timed transition system.

Definition. *Given a TPG* Γ *and a concurrent timed transition system* $\mathcal{S}^+$, *a TET* γ *of* Γ represents *a TES* Π *of* $\mathcal{S}^+$ *iff* $\langle \forall \sigma, \exists G : \sigma \in \Pi \wedge G \in \gamma :: G \in h(\sigma) \rangle$.

A TPG $\Gamma_D = \langle V, \Sigma, \Lambda, \tau \rangle$ represents a concurrent timed transition system $\mathcal{S}_D^+ = \langle V, \Sigma, \sigma_C, \tau \rangle$, both corresponding to timed transition diagram set D, iff initial point G^C represents initial state σ_C and every TES in $\mathcal{S}_D$ is represented by some TET in Γ_D.

Definition. *Given a concurrent timed transition system* $\mathcal{S}^+ = \langle V, \Sigma, \sigma_C, \tau \rangle$ *and TPG* $\Gamma = \langle C, \Lambda, G^C, f \rangle$, Γ represents $\mathcal{S}^+$ *iff the following conditions are satisfied:*

E1: $G^C \in h(\sigma_C)$, *and*

E2: Given any TES Π *of* $\mathcal{S}^+$, $\langle \exists \gamma : \gamma$ *is a TET in* $\Gamma :: \gamma$ *represents* $\Pi \rangle$.

In general, a TET consists of a *transient portion* followed by an infinite number of repetitions of a LCET. Either portion may be empty. The final state in the transient portion is the initial state of the first cycle of the LCET. These concepts are formalized in the following definition.

Definition. *Consider a TET* γ. *A directed, continuous path* $\hat{\gamma}$ *is a* 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.*

Consider an equivalent concurrent timed transition system $\mathcal{S}^+$ and TPG Γ. Consider any TET in Γ and the TES in $\mathcal{S}^+$ that the TET represents. The transient (respectively, limit cycle) execution trajectory of the TET represents the transient (respectively, limit cycle) execution sequence of the TES.

Example 11 In Fig. 3, the TET contains a transient execution trajectory with initial and final points $(0,0)$ and $(3,2)$, respectively, followed by an infinite number of congruent LCETs. One is $\overline{[(3,2),(5,2)),}$ $\overline{[(5,2),(8,5))}$; another, congruent LCET is $\overline{[(10,5),(13,8))}$, $\overline{[(13,8),(15,8))}$. The two are congruent because the first and second rays of the first subtrajectory are congruent to the second and first rays of the second subtrajectory, respectively. □

6 Computational Geometric Analysis of TPGs

This section solves four software performance problems by casting them to an equivalent geometric problem.

Problem P1: *State the necessary and sufficient conditions for a TET in a TPG to contain a LCET.*

Problem P2: *Given a TPG and any point* $G \in R^2$, *output a representation of the set of TETs rooted at* G *in* Γ.

Precise statement of P3 requires the following definition. A point G is free, denoted $C_F(G)$, if G represents some deterministic state in which all processes are running, and in all TESs containing the state, in some subsequent state some process is block. (A diagonal ray rooted at a free point never intersects a constraint line in Λ^* thus has infinite length; hence the name "free.") Formally, $C_F(G) \stackrel{def}{=} \langle \nexists L : L \in \Lambda^* :: G \in L \rangle \wedge S_2(G) = (\infty, \infty)$.

Problem P3: *Given a TPG, determine if there exists any free initial point G^C; if there is, output one such point.*

Statement of P4 requires categorization of LCESs as *blocking* or *non-blocking*. A non-blocking LCES contains only running states. Geometrically, the corresponding LCET is a slope one diagonal ray. A blocking LCES contains a state in which a process is blocked. Geometrically, the corresponding LCET contains a horizontal or vertical ray. Furthermore, we say that two TETs are *homotopic* if they are continuously transformable avoiding the constraint lines; the term originates with Lipski and Papadimitriou [21] for paths in UPGs. For example, the TETs rooted at all points in $\{G \mid 5 \leq G_0 \leq 6 \wedge G_1 = 0\}$ in Fig. 4 are homotopic.

Problem P4: *Given a TPG, output one element of each equivalence class of blocking LCETs, and one (non-blocking) LCET from each set of homotopic TETs rooted at a free initial point.*

6.1 Problem P1

The following theorem, along with the equivalence of transition systems and TPGs, implies that any TES that does not contain a dead state and contains a finite (possibly zero) number of nondeterministic states reaches a limit cycle. Therefore Theorem 1 solves P1. Appendix B contains the theorem proof.

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

6.2 Computing Function f

Essential to solving P2 to P4 is a method of computing transition function f. By definition, computation of $f(G)$ for any point G in R^2 requires either computing $f_1(G)$ if $mod(G)$ lies on a line in Λ; otherwise we compute $f_2(G)$. Computation of f_1 and f_2 is discussed below.

A practical consideration: We henceforth assume that the delays in a timed transition diagram are integers, rather than reals: $\langle \forall r :: t_r^{-1} \in Z \wedge \langle \forall i : 0 \leq i < n_r :: t_r^i \in Z^+ \rangle \rangle$. Otherwise the computational geometric algorithms to be presented will not work correctly with finite precision arithmetic (e.g., computation of the *mod* operation is subject to roundoff error). The assumption of integer delays is not unreasonable in practice for software performance evaluation. For example, in measurements from a computer with a microsecond period clock and all measured times are rational numbers of the form $\frac{x}{10^6}$, where $x \in Z$. Therefore scaling all measurements by the inverse of the clock period (e.g., 10^6) yields the integer quantities required by the proposed algorithms.

Computation of $f_1(G)$: Computation of $f_1(G)$ for point G on line L requires computation of $S_1(G, L)$, which in turn requires computation of all points of intersection of line $\overline{(L.i, L.f)}$ with all lines except L in Λ^*. We cannot directly compute S_1 by its definition because $||\Lambda|| = \infty$.

The problem of Λ^* containing an infinite number of lines will recur throughout this section, so we introduce a partitioning of the R^2 plane into rectangles called *quadrants*. The axes and dotted lines in Figs. 3 and 4 form quadrant boundaries. Formally, quadrant boundaries are formed by the lines $x = i_0\phi_0$ and $y = i_1\phi_1$, where $\langle \forall r :: i_r \in Z \rangle$. The quadrant bounded by the x-axis, y-axis, and lines $x = \phi_0$ and $y = \phi_1$ is called the *initial quadrant*. Thus the subset of Λ^* contained in the initial quadrant is exactly Λ.

To compute S_1, we map G to a congruent point in the initial quadrant, then compute $G' = S_1(mod(G), mod(L))$ (because $G \in L \Rightarrow mod(G) \in mod(L)$) by Lemma 1), and finally map G' back to the quadrant containing G. Formally,

$$S_1(G, L) = S_1(mod(G), mod(L)) + (\lfloor G/\phi_0 \rfloor, \lfloor G/\phi_1 \rfloor).$$

The correctness follows from Lemma 1.

An algorithm solving problems P2 to P4 may require computation of $f_1(G)$ and hence S_1 for many points G. Therefore we can reduce the execution time of each evaluation of S_1 by precomputing $\hat{S}(L) \stackrel{def}{=} \{G' \mid \langle \exists L' : L' \in \Lambda :: \overline{(L.i, L.f)} \cap L' \rangle\}$ for all $L \in \Lambda$, which then reduces the computation of each evaluation of $S_1(G, L)$ to evaluating the smallest point in set $\{mod(L).f\} \cup \{G' \mid G' \geq mod(G) \wedge G' \in \hat{S}_1(L)\}$. Computing $\hat{S}_1$ is equivalent to computing all intersections of a collection of horizontal and vertical lines, which is a well known computational geometric problem (e.g., see [34, Ch. 27]).

Computation of $f_2(G)$: We propose algorithm F2 (Fig. 7) to compute $f_2(G)$. This and successive algorithms are written in a Pascal-like psuedocode. They use a data structure called *point*:

`point: record x0,x1: integer end.`

Recall that $f_2(G)$ is the smallest point at which a slope one ray rooted at G intersects a constraint line instance, or, if there is no such intersection point, $\hat{G} = (\infty, \infty)$. The well known problem of ray shooting with line segments [4, pp. 234-247] can be used to find $f_2(G)$: Given finite set of line segments in a plane, a point, and a direction, find the first line segment intersected by a ray rooted at the point with the given direction. The typical ray shooting solution first stores the line segments in a data structure, so that subsequent queries consisting of a point and a direction can be answered in sublinear time. We assume that we have a ray shooting algorithm, invoked as

`ShootRay(in point P, direction D, set of line L; out point Poi, line Loi).`

The input parameters (labeled by `in`) are point P; direction D (either + for a ray directed away from the axes or - for a ray directed toward the axes); and set L, containing a finite number of line segments. The output parameters (denoted by `out`) are *Poi*, containing the point of intersection or (∞, ∞) if the ray does not intersect a line segment; and *Loi*, containing the line in L on which *Poi* lies, if *Poi* is not (∞, ∞).

Finding $f_2(G)$ is equivalent to the ray shooting problem using as the set of line segments Λ^*. However, because Λ^* is an infinite set, `ShootRay` cannot be used directly. Earlier, we proposed computing $f_1(G)$ by mapping G to the initial quadrant. Similarly, we compute $f_2(G)$ by shooting a ray with initial point $mod(G)$ and using as the line segment set Λ unioned with two more line segments, closed at both end points, which are the top and right edges of the initial quadrant rectangle. Algorithm F2 repeatedly calls `ShootRay` until `ShootRay` returns either $P = (\infty, \infty)$ or a *Loi* that is not the top or right initial quadrant edge. A TPG and point G requiring two `ShootRay` operations because ray $\overline{(G, f_2(G))}$ lies in exactly two quadrants arises in Fig. 3 and is illustrated in (Fig. 8). In general, a TET whose points lie in n different quadrants can be partitioned into n subtrajectories, each contained within exactly one quadrant. Formally, any TET γ can be partitioned into $\gamma_0, \gamma_1, \ldots, \gamma_{n-1}$ satisfying

$$\gamma.i = \gamma_0.i \wedge \gamma.f = \gamma_{n-1}.f \wedge \langle \forall j : 0 \leq j < n :: \gamma_j.f = \gamma_{j+1}.i \rangle \wedge$$
$$\langle \forall j : 0 \leq j < n-1 :: \langle \exists G : G = \gamma_j.f \wedge (G_0 \equiv \phi_0 \vee G_1 \equiv \phi_1) \rangle.$$

So to compute γ, each iteration of the while loop in Fig. 7 computes γ_0, then γ_1, and so on, using *only* the initial quadrant.

```
point function F2(in point G)
var
    i_0, i_1:   integer;
    G':         point;
    P:          set of point;
    L:          line;
    C:          set of line;
begin
    {line segments to shoot are those in Λ along with top and right initial quadrant edges}
        C := Λ ∪ { [(0, φ_1), (φ_0, φ_1)], [(φ_0, 0), (φ_0, φ_1)] };
    {map G to initial quadrant}
        ⟨∀r :: i_r := ⌊G_r / φ_r⌋; G'_r := G_r mod φ_r⟩;
    ShootRay(G', +, C, G', L);
    while (L∉Λ) begin
        if (L ∈ Λ) then exitloop
        else if G' ∈ P then return (∞, ∞)
        else P := P ∪ {G'};
        {map G' to left or bottom edge or initial quadrant}
            ⟨∀r :: i_r := i_r + ⌊G'_r / φ_r⌋; G'_r := G'_r mod φ_r⟩;
        ShootRay(G',+, C, G', L);
        end
    {¬C_F(G) holds; map G' to correct quadrant }
        ⟨∀r :: G'_r := G'_r + i_r φ_r⟩;
    return G'; end
```

Figure 7: Algorithm to compute $f_2(G)$.

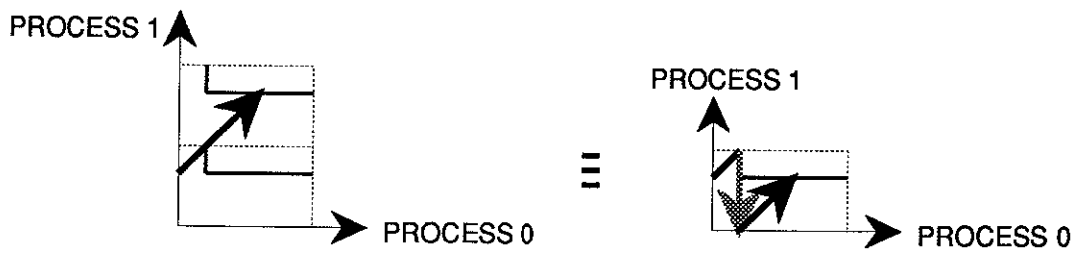


Figure 8: Illustration of using `ShootRay`, but only within initial quadrant.

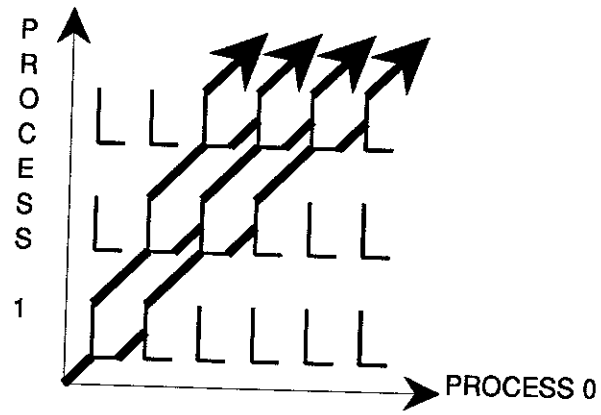


Figure 9: TPG containing an infinite number of TETs.

6.3 Problem P2

The definition of TET suggests a method of constructing a TET. The definition cannot directly be used, because $f(G)$ may contain either one (for deterministic G) or two (for non-deterministic G) points. If $f(G)$ contains two points, then there are at least two possible TETs rooted at G. We say "at least two" because if a point on one of the TETs rooted at G contains a point distinct from G that is nondeterministic, there will be more than two TETs rooted at G. Therefore a solution to P2 requires calculating a set of TETs. More importantly, there could be an infinite number of TETs rooted at a point, as illustrated in Fig. 9. (This occurs when the program timings are such that in any state in any possible TES, the program will eventually reach another race condition.) To insure that algorithm B2 terminates, we choose an integer value *maxNPaths* such that if we find more than *maxNPaths* possible TETs, we assume that there are an infinite number of TETs and terminate the algorithm without further exploration. Another consideration is that the TET either has finite length and ends in a dead point (in which case the entire TET can be explicitly output) or has infinite length and ends in a LCET (in which case, algorithm B2 should output the transient execution trajectory and the first LCET).

The proposed algorithm (B2 in Fig. 10) constructs a graph. If the set of all possible TETS rooted at G^0 is simply the point G^0, then the graph contains one vertex, labeled G^0. Otherwise the graph contains one vertex for each ray end point in both the transient execution trajectory and the first LCET of all possible TETs rooted at G^0. A vertex is labeled by the TPG point to which it corresponds. An edge directed from vertices labeled by TPG points G' to G'' exists if either $G'' \in f(G')$ or G'' is on the graph path from G^0 to G' and $G' \equiv G''$. Therefore, with a sufficiently large value of *maxNPaths*, each LCET is represented by a graph cycle, and that if the set of all possible TETs rooted at G^0 contains no LCETs then the graph is a tree.

Each vertex is colored green or red. Initially, a green vertex for G^0 is inserted. The for all loop picks a green node (let the node be labeled G) and adds $||f(G)||$ vertices, each labeled by a unique point in $f(G)$. The initial vertex color is red iff the point labeling the vertex is either dead or congruent to a vertex on the path back to the root (and hence in a LCET). The algorithm terminates when all vertices are red.

```
function B2(in point GI, out vertex v)
type
    colors:    enum { green, red };
    vertex:    record
               label:      point;
               color:      colors;
               children:   set of pointer to vertex;
               lcet:       pointer to vertex;
               end
var
    nPaths:    integer;
    P:         point;
    G, G', G'': vertex;
    P :        set of point;
begin
    nPaths := 0;
    {Insert tree root}
        G := new(vertex);
        G.label := GI; G.color := green; G.nPaths := 1; G.children := ∅; G.lcet := null;
    v := G;

    repeat
        begin
        G := any point in v such that G.color=green;
        if (C_D(G)) then G.color := red
        else begin
            P := f(G);
            nPaths := nPaths+||P||-1; if (nPaths>maxNPaths) then return;
            for each P ∈ P begin
                G' := new(vertex);
                G'.label := P;
                G'.color := green;
                G.children := G.children ∪ {G'};
                if (⟨∃ G'' :: G'' is on path from GI to G' ∧ G''.label ≡ G'.label ⟩) then
                    begin G'.lcet := G''; G'.color := red; end
                end
            end
        end
    until (⟨∀G : G ∈ graph rooted at v :: G.color=red ⟩);
end
```

Figure 10: Algorithm solving P2.

```
point function B3()
var
    G^C:              point;
    L,left,bottom:    line;
    L:                set of line;
begin
    G^C := (∞, ∞);
    {line segments to shoot are those in Λ along with bottom and left initial quadrant edges}
        left := [(0,0), (φ0, 0)]; bottom := [(0,0), (0, φ1)];
        L := Λ ∪ { left, bottom};
    for all (L ∈ Λ) begin
        ShootRay(L.f, −, L, G^C, L);
        if (L ∈ {left,bottom}) then begin G^C := f2(L.f); if (G^C = (∞, ∞)) then return G^C; end
    end
end
```

Figure 11: Algorithm solving P3.

6.4 Problem P3

There exists a free initial point G^C in a TPG iff there exists a constraint line generator $L \in \Lambda$ satisfying (1) $f(L.f) = (\infty, \infty)$ and (2) a diagonal line segment whose end points are $L.f$ and a point on an axis does not cross a generator in Λ. Condition (1) identifies a potential LCET rooted at a free point, and condition (2) insures that this LCET is reachable from some initial condition. Algorithmically (B3 in Fig. 11), we must examine the final point of each generator in Λ until we find one that satisfies (1) and (2); if no such point exists, then the algorithm reports that a free trajectory does not exist. Evaluating condition (1) simply requires algorithm F2 (Fig. 7). Evaluating (2) requires shooting a ray rooted at the final point of a constraint line generator in the *negative* direction and seeing if its first intersection point lies on an axis. B3 returns (∞, ∞) if there exists no free initial point G^C. As in algorithm F2, we add two edges of the initial quadrant (in this case, the left and bottom) to the set of line segments Λ that are used in the ray shooting algorithm.

6.5 Problem P4

The set of all *non-blocking* LCETs can be found by modifying function B3 as follows. In Fig. 11, add the declaration "$\mathcal{L}'$: set of lines" with initial value $\emptyset$. Change the return value of B3 from point to "set of line." Change "return G^C" to "$\mathcal{L}' := \mathcal{L}' \cup \{\overline{[L.f,\ L.f + (\phi_0, \phi_1)]}\}$." Finally, add "return $\mathcal{L}'$" just before the final "end."

Informally, the set of all *blocking* LCETs can be found as follows. Observe that a set of TETs that intersects a given constraint line instance, denoted L, can share a common point that lies on L. The common point is either a dead point, in which case the point is the final point of all the TETs, or $L.f$. In the later case, this set must either have the same LCET or reach the same dead state that lies on another constraint line instance. If they reach a LCET, it is therefore only necessary to consider the TET rooted at $L.f$ and determine if it contains another point congruent to $L.f$. Therefore, to find all LCETs that intersect a constraint line, we must generate just the initial portion of each TET rooted at the final point of each constraint line in Λ with a final point either congruent to the initial pont or a nondeterministic point. (By Lemma 1, we need only consider the lines in Λ and not Λ^*.) Lemma 2 makes this precise.

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

Proof: See Theorems 3 and 4 in [3]. □

Let ζ denote the set of LCETs that satisfy the Lemma 2. We can determine which LCETs in ζ are reachable by calculating, for each line L in a LCET in ζ, `ShootRay`$(L.f, -, \mathcal{L}, P, L)$, where $\mathcal{L}$ is the same line set as in algorithm B3. If P, the point of intersection returned, lies on the bottom or left edge of the initial quadrant, then the LCET containing L is reachable. Therefore we propose as a solution to P4 the exhaustive testing of all L and i, evaluating $f^i(L.f)$ as described in §6.2.

The final solution to P4 is the union of the set of non-blocking and blocking LCETs.

7 Conclusions

This paper demonstrated that properties about the set of all possible timed execution sequences of certain parallel programs can be exactly analyzed by solving an equivalent computational geometric problem. The obvious question is whether other program classes can also be analyzed with geometry.

The analysis is limited to two processes. The extension to d processes requires ray shooting in a d dimensional Cartesian graph with $d-1$ dimensional hyperplanes in a non-simple arrangement that are bounded in one dimension is required. To our knowledge, this is an open computational geometric problem, whose solution would allow solution of problems P1 through P4 for an arbitrary number of processes. The closest problem solved in d dimensions is ray shooting with unbounded hyperplanes that form a simple arrangement (e.g., see [26]).

A second open problem, in two dimensions is the following. Consider a TPG for which we have already computed the TET (e.g., Fig. 3). How does the solution change if we vary one delay t_r^i by a small amount? This is a common question asked when maintaining a program, and changing one code segment in one process by a small amount. Recall from the introduction the analogy of turning a knob: a small change retains the same location sequence of a LCES, but when the knob is turned past a "critical point" the location sequence of the LCES suddenly changes. Geometrically, changing one delay represents stretching or shrinking (dilating) each interval of the process r axis, corresponding to location i. The consequence of dilation is for some non-free diagonal rays in a TET to change to point at which they intersect a constraint line. If the dilation is sufficiently large, the ray will *miss* a constraint line altogether, which results in the location sequence of a LCES changing. The geometric problem is: given a TPG and a TET, determine how much one delay can be changed before any diagonal ray in the TET misses a constraint line that it formerly intersected. This yields the critical point values.

Another open problem is to replace A2.3 by the assumption that execution of both processes is interleaved on the same processor. The resulting TET no longer contains diagonal rays. The geometric problem is to solve P2, given a schedule of execution (e.g., round robin scheduling).

A final open problem is the following. The program class considered permits no program variables. Can this assumption be relaxed? Because parallel programs can be represented as transition systems, the ability to represent variables and their changes in values would allow geometric representation of an arbitrary parallel program.

References

[1] M. Abrams. *Performance Analysis of Unconditionally Synchronizing Distributed Computer Programs Using the Geometric Concurrency Model.* Ph.D. diss., Dept. of Computer Science, Univ. of Maryland, TR-1696, Aug. 1986.

[2] M. Abrams, N. Doraswamy, and A. Mathur, Chitra: visual analysis of parallel and distributed programs in the time, event, and frequency domain, *IEEE Transactions on Parallel and Distributed Systems 3*, 6, Nov. 1992, 672-685.

[3] M. Abrams, *Geometric Performance Analysis of Semaphore Programs*, Technical Report TR-93-29, Virginia Tech, July 1993.

[4] P. K. Agarwal, *Intersection and decomposition algorithms for planar arrangements*, Cambridge: Cambridge Univ. Press, 1991.

[5] G. Balbo, G. Chiola, 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*, Vol. 2, March 1992, 221-240.

[6] R. F. Berry and J. L. Hellerstein, Characterizing and interpreting periodic behavior in computer systems, *Proc. SIGMETRICS 92*, Newport, RI, June, 1992, pp. 241-242.

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

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

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

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

[11] E. W. Dijkstra. Cooperating Sequential Processes, Tech. Rep. EWD-123, Technological University, Eindhoven, The Netherlands, 1965.

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

[13] I. Gorton, Parallel program design using petri nets, *Concurrency Practice and Experience 5*, 2, (Apr. 1993) 87-104.

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

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

[16] T. A. Henzinger, Z. Manna, and A. Pnueli, *Timed Transition Systems,* Dept. of Comp. Sci., Cornell Univ., TR 92-1263, Jan. 1992.

[17] T. A. Henzinger, Z. Manna, and A. Pnueli, *Temporal Proof Methodologies for Timed Transition Systems,* Dept. of Comp. Sci., Cornell Univ., TR 93-1330, March 1993.

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

[19] T. Lehr, Z. Segall et. al., Visualizing performance debugging, *IEEE Computer*, Oct. 1989, 38-51.

[20] B. P. Lester, *The Art of Parallel Programming*, Englewood Cliffs: Prentice Hall, 1993.

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

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

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

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

[25] A. D. Malony, JED: just an event display, in *Performance Instrumentation and Visualization*, ed. M. Simmons and R. Koskela, ACM Press, pp. 99-114, 1989.

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

[27] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski, IPS-2: the second generation of a parallel program measurement system, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, pp. 206-17, April 1990.

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

[29] B. Plateau and A. Staphylopatis. Modeling of the parallel resolution of a numerical problem on a locally distributed computing system. *Proc. SIGMETRICS 82,* 108-117.

[30] B. Plateau and K. Atif. Stochastic automata network for modeling parallel systems, *IEEE Trans. on Soft. Eng. 17,* 10, Oct. 1991, 1093-1109.

[31] F. P. Preparata and M. I. Shamos, *Computational Geometry,* New York: Springer Verlag, 1985.

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

[33] G. Roman and K. Cox, "A Declarative Approach to Visualizing Concurrent Computations," *IEEE Computer 22,* 10, Oct. 1989, 25-36.

[34] R. Sedgewick, *Algorithms in C,* Reading, MA: Addison-Wesley, 1990.

[35] M. Simmons, R. Koskela, and I. Bucher, *Instrumentation for Future Parallel Computing Systems,* New York: ACM Press, 1989.

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

[37] L. Zhang, S. Shenker, D. D. Clark, Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic, *Proc. SIGCOMM 91,* Zurich, Sept. 1991, pp. 133-147.

A Proof of Lemma 1

The following lemma will simplify the proof of Lemma 1.

Lemma 3 *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 timed progress graph Γ with initial points G and G', respectively. If $G \equiv G' \wedge \langle \nexists \hat{G} : \hat{G} \in \gamma \vee \hat{G} \in \gamma' :: C_N(\hat{G}) \rangle$ then $\gamma \equiv \gamma'$.*

Proof: 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 3, $\gamma = \gamma'$.

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 $f = f_1$ or $f = f_2$.

C1: $C_D(G^1)$: Follows from Lemma 3.

C2: $\neg C_D(G^1) \wedge f(G^1) = f_1(G^1)$

$f(G^2) = f_1(G^2)$

, by hypothesis $G^1 \equiv G^2$ and the definitions of f_1 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 $f(G^1) = f_1(G^1)$, last deduction, and the definition of f

$S_1(G^1, L_1) \equiv S_1(G^2, L_2)$

, by the definition of S_1, 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: $\neg C_D(G^1) \wedge f(G^1) = f_2(G^1)$

$f(G^2) = f_2(G^2)$

, by hypothesis $G^1 \equiv G^2$ and the definitions of f_2 and Λ^*

$\langle \exists L : L \in \Lambda^* :: G^1 \in L \vee G^2 \in L \rangle$

, by hypothesis $f(G^1) = f_2(G^1)$, last deduction, and the definition of f_2

$\langle \forall L : L \in \Lambda^* :: \overline{[G^1, (\infty, \infty))} \cap L \neq \emptyset \Rightarrow \overline{[G^2, (\infty, \infty))} \cap L \neq \emptyset \rangle$

, by hypothesis $G^1 \equiv G^2$ and the definition of Λ^*

$S_2(G^1) \equiv S_2(G^2)$

, by last deduction and the definition of S_2

There must exist points $\hat{G}^1$ and $\hat{G}^2$ such that $\langle \forall i : i \in \{1, 2\} :: f_2(G^i) = \{\hat{G}^i\} \rangle$

, by last deduction (and observing that $\langle \forall G : G \in F^2 - (\infty, \infty) :: ||S_2(G)|| = 1 \rangle$)

$\hat{G}^1 \equiv \hat{G}^2$

, combine previous two deductions

□

B 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 two lemmas.

Lemma 4 $||\Lambda|| < \infty$.

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

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, $C_L(G) \stackrel{def}{=} \langle \exists L : L \in \Lambda^* :: G \in L \wedge S_1(G, L) = L.f \rangle$.

Lemma 5 *Consider any TET γ in a TPG. If $||\{G \mid G \in \gamma \wedge C_N(G)\}|| < \infty$ 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 $\langle \exists G : G \in \gamma :: \langle \nexists L : L \in \Lambda^* :: G \in L \rangle \wedge S_2(G) = \{(\infty, \infty)\} \rangle$.

case (a):

There must exist a G such that $G \in \gamma \wedge \langle \nexists L : L \in \Lambda^* :: G \in L \rangle \wedge S_2(G) = \{(\infty, \infty)\}$

, by hypothesis

$f(G) = f_2(G) \wedge f_2(G) = \{(\infty, \infty)\}$

, 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 $\langle \exists L : L \in \Lambda^* :: G \in L \rangle \vee S_2(G) \neq \{(\infty, \infty)\}$.)

If $\langle \exists G : G \in \gamma :: C_N(G) \rangle$ then choose G^N to be $\max\{G \mid G \in \gamma \wedge C_N(G)\}$ 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 $S_2(G) \neq \{(\infty, \infty)\}$ and definition of C_F and S_2

(†) $\langle \forall G : G \in \gamma :: \neg C_F(G) \vee C_L(G) \rangle$
, 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 4

$\langle \forall G : G \in \hat{\gamma} \cap \Lambda^* :: C_L(G) \rangle$
, 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_1

$\langle \exists G, G' : G, G' \in \hat{\gamma} :: G \equiv G' \rangle$
, 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 5 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

□