

Original citation:

Moitra, A. and Joseph, M. (1991) Determining timing properties of infinite real-time programs. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-172

Permanent WRAP url:

<http://wrap.warwick.ac.uk/60868>

Copyright and reuse:

The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:

The version presented in WRAP is the published version or, version of record, and may be cited as it appears here. For more information, please contact the WRAP Team at: publications@warwick.ac.uk



<http://wrap.warwick.ac.uk/>

_____Research report 172_____

DETERMINING TIMING PROPERTIES OF INFINITE REAL-TIME PROGRAMS

ABHA MOITRA MATHAI JOSEPH

(RR172)

This paper describes a method for determining whether timing deadlines over the commands of a real-time program will be satisfied during execution. A program is assumed to consist of a fixed number of nonterminating processes, each having a simple cyclic structure. Processes may contain nondeterministic commands any may inter-communicate synchronously or asynchronously. Each command has an execution time which takes any value from a non-empty real interval.

There are infinitely many possible executions of such a program, and each execution is of infinite length. And if the program P has n processes, it may be implemented on any system with m processor $1 \leq m \leq n$. We show how this set of executions can be reduced to a finite set, $R-Exe_m(P)$ of finite length executions, and then prove that any deadline that is violated in any infinite execution will be violated in one of these finite executions. Thus the feasibility of executing the infinite program P on m processors can be determined by examination of the set $R-Exe_m(P)$ of executions of P .

Determining Timing Properties of Infinite Real-Time Programs

Abha Moitra*
General Electric CR&D

Mathai Joseph†
University of Warwick

Abstract

This paper describes a method for determining whether timing deadlines over the commands of a real-time program will be satisfied during execution. A program is assumed to consist of a fixed number of nonterminating processes, each having a simple cyclic structure. Processes may contain nondeterministic commands and may intercommunicate synchronously or asynchronously. Each command has an execution time which takes any value from a non-empty real interval.

There are infinitely many possible executions of such a program, and each execution is of infinite length. And if the program P has n processes, it may be implemented on any system with m processors, $1 \leq m \leq n$. We show how this set of executions can be reduced to a finite set, $R\text{-Exe}_m(P)$ of finite length executions, and then prove that any deadline that is violated in any infinite execution will be violated in one of these finite executions. Thus the feasibility of executing the infinite program P on m processors can be determined by examination of the set $R\text{-Exe}_m(P)$ of executions of P .

1 Introduction

Scheduling theory analyses (cf. Liu and Layland [4]) have generally considered a real-time program to consist of disjoint, deterministic, periodic processes and have then examined the timing constraints or deadlines that might be satisfied by the program under various conditions: e.g. when executed on specific architectures, or under different scheduling disciplines or with different resource requirements (e.g. [5]). Recent extensions to this kind of analysis enable some account to be taken of the particular features of programming languages

*K1-3C18, Research and Development Center, General Electric Company, P.O. Box 8, Schenectady, N.Y. 12301, U.S.A.

†Department of Computer Science, University of Warwick, Coventry CV4 7AL, U.K. (supported by Research Grants GR/D/73881 & GR/F 57960 from the Science and Engineering Research Council).

and offer solutions such as the use of priority inheritance protocols and priority ceilings for AdaTM tasks [3].

In practice, real-time programs consist of processes (or tasks) which are *cyclic* rather than periodic, have *inter-process communication* and are *nondeterministic*. Such processes have no fixed repetition frequency and have communication dependencies which make simple scheduling predictions inapplicable. If the execution time of any command in the program is realistically assumed to lie in a bounded, real interval, then each command, and thus the program, may have infinitely many possible executions which differ in their timing properties. And since most real-time programs are intended to run forever, each execution is of infinite length. Thus a major problem in analysing the timing properties of such programs arises because the set of possible executions is infinite and each execution in the set is of infinite length. Without making simplifying assumptions to consider only commands with fixed execution times and to have very restricted interaction between deterministic processes, it has so far not been possible to establish whether each time constraint or deadline for a program is satisfied in every possible execution.

This paper introduces a method of analysis for real-time programs which shows how the satisfaction of deadlines (or the *feasibility*) can be precisely determined by examining a finite set of executions obtained from the infinite set of possible executions, and with each execution limited to a finite prefix of its infinite execution. Processes are assumed to have a simple cyclic structure, but there are no restrictions about interprocess communication (which may be synchronous or asynchronous) or about nondeterminism. A time constraint is assumed to be a deadline between the execution of one command and the execution of one of a set of commands; such commands may occur in one process or in different processes.

We first show how to limit examination of a program's executions to the finite initial prefixes which must be used to establish infeasibility, and prove that if each of these truncated executions is feasible then every infinite execution of the program is also feasible. This applies to any target architecture for which the following assumption is satisfied.

Assumption A processor is never idle if it can execute a command.

We also show that if an infinite execution of a program is infeasible, then there is an infeasible execution of the same program in which each command takes either its minimum execution time or its maximum execution time. Thus in order to determine feasibility, it is only necessary to consider minimum and maximum execution times for each command. This reduces the set of possible executions to be analysed to a finite set, each member of which is a finite prefix of an infinite execution. The analysis does not consider the effect of using different scheduling policies, such as assigning priorities to processes and allowing the pre-emption of executions, but this can be accommodated into the method if necessary.

2 Program Model

Consider a simple programming language with a set Seq of sequential commands (such as assignment, alternation and repetition) and sets $Send$ and $Receive$ of synchronous and asynchronous communication commands to send and receive messages. For any command C , let $Kind(C) \in Seq \cup Send \cup Receive$. Assume that communication between processes takes place through point-to-point channels: a channel transmits in order values sent by one process to exactly one other process. Let a synchronous communication be implemented by an asynchronous send command and a positive acknowledgement from the receiver. The execution time of an asynchronous communication command consists of a fixed synchronisation time and a transmission time, T_t , which is the time taken for the message to be physically transferred through the channel from the sender to the receiver.

Let a program P be defined as the parallel composition of the sequential processes $P_1 \parallel P_2 \parallel \dots \parallel P_n$. Let each process P_i consist of sequential and communication commands with the following structure:

$$P_i :: C_1; C_2; \dots ; C_{j-1}; cycle(C_j; \dots ; C_k)$$

where initialization is performed by the commands $C_1; C_2; \dots ; C_{j-1}$ and is followed by the nonterminating iteration of $cycle(C_j; \dots ; C_k)$.

A command C in a process P_i is either its first command or will be executed after one of a number of possible preceding commands in P_i has completed execution. Assume that the program P is divergence-free and deadlock-free, so that there are no unbounded repetition commands and the execution of every selected communication command is completed in finite time.

Define $Before(C)$ to be the set of commands from P_i which are the predecessors of C in all possible executions of the program: in any execution, if C is the k th command to be executed by process P_i and $k > 1$, then one of the commands in $Before(C)$ will be the $(k - 1)$ th command of process P_i to be executed.

2.1 Timing and Deadlines

Let $MinTime(C)$ and $MaxTime(C)$ be the the minimum and maximum times for executing the command C and let $Length(C)$ be the real interval $[MinTime(C), MaxTime(C)]$. If $MinTime(C) \leq L \leq MaxTime(C)$ then we will write L in $Length(C)$.

In general, a deadline is of the form “if a command C starts (or finishes) execution at time t then within the time interval $(t, t + d]$, $d > 0$, one of the commands from the set $\{C_1, \dots, C_k\}$ must start (or finish) execution”. Such a timing constraint is represented as

$$D(C) = \langle \{C_1, \dots, C_k\}, d, SF_1, SF_2 \rangle$$

where $SF_1, SF_2 \in \{start, finish\}$ refer to the constraint on C and $\{C_1, \dots, C_k\}$ respectively. For a command C with a periodic deadline d , we have $D(C) = \langle \{C\}, d, start, start \rangle$. A real-time system will be said to *satisfy* its time constraints if it meets all its deadlines. (The notation for specifying deadlines can be extended to specify more complex relations between commands but for simplicity we shall not consider that here.)

3 Program Executions

An execution of a program is defined by specifying the commands that are executed and their ordering. In general programs are non-deterministic, so there is a set of possible executions for each program. Throughout the following we will assume that the execution of a command is never preempted.

Let C be a command in process P_j and let c_i be an execution of C ; we will refer to c_i as an executed command, or an *e-command*. Let F be a function such that $F(c_i) = C$. If $F(c_i)$ is a command which is executed more than once, there will be several e-commands, e.g. c_{i1}, \dots, c_{ih} , which map back to it: this may occur, for example, inside a repetitive command. In the infinite execution of the *cycle* command of a process, an infinite number of e-commands will map to the same program command.

3.1 Proper schedules

A *processor schedule* S_i for a processor i is a sequence of e-commands c_{i1}, \dots, c_{im} executed on that processor. The execution of c_{ij} spans the non-empty interval of time $[c_{ij}/start, c_{ij}/end)$. Let $e\text{-Length}(c_{ij}) = c_{ij}/end - c_{ij}/start$. Further,

$$\forall c_{ij} \in S_i, 1 \leq j < m, c_{ij}/end \leq c_{i(j+1)}/start$$

A *schedule SCH* for a program is a sequence of processor schedules, one for each processor on which part of the program is executed. In practice it might be appropriate to require that all the commands of a process are executed on the same processor but that is not essential for this analysis.

Dependencies between the e-commands of a program P induce a relation which determines when a schedule SCH is a *ProperSeq* schedule. If c_{ij} is the k th e-command executed by

process P_h and $k > 1$, then the set $e\text{-Before}_s(c_{ij})$ of the sequential predecessors of c_{ij} will contain the $(k-1)$ th e-command of process P_h to be executed. Additionally, if c_{ij} is a *receive* e-command and is the t th receive e-command executed by process P_h to receive a value sent by process P_q , then the set $e\text{-Before}_m(c_{ij})$ will contain the t th *send* e-command executed in process P_q to send a value to P_h .

$$\begin{aligned} \text{ProperSeq}(P, SCH) &= \forall S_i \in SCH, \forall c_{ij} \in S_i, \\ &\quad (e\text{-Length}(c_{ij}) \text{ in } \text{Length}(F(c_{ij})) \\ &\quad \wedge c_{ij}/\text{start} \geq \text{Earliest-Start}(SCH, c_{ij})) \end{aligned}$$

where

$$\begin{aligned} \text{Earliest-Start}(SCH, c_{ij}) &= \text{if } c_{ij} \in SCH \text{ then} \\ &\quad \text{if } \text{Kind}(c_{ij}) \in \text{Receive} \text{ then} \\ &\quad \quad \max(\{x/\text{end} \mid x \in e\text{-Before}_s(c_{ij})\} \\ &\quad \quad \cup \{x/\text{end} + T_t \mid x \in e\text{-Before}_m(c_{ij})\}) \\ &\quad \text{else } \max(\{x/\text{end} \mid x \in e\text{-Before}_s(c_{ij})\}) \text{ endif} \\ &\text{endif} \end{aligned}$$

Given a program P and a schedule SCH such that $\text{ProperSeq}(P, SCH)$, every executed command is executed in the correct order. To guarantee that the commands are also executed with no unnecessary delay (see the Assumption above), we have the following predicate:

$$\begin{aligned} \text{FirstProperSeq}(P, SCH) &= \text{ProperSeq}(P, SCH) \\ &\quad \wedge (\text{ShiftedSch}(SCH, SCH') \wedge \text{ProperSeq}(P, SCH') \Rightarrow SCH = SCH') \end{aligned}$$

where

$$\begin{aligned} \text{ShiftedSch}(\langle S_1, \dots, S_n \rangle, \langle S'_1, \dots, S'_n \rangle) &= \text{ShiftedSeq}(S_1, S'_1) \wedge \dots \wedge \text{ShiftedSeq}(S_n, S'_n) \\ \text{ShiftedSeq}(S_i, S_j) &= \#S_i = \#S_j \end{aligned}$$

$$\wedge 1 \leq h \leq \#S_i, (F(c_{ih}) = F(c_{jh}) \wedge c_{ih}/start \geq c_{jh}/start \\ \wedge e\text{-Length}(c_{ih}) = e\text{-Length}(c_{jh}))$$

A program P can have an infinite number of different executions on any specific target architecture. If the target architecture has m processors then the possible executions of the program P are syntactically defined as

$$Exe_m(P) = \{\langle S_1, \dots, S_m \rangle \mid FirstProperSeq(P, \langle S_1, \dots, S_m \rangle)\}$$

The set of actual executions of P is a subset of $Exe_m(P)$ as in general there may be ‘executions’ in $Exe_m(P)$ that are not semantically valid.

3.2 Feasible schedules

A *feasible* schedule is a *proper* schedule which always meets its deadlines. To determine whether every process in a program meets its deadlines, the possible executions of all the processes must be unrolled upto a point where this can be determined.

Thus for a program P and schedule SCH ,

$$\begin{aligned} Feasible(P, SCH) \\ &= ProperSeq(P, SCH) \\ &\wedge \forall S_i \in SCH, \forall c_{ij} \in S_i, \\ &\quad D(F(c_{ij})) = \langle Set_1, d, SF_1, SF_2 \rangle \wedge Set_1 \neq \emptyset \wedge d > 0 \\ &\quad \wedge Extends\text{-upto}(SCH) \geq c_{ij}/SF_1 + d \\ &\quad \Rightarrow \exists l, h : h \leq \#S_l \wedge F(c_{lh}) \in Set_1 \wedge c_{lh}/SF_2 > c_{ij}/SF_1 \\ &\quad \quad \wedge c_{lh}/SF_2 - c_{ij}/SF_1 \leq d \end{aligned}$$

where

$$Extends\text{-upto}(\langle S_1, \dots, S_n \rangle) = \max(\{c_{pq}/end \mid 1 \leq p \leq n, 1 \leq q \leq \#S_i\})$$

3.3 Limited execution

The programs we consider are in general nondeterministic and nonterminating, so for any specific architecture there will be infinitely many different possible executions of a program, with each execution of infinite length. Our objectives are to show the following.

1. The timing properties of the whole program can be determined for an architecture by examining a *finite initial segment* of each execution.
2. If a *finite set* of such initial execution segments meet all the deadlines then so does every infinite execution.

To make it possible for the analysis to be efficiently automated, the length of the initial segments and the size of the finite set of executions should both be as small as possible.

Assume that the program P consisting of n processes P_1, \dots, P_n is executed on m processors, where $m \leq n$. Consider a time instance t during any execution SC of this program when *each* processor has either just completed execution of an e-command or is idle, and for each process P_i , the e-command c_i is the next command to be executed. Let this point be defined as the *snapshot* $\langle F(c_1), \dots, F(c_n) \rangle$ of the program at time t . (Note that not all time points have an associated snapshot.) Let the *snapshot repetition point*, SRP be the first point of time at which a snapshot is repeated. Then truncate the execution at the point *after* SRP where all the deadlines associated with the commands upto SRP are over.

The *truncation point*, TP is defined as follows:

$$LD = \max(\{MaxTime(F(c_i)) + d_i \mid D(F(c_i)) = \langle Set_1, d_i, SF_1, SF_2 \rangle \\ \wedge Set_1 \neq \emptyset \wedge d_i > 0\})$$

$$TP = SRP + LD$$

Note that each execution has its own truncation point TP . For any execution, define the *limited execution* as the truncation of that execution at its truncation point TP .

Let $Exe_m(P)$ be the (possibly infinite) set of executions of program P on m processors. Let $T-Exe_m(P)$ be a set of limited executions obtained from $Exe_m(P)$. $T-Exe_m(P)$ is an infinite set because an execution of any e-command c can take any one of the infinite set of values in the non-empty real interval $[MinTime(F(c)), MaxTime(F(c))]$.

The next step is to reduce the set $T-Exe_m(P)$ to a finite set, and this is done as follows.

Consider some schedule Sch in $T-Exe_m(P)$. The conditions under which this schedule can be dropped from consideration depend on what we want to prove about the pruned set. In this case, we need conditions under which we can determine the feasibility of executing a program on a particular target architecture by considering the feasibility of all the (finite) executions in the set $T-Exe_m(P)$. Most importantly, if an infeasible execution is present in the set $T-Exe_m(P)$, then the pruned set must also contain an infeasible schedule. We prune the set as follows and then prove later that the pruned set satisfies our requirements.

$$R-Exe_m(P) = \{Sch \mid Sch \in T-Exe_m(P) \\ \wedge \forall c \in Sch, e-Length(c) \in \{MinTime(F(c)), \\ MaxTime(F(c))\} \}$$

Theorem 1 (Finiteness): *For any program P , $R-Exe_m(P)$ is a finite set of schedules, each of which is finite.*

Proof: Each schedule in the set $T-Exe_m(P)$ is finite. Since the number of commands in each schedule is finite, if each command c takes either $MinTime(F(c))$ or $MaxTime(F(c))$, then $R-Exe_m(P)$ is a finite set of finite schedules. □

If the processes are deterministic and for all commands C , $MinTime(C) = MaxTime(C)$, the set of finite schedules $R-Exe_m(P)$ is even smaller. Suppose in addition that each process P_i executes on a separate processor and has a fixed period T_i . Then if the processes start execution simultaneously at $time = 0$, the first snapshot will also be at $time = 0$. Assuming that the processors are sufficiently fast there will be another snapshot which, in the worst case, will be at $time = LCM(\{T_i \mid 1 \leq i \leq n\})$. However, in all but the worst case the SRP will be closer to the first snapshot than is the LCM .

Example: Consider a simple program P with 3 cyclic processes, each consisting of a single command: $P_1 :: cycle(C_1)$, $P_2 :: cycle(C_2)$ and $P_3 :: cycle(C_3)$. Let the deadlines for the program be

$$D(C_1) = \langle \{C_1\}, 5, end, end \rangle \\ D(C_2) = \langle \{C_2\}, 7, end, end \rangle \\ D(C_3) = \langle \{C_3\}, 9, end, end \rangle$$

Assume that there is no communication between the processes and that the fixed execution times of the commands C_1 , C_2 and C_3 are 1, 2 and 2 time units respectively. Let each process be executed on a separate processor.

If the processes start execution simultaneously, then the first snapshot is at time 0. The next snapshot will be at time 2, when each process has completed its command. Since the second snapshot is a repetition of the first, $SRP = 2$. The latest deadline, LD , will be at time $2 + 9 = 11$. Thus, the TP will be at time 13. On the other hand, the LCM of the periods of the processes is $5 \times 7 \times 9$, and so an analysis based on the LCM will require examination of the execution until time 315. If the execution times of the commands equals the periods of the processes then the LCM will occur at the same time as the SRP .

4 Variation in Execution Times

The set $R-Exec_m(P)$ is a finite set of finite schedules and it must now be shown how the timing properties of the actual schedules of program P are related to the properties of the schedules in $R-Exec_m(P)$. As a first step towards this we need to investigate how the starting or ending times of commands vary as a result of changing the execution time of a single command.

Let sch be a schedule in $Exec_m(P)$ for some program in which there are e-commands c and c_1 . Consider infinite set of executions $SetA(sch, c)$ produced from the single schedule sch by allowing the e-command c to take all real values between $MinTime(F(c))$ and $MaxTime(F(c))$ (the execution times of the other commands are not changed). Let function

$$Vary(sch, c, e-Length(c), c_1, start)$$

describe the variation of the starting time of the command c_1 in $SetA(sch, c)$ as a function of $e-Length(c)$. Similarly, let

$$Vary(sch, c, e-Length(c), c_1, end)$$

represent the variation of the ending time. Note that if c_1 is not equal to c then

$$Vary(sch, c, e-Length(c), c_1, end) = Vary(sch, c, e-Length(c), c_1, start) \oplus e-Length(c_1)$$

where \oplus is pointwise addition.

Assume that c_1 immediately follows c if they are placed consecutively on the same processor with c_1 after c . If c_1 immediately follows c and c_1 is not a receive command then

$$Vary(sch, c, e-Length(c), c_1, start) = Vary(sch, c, e-Length(c), c, end)$$

If instead c_1 immediately follows c_1 and is a receive command then let c_2 be the matching send command for c_1 . Then

$$Vary(sch, c, e-Length(c), c_1, start) = Max(Vary(sch, c, e-Length(c), c, end), T_t \oplus Vary(sch, c, e-Length(c), c_2, end))$$

(where Max is the pointwise maximum).

Example: Consider a simple program with two processes, each consisting of two commands.: $P_1 :: C1; C2, P_2 :: C3; C4$; where $C1$ is a send command with $C4$ as its matching

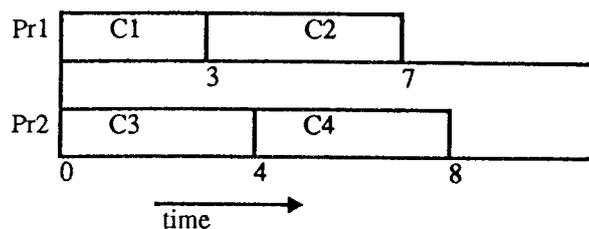


Figure 1: Schedule sch .

receive command. Commands $C2$ and $C3$ are sequential commands. Let $MinTime$ and $MaxTime$ for command $C1$ be 2 and 4 respectively. For the remaining commands let their $MinTime$ equal 4, and $MaxTime = MinTime + 2$. Consider a 2 processor implementation where $T_i = 1$.

Let a particular schedule, sch for this program be as in Fig. 1. (For simplicity, the processes are not cyclic - or it could be assumed that we are considering a truncated schedule.) Since each command is executed once only, we will use C_i to mean the command as well as the e-command.

The set $SetA(sch, C1)$ consists of all the schedules generated from sch by varying the execution time taken by command $C1$. Now consider how the start times of the various commands vary as a function of the execution length of $C1$.

The dependency relationship implies that the start time of $C2$ is the same as the end time of $C1$. Hence $Vary(sch, C1, e-Length(C1), C2, start)$ is a diagonal line (and if the same time unit scales are used for the X and Y axis, then the diagonal line will be a 45° line).

Since $C3$ does not depend on anything, $Vary(sch, C1, e-Length(C1), C3, start)$ is a vertical line.

$C4$ depends on $C1$ and $C3$. Since T_i is 1, it follows that its communication dependency requires that its start time be $e-Length(C1) + 1$ (or later). Its sequential dependency requires that its start time be 4 (or later). Hence, when the execution time taken by $C1$ is between 2 and 3, the start time of $C4$ is 4; but when the execution time of $C1$ is between 3 and 4 then the start time of $C4$ is $e-Length(C1) + 1$. Hence $Vary(sch, C1, e-Length(C1), C4, start)$ is a vertical line topped by a diagonal line.

Theorem 2 (Time Variation): *For any schedule sch and e-commands c and c_1 , $Vary(sch, c, e-Length(c), c_1, start)$ and $Vary(sch, c, e-Length(c), c_1, end)$ are of one of forms in Fig. 2. (where the X and Y axis use same time unit scale).*

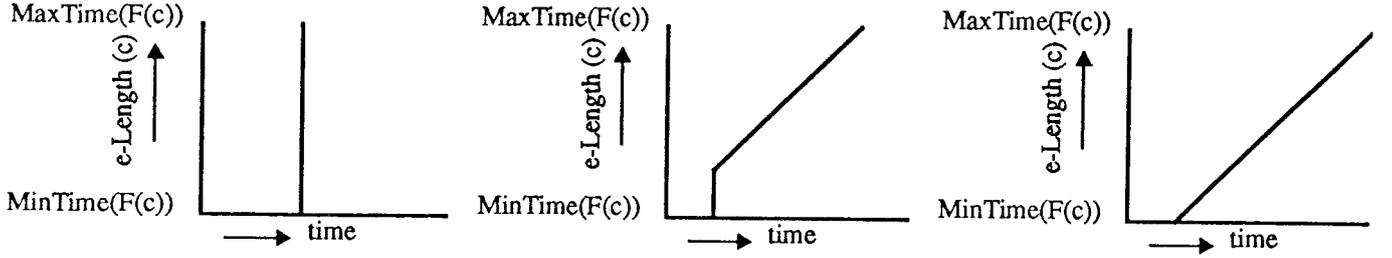


Figure 2: Structure of $Vary(sch, c, e-Length(c), c_1, SF)$

Proof: Since the X and Y axis use the same time unit scales, the diagonal lines will be at 45° .

Let $F1 = Vary(sch, c, e-Length(c), c_1, start)$ and $F2 = Vary(sch, c, e-Length(c), c_1, end)$. Proof by induction.

Base Case. Here we show how the starting and ending times of the commands with the earliest execution start times vary as a function of $e-Length(c)$. Consider a command c_1 with the earliest execution starting time. The starting time of c_1 cannot vary as a function of $e-Length(c)$; so $F1$ is a vertical line. If c_1 is not c then $F2$ is a vertical line; otherwise $F2$ by definition is a 45° line.

Induction step. Consider an arbitrary command c_1 . Let each e-command that starts before c_1 satisfy the induction hypothesis.

1. $c_1 = c$. Then $Vary(sch, c, e-Length(c), c, start)$ is a vertical line and $Vary(sch, c, e-Length(c), c, end)$ is by definition a 45° line.
2. $c_1 \neq c$. If c_1 is the first e-command executed on that processor then it either has the earliest execution starting time, which has been dealt with in the Base Case, or is a receive command. In the second case, let cb be the matching send command for c_1 . Then, $F1 = Vary(sch, c, e-Length(c), cb, end) \oplus T_i$ and since cb starts before c_1 , $F1$ is one of the specified forms.

Otherwise c_1 has an immediate predecessor - say ca .

If c_1 is not a receive command then $F1 = Vary(sch, c, e-Length(c), ca, end)$ and by the induction hypothesis it follows that $F1$ is one of the forms specified.

Otherwise

$$F1 = \text{Max}(Vary(sch, c, e-Length(c), ca, end), \\ Vary(sch, c, e-Length(c), cb, end) \oplus T_i)$$

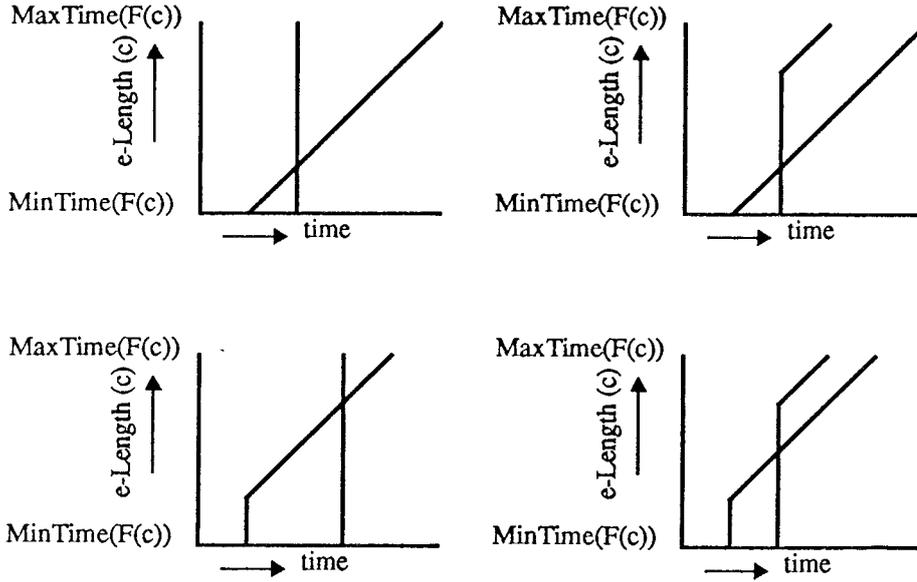


Figure 3: Combining Two Forms.

where cb is the matching send command for c_1 . Then it must be shown that if both $F3$ and $F4$ are of one of the forms specified, then $Max(F3, F4)$ is also in a specified form. If $F3$ and $F4$ do not intersect then this follows immediately. If $F3$ and $F4$ do intersect, there are four cases for $F3$ and $F4$, as shown in Fig. 3.

In each of the cases in Fig. 3, $Max(F3, F4)$ is of one of the specified forms. Hence $F1$ is of one of the specified forms.

Finally, since $F2 = F1 \oplus e\text{-Length}(c_1)$, $F2$ is also one of the forms specified.

□

Theorem 3 (Extremal Point Infeasibility): *If there is an infeasible schedule in $Exe_m(P)$ then there is an infeasible schedule in $Exe_m(P)$ such that each e-command c in that schedule takes either $MinTime(F(c))$ or $MaxTime(F(c))$.*

Proof: Consider an infeasible schedule Sch in $Exe_m(P)$. Let the first unsatisfied deadline be $D(F(c_1)) = \langle Set, d, SF_1, SF_2 \rangle$ and let the violation occur between e-commands c_1 and c_2 with $F(c_2) \in Set$. We will now construct another infeasible schedule in $Exe_m(P)$ where each e-command takes its $MinTime$ or $MaxTime$.

Define $Gap = c_2/SF_2 - c_1/SF_1$. Any schedule obtained by altering the execution times of the e-commands and for which Gap remains the same or increases (i.e. ‘widens’) is infeasible. We will take an arbitrary e-command c and show that either when c takes value $MinTime(F(c))$ or $MaxTime(F(c))$ then Gap is widest. Let $F1 = Vary(sch, c, e-Length(c), c_1, SF_1)$ and $F2 = Vary(sch, c, e-Length(c), c_2, SF_2)$.

First, $F1$ and $F2$ do not intersect (since the deadline requires that c_2/SF_2 happens after c_1/SF_1). Theorem 2 specifies all the possible forms of $F1$ and $F2$, in Fig. 4 we give all possible combinations of $F1$ and $F2$ and in each case $F1$ appears before $F2$.

In all these cases, if c takes either $MinTime(F(c))$ or $MaxTime(F(c))$ appropriately, the Gap either widens or is unchanged, and hence we have constructed an infeasible schedule where each command takes either its $MinTime$ or $MaxTime$. □

5 Feasibility Checking

Theorem 4 (Infeasibility Preservation): *There is an infeasible execution in $Exe_m(P)$ iff there is an infeasible execution in $R-Exe_m(P)$.*

Proof:

(\Leftarrow) Since each execution in $R-Exe_m(P)$ is an initial prefix of some execution in $Exe_m(P)$; this direction follows directly.

(\Rightarrow) We prove this by selecting a particular infeasible execution in $Exe_m(P)$ and considering its structure.

$$Infeasible(P) = \{Sch \mid Sch \in Exe_m(P) \wedge \neg Feasible(P, Sch) \\ \wedge \forall c \in Sch, e-Length(c) \in \{MinTime(F(c)), \\ MaxTime(F(c))\} \}$$

If there is an infeasible schedule in $Exe_m(P)$ then by Theorem 3 there is an infeasible schedule in $Exe_m(P)$ where the execution time of every e-command is either its $MinTime$ or its $MaxTime$. Hence $Infeasible(P)$ is not an empty set.

Let SCH in $Infeasible(P)$ be the schedule where the number of commands executed upto the point at which the infeasibility is detected is minimal (i.e. no other schedule in $Infeasible(P)$ has fewer commands executed upto the point at which the infeasibility is detected).

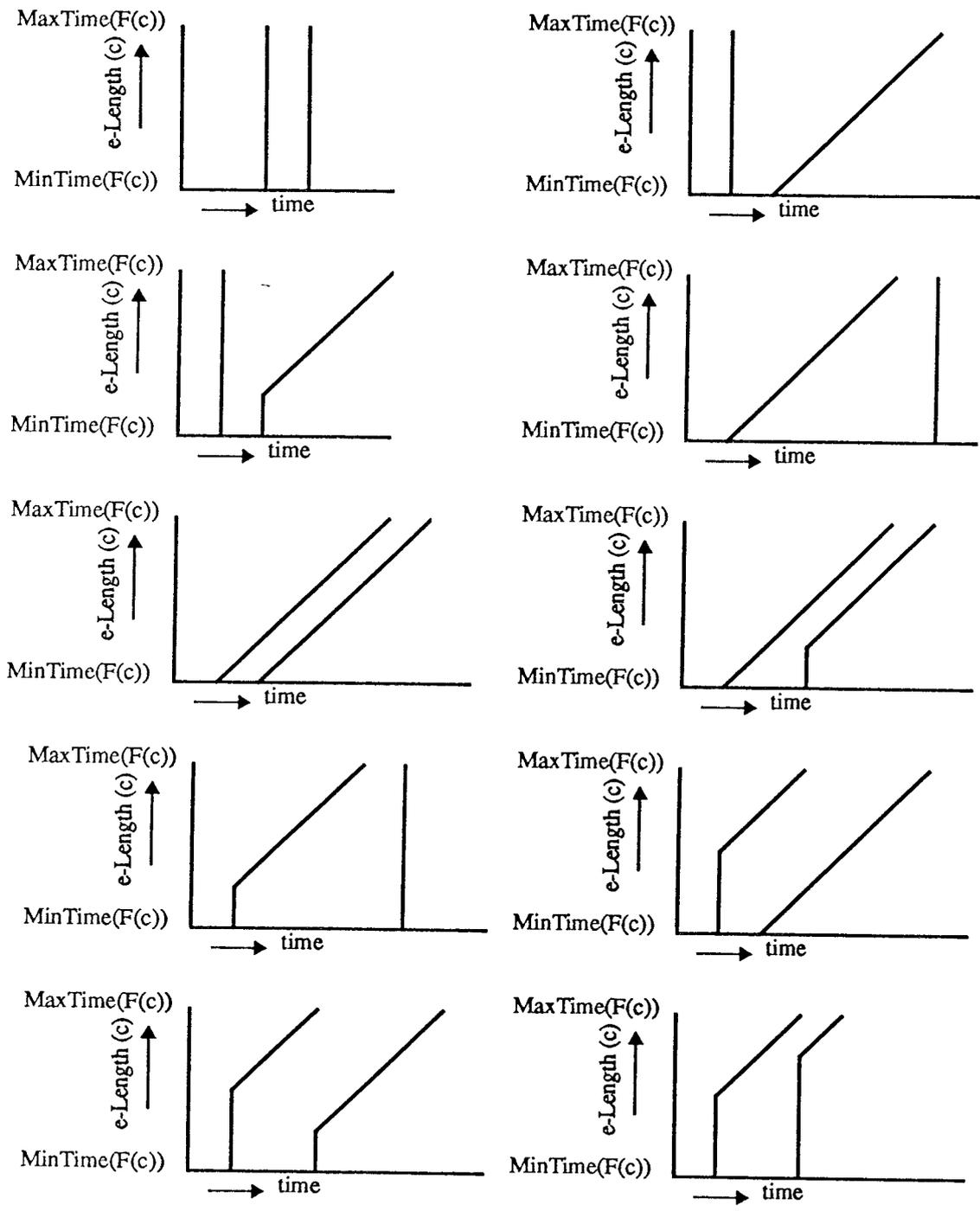


Figure 4: $F1$ and $F2$ Relationship.

Consider the structure of SCH . Let the earliest missed deadline in it be for e-command c_{ij} which starts at t_1 and finishes at t_2 . (If the deadline for c_{ij} is of the form $D(F(c_{ij})) = \langle Set, d, Start, SF_2 \rangle$ then the missed deadline is detected at time $t_1 + d$ otherwise it is detected at time $t_2 + d$). We now have three possibilities for the SRP of SCH .

1. $SRP \geq t_2$. So, $TP \geq t_2 + MaxTime(F(c_{ij})) + d$. So, the missed deadline will be detected in the truncated execution.
2. $SRP = t_1$. So, $TP \geq t_1 + MaxTime(F(c_{ij})) + d$. Since $MaxTime(F(c_{ij})) \geq t_2 - t_1$, the missed deadline is detected in the truncated execution.
3. $SRP < t_1$. Since SRP is the point where a snapshot is repeated, it follows that we can construct another infeasible schedule from SCH by cutting out the portion of the execution between the two points at which the snapshots are equal. The resulting schedule has fewer commands executed by the infeasibility point, contradicting the way SCH was selected.

Note that by definition SRP cannot lie strictly between t_1 and t_2 . So, the truncated schedule corresponding to SCH is infeasible.

Now we have to show that the truncated schedule corresponding to SCH is not discarded in the process of going from $T-Exe_m(P)$ to $R-Exe_m(P)$. This follows trivially from the fact that each e-command in SCH takes either its $MinTime$ or $MaxTime$. Hence there is an infeasible schedule in $R-Exe_m(P)$. □

Theorem 5 (Feasible Scheduling): *If every schedule in $R-Exe_m(P)$ is feasible then P can be feasibly scheduled on m processors.*

Proof: We prove this by showing that if there is an infeasible schedule of P on m processors then there is an infeasible schedule in $R-Exe_m(P)$. If there is an infeasible schedule of P on m processors then there is an infeasible schedule in $Exe_m(P)$. By Theorem 4 it then follows that there is an infeasible execution in $R-Exe_m(P)$. □

A useful extension of this theorem would be to prove that every schedule in $R-Exe_m(P)$ is feasible if and only if P can be feasibly scheduled on m processors. However, that is not true in general. For instance, it may be that P can be feasibly scheduled on m processors even if there is a syntactically possible (but not semantically possible) execution of P that is not feasible.

6 Discussion and Conclusions

The best known comparable design and analysis methods for real-time systems are associated with the Esterel language [1] (and the related languages Signal and Lustre), and Statecharts [2] which are supported by the Statemate system. Ignoring the form of presentation used in the user interface (Statecharts are composed using a visual representation in the form of higraphs), both methods have several similarities: they are based on executable specifications for deterministic programs (e.g. Esterel programs are compiled into a finite state form for analysis and code generation), and they are concerned with *synchronous* systems where the time for a computation is assumed to be negligible when compared to the time for an external action.

Our method permits the analysis of a class of *asynchronous* real-time programs which are executed on systems where the time for each computation is finite and not negligible. For this class of programs, and this would include programs written in occam or Ada, we have shown how the timing properties can be determined by examining a finite prefix of an infinite execution. It is important to note that apart from restricting processes to a cyclic structure (which in any case applies to many real-time programs) there are no other major restrictions on the use of language features or on the kinds of system on which the programs may be implemented. Thus, for example, programs may be nondeterministic, they may use synchronous or asynchronous communication, and they may be implemented on single or multiple processor systems or on distributed systems.

The availability of this method makes it clear that the analysis of asynchronous real-time programs can be automated. This requires the set of limited executions of a program to be generated by a syntactic analysis of the program text, and the timing properties of these executions to be compared with the specified deadlines. Going further, the automated analysis could also consider executions of the program on systems with different numbers of processors, and under different scheduling disciplines, thus enabling the system designer to experiment with implementation strategies. A software tool to perform such analysis is under development at the University of Warwick.

References

- [1] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Lecture Notes in Computer Science 197*, pages 389–449. Springer-Verlag, Heidelberg, 1985.
- [2] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

- [3] J.P. Lehoczky L. Sha, R. Rajkumar. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
- [4] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprocessing in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.
- [5] W. Zhao, K. Ramamritham, and J.A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, 36(6):949–960, 1987.