Timing Analysis of Cyclic Concurrent Programs

C. Samuel Hsieh
Computer Science Department, Vanderbilt University
Nashville, TN 37235, U.S.A.
e-mail: samuel@vusc.vanderbilt.edu

ABSTRACT

An approach to timing analysis of cyclic concurrent programs is presented. GR₀ path-expressions are used to describe synchronization and concurrency of atomic operations in cyclic concurrent programs. The behavior of a cyclic concurrent program is represented as a partial order of atomic operations, and a technique to derive this partial order from a GR₀ program is developed. Given the execution times of the individual atomic operations of a GR₀ program and a set of timing constraints, our timing analysis technique uses the partial order to determine whether the concurrent program, when executed, will satisfy the set of timing constraints. The timing analysis technique can be completely automated.

I. Introduction

A concurrent program that interacts with physical processes (external objects) that may fail to wait for appropriate synchronization with the program is often called a real-time program. For the system - the program plus the physical processes it controls or monitors - to function properly, the program must meet the timing constraints imposed by the physical processes. Wirth [22] suggested an approach to real-time programming: first, write and verify a real-time program as an ordinary concurrent program assuming that the physical processes will wait to synchronize, and then try to establish that the timing constraints imposed by the physical processes are met. This approach separates two distinct concerns over verification of real-time programs, namely, logical correctness of programs and satisfaction of timing requirements imposed by physical processes. The latter requires techniques to establish timing properties of a concurrent program. This paper presents a technique for timing analysis of cyclic concurrent programs.

We shall view a sequential process as a sequential program that, when executed, will perform a sequence of atomic operations. A concurrent program is considered as a set of cooperating sequential processes whose executions may proceed asynchronously in parallel but may have to synchronize with one another at some points of their executions. The atomic operations are purely sequential in nature and their execution times can be determined without considering the synchronization aspect of a concurrent program. We assume that the execution times of atomic operations can be determined by other methods [e.g., 12]. When the execution times of the atomic operations of a program are given, the timing analysis technique developed in this work can be used to determine if a set of timing constraints are satisfied by the program.

II. Relation to Other work

Some researchers have proposed a number of temporal logics [e.g., 2, 19] for reasoning about timing of concurrent programs. However, temporal logic deals with the relative order of executions of atomic actions instead of the execution times of the actions, but real-time properties depend on the execution times of the actions. Lamport [13] suggested that time can be modeled by introducing the notion of a clock and then a real-time property can be expressed and proved as a safety property involving the clock variable. In a program whose actions are executed strictly in sequence, maintaining a clock variable to accumulate the elapsed time in executing the atomic actions may not be a problem, but it is unclear how (or whether) such a clock variable can be properly maintained when executions of several atomic actions overlap, perhaps partially, in time.

Recently, Jahanian and Mok [7, 8] proposed a formal logic RTL (Real-Time Logic) for specification and analysis of real-time systems. RTL formulas are used to express both the timing specification of a system and safety (timing) assertions about the system. Their work focuses on determining the consistency of a safety assertion with respect to a timing...
specification expressed in RTL, rather than on timing analysis of concurrent programs. To capture the semantics of systems, they introduced a computation model, called event-action model [7]. The event-action model has certain composition constructs similar to those of some concurrent languages and, hence, a specification in the event-action model might resemble a concurrent program. However, time is explicitly used in the model, and a specification is therefore time-dependent.

In this work, we take a complementary approach to timing analysis of concurrent programs: programs contain no explicit mention of time, and the technique presented in this paper is used to perform timing analysis for a program if the program needs to satisfy some real-time constraint. Hence, concern over logical correctness of a program is separated from that over real-time behavior of the program. Schneider suggested that new abstractions for real-time programming should build on existing abstractions for writing programs in which physical time is irrelevant, and that the new abstractions should degenerate into the existing ones when time can be ignored [20]. Our approach to timing analysis seems in concordance with this point of view.

III. GR₀ Path Programs

Since this work does not deal with methods to find execution times of individual atomic operations, we need a notation that allows us to abstract away from purely sequential operations and focus on parallel execution and synchronization in concurrent programs. The path expression notation [4, 14] reasonably meets this requirement.

The notation was first introduced by Campbell and Habermann [4] as a linguistic construct for programmers to express synchronization constraints. It was later developed more fully and given formal semantics in terms of Petri-Nets [14] and in terms of Vector Firing Sequences [21, 16]. Lauer, Shields and others have shown that path expressions yield to mathematical analysis and are useful for precise specification of the semantics of concurrent computation [3, 9, 10, 21]. The notation has been implemented in Path Pascal [5] and Distributed Path Pascal [6], and is the basis for the system specification language COSY [15, 16]. Path expressions have been applied to VLSI design [1, 17] and found useful for parallel applicative languages [11].

There are a number of variants of path expressions. We shall use the GR₀ paths* notation [3] as a formalism for describing cyclic concurrent systems. Complete descriptions of path expressions can be found in the references [14, 3]. Here we briefly sketch the syntax and semantics of GR₀ programs. A GR₀ program is a finite set of GR₀ paths. A GR₀ path defines a total ordering on activations (executions) of atomic operations, e.g., let A, B, and C be some atomic operations, the path†

\[
\text{path A ; B ; C end}
\]

specifies that an activation of A should precede B, which must precede C, which must precede the next activation of A. In short, a semicolon (:) denotes sequentialization, and the key words path...end denote repetition. A distinction should be made between an operation and an activation of an operation, which means an execution of an operation. An operation can be activated (executed) many times.

There are generally several paths in a program, and operation activations must satisfy all of the ordering constraints specified by all of the paths of a program. For example, the program

\[
\text{path A ; B end path C ; B end}
\]

specifies that every activation of B must be preceded by an activation of A and an activation of C. Together, they define the constraint illustrated in Fig. 1. We observe that A and C in Fig. 1 can be activated concurrently since no ordering constraint between them has been specified.

\[
\text{Fig.1 ordering constraint}
\]

A path program defines a partial ordering relation < on activations of atomic operations. Two activations X, Y of atomic operations are said to be concurrent if neither X < Y nor Y < X. Activations that are concurrent can be executed in parallel or in any arbitrary sequence without violating the partial ordering constraint imposed by the path program. Timing properties of a path program generally depend on how concurrency is implemented. In this work, we consider the case where an operation activation begins to execute as soon as the ordering constraint defined by the program does not prevent it from execution. Hence, activations that are concurrent are executed truly in parallel. For instance, in Fig. 1 the nth activation of A and the nth activation of C are concurrent (for all n > 0) and, hence, can be executed truly in parallel or in an arbitrary sequence. In this work we consider the case that they are executed in parallel.

In addition, this work will assume that a computer system is dedicated. That is, a concurrent program does not share its underlying computer system with other programs. Generally, it is impossible to establish hard real-time properties of a program whose progress in execution may be arbitrarily interrupted by other programs that also execute on the same computer system.

IV. Timing Constraints

It is necessary to define several commonly-used terms in order to use them in the context of this work.

† Unless otherwise specified, a path means a GR₀ path, and a program means a GR₀ program in this paper.
An event is an identifier used to mark certain points in time. An event occurrence marks a single point in time. An event is periodic if every occurrence of the event is temporally followed by an occurrence of the same event within a bounded amount of time.

At the level of abstraction adopted in this work, each operation is associated with two events of special interest: one marks the start of an operation, and the other marks its end. Accordingly, each activation of an operation is associated with two event occurrences: an occurrence of its start event and an occurrence of its end event.

The behavior of a path program is a temporal ordering on all of the activations of the operations in the program when the program is executed. Such a temporal ordering is generally a partial ordering relation.

A sporadic timing constraint is a specification that an event occurrence $e_1$ is temporally followed by another event occurrence $e_2$ within a certain interval of time (or may be occurrences of the same event). A periodic timing constraint is a specification that every occurrence of a periodic event $e_1$ is temporally followed by an occurrence of a periodic event $e_2$ within a certain interval of time (or may be the same event). Sporadic and periodic timing constraints are dictated by the external objects (physical processes) that a program controls or monitors. We believe that a wide variety of useful timing requirements of physical processes can be specified by these two basic types of timing constraints.

When the execution times of individual atomic operations are given, the analysis techniques described in this paper can be used to determine whether the behavior of a program satisfies a set of sporadic and/or periodic constraints.

V. Program Behavior

In the absence of deadlocks, the sequence of operations in an individual path is activated cyclically. It is interesting to ask whether, in the absence of deadlocks, the behavior of a path program is also cyclic. This, indeed, is the case. That is, the behavior is an infinite repetition of some basic cycle. Such a cycle is generally not a sequence, but a partial order, of operation activations. This section describes how to find such a partial-order cycle for a program.

Let the collection of paths $P_1, P_2, ..., P_n$ be a program. Let $OP(P_i)$ denote the set of operation names used in the path $P_i$. We can draw a graph, called intersection graph, to depict how the operation names are shared by the paths. The vertices of the graph are the paths $P_1, P_2, ..., P_n$, and an edge $(P_i, P_j)$ exists whenever $OP(P_i) \cap OP(P_j) \neq \emptyset$ (i.e., $\neq 0$). Each edge $(P_i, P_j)$ is labeled with operation names in $OP(P_i) \cap OP(P_j)$. As an example, a path program and its intersection graph is shown in Fig. 2.

Since every path starts from its beginning when execution of a program begins, if the behavior of the program is cyclic, then, upon completion of a cycle, every path must, again, begin from its beginning. Let $P_i(X)$ denote the number of times an operation name $X$ occurs in a $GR_0$-path $P_i$. Suppose that, in one cycle, $P_i$ is repeated $r_i$ times, $P_j$ is repeated $r_j$ times. If an operation name $X$ appears in both $P_i$ and $P_j$, then $r_i*P_i(X) = r_j*P_j(X)$, because every activation of $X$ in the basic cycle requires occurrence of $X$ in $P_i$ and in $P_j$. All such $r_i (1 \leq i \leq n)$ can be found by the following steps:

1. Construct the intersection graph for the program $P_1, P_2, ..., P_n$.
2. Choose a spanning tree of the intersection graph, and label each edge $(P_i, P_j)$ of the spanning tree with an operation name $X \in OP(P_i) \cap OP(P_j)$.
3. For each edge of the spanning tree $(P_i, P_j)$ labeled $X$, write down an equation $r_i*P_i(X) = r_j*P_j(X)$.
4. Find the smallest positive integer solution to the system of linear equations obtained in step 3.

$$r_1 = r_2 \quad r_2 = 2r_3 \quad r_4 = 2r_3$$

In step 3 we obtain $n$ equations in $n$ unknowns $r_1, r_2, ..., r_n$. So there can be many linearly dependent solution vectors for the system of equations. We should pick, for our purpose, the smallest positive integer solution. This is always possible and can be done easily. As an example, from the program and the intersection graph in Fig. 2, we choose a spanning tree and write down the linear equations (Fig. 3). Obviously, the solution we need is $r_1 = 2, r_2 = 2, r_3 = 1, r_4 = 2$. The following step is a check for possible deadlocks:

5. For each edge and each label in the intersection graph that are not included in the spanning tree, write down an equation as in step 3. If the solution obtained in step 4 does not satisfy every equation obtained in this step, then the given program will deadlock.

Generally, it is possible to obtain several different spanning trees from an intersection graph. Step 5 says that if a program is free of deadlocks, then the values of $r_1, r_2, ..., r_n$ do not depend on which spanning tree is chosen in step 2. This is so because if the solutions for $r_1, r_2, ..., r_n$ obtained from two different spanning trees contradict each other, then the program...
is unable to complete even one cycle (no matter which of these two contradictory solutions is used to define a cycle). In our example, the edge \((P_1, P_4)\) labeled C and the label D on the edge \((P_2, P_3)\) are not included in the spanning tree in Fig. 3, hence we write

\[ r_1 = r_4 \quad r_2 = 2r_3 \]

Our solution for \(r_1 \ldots r_4\) satisfies these two equations. However, this still can not guarantee absence of deadlocks, as will become clear later.

Now we are ready to construct the partial order behavior for one cycle. First, the total order on operation activations in individual paths are represented graphically,

6. For each \(P_i\), represent the total order of \(P_i\) repeated \(r_i\) times as an acyclic directed graph \(G_i = (V_i, E_i)\) in which a vertex \(X_\ell \in V_i\) represents the \(\ell\)th activation of the operation \(X\) in the total order, and an edge \((A_j, B_k) \in E_i\) exists if the \(j\)th activation of the operation \(A\) is the immediate predecessor of the \(k\)th activation of \(B\) in the total order.

The graphs \(G_1, G_2, G_3\) and \(G_4\) for \(P_1 \ldots P_4\) of the example are shown in Fig. 4. Step 7 constructs the partial order behavior for one cycle,

7. Merge all of the graphs \(G_1 \ldots G_n\) into a single directed graph \(G = (V, E)\) such that \(V\) is the union of \(V_1 \ldots V_n\), and \(E\) is the union of \(E_1 \ldots E_n\).

Basically, step 7 merges the graphs by identifying vertices that represent the same operation activation in different paths. The following step is not necessary, but it simplifies the graph \(G\).

7a. Delete an edge \((X, Y) \in E\) in \(G\) if there is a directed path from the vertex \(X\) to the vertex \(Y\) and the edge \((X, Y)\) is not included in the path. Repeat this step until no edge can be deleted this way.

Step 7a deletes edges that are redundant for representing the precedence relation on operation activations. It has no effect on the precedence relation. In fact, steps 6 - 7a can be used to construct the partial order behavior for \(n\) (any positive integer) cycles by repeating each \(P_i\) for \(n \times r_i\) times in step 6.

The graph in Fig. 5 shows the partial order behavior of the example for two consecutive cycles. The following step determines if a program is deadlock-free.

8. If the graph \(G\) obtained in step 7 (7a) is acyclic, then the corresponding path program is deadlock-free; otherwise, the program is deadlockable.

Since the graph in Fig. 5 is acyclic, our example program is free from deadlocks. Best [3] proposed a technique for determining whether a \(GR_0\) program can deadlock. His technique relies on interleaving atomic operations. In contrast, our approach does not use interleaving and, in addition, produces the partial ordering needed for timing analysis.

VI. Timing Analysis

By our assumption on implementation of concurrency (section III), we may regard the behavior of a deadlock-free \(GR_0\) program as an infinite activity network. However, if there is a point of convergence in the partial-order cycle, no more than two cycles of the program behavior need be analyzed to determine whether certain timing constraints are satisfied by the entire program behavior. Hence, well-known and efficient algorithms for critical-path analysis of activity networks can be applied. When the behavior of a program is represented as a directed graph, a point of convergence is an operation activation that appears in every directed path leading from an operation activation of a cycle of the behavior to the corresponding operation activation of the next cycle.
As an example, we consider the graph in Fig. 5. Any directed path from an operation activation of the first cycle to the corresponding operation activation of the second cycle must pass the operation activation $E_1$ (of either the first or the second cycle). Hence, $E_1$ is a point of convergence. The time it takes for the program to progress from the start of a point of convergence of a cycle to the start of the corresponding point of convergence of the next cycle is a constant, called the cycle time. Given the following execution times of the operations:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

a critical-path analysis of the portion of the graph in Fig. 5 from $E_1$ of the first cycle to $E_1$ of the second cycle produces

$$
\begin{array}{cccccc}
A_1 & B_1 & C_1 & D_1 & E_1 & F_1 \\
8 & 10 & 13 & 2 & 0 & 2 \\
A_2 & B_2 & C_2 & D_2 & E_2 & F_2 \\
3 & 4 & 5 & 6 & 8 & 10
\end{array}
$$

The table shows the start time of each operation activation, relative to the time at which $E_1$ begins in the first cycle. The end time of an operation activation is its start time plus its execution time. For example, the end times of $C_1$ and $B_1$ (both of the second cycle) are, respectively, 13+3=16 and 10+2=12. The cycle time of this example, i.e., the start time of $E_1$ of the second cycle relative to that of $E_1$ of the first cycle, is 16, the larger of the end times of $C_1$ and $B_1$. All questions about timing of the program after the first occurrence of $E_1$ can be answered from this table. For example, consider the periodical timing constraints: the interval of time between two consecutive occurrences of the start event of the periodical operation $F$ does not exceed 8 time units. From the table, we see that $F_2$ is activated within 10 - 2 = 8 time units after the start of $F_1$, and that $F_1$ of the next cycle will be started within 16 + 2 - 10 = 8 time units after the start of $F_2$ of the current cycle. Hence, this timing constraint is satisfied. Since the cycle time is a constant, timing constraints that relate events in different cycles can be easily answered.

Since some operations of the initial cycle of a program behavior may precede the first occurrence of the point of convergence, the portion of the program behavior preceding the first occurrence of the point of convergence need to be analyzed separately. In the example, the timings of $A_1$, $B_1$ and $C_1$ of the initial cycle of the program behavior are not covered by the above table and, hence, should be analyzed separately. Such an analysis is needed only for a part of the very first cycle of the program behavior and is quite easy.

When there are no points of convergence in a cycle, then timing analysis will be more complicated. Fortunately, it can be shown that, for every program that does not deadlock, a constant cycle time exists after some bounded number of initial cycles, even if no points of convergence exist in the program’s behavior.

A formal proof of this result is too lengthy to be included here. We sketch the main points of the proof below. Let $G_1, G_2, \ldots, G_n$ be the graphs (for a single cycle) obtained in step 6 (section V) for a program and $G$ be the directed graph (for a single cycle) constructed in step 7a for the program. Define a cycle-path $CP$ of the graph $G$ to be a directed path in $G$ such that there is a graph $G_i$ (constructed in step 6) which begins with the first vertex of $CP$ and ends with the last vertex of $CP$. In each cycle of the program’s behavior, all operation activations (vertices) along a cycle-path $CP$ are executed once in the total-order sequence defined by $CP$, and the last vertex of $CP$ in the $n$th cycle precedes the first vertex of $CP$ in the $(n+1)$th cycle. Define the length of a directed path in $G$ to be the sum of the execution times of all operation invocations along the directed path. Call the longest cycle-path(s) critical cycle-path(s) of $G$, and the time at which the first operation activation of a cycle-path begins execution in a cycle the start time of the cycle-path in that cycle.

Consider a program with several cycle-paths. Different cycle-paths may be started at different times in the first cycle. Since a cycle-path $CP$ must synchronize with all other cycle-paths either directly (through shared vertices) or indirectly (through some cycle-paths that have synchronized with $CP$), any influence that the initial start time of a cycle-path may have on the future start times of other cycle-paths will be realized in no more than $d$ cycles, $d$ being the diameter of the intersection graph of a program. Then, in each subsequent cycle, the start time of a critical cycle-path will be incremented by a larger amount than that of a non-critical cycle-path will be, if there is no synchronization between the critical cycle-path and the non-critical one. However, any cycle-path synchronizes directly or indirectly with all other cycle-paths in a program. Hence, after a bounded number of cycles, the start times of critical cycle-paths will act as the pacemakers for all cycle-paths and, then, the start time of every cycle-path, critical or non-critical, will be incremented in each subsequent cycle by the length of the critical cycle-paths. Hence, after a bounded number of initial cycles, a constant cycle time exists: the length of a critical cycle-path.

This general characteristic of all $GR_0$ programs is quite significant. It assures us that verification of any timing constraint can be algorithmically carried out (automated on a computer), because only a bounded number of cycles need be analyzed in order to determine whether a given timing constraint, periodic or sporadic, will be violated by any portion of the entire behavior of a program.

*We consider only those programs whose intersection graphs are connected.*
We give an example. Fig.6 shows a program and its behavior for the first three cycles. Suppose that initially A starts at time 200, B at time 100, and C at time 0, and the execution times of the operations are

\[
\begin{array}{cccccccccccc}
A & B & C & D & E & F & G \\
5 & 3 & 2 & 10 & 5 & 15 & 18 \\
\end{array}
\]

Fig.7 shows the start times of A, B, and C in the first 9 cycles. By the third cycle, the influence of the late start time of A has been propagated to all cycle-paths. By the 8th cycle, the critical cycle-path CG, whose length is 20, has been able to synchronize with all other cycle-paths and, in every subsequent cycle, the start times of A, B, and C will be incremented by the constant 20.

VII. Summary and Discussion

We have presented a technique for timing analysis of \( GR_0 \) programs. The main contributions of this work are 1) the timing analysis technique, and 2) the conclusion that timing analysis of the entire class of \( GR_0 \) programs can be carried out algorithmically and, hence, can be fully automated.

The obvious next step is to extend the results of this work to timing analysis of a wider class of path programs, say, the class of programs which may use a comma (,) to denote arbitrary choice\(^1\). With the use of the arbitrary choice operator, there can be many possible behaviors for a given program and, furthermore, even in the absence of deadlocks some of the possible behaviors of a program may not be cyclic. For example, a behavior of the following simple program, which uses the arbitrary choice operator,

\[
\text{path A, B end}
\]

\[
\text{path A; D end, path B; F end}
\]

\[
\text{path A; E end, path B; E end, path C; F end}
\]

\[
\text{path B; F end, path C; G end}
\]

\[
\text{A} \rightarrow \text{D} \rightarrow \text{A} \\
\text{B} \rightarrow \text{E} \rightarrow \text{B} \\
\text{C} \rightarrow \text{F} \rightarrow \text{C} \\
\text{G} \rightarrow \text{C} \rightarrow \text{G}
\]

\( \text{First Cycle} \rightarrow \text{Second Cycle} \rightarrow \text{Third Cycle} \)

\[
\begin{array}{cccccccccccc}
\text{Cycle} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\text{Start of A} & 200 & 215 & 230 & 245 & 260 & 275 & 293 & 313 & 333 \\
\text{Start of B} & 100 & 210 & 228 & 246 & 265 & 285 & 305 & 325 & 345 \\
\text{Start of C} & 0 & 118 & 228 & 248 & 268 & 288 & 308 & 328 & 348 \\
\end{array}
\]

Fig.6: a program with no points of convergence

Fig.7: timing analysis of the program in Fig. 6

\[\text{can be viewed as a string over the alphabet } \{A, B\} \]. There are infinitely many such strings and, moreover, since it is always possible to construct a cube-free* string of arbitrary length from an alphabet with cardinality \( \geq 2 \) (e.g., see [18]), it can be concluded that some possible behaviors of the program are non-cyclic. Development of timing analysis techniques for path programs that use the arbitrary choice operator is a non-trivial task.

References


\( t \) Often referred to as \( GR \) programs.

* A string is cube-free if it contains no substring of the form \( x^3 \) where \( x \) is a nonempty string.


