AUTOMATED SYMBOLIC TIMING ANALYSIS FOR DISTRIBUTED SYSTEMS

Scott D. Stoller and Leena Unnikrishnan
Computer Science Dept., Indiana University, Bloomington, IN 47405 USA
{stoller,lunnikri}@cs.indiana.edu

Abstract

A *timing property* of a distributed system is an assertion about the time intervals between events in executions of that system. There are three traditional approaches to determining timing properties of distributed systems: measurement, simulation, and analysis. Results from analysis can be symbolic and therefore much more general than results from measurement and simulation. For example, analysis can express the end-to-end delay of an atomic commitment protocol symbolically as a function of message delay and other system parameters. However, manual analysis of distributed algorithms is often tedious and error-prone. We have developed and implemented a state-exploration-based analysis that can automatically determine a large class of timing properties of distributed systems.

1 Introduction

A *timing property* of a distributed system is an assertion about the time intervals between events in executions of that system. A timing property might characterize the probability distribution of time intervals between events, or the minimum, maximum, or mean of that distribution. For example, for a distributed database system, typical timing properties of interest are the mean and maximum times from when a transaction starts until it completes. There are three traditional approaches to determining such properties: measurement, simulation, and analysis. Measurement can provide very accurate information, but the results are specific to a particular compiler, operating system, CPU, network, *etc.* Similarly, the results of a simulation reflect specific values of or distributions for the time needed for low-level operations. Analysis can provide much more general results by expressing timing properties symbolically as functions of *configuration parameters* (*e.g.*, number of processes, maximum number of failures, and network diameter), and *timing parameters*, which characterize the time needed for low-level operations (*e.g.*, minimum and maximum network delay, hard-disk access time, and CPU time spent in protocol stack). For example, De Prisco, Lampson, and Lynch's analysis shows that the time needed for the PAXOS algorithm (based on Lamport's part-time parliament [Lam89]) to reach consensus after failures cease is at most $24\ell + 10n\ell + 13d$, where n is the number of processes, and ℓ and d are bounds on local processing time and message delay, respectively. Determining and verifying such formulas by hand is tedious and error-prone, due to non-determinism from concurrency and failures.This non-determinism also makes traditional methods for analysis of the time complexity of sequential algorithms difficult to apply.

This paper describes a method for automatically determining a large class of timing properties of distributed systems. Timing parameters are handled symbolically (or numerically, if desired) throughout the analysis. Configuration parameters must be instantiated with particular values. One can often obtain a good idea of the dependence of timing properties on configuration parameters by repeating the analysis for various values and plotting the results. Extending the analysis to handle configuration parameters symbolically, perhaps along the lines of [KM95], is an area for future work.

We envision two main uses for this analysis. One is to determine "end-to-end" timing properties of distributed algorithms, such as the property of PAXOS mentioned above. The other is to facilitate the development of such algorithms by helping determine time-outs. For example, the PAXOS algorithm, like many fault-tolerant algorithms, uses a timer that should expire just after a certain message would have arrived if failures did not interfere; specifically, the time-out in the PAXOS algorithm is $7\ell + 4n\ell + 4d$. For particular values of n, our timing analysis can determine symbolically the time interval in which the message can arrive; the upper limit of this time interval provides the correct time-out.[1]

Our analysis constructs a graph: each node corresponds to a (global) state of the system, and each edge corresponds to an event (transition). The graph is constructed by starting with the initial state and repeatedly generating new states by exploring events. Each event is tagged with symbolic times that characterize when it can occur. A *symbolic time* is either an expression built from timing parameters and arithmetic operators or the symbol ∞ (infinity). Symbolic times are used to determine the order in which events can occur and to compute bounds on the minimum and maximum time intervals between specified classes of events.

The use of (symbolic) time intervals (instead of numerical times, as in most simulators) has many ramifications in the analysis. One is the inadequacy of the straightforward approach in which each event is tagged with one symbolic time interval, representing the range of times that it can occur, measured from the start of the execution. As we show in Section 4, it is often necessary to tag a pending event e with multiple symbolic time intervals, each representing the range of times that e can occur, measured from a specified point during the execution. We call these points *anchors*.

Section 2 describes our system model. Section 3 describes a simplified symbolic timing analysis, called the one-anchor analysis. Sections 4 and 5 motivate and describe, respectively, the multi-anchor analysis. Section 6 describes a modular way of introducing failures. The analysis has been implemented in Java in a prototype tool called TADA (Timing Analyzer for Distributed Algorithms). Section 7 describes the application of TADA to a new message-efficient algorithm for uniform timed reliable broadcast [MS98]. To the best of our knowledge, this is the first automated analysis of symbolic timing properties of that or any similar distributed algorithm.

Future work includes: (1) optimizing the analysis by developing a suitable partial-order optimization [God96, PPH97] that avoids exploring all interleavings of concurrent pending events that "commute" (experience with analysis of untimed systems shows that this reduces the analysis cost by an exponential factor for many systems), (2) optimizing the analysis by exploiting "monotonicity" to justify exploring some pending events only at their earliest or latest possible occurrence time; (3) integrating the analysis with a symbolic timing analysis for sequential programs written in a conventional programming language (*cf.* Section 3.1); (4) empirically checking the accuracy of timing properties calculated by substituting measured values for timing parameters into formulas obtained from the analysis; if necessary, the accuracy can be increased by introducing additional timing parameters, possibly like those in the LogP model of parallel computation [CKP$^+$96]; (5) applying the analysis to agreement algorithms like those in [Lyn96, Part III] or [PLL97], group communication protocols like those in [CZ85, RVR93, MR97], *etc.*

[1] How can the algorithm be analyzed before it is completely designed? The arrival time of the message in question does not depend on this time-out. Thus, during this analysis, this timer can be removed from the algorithm or set to a very large value.

The work most closely related to ours is work on timed automata [AD94]. To the best of our knowledge, of the numerous papers on timed automata, only [AHV93] treats timing parameters symbolically. The analysis method described there is (apparently) unimplemented. More importantly, it represents passage of time by explicitly subtracting the elapsed time from every clock on every transition. Such "global updates" make partial-order optimizations difficult, although they do allow more accurate analysis of some systems (such as UTRB2 [BT93a, BT93b]), for which our method gives approximate results. Alur *et al.*'s work on analysis of hybrid systems can also treat some system parameters symbolically [ACH$^+$95]. Generalizing from timed systems to hybrid systems complicates the analysis and makes development of partial-order optimizations difficult. We are currently developing partial-order optimizations for timed systems, based on ideas in [God96, BJLY98], and plan to make that analysis parametric.

2 System Model

We model a distributed system as a collection of processes that communicate by message-passing over reliable FIFO channels with unbounded capacity and bounded delay. It is easy to modify the analysis to accommodate unreliable, unordered, or finite-capacity channels. By convention, the symbols δ_1 and δ_2 are lower and upper bounds, respectively, on message delay. Specifically, message delay is at least δ_1 and is less than δ_2. Note that this implies $\delta_1 < \delta_2$.

We use a reactive model of processes: a process is inactive except when reacting to an input event. There are three kinds of input events: message reception (with type MSG), timer expiration (TMR), and signal reception (SIG). Each process can use multiple timers, so when a timer expires, the input event contains a timer identifier that indicates which timer expired. Signals are like messages except that signals are transmitted instantaneously. Signals are useful for modeling communication between processes running on the same machine; in this sense they are like UNIX signals. For example, communication between a failure-detector process and its clients, which must be on the same machine as the failure detector, would be modeled by signals rather than messages. Signals are also useful for enforcing global invariants; an example of this appears in Section 6.

In reaction to an input event, a process may change its local state and produce a sequence of outputs. There are four kinds of outputs: sending a message (MSG), setting a timer (SETTMR), cancelling a timer (DELTMR), and sending a signal (SIG). Processes may be non-deterministic. We assume that a process's reaction to an input event is not interrupted by processing of other input events. However, we do not assume that the sequence of outputs is produced instantaneously.

We assume all timers run at the same rate as real time. It is straightforward to accommodate timers that run too fast or too slow; bounds on their rates could be numerical or symbolic.

3 One-Anchor Analysis

This section describes a simplified analysis, called the *one-anchor analysis.* The following system serves as a running example. The system contains two processes, p_0 and p_1. At the start of the execution, p_0 sends a message m_0 to p_1 and sets a timer to $2\delta_2$. On receiving m_0, p_1 sends a reply m_1 to p_0. On receiving m_1, p_0 sends an acknowledgment m_2 to p_1 and cancels the timer, if it has not yet expired. If the timer expires, then p_0 retransmits m_0. The first timing property that we consider is the minimum and maximum time from when p_0 sends m_0 until p_0 receives m_1.

3.1 Processes

We model a process as an automaton, defined by: a set of (local) states, a transition function, an initial state, and an optional initial output. The arguments of the transition function are the current local state of the process and an input event. Processes are non-deterministic, so the transition function returns a set of possible reactions, each represented by a pair $\langle s, outs \rangle$, where s is a state of the process and $outs = \langle\!\langle o_0, o_1, \ldots, o_n \rangle\!\rangle$ is a sequence of outputs (we use double angle brackets to denote sequences). Each output o_i is tagged with a time interval called its *offset*, which bounds the time that elapses from when o_{i-1} is produced until o_i is produced; as a special case, the offset of o_0 is measured from when the input event occurs. The optional initial output, if present, must be of type SETTMR. Thus, each process can set an initial timer, and when that timer expires, the process can send messages, *etc.*

Let r_0 be the transition function for p_0 in the running example. The initial output of p_0 sets a timer to expire after a delay of zero; thus, that timer expires immediately at the start of the execution. Let s_0 be the initial state of p_0, and let $itmr_0$ be an input event representing the expiration of p_0's initial timer. When applied to the arguments s_0 and $itmr_0$, r_0 returns $\{\langle s_0', \langle\!\langle omsg_0, otmr_0 \rangle\!\rangle \rangle\}$, where s_0' is a state of p_0, and outputs $omsg_0$ and $otmr_0$ correspond to sending m_0 and setting a timer to $2\delta_2$, respectively.

Using automata simplifies the implementation of the analysis, but writing distributed algorithms as automata is often inconvenient. Programs in a conventional programming language augmented with Send and Receive statements could be used instead, provided the duration of local processing is expressed symbolically (as in the offsets in automata outputs). One approach is to have the user annotate program segments with symbolic times. A more automatic approach is to associate timing parameters with each program construct and use, *e.g.*, Shaw's timing analysis to symbolically determine execution times for sequential code fragments [Sha89]. The analysis in [Sha89] is suitable for many real-time programs but does not deal with message passing or (more importantly) the high degree of non-determinism resulting from failures.

3.2 Timing Properties

We consider timing properties that characterize the minimum and maximum time between classes of events. A timing property is expressed as a start condition and a stop condition, each being a predicate on message sending or receiving events, *e.g.*, "p_1 receiving a message containing *dlvr_mcast* from any process". A *start event* or *stop event* is an event satisfying the start or stop condition, respectively. In the running example, the start condition is "p_0 sends m_0 to p_1", and the end condition is "p_0 receives m_1 from p_1".

Our analysis provides symbolic bounds on the minimum and maximum time between a start event and a causally subsequent stop event. The following restricted notion of causality simplifies the calculations in Section 5 and is sufficient for many examples, so we adopt it in this paper. However, using Lamport's causality relation ("happened before") [Lam78] or temporal ordering (instead of causality) is not difficult. Our causality relation $\rightarrow$ for a computation c is the smallest transitive relation on pending events and outputs in c such that: (1) for each pending event pe and each output o produced by pe, $pe \rightarrow o$ and, if there is a pending event pe' corresponding to o (*e.g.*, o is not of type SIG), then $pe \rightarrow pe'$; (2) for consecutive outputs o and o' in a sequence of outputs produced by some pending event, $o \rightarrow o'$. Note that $\rightarrow$ is a subset of Lamport's causality relation. Specifically, $\rightarrow$ ignores local orderings between processing of different inputs.

Semantics. Let σ be an instantiation of the timing parameters of the system, with $\sigma(\delta_1) < \sigma(\delta_2)$; for example, $\sigma(\delta_1) = 1.5$ msec, *etc.* Let $\hat{t}_{min}$ and $\hat{t}_{max}$ be the least and greatest times, respectively, between the first occurrence of a start event and the first causally-subsequent occurrence of an end event in any possible execution of the system with timing parameter values given by σ. If there is no execution in which a start event is causally followed by a stop event, then $\hat{t}_{min}$ is infinity; if there is some execution not containing a start event, or in which the first occurrence of a start event is not causally followed by an end event, then $\hat{t}_{max}$ is infinity. The analysis computes sets *mins* and *maxs* of symbolic times such that there exist $t_{min} \in mins$ and $t_{max} \in maxs$ such that $\sigma(t_{min}) \leq \hat{t}_{min}$ and $\hat{t}_{max} \leq \sigma(t_{max})$, where for a symbolic time t, $\sigma(t)$ is the value of t with the timing parameters instantiated with the values in σ. Thus, *mins* and *maxs* provide a lower bound on $\hat{t}_{min}$ and an upper bound on $\hat{t}_{max}$, respectively.

To see why *mins* and *maxs* sometimes need to be sets of symbolic times, suppose the analysis determines that the minimum separation between the start and end events is either $3\delta_1$ or $2\delta_2$, depending on some non-deterministic choice. Given only that $\delta_1 < \delta_2$, there is no way to determine whether $3\delta_1$ or $2\delta_2$ is smaller, so the analysis would return $mins = \{3\delta_1, 2\delta_2\}$. Ordering relationships between symbolic times are discussed further in Section 3.3.

3.3 One-Anchor Analysis Algorithm

Timing properties are evaluated by constructing a graph with nodes corresponding to global states of the system and with edges corresponding to transitions, and then calculating *mins* and *maxs* from the occurrence intervals of the start and end events. Each global state g has a field $g.LS$ containing an array of local states and a field $g.PE$ containing a set of pending events. The set of pending events reflects the state of the operating systems and network. There are two kinds of pending events: messages in transit (MSG), and ticking timers (TMR). Each pending event *pe* has a field $pe.PID$ containing the name of the process that will execute *pe* and a field $pe.OI$ containing an *occurrence interval* $[t_1, t_2]$, where t_1 and t_2 are lower and upper bounds, respectively, on the time at which *pe* occurs. Occurrence intervals are implicitly anchored at (*i.e.*, measured with respect to) the start of the execution. Each edge in the graph is labeled with the pending event that occurs along that edge.

The graph constructed for the running example is shown in Figure 1, where msg_i corresponds to reception of message m_i, tmr_0 is p_0's initial timer, *etc.*

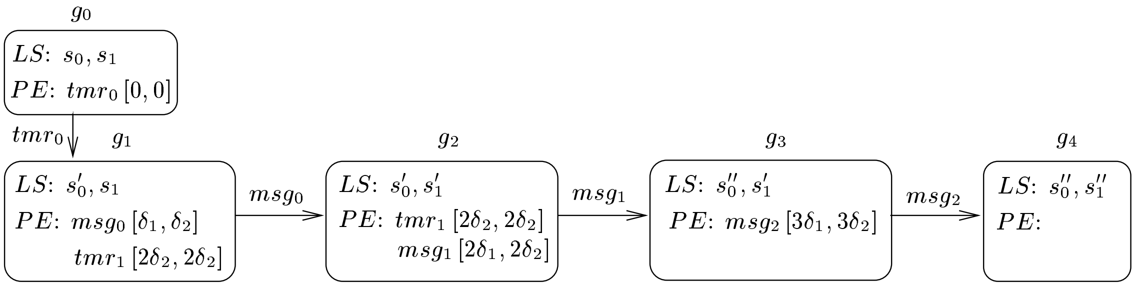

g_0
LS: s_0, s_1
PE: $tmr_0\,[0,0]$
tmr_0
g_1
LS: s_0', s_1
PE: $msg_0\,[\delta_1, \delta_2]$
$tmr_1\,[2\delta_2, 2\delta_2]$
msg_0
g_2
LS: s_0', s_1'
PE: $tmr_1\,[2\delta_2, 2\delta_2]$
$msg_1\,[2\delta_1, 2\delta_2]$
msg_1
g_3
LS: s_0'', s_1'
PE: $msg_2\,[3\delta_1, 3\delta_2]$
msg_2
g_4
LS: s_0'', s_1''
PE:

Figure 1: The graph computed in the one-anchor analysis of the running example.

Exploring a Pending Event. The graph is constructed by starting with the initial state (*i.e.*, the global state with every process in its local state and with pending events corresponding to the initial outputs) and repeatedly selecting a global state g and a pending event *pe* in $g.PE$ and exploring *pe*, as follows. Apply the transition function for $pe.PID$ to the local state $g.LS[pe.PID]$ of $pe.PID$ and the pending event *pe* (now regarded as an input event) to obtain the set S of possible outcomes. For each $\langle s, outs\rangle \in S$, construct a global state g' by copying g and then applying the following updates: set $g'.LS[pe.PID]$ to equal s; remove *pe* from $g'.PE$; process the elements of *outs*, as follows, in the order they appear. A MSG output adds to $g'.PE$ a MSG pending event with the specified destination, message contents, *etc.* A SETTMR output adds to $g'.PE$ an appropriate TMR pending event. A DELTMR output removes a specified TMR pending event from $g'.PE$. A SIG output is regarded as an input event of the specified destination process q and is processed in essentially the same way that a pending event is explored, namely, by updating the local state of q and processing the resulting outputs of q, if any.[2] Calculation of occurrence intervals associated with new pending events and signals is discussed below. If the resulting global state g' does not already appear in the graph, then it is added. Also, an edge from g to g' labeled with *pe* is added. An enhancement that sometimes yields tighter bounds is: update the lower bounds of other pending events in g' to reflect that in g', those pending events must occur after *pe*, and update the upper limit of *pe* on the edge from g to g' to indicate that it must occur before all other pending events in $g.PE$. For example, let *pe* be a pending event with occurrence interval $[\delta_1+\delta_2, 3\delta_2]_{init}$ in global state g. Let *pe'* be a pending event with occurrence interval $[2\delta_1, 2\delta_2]_{init}$ in global state g. Then, in the global state g' reached from g by executing *pe*, we can replace *pe'* with a new pending event whose occurrence interval is $[\delta_1+\delta_2, 2\delta_2]_{init}$. Similarly, the upper limit of the occurrence interval of *pe* on this edge can be changed to $2\delta_2$. If *pe* or a causally subsequent event is an end event, this can lead to tighter *mins*.

Computing Occurrence Intervals. When a pending event *pe* with occurrence interval $[t_1, t_2]$ is explored, occurrence intervals are computed as follows for the resulting signals and pending events. Let *out* be an output produced by exploring *pe*. Let $[\mathit{off}_1, \mathit{off}_2]$ be the offset associated with *out*. If *out* is the sending of a message m, then the occurrence interval of the pending event corresponding to reception of m is $[t_1 + \mathit{off}_1 + \delta_1, t_2 + \mathit{off}_2 + \delta_2]$. If *out* is the setting of a timer to expire after time t, then the occurrence interval of the pending event corresponding to expiration of that timer is $[t_1 + \mathit{off}_1 + t, t_2 + \mathit{off}_2 + t]$. If *out* is a signal, then the occurrence interval of the input event corresponding to reception of that signal is $[t_1 + \mathit{off}_1, t_2 + \mathit{off}_2]$.

In the running example, the offset intervals are all $[0, 0]$. When msg_1 is explored in global state g_2, p_0 produces two outputs: it sends m_2 and cancels the timer. The former output creates a pending event msg_2 with occurrence interval $[\delta_1 + 0 + \delta_1, \delta_2 + 0 + \delta_2]$.

Enabled Pending Events. For a global state g, only pending events in $g.PE$ that could occur next in an execution of the system should be explored in g. We introduce an ordering $\dashrightarrow_1$, called "can occur before", and explore a pending event *pe* in $g.PE$ iff *pe* can occur before every other pending event in $g.PE$, in which case we say that *pe* is *enabled* in g. The ordering $\dashrightarrow_1$ is determined by comparing symbolic times in occurrence intervals. The comparisons are based on arithmetic identities (*e.g.*, δ_1 is less than $2\delta_1$) and the premise that δ_1 is less than δ_2.[3] The results of some comparisons (*e.g.*, between $3\delta_1$ and $2\delta_2$) are undetermined, a situation denoted by $\perp$. Thus, for symbolic times t and t', we define

$$cmp(t, t') = \begin{cases} EQ & \text{if } t \text{ and } t' \text{ are equal} \\ LT & \text{if } t \text{ is definitely less than } t' \\ GT & \text{if } t \text{ is definitely greater than } t' \\ \perp & \text{otherwise} \end{cases}$$

[2] If the outputs of two processes p and q both contain signals sent to some process r, then one must be careful to specify the order in which those signals are processed by r. Our current implementation simply prohibits such situations, *i.e.*, it prohibits race conditions involving signals.

[3] It is straightforward to allow linear inequalities involving all timing parameters to be supplied as premises. If the symbolic times are all linear expressions, comparisons can be evaluated using, *e.g.*, Shostak's loop residue approach [Sho91].

A pending event pe with occurrence interval $[t_1, t_2]$ *can occur before* a pending event pe' with occurrence interval $[t'_1, t'_2]$, denoted $pe \dashrightarrow_1 pe'$, iff $cmp(t_1, t'_2) \in \{LT, EQ, \bot\}$ and it is not the case that pe is a timer expiration, pe' is a message reception, and $cmp(t_1, t'_2) = EQ$. The exception reflects the fact that time intervals associated with message reception are implicitly right-open, because δ_2 is a *strict* upper bound on message delay. In the running example, $tmr_1 \dashrightarrow_1 msg_1$ does not hold because of this exception, so tmr_1 is not enabled in g_2. Our "can occur before" relation is reminiscent of Lamport's "can affect" relation [Lam86]; however, we consider events to be atomic, so the lack of a definite event ordering stems only from the lack of exact (numerical) occurrence times for events.

Evaluating Timing Properties. Timing properties can be evaluated as follows. (1) The entire graph is constructed. (2) The graph is searched to obtain a set S containing every pair $\langle e, e' \rangle$ of events such that e is a first occurrence of a start event and e' is a first causally-subsequent occurrence of an end event. (3) For each pair $\langle e, e' \rangle$ in S, $t'_1 - t_2$ is inserted in $mins$ and $t'_2 - t_1$ is inserted in $maxs$, where $[t_1, t_2]$ and $[t'_1, t'_2]$ are the occurrence intervals of e and e', respectively.[4] (4) $mins$ is simplified by removing each element t such that there is some $t' \in mins$ such that $cmp(t', t) = LT$. $maxs$ is simplified by removing each element t such that there is some $t' \in maxs$ such that $cmp(t', t) = GT$.

For brevity, we have elided checks for special cases, *e.g.*, existence of a start event not causally followed by a stop event, in which case $maxs = \{\infty\}$. As an optimization, steps (1) and (2) are combined in our implementation. This sometimes enables us to avoid constructing part of the graph, because there is no need to explore pending events that occur after a stop event that causally follows a start event.

For the running example: (1) construct the graph in Figure 1; (2) the only start event is the output that produces msg_0, and the only end event is msg_1; these events have occurrence intervals $[0, 0]$ and $[2\delta_1, 2\delta_2]$, respectively; (3),(4) the result of the analysis is $mins = \{2\delta_1\}$ and $maxs = \{2\delta_2\}$. Note that these bounds are tight.

4 Limitations of One-Anchor Analysis

The one-anchor analysis often produces loose bounds. We illustrate this with a different timing property of the running example. The start condition is "p_1 sends m_1 to p_0"; the stop condition is "p_1 receives m_2 from p_0". Referring to the graph in Figure 1, there is one start event (the output that produces msg_1) and one end event (msg_2), with occurrence intervals $[\delta_1, \delta_2]$ and $[3\delta_1, 3\delta_2]$, respectively, so $mins = \{3\delta_1 - \delta_2\}$ and $maxs = \{3\delta_2 - \delta_1\}$. These bounds are loose; tight bounds are $2\delta_1$ and $2\delta_2$, respectively.

The root of the problem is that occurrence intervals are measured only from the start of the execution, so they get wider and wider along paths from the initial state. Thus, the later the start event occurs in the execution, the looser are the bounds computed by the one-anchor analysis. The bounds obtained for the example in Section 3 are tight because the start event occurs at the start of the execution.

This phenomenon also affects calculations of enabledness. For example, consider a modified version of the running example in which p_1 sets a timer to $2\delta_2$ (and sends m_1) when it receives m_0. Let tmr_2 denote the pending event corresponding to that timer. tmr_2 has occurrence interval $[\delta_1 + 2\delta_2, 3\delta_2]$. Let g'_3 be the global state with pending events tmr_2 and msg_2, where msg_2 is as in Figure 1. tmr_2 is enabled in g'_3, because $tmr_2 \dashrightarrow_1 msg_2$, because $cmp(\delta_1 + 2\delta_2, 3\delta_2) = LT$. However, it is clear that tmr_2 cannot actually occur before msg_2. Thus, the one-anchor analysis produces a graph containing paths that do not correspond to possible

[4] For some values of the timing parameters, $t'_2 - t_1$ might be negative. It is still a legitimate lower bound on the minimum time from a start event until an end event.

executions, and this may cause the bounds in *mins* and *maxs* to be loose. The one-anchor analysis is sound, because the graph does contain a path corresponding to each possible execution.

5 Multi-Anchor Analysis

Tighter bounds are obtained by tagging each pending event with a set of occurrence intervals, each of which is an *anchored time interval* $[t_1, t_2]_a$, where the *anchor* a is an event from whose occurrence t_1 and t_2 are measured. As a special case, the anchor can be *init*, representing the start of the execution. Every event has an occurrence interval anchored at *init*. For efficiency, only selected events are used as anchors. Specifically, all start events and SETTMR or MSG outputs are used as anchors, except those that occur at the start of the execution (*i.e.*, have occurrence interval $[0,0]_{init}$); we call these *anchor events*. Pending events are explored in the same way as in the one-anchor analysis, with the calculation of occurrence intervals and the definition of enabledness modified as follows.

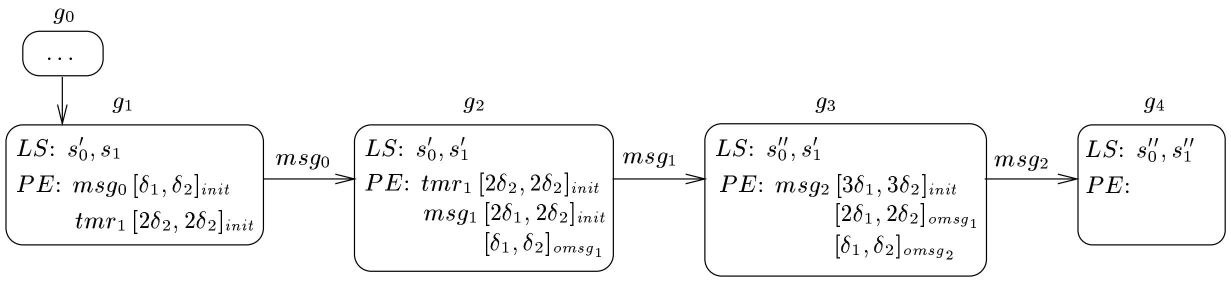
Computing Occurrence Intervals. When a pending event pe is explored, producing a set S of signals and pending events, each occurrence interval of pe is propagated to all elements of S in essentially the same way that occurrence intervals are propagated in the one-anchor analysis. Furthermore, if processing of pe involves an anchor event a (which could be pe itself or a MSG output in S) that does not occur at the start of the execution, then occurrence intervals anchored at a are added to a and all events in S that causally follow a; these occurrence intervals are easily computed from the offsets. For example, if a is pe and the offset of some MSG output is $[\mathit{off}_1, \mathit{off}_2]$, then the corresponding MSG pending event in S has an occurrence interval $[\mathit{off}_1 + \delta_1, \mathit{off}_2] + \delta_2$. A simple inductive proof shows: for every anchor event a that does not occur at the start of the execution, every event that causally follows a has an occurrence interval anchored at a.

Enabled Pending Events. A pending event pe can occur before another pending event pe' only if this ordering is consistent with all of the occurrence intervals of those events. Let A be the set of anchors common to pe and pe'; thus, for all $a \in A$, pe and pe' have occurrence intervals $[t_1(a), t_2(a)]_a$ and $[t'_1(a), t'_2(a)]_a$, respectively. Then *pe can occur before* pe', denoted $pe \dashrightarrow pe'$, iff for all $a \in A$, $cmp(t_1(a), t'_2(a)) \in \{LT, EQ, \bot\}$ and it is not the case that pe is a timer expiration, pe' is a message reception, and $cmp(t_1(a), t'_2(a)) = EQ$. For a global state g and a pending event pe in $g.PE$, pe is *enabled* in g iff for all other pending events pe' in $g.PE$, $pe \dashrightarrow pe'$.

Evaluating timing properties. Timing properties are evaluated in essentially the same way as in the one-anchor analysis, except step (3) is modified to be: for each pair $\langle e, e' \rangle$ in S, t'_1 is added to *mins* and t'_2 is added to *maxs*, where $[t'_1, t'_2]_e$ is the occurrence interval of e' anchored at e (if e occurs at the start of the execution, the occurrence interval of e' anchored at *init* is used instead). The definition of causality in Section 3.2 ensures that e' has an occurrence interval anchored at e. If we base the semantics of timing properties on Lamport's causality relation or temporal ordering instead, this might not be the case, so occurrence intervals with other anchors would need to be considered in this calculation.

For the running example of Section 3, the graph produced by the multi-anchor analysis is the same as in Figure 1, except with $[\cdots]$ changed to $[\cdots]_{init}$. The start event (*i.e.*, the sending of msg_0) and the SETTMR output of p_0 have occurrence interval $[0,0]_{init}$, so they are not anchor events. The calculations of *mins* and *maxs* yield the same results as in Section 3.

For the timing property considered in Section 4: (1) construct the graph in Figure 2, where $omsg_1$ denotes the MSG output of p_1 representing the sending of m_1; (2) $omsg_1$ is the only start event and msg_2 is the only stop event; (3),(4) the result of the analysis is $mins = \{2\delta_1\}$ and $maxs = \{2\delta_2\}$. These bounds are tight.



g_0
$\ldots$
g_1
LS: s_0', s_1
PE: $msg_0\,[\delta_1, \delta_2]_{init}$
$tmr_1\,[2\delta_2, 2\delta_2]_{init}$
msg_0
g_2
LS: s_0', s_1'
PE: $tmr_1\,[2\delta_2, 2\delta_2]_{init}$
$msg_1\,[2\delta_1, 2\delta_2]_{init}$
$[\delta_1, \delta_2]_{omsg_1}$
msg_1
g_3
LS: s_0'', s_1'
PE: $msg_2\,[3\delta_1, 3\delta_2]_{init}$
$[2\delta_1, 2\delta_2]_{omsg_1}$
$[\delta_1, \delta_2]_{omsg_2}$
msg_2
g_4
LS: s_0'', s_1''
PE:

Figure 2: The graph computed in the multi-anchor analysis of the running example. g_0 is the same as in Figure 1, except $[\cdots]$ is changed to $[\cdots]_{init}$. Two occurrence intervals are shown for msg_1; three occurrence intervals are shown for msg_2.

6 Failures

Failures can be introduced in two ways: by adding them to the system model or the processes. We take the latter approach, because it keeps the analyzer simple and because a failure mode can be added to processes in a modular way, by embodying the failure mode as a *process transformer*, *i.e.*, a function that takes a process (*i.e.*, an automaton) as input and returns a new process. For example, consider a process transformer $\mathcal{T}$ that embodies crash failures. Given a system with processes $p_0, \ldots, p_N$, the system with processes $\mathcal{T}(p_0), \ldots, \mathcal{T}(p_N)$ is the same except that each process might crash at any time. The set of states of $\mathcal{T}(p)$ is the set of states of p plus a new state s_{dead}. The transition function r of $\mathcal{T}(p)$ is defined as follows. When $\mathcal{T}(p)$ is in state s_{dead}, inputs are "ignored", *i.e.*, r returns $\{\langle s_{dead}, \varepsilon\rangle\}$, where ε is the empty sequence. We do not assume that a sequence of outputs produced by an input event is failure-atomic; thus, there is a possibility of crashing before each output. When $\mathcal{T}(p)$ is in a state of p, r calls the transition function of p to obtain the set S of p's possible behaviors and then returns $S \cup (\bigcup_{\langle s, outs\rangle \in S} \bigcup_{os \prec outs} \{\langle s_{dead}, os\rangle\})$, where $x \prec y$ means x is a prefix of y. Similarly, one can define process transformers that add crashes and recoveries, timing failures, message loss, *etc.*

Timing properties for executions involving limited numbers of failures are often of interest, so we parameterize process transformers by the number of failures. For example, $\mathcal{T}_f$ is defined so that f crashes occur in executions of the system $\mathcal{T}_f(p_0), \ldots, \mathcal{T}_f(p_N)$. To enforce this, when $\mathcal{T}_f(p)$ crashes, it informs every other process of this by sending signals. Each process $\mathcal{T}_f(p)$ keeps track of the number of such signals received and does not crash if this number equals f. The synchronous nature of signals is essential here. Of course, these signals do not correspond to actual communication in an implementation of the system.

7 Example: Uniform Timed Reliable Broadcast

We are applying TADA to a new rotating-coordinator algorithm, called UTRB4 [MS98], for uniform timed reliable broadcast (UTRB) [BT93a, BT93b] that tolerates crash failures. A UTRB algorithm can form the heart of an implementation of non-blocking atomic commitment in a distributed database system [BT93a, BT93b]. UTRB4 uses carefully designed patterns of time-outs to help each process determine the status of other processes (*e.g.*, whether they received certain messages) without sending messages. UTRB4 has a worst-case message complexity of $2(N-1)+\frac{1}{2}(f-1)f$, where N is the number of processes and f is the number of crashes, compared to $2(N-1)+2(N-1)f$ for UTRB2 [BT93a, BT93b], which is the most message-efficient UTRB algorithm we have found in the literature. For example, when $f=1$, UTRB4 uses half as many messages as UTRB2. The manual calculations of the time-outs are non-trivial, because they require consideration of chains of events involving multiple failures. The times for these chains of events can be expressed as timing properties and checked with TADA. That will be an interesting test of those calculations and the analysis.

We have used TADA to verify for particular values of N and f that the worst-case delay from when a broadcast is initiated until all non-faulty processes have delivered the message is $\delta_2+\tau$ for $f=0$, $3\delta_2+\tau$ for $f=1$, and $2^{f+1}\delta_2+2^{f-2}\tau-\delta_2$ for $f>1$, where the timing parameter τ characterizes the local overhead of sending messages to all other processes. For $N=7$ and $f=0$, the analysis took approximately 26.5 min (on a 75 MHz MIPS R8000) and 1.8 MB of heap memory; for $N=4$ and $f=3$ (the case $f=N$ isn't interesting), the analysis took approximately 4.5 min and 2.2 MB of heap memory; for $N=5$ and $f=4$, the analysis took approximately 6.5 hours and 62 MB of heap memory.

The current implementation is simple but inefficient. The performance can be improved by a significant constant factor by using more efficient data structures and algorithms (*e.g.*, incrementally maintaining the set of enabled events, rather than recomputing it from scratch after each transition). More importantly, as mentioned in the Introduction, we expect a partial-order optimization to reduce the analysis cost (time and space) by an exponential factor for many systems, including UTRB4.

References

[ACH$^+$95] R. Alur, C. Coucoubetis, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AHV93] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. Technical Report 93-1345, Cornell University, Department of Computer Science, 1993.

[BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *Proc. 9th Int'l. Conference on Concurrency Theory (CONCUR)*, 1998.

[BT93a] Özalp Babaoğlu and Sam Toueg. Non-blocking atomic commitment. In Sape Mullender, editor, *Distributed Systems*, chapter 6, pages 147–168. Addison Wesley, 2nd edition, 1993.

[BT93b] Özalp Babaoğlu and Sam Toueg. Understanding non-blocking atomic commitment. Technical Report UBLCS-93-2, University of Bologna, Laboratory for Computer Science, 1993.

[CKP$^+$96] David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11), November 1996.

[CZ85] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, 1985.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[KM95] R. P. Kurshan and K. L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117(1), 1995.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–564, 1978.

[Lam86] Leslie Lamport. On interprocess communication: Part 1. *Distributed Computing*, 1:76–101, 1986.

[Lam89] Leslie Lamport. The part-time parliament. Technical Report SRC-49, Digital Equipment Corporation, Systems Research Center, 1989.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.

[MR97] Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *The Journal of Computer Security*, 5:113–127, 1997.

[MS98] Yu Ma and Scott D. Stoller. Message-efficient uniform timed reliable broadcast. Technical Report 515, Computer Science Dept., Indiana University, September 1998.

[PLL97] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the Paxos algorithm. In M. Mavronicolas and P. Tsigas, editors, *Proc. 11th International Workshop on Distributed Algorithms (WDAG '97)*, volume 1320 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[PPH97] Doron Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors. *Proc. Workshop on Partial Order Methods in Verification*, volume 29 of *DIMACS Series.* American Mathematical Society, 1997.

[RVR93] L. Rodrigues, P. Veríssimo, and J. Rufino. A low-level processor group membership protocol for lans. In *Proc. IEEE 13th International Conference on Distributed Computing Systems (ICDCS).* IEEE Computer Society Press, 1993.

[Sha89] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[Sho91] Robert E. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1991.