Timing Tolerances in Safety-Critical Software

Alan Wassyng[⋆], Mark Lawford[⋆], and Xiayong Hu

Software Quality Research Laboratory, Department of Computing and Software,
McMaster University, Hamilton, Canada
{wassyng, lawford, huxy}@mcmaster.ca

Abstract. Many safety-critical software applications are hard real-time systems. They have stringent timing requirements that have to be met. We present a description of timing behaviour that includes precise definitions as well as analysis of how functional timing requirements interact with performance timing requirements, and how these concepts can be used by software designers. The definitions and analysis presented explicitly deal with tolerances in all timing durations. Preliminary work indicates that some requirements may be met at significantly reduced CPU bandwidth through reduced variation in cycle time.

Keywords: safety-critical, real-time, timing tolerances, requirements.

1 Introduction

Specifying, implementing and verifying real-time requirements for embedded software systems can be a difficult and time consuming task. Hence real-time systems have become an active area of research in the formal methods community. Practical implementations have to worry about sampling rates, schedulability, computation time, latency, and jitter, all of which involve tolerances in some form when interfacing a physical plant and a software control system. In this paper we make the case that several different types of tolerances need to be fully specified at the requirements level in order to properly deal with the timing tolerances that are inherent in the system implementation. These include tolerances on functional timing requirements, and tolerances that allow for deviation from the idealized behaviour specified by the requirements models. This work builds on analysis and definitions that were used in safety-critical software applications over many years at Ontario Power Generation in Canada [9].

The extensive survey of formal methods for the specification and verification of real-time systems in [1] contains references to over 200 publications. The overwhelming majority of the cited works are dedicated to the specification and validation of real-time requirements. Despite this intensity of research, relatively little work has been done on formally modeling timing tolerances.

Recent work has begun to address the issue of timing tolerances required to verify implementations of requirements modeled as timed automata with ASAP

[⋆] Partially supported by NSERC.

semantics [2, 3]. Wulf, et al, consider the case of implementing a continuous-time controller with a discrete-time system, assuming that there is a delay Δ associated with the controller's reaction to the environment. Both the controller and the plant are first modeled as timed automata. Their control objective is to ensure that the closed-loop system satisfies a safety property by avoiding bad states. Provided that all control actions can be delayed by up to some fixed $\Delta > 0$ without violating the safety property, they say that the controller is "implementable". A PSPACE-complete decision procedure to test implementability is described in [3], while [2] provides a semi-decision procedure to compute the maximal reaction delay Δ allowable by the implementation that still preserves the correctness of the closed loop system. It further shows that the system is implementable by a cyclic executive with loop time upper bound Δ_L and a finite precision clock with a resolution of Δ_P, provided that $\Delta > 3\Delta_L + 4\Delta_P$. In this work response allowance ra and sample interval ts correspond most closely to Δ and Δ_L in [2] and implicitly we assume a clock resolution of 1 time unit. Based on our definitions, and using simple mathematical arguments, we are able to come to a somewhat surprising result that allows some timing requirements to be verifiably implemented at a significantly lower CPU bandwidth.

The remainder of this paper is organized as follows: Section 2 provides the notation and definitions of terms and operators, and specifically differentiates between functional and performance timing requirements. Section 3 describes the relationship between the two performance timing requirements, while Section 4 details the interaction of functional and performance timing requirements. Conclusions are provided in Section 5.

2 Definitions

2.1 The Requirements Model

The requirements model we use is a finite state machine with an arbitrarily small clock-tick. This enables us to straddle the time continuous and time discrete domains. Many other models could be used and would require minimal changes in the following definitions.

Stimuli are referred to as *monitored variables*, and responses are *controlled variables.*

The finite state machine is assumed to describe idealized behaviour, i.e. results are produced instantaneously. If $C(t)$ is the vector of values of all controlled variables at time t, $M(t)$ is the vector of values of all monitored variables at time t, $S(t)$ is the vector of values of all state variables at time t, we can define relations R (requirements) and NST (next state) as follows:

$$\begin{aligned} C(t_k) &= R(M(t_k), S(t_k)) \\ S(t_{k+1}) &= NST(M(t_k), S(t_k)), \text{ for } k = 0, 1, 2, \ldots \end{aligned} \tag{1}$$

where the time of initialization is t_0, and the time between t_k and t_{k+1} is an arbitrarily small time, δt. It is almost always necessary to decompose the construction of R and NST into a number of intermediate functions. NST in our

$$\begin{array}{|c|}\hline Condition_1 \\ \hline Condition_2 \\ \hline \cdots \\ \hline Condition_n \\ \hline \end{array} \quad \begin{array}{||c||}\hline f\_name \\ \hline\hline result_1 \\ \hline result_2 \\ \hline \cdots \\ \hline result_n \\ \hline\hline \end{array} \quad \equiv \quad \begin{array}{l} \text{if } Condition_1 \text{ then } f\_name = result_1 \\ \text{else if } Condition_2 \text{ then } f\_name = result_2 \\ \text{else if ... then ...} \\ \text{else if } Condition_n \text{ then } f\_name = result_n \end{array}$$

Disjointness: $Condition_i \wedge Condition_j \Leftrightarrow False, \forall i, j = 1..n, i \neq j$, and
Completeness: $Condition_1 \vee \ldots \vee Condition_n \Leftrightarrow True$.

Fig. 1. Horizontal Condition Tables

formulations is usually trivial since we strive to keep state data at the requirements level to a very simple form, namely the previous values of intermediate functions and variables.

2.2 Notation

Current time is denoted by t_{now}. We indicate elements of state data by subscripting the identifiers. $variable_{-n}$ means the value of *variable*, n clock-ticks prior to the current one.

Where possible, we use *tabular expressions* to define functions. We are convinced that tabular expressions (function tables) are a superb notation for describing software functions. Disjointness and completeness criteria help us in ensuring that the functional descriptions are unambiguous and complete [6]. There have been a number of publications on the semantics and usage of tabular expressions (e.g. [7, 8, 9]). The tabular expressions we use here are particular simple (they are called *horizontal condition tables*). Fig. 1 presents an example table together with its informal semantics.

2.3 Functional Timing Requirements

Functional timing requirements are timing requirements that are directly related to the required behaviour of the application. Some of the more common functional timing requirements are described below, and mathematical definitions are provided.

Sustained Timing Requirements: A common functional timing requirement is one that specifies that a condition must be sustained over a particular time duration. For example, to filter out the effect of a noisy signal we may specify that an event in which a sensor signal is above its setpoint should be sustained for 300 ms before it can cause a “trip”. This means that the implementation must guarantee that if the sensor event is sustained for less than 300 ms, the trip must not occur. Similarly, if the sensor event is sustained for 300 ms or longer, the trip must be generated. Without tolerances on the time duration, these requirements would be impossible to meet.

Many of the concepts and analyses we present are best illustrated when applied to sustained timing requirements. For this reason we discuss this example in detail.

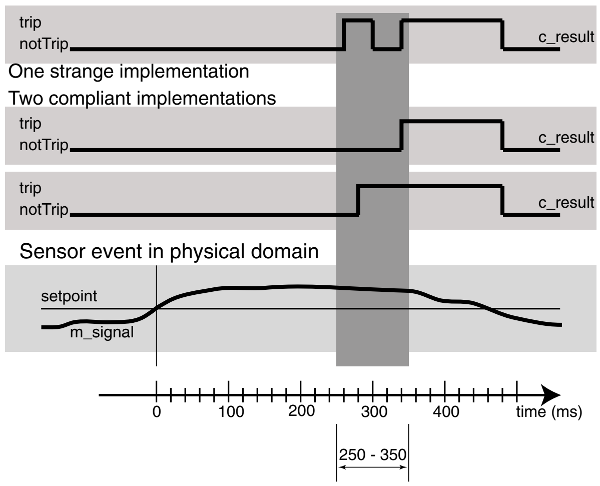

Fig. 2. Two Valid Implementations of a Sustained Timing Requirement

We can introduce tolerances on the time duration in the above example. Assume that the sensor trip condition should be sustained for 300 ±50 ms.

Fig. 2 shows an implementation of the behaviour specified above for a controlled variable *c_result* and sustained condition $m\_signal \geq setpoint$. The strange behaviour in the top implementation is almost certainly not what the specifier intended, but it may be compliant with its specification. How should we interpret this specification? A logical interpretation is that *c_result* should not equal trip until $m\_signal \geq setpoint$ has been True for at least 250 ms, and that *c_result* must equal trip if $m\_signal \geq setpoint$ has been True for 350 ms.

The problem is: what happens in the range 250–350 ms? Fig. 2 shows another two possible implementations that really would be compliant with this requirement. The difference here is that for each event we have effectively restricted ourselves to a single representative duration inside the specified range.

There are a number of important points to emphasize. i) The time duration is measured from when the event started in the physical application domain. It is not measured from the time it is detected. Since the requirements are (supposed to be) developed by the domain experts, and should be independent of any implementation, it does not make sense to define timing requirements with reference to when events are detected. ii) Many different implementations are valid. The behaviour in the dark shaded interval representing time in the interval [250, 350] ms is not deterministic. It is vital that everyone has the same understanding of what the requirement means. iii) Even though we have introduced tolerances into the requirement, the requirement still describes idealized behaviour understood within the constraints of the requirements model. For instance, it does not take into account that processing time is not infinitely small, and it makes no reference to how often the application samples the values of the sensor.

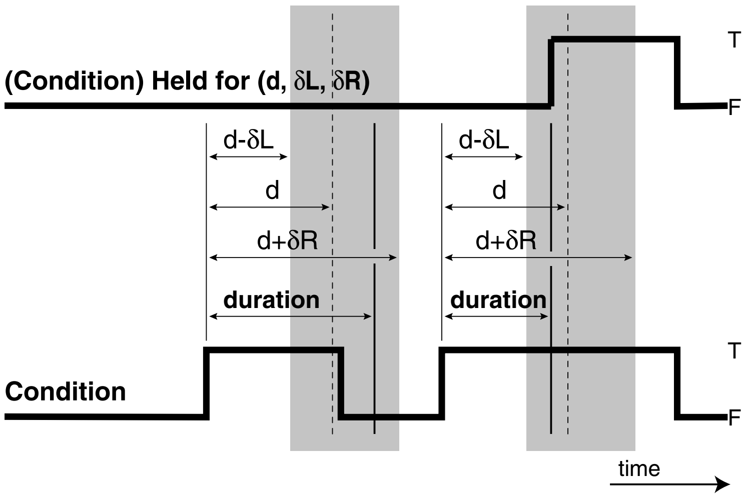

Fig. 3. "Held for" Functional Timing Requirement

(Condition :bool) **Held for** (d: $\mathbb{R}^{>0}$, δL, $\delta R : \mathbb{R}^{\geq 0}$) :bool
where duration(Condition: bool): $[\mathrm{d} - \delta L, \mathrm{d} + \delta R]$
Event_start_time(Condition :bool) : $\mathbb{R}^{\geq 0}$
Initially: duration = any value in $[d - \delta L, d + \delta R]$
Event_start_time$_{-1}$ = 0
Condition$_{-1}$ = False

	duration	Event_start_time
(Condition = True) & (Condition$_{-1}$ = False)	Any value in $[d{-}\delta L, d{+}\delta R]$	t_{now}
(Condition = False) OR (Condition$_{-1}$ = True)	No Change	No Change

		Held for
Condition = True	$t_{now}-$ Event_start_time$\geq$ duration	True
	$t_{now}-$ Event_start_time$<$ duration	False
Condition = False		False

Fig. 4. Formal Definition of "(Condition) Held for (d, δL, δR)"

To model sustained events, we developed an infix operator, *(Condition) Held for (d, δL, δR)*, which uses a *duration* defined by the constant time d (> 0), with tolerances $-\delta L$, $+\delta R$, $0 \leq \delta L < d, 0 \leq \delta R$. "Held for" is illustrated in Fig. 3, and is defined formally using tabular expressions in Fig. 4. A critical concept is that although duration can be any value in the interval $[d - \delta L, d + \delta R]$, it must be constrained so that duration has only a single value throughout an event. An event in this case means that *Condition* changes from False to True. Without this constraint, many different bizarre behaviours are possible, all of them clearly not the intent of the function.

Periodic Timing Requirements: Periodic timing requirements are common in hard real-time systems. To help us model periodic timing requirements we developed a function, *Periodic(Condition, d, δL, δR)*. This function (*Periodic*) is True for 1 clock-tick at the instant that *Condition* changes from False to True, and, as long as *Condition* remains True, the function is True again, some time "period" after the most recent time it changed from False to True. The effective *period* of the function is defined by the constant duration d (> 0), with tolerances $-\delta L, +\delta R, 0 \leq \delta L < d, 0 \leq \delta R$. *Periodic* is illustrated in Fig. 5, and is defined formally using tabular expressions in Fig. 6. A different kind of periodic function is one that is synchronized with an external clock as illustrated in Fig. 7.

If the periodic functional requirement is synchronized with an external clock, definitions equivalent to t **mod** period = 0 are useless when the period involves tolerances. The requirement t **mod** 400±50 ms = 0 results in milli-second intervals of [350-450], [700-900], [1050-1350], [1400-1800], [1750-2250], [2100-2700], ..., and after a relatively short time period the requirement does not constrain behaviour much at all. A practical, formal specification of this periodic functional requirement can be developed from $\forall n : \mathbb{N} \cdot t_n \in [n \cdot d - \delta L, n \cdot d + \delta R]$, and is defined using tabular expressions in Fig. 8. This definition does not deal explicitly with a consistent clock drift, but this could be included by specifying d as a constrained function of time.

2.4 Performance Timing Requirements

Functional behaviour of the application is (typically) described using a model that describes the ideal behaviour of the application. It totally ignores the fact that an implementation cannot continuously monitor sensor values and requires a finite, non-zero amount of time to process its results. To complete the description of the required behaviour, a requirements document must also specify the performance tolerances that are allowed in meeting functional timing requirements. There are two different performance timing requirements, *timing resolution* and *response allowance*. These are defined and discussed in the following two sections.

Timing Resolution: Each monitored variable has a timing resolution associated with it. The definitions for this interval are different for time continuous and time discrete monitored variables.

The timing resolution (TR) for a time continuous monitored variable is the minimum time duration of an initiating event dependent on that monitored variable for which the application must guarantee that it will detect that event. Thus, the TR is also an indication of the maximum time interval that the trip computer can allow between successive sampling instances for that stimulus.

The TR for a time discrete monitored variable is the smallest time interval separating two events dependent on that monitored variable, in which the application must guarantee that it will detect both events.

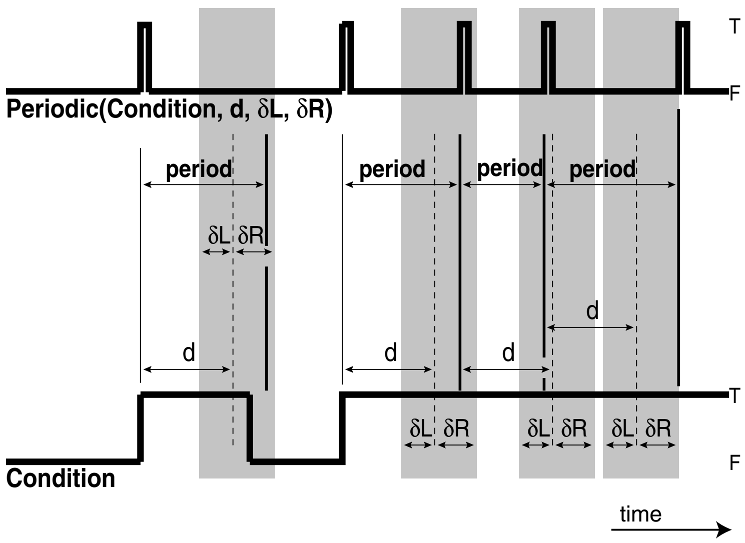

Fig. 5. A Periodic Functional Timing Requirement

Periodic(Condition :bool, d :$\mathbb{R}^{>0}$, δL, $\delta R : \mathbb{R}^{\geq 0}$) :bool
where period(Periodic$_{-1}$: bool): $[\text{d} - \delta L, \text{d} + \delta R]$
previous_pulse_time(Condition :bool) : $\mathbb{R}^{\geq 0}$
Initially: period = any value in $[0, \delta R]$; previous_pulse_time$_{-1}$ = 0; Periodic$_{-1}$ = False

	period
Periodic$_{-1}$ = True	Any value in [d-δL, d+δR]
Periodic$_{-1}$ = False	No Change

			Periodic	previous_pulse_time
Condition = True	Condition$_{-1}$ = False		True	t$_{now}$
	Condition$_{-1}$ = True	t$_{now}$ $\geq$ previous_pulse_time$_{-1}$ + period	True	t$_{now}$
		t$_{now}$ $<$ previous_pulse_time$_{-1}$ + period	False	No Change
Condition = False			False	No Change

Fig. 6. Formal Definition of "Periodic(Condition, d, δL, δR)"

These situations are illustrated in Fig. 9. Note that if a monitored variable is used in determining the behaviour of two (or more) controlled variables, it is probable that at least two different events (one on each controlled-monitored variable path) are dependent on that monitored variable, and that the monitored variable could have two different TRs associated with it. In general, we assign a TR for each controlled-monitored variable pair in which the controlled variable value can be affected by the value of the monitored variable.

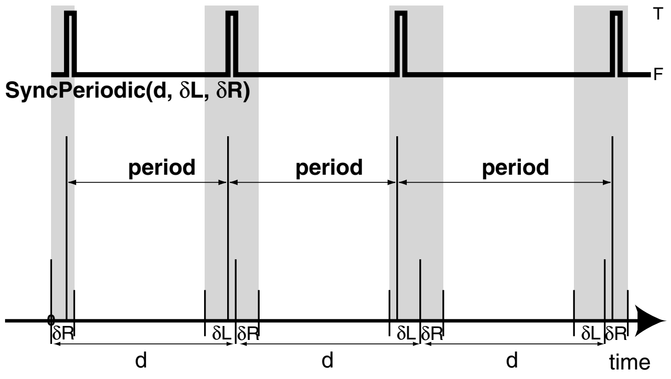

Fig. 7. Synchronized Periodic Functional Timing Requirement

SyncPeriodic$(d : \mathbb{R}^{>0}, \delta L, \delta R : \mathbb{R}^{\geq 0})$: *bool*
where n: $\mathbb{N}$, and $\varDelta : \mathbb{R}$
Initially: $n = 0$; $\varDelta$ = any value in $[0, \delta R]$; SyncPeriodic$_{-1}$ = False

	$\varDelta$	**n**
SyncPeriodic$_{-1}$ = True	Any value in [-δL, δR]	n + 1
SyncPeriodic$_{-1}$ = False	No Change	No Change

	SyncPeriodic
t$_{now}$ $\geq$ n·d + $\varDelta$	True
t$_{now}$ $<$ n·d + $\varDelta$	False

Fig. 8. Formal Definition of "*SyncPeriodic*$(d, \delta L, \delta R)$"

Response Allowance: The Response Allowance (RA) for a controlled-monitored variable pair specifies an allowable processing delay. Each controlled variable must have an RA specified for it. The RA applies to the controlled variable and the particular monitored variable on which the controlled variable's behaviour depends. The RA is measured from the time the event actually occurred in the physical domain, until the time the value of the controlled variable is generated and crosses the application boundary into the physical domain.

Some important considerations:

1. The RA for the pair *c*-*m* is meaningless if *c* does not change its value in response to a change in the value of *m* (the effect must be visible externally).
2. The time sequence of externally generated values of a controlled variable *c* cannot be altered by consideration of the RAs for each *c*-*m* pair. For instance, we cannot allow *c* to change from *trip* = True (evaluated at time t) to *trip*

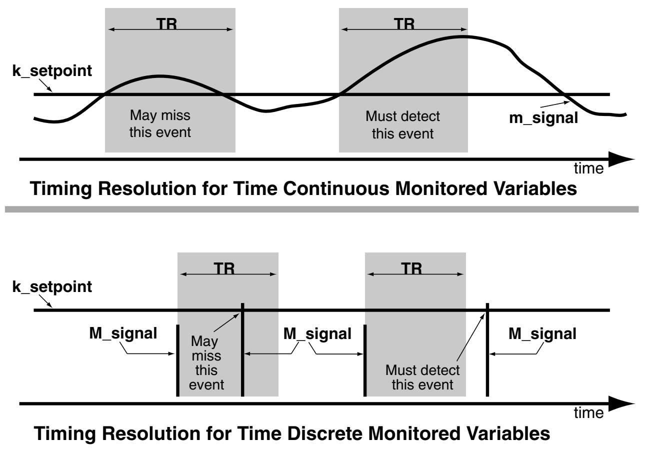

Fig. 9. Timing Resolution

= False (evaluated at time $t - \delta t$, δt is an arbitrarily small positive number) simply because the RA was large enough to allow this.

3 Relationship Between Response Allowance and Timing Resolution

Consider the case where c, a controlled variable, depends solely on m, a monitored variable. We must specify both a TR (value tr) and RA (value ra) for the c-m pair. Now, in the implementation, let ts represent the sampling interval used for m, and tp the processing time required to output c, measured from the instant that the value of m was sampled. Then, if the implementation is to comply with its timing requirements, it is clear that we must insist that $ts + tp \leq ra$. Since $tp > 0$ and $ts \leq tr$ (it is permitted to equal ra), it follows that $tr < ra$. So, unless there is a reason to use a more restrictive TR for m, we can assume a default upper limit for TR equal to the RA for the c-m pair. This is the least restrictive requirement that we can place on the software design. It leaves the designer free to choose a sampling interval anywhere in the range $[0, ra)$ as long as the RA is satisfied.

In most real applications, the TR for a monitored variable would be determined, initially, from a study of the possible transients associated with the particular monitored variable. If the physically motivated TR is larger than the associated RA then the TR would have to be constrained by the value of the RA. If the physically motivated TR is smaller than the associated RA, then that smaller value must be used as the specified TR.

The RA itself is always derived from consideration of the physical application. In safety-critical applications, absolute compliance with the RA is clearly just as important as compliance with any other requirement.

In the following section we see that both the TR and RA may need to be modified once we consider the effect of functional timing requirements.

4 Interaction Between Functional and Performance Timing Requirements

There are a number of interactions between functional and performance timing requirements. Some of them affect the timing resolution by imposing restrictions on sampling intervals in the implementation. Other interactions force us to consider exactly how to specify response allowances for controlled-monitored variable pairs that are also involved in functional timing behaviours.

We use sustained events to illustrate these interactions.

4.1 Timing Resolution for Sustained Events

Given a sustained timing requirement we need to consider whether it is possible to implement a design so that the requirement can be met. We can identify two different categories of sustained events. The first one, as discussed in Section 2.3, is where the behaviour depends on values of one or more monitored variables. In this case the event is timed from the time at which the event was initiated in the physical domain. The second kind is one in which the sustained event depends only on the values of controlled variables (or is synchronized in some way with an external clock). In this case the event is timed from the instant at which the event is initiated within the software domain. This kind of event is typically easier to deal with since the inherent uncertainty of when the event actually occurred is removed from consideration.

The following two sections present analyses of these cases.

Sample Intervals for Events That Depend on Monitored Variables: We know from earlier discussion (Section 2.3) that if we specify behaviour in the form of $(Condition)\ Held\ for\ (d,\ \delta L,\ \delta R)$, and $duration \in [d - \delta L, d + \delta R]$, then the requirement means that we cannot make the final decision as to whether "Held for" generates True or False based on values that were sampled before we are sure that $d - \delta L$ time has elapsed since the event occurred in the physical domain. We also cannot delay the decision past $d + \delta R$.

The situation is illustrated in Fig. 10. Let us assume that the sample intervals are ts_0, ts_1, ts_2, etc. Since our analysis has to hold for real industrial applications, we do not assume a constant sample interval. We do assume that we can place limits on the sample intervals. We call these limits ts_min and ts_max. Once we have these limits, we know that $ts\_min \leq ts_j \leq ts\_max$ for each $j \in \{0..n\}$. We will see later that any variation in sample intervals results in fewer feasible implementations. If the event is detected at sample time 1, then we know that the event must have occurred sometime between sample time 0 and sample time 1. We can now assume that *Condition* remains True at sample times 2, 3, ,..., n-2. (If it does not remain True, we simply terminate the event and the "Held for" value becomes False.)

If we study the situation in Fig. 10, we see that the only way we can be certain that we base our decision on values sampled in the time interval $[d-\delta L, d+\delta R]$ is to ensure that we have at least two sample points inside that interval. It turns out this is a necessary condition, but it is not sufficient.

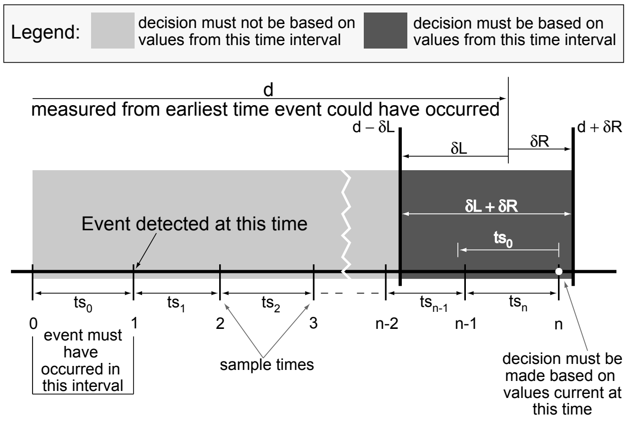
Legend:
decision must not be based on values from this time interval
decision must be based on values from this time interval
d
measured from earliest time event could have occurred
d – δL
d + δR
δL
δR
δL + δR
ts0
Event detected at this time
ts0
ts1
ts2
tsn-1
tsn
0
1
2
3
n-2
n-1
n
event must have occurred in this interval
sample times
decision must be made based on values current at this time

Fig. 10. Sample Intervals Required for Sustained Events

The earliest the event could have occurred is immediately after values were sampled at sample point 0. The latest the event could have occurred is immediately prior to sample point 1. We choose to measure all relevant times from sample point 0, i.e. from the earliest time it could have occurred. Now we can consider the two sample points in the interval $[d - \delta L, d + \delta R]$ (assuming we manage to get two samples in that interval). The later time in the interval (sample point n in Fig. 10) must be to the left of the $d + \delta R$ boundary because the times are measured from the earliest time the event could have occurred. So we know that decisions based on values sampled at sample point n are not too late. For it to be too early, the actual event would have had to occur immediately prior to sample point 1. In this case, we should subtract ts_0 from the time at sample point n and check to see if the resulting time is less than $d - \delta L$. If all sample intervals were equal, having two sample points in the interval would be sufficient to prove that sample point n could not be too early. However, since $ts_n \leq ts\_max$, there is a chance that $(sample\ point\ n) - ts_0$ could lie outside the interval, in which case the decision would be made based on values that are too early. The following analysis copes with all the questions we have raised. Note that we could have chosen to measure times from time of detection. The analysis would have to be adjusted accordingly.

Case 1: $\boldsymbol{0 < ts\_max \leq \frac{1}{2}(\delta L + \delta R)}$: In this case it is easy to see that it is always possible to implement the sustained event.

Case 2: $\boldsymbol{\frac{1}{2}(\delta L + \delta R) < ts\_max \leq (\delta L + \delta R)}$: It may happen that the hardware platform is not fast enough for us to arrange a sample interval that always works as defined in Case 1. It is still possible to find sample intervals that allow us to implement the sustained event.

It is crucial to realize that if $ts\_max > \frac{1}{2}(\delta L + \delta R)$ then the only way we can ensure that two samples, ts_max apart, fall in the duration interval, is if the last sample point to the left of the interval is not "too close" to $d - \delta L$.
Let $k_{min} = int(\frac{d-\delta L}{ts\_max})$, and $k_{max} = int(\frac{d-\delta L}{ts\_min})$, where int(r) truncates r to an integer.
$k_{min} \neq k_{max}$ implies that $k_{max} \cdot ts\_min \leq d - \delta L$ and
$k_{max} \cdot ts\_max > d - \delta L$, since $k_{max} > k_{min}$.

This means that there is always some combination of sample intervals such that $\sum_{j=1}^{k} ts_j = d - \delta L - \epsilon$, where ϵ is arbitrarily small. This implies that there are always sample intervals within the range $[ts\_min, ts\_max]$ such that there is only one sample point within $[d - \delta L, d + \delta R]$. Thus we can conclude that if $k_{min} \neq k_{max}$ then there is no feasible implementation.
So, $k_{min} = k_{max}$ is a necessary condition for a feasible implementation. Unfortunately it is not sufficient. Let $k = k_{min} = k_{max}$. Then $\sum_{j=1}^{k} ts_j \leq d - \delta L$, and $\sum_{j=1}^{k+1} ts_j \geq d - \delta L$, for any combination of sample intervals within $[ts\_min, ts\_max]$. The worst case is when $ts_j = ts\_max$ for each $j \in \{1, 2, ..., k + 2\}$. So, a sufficient condition when $k_{min} = k_{max}$ is that $(k+2) \cdot ts\_max \leq d + \delta R$.
Case 3: $(\delta L + \delta R) < ts\_max$: The sustained event cannot be implemented.

Examples of Feasible Sample Interval Ranges for Sustained Events: It is instructive to examine the ranges of sample intervals that result in feasible implementations of sustained events that are dependent on monitored variables. The analysis from Case 2 was implemented in a spreadsheet and graphs showing the feasible sample intervals were generated (Fig. 11). Each graph lists [d-δL, d+δR]. It also shows the nominal sample intervals as labels along the x-axis, and lists the deviations as $(-\ell, +r)$. So, for ts=50, with deviation $(-3, +2)$ we have ts_min=47 and ts_max=52. A deviation of $(-0, +0)$ indicates a constant sample interval (pretty much impossible to achieve).

Fig. 11 shows that in the case when $duration \in [400 - 50, 400 + 60]$, rather than requiring the code to run with every $ts \leq 50ms$ (a 20Hz or faster task), it is possible to detect the event with every $ts \in$ [74-1, 74+2] ms (roughly a 13.5Hz task). This represents an approximately 32% reduction in CPU time required for the task! This pattern results in a positive cycle. Making execution times more precise may present the opportunity to reduce the CPU load, which in turn should make it easier to meet timing requirements. While scheduling conflicts may be more difficult to resolve with the tighter constraints on a larger ts, we note that the tolerances only restrict when the sample of input m must be taken, not when output c must be updated, which is specified by the response allowance.

Intuitively, when tolerances are allowed on the sample time (non-zero jitter), it is more difficult to detect sustained conditions of longer duration with the same precision. E.g., as the duration changes from [200-50, 200+60] to [300-50, 300+60] to [400-50, 400+60] in Fig. 11, the available sample times in [50,110] are first significantly reduced then completely eliminated.

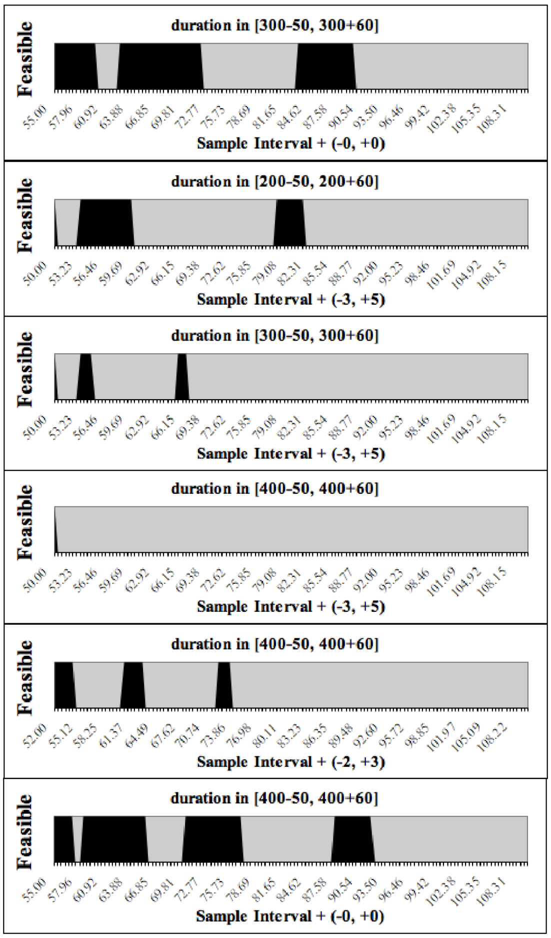

Fig. 11. Feasible Sample Intervals for Various Durations and Tolerances

Sample Intervals for Events That Depend on Controlled Variables: In this case the uncertainty as to when the event occurred has been removed. Thus, any sample interval less than or equal to $(\delta L + \delta R)$ suffices since we need only a single sample point in the interval. Smaller sample intervals allow us to define the boundaries of the interval more precisely, but any point in the interval satisfies the requirement. However, larger sample intervals are also possible. For instance, sample intervals in the range $d - \delta L \leq ts \leq d + \delta R$ also work, giving us two sample points, one at the start of the event and another in the desired interval. We have called these "sample points", however, it is more accurate to term them "evaluation points", since no monitored variable is sampled, the current value of a controlled variable is simply used in a function evaluation. We can see therefore, that this kind of sustained event is affected by specified RAs rather than TRs. This is discussed further in Section 4.2.

Specifying Timing Resolution Affected by Sustained Events: We have presented analyses that show how sample intervals must be restricted to be able to implement sustained events. Since timing resolution specifies a maximum sample interval for time continuous monitored variables, it is clear that sustained events may affect the timing resolution we must specify for monitored variables involved in those events.

The restrictions imposed on the sample intervals however, are not enforceable by specifying a more or less restrictive timing resolution. If we look at Cases 1, 2 and 3 for the sustained events dependent on monitored variables, we see that all sample intervals in Case 1 are feasible, there are disjoint ranges of feasible sample intervals for specific events for Case 2, and no feasible sample intervals for Case 3. We can specify a timing resolution of $(\delta L + \delta R)$ since we know that anything greater than that results in an infeasible implementation, but in fact, this is not sufficient. One way of dealing with this problem is to shift the responsibility of determining exactly what sample intervals are feasible to the software design phase.

In the case of sustained events that are not dependent directly on monitored variables, timing resolution is not an issue since monitored variables cannot directly affect the event.

4.2 Response Allowance for Sustained Events

There are two specific concerns related to specifying the response allowance for variables involved in sustained events. The first is a general one. How do we cope with specifying the RA for a sustained event so that it is clear what the requirement allowance is for both the successful continuation of a sustained event, as well as the cancellation of a sustained event. The second concern is what restrictions must be placed on RAs so that the sustained event can be implemented.

1. We begin by assuming that monitored variable m and controlled variable c are involved in the sustained event. If the functional requirement does not involve a sustained event the RA is based on a physical analysis of the required behaviour. We call this ra_{c-m}. This is a suitable RA to use for the case when the sustained event is canceled. In other words, given a sustained event specified by $(Condition)\ Held\ for\ (d,\ \delta L,\ \delta R)$, if $Condition$ changes from True to False, the application must generate the value of c within ra_{c-m} measured from the time the event occurred in the physical domain. Now what if the sustained event is successful? We know that we have $d + \delta R$ within which to determine that fact (measured from the initiation of the event in the physical domain). We also have some time in which to calculate the value of c. The problem is that we do not know how much of ra_{c-m} to add to $d + \delta R$. One solution is to add the entire ra_{c-m}, in spite of the fact that this "double counts" any portion of ra_{c-m} that was allocated to detecting the event. We are examining alternative strategies but this is the best we have to date. Thus, the RA for sustained events is specified in the form: $ra_{true}\ Held\ for\ (d,\ \delta L,\ \delta R)/ra_{false}$. This is interpreted as specifying a response allowance of ra_{true} when the

event continues to completion (because of that "Held for" event), and ra_{false} when the event is canceled. Example: an RA of 250 ms is specified for a *c-m* pair, and the event "$(f\_sentrip = e\_trip)$ *Held for* (k_delay)" also involves that pair, where $k\_delay = 500\ ms \pm 25\ ms$. ra_{c-m} is documented as: 775 *ms Held for* (k_delay) / 250 *ms*. We could specify ra_{c-m} simply by 775 ms/250 ms, but the *Held for* (k_delay) provides useful information.

2. In the case of a sustained event that depends on controlled variables, we saw earlier that it is relatively easy to arrange that at least one evaluation point lies in the interval of interest. Since the evaluation depends on the previous value of a controlled value, the RA for that variable serves the same purpose as the timing resolution does for monitored variables. Thus, to ensure that a "fresh" value of the controlled variable is used in the evaluation, we specify that the RA for that controlled variable must be no larger than $d + \delta R$. Of course, it may already have been specified to be more restrictive than that by the domain experts. In such cases the more restrictive value is used.

5 Conclusion

We have presented precise definitions for timing requirements that include tolerances on the time durations. Our analysis, based on these definitions, shows that it is possible to specify and verify critical timing requirements using simple mathematics that is accessible to both software engineers and domain experts. These definitions and related analyses can form the basis of a comprehensive, practical approach to specifying timing requirements in high reliability real-time and embedded systems.

In many safety-critical applications, when operating at the limits of the available hardware, sampling faster is simply not an option. Thus in order to meet all system deadlines, we may be forced into a situation where $ts\_max > \frac{1}{2}(\delta L + \delta R)$ for a given requirement. We have shown that it is still possible to find implementable sampling intervals that satisfy the relevant timing requirements. Our analysis also demonstrates that even low jitter in the sampling can prevent our being able to design an implementation that satisfies its timing requirements.

Acknowledgments

The work presented in this paper is based on the efforts of many current and former employees and consultants of Ontario Power Generation Inc., and AECL, including: Glenn Archinoff, Dominic Chan, Peter Froebel, Rick Hohendorf, David Lau, Jeff McDougall, Greg Moum, Mike Viola, and Alanna Wong. We also acknowledge and thank David Parnas. This work reflects the successful application of many of his pioneering and fundamental ideas regarding software engineering.

References

1. Wang, F.: Formal verification of timed systems: A survey and perspective. Proceedings of the IEEE **92** (2004) 1283–1307
2. Wulf, M.D., Doyen, L., Raskin, J.F.: Almost asap semantics: From timed models to timed implementations. In: HSCC04. Vol. 2993 of LNCS. (2004) 296–310
3. Wulf, M.D., Doyen, L., Markey, N., Raskin, J.F.: Robustness and implementability of timed automata. In: FORMATS04,. Vol. 3253 of LNCS., Grenoble (2004) 152–166
4. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Transactions on Programming Languages and Systems **16** (1994) 1543–1571
5. Shankar, N.: Verification of real-time systems using PVS. In Courcoubetis, C., ed.: CAV '93. Vol. 697 of LNCS., Elounda, Greece, Springer-Verlag (1993) 280–291
6. Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming **25** (1995) 41–61
7. Janicki, R., Khédri, R.: On a formal semantics of tabular expressions. Science of Computer Programming **39** (2001) 189–213
8. Wassyng, A., Janicki, R.: Using tabular expressions. In: Int. Conf. on Software and Systems Engineering and their Applications. Vol. 4., Paris (2003) 1–17
9. Wassyng, A., Lawford, M.: Lessons learned from a successful implementation of formal methods in an industrial project. In Araki, K., Gnesi, S., Mandrioli, D., eds.: FME 2003. Vol. 2805 of LNCS., Springer-Verlag (2003) 133–153