Formal Verification of Implementability of Timing Requirements

Xiayong Hu and Mark Lawford* and Alan Wassyng*

Software Quality Research Laboratory, Department of Computing and Software
McMaster University, Hamilton, Canada L8S 4K1
huxy@mcmaster.ca, lawford@mcmaster.ca, wassyng@mcmaster.ca

Abstract. While there has been a large amount of work on validation of timing requirements, there has been relatively little work on the implementability of timing requirements. We have previously provided definitions of fundamental timing operators that explicitly considered tolerances on property durations and intersample jitter [1]. In this work we refine the model and formalize the analysis of the *Held for* operator of [1] in the PVS theorem prover. We formalize different implementation environments incorporating nonuniform samples with bounded jitter and for one of the environments provide a full formal proof of necessary and sufficient conditions for when it is possible to implement a *Held for* requirement. We briefly describe how these results can then be used to formally verify a sample module implementing a requirement that utilizes the *Held for* operator and describe an example application.

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. The extensive survey of formal methods for the specification and verification of real-time systems in [2] 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.

Implicit in many of the formal models of timing requirements is the assumption that the real-time system implementing the timing requirements continuously monitors its inputs and can instantaneously react to the occurrence of an "event" (a significant change in the inputs). Due to their clock driven nature, computer control systems must typically sample some set of inputs and then update a set of outputs. Models that consider the sampling required for a computer controlled implementation of system requirements will often make the simplifying assumption that all samples are uniformly spaced and sufficiently fast to guarantee system response.

* Partially supported by the Natural Sciences and Engineering Research Council of Canada

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.

Motivated by our work on the Darlington Nuclear Generating Station Shutdown Systems software redesign project [3] and the difficulties and effort involved with the verification of timing requirements on that project, we began studying timing requirements with tolerances. In [1] we justified use of several different types of tolerances that must be fully specified at the requirements level in order to properly deal with the timing tolerances that are inherent in the system implementation. These included tolerances on *functional timing requirements*, and tolerances on *performance timing requirements* that allow for deviation from the idealized behaviour specified by the requirements models. By modeling these requirements, we were able to come to a somewhat surprising result that allows some timing requirements to be verifiably implemented at a significantly lower CPU bandwidth than normally assumed.

In this work we refine the model and formalize the analysis of the *Held for* with tolerance operator of [1] in the PVS[1] theorem prover. We identify different clock information assumptions for implementations, showing how changing the order of quantifiers in a high order logic formula can capture what is required to implement the *Held for* requirement under each of the clock information assumptions. For the "perfect clock at sample times case" that was implicit in the manual analysis of [1], we provide a full formal proof of necessary and sufficient conditions for when it is possible to implement a requirement that a boolean condition has been sustained for a duration d within a tolerance of $[d - \delta_L, d + \delta_R]$ with a discrete implementation with nonuniformly spaced samples where the intersample spacing is bounded by minimum and maximum sample intervals. As a result of the formalization in PVS, we discovered a missing boundary case in the original theorem statement of [1] (see section 4 for details). Similar to the original theorem statement of [1], the fixed theorem statement provides an easy to evaluate formula relating the tolerances on duration to the tolerances on intersample spacing that must be satisfied for the requirement to be implementable when the boolean condition is filtered so that it changes state no more than once per sample interval.

At the requirements level we assume that inputs are sampled arbitrarily fast while at the implementation level sampling occurs at most once every T_{min} time units and at least once every T_{max} time units. We provide an intermediate representation of the *Held for* requirement on the implementation's sampled signals that we use to prove a code level implementation model of the *Held for* requirement correct via a two step process. First we show that the implementation is equivalent to the intermediate representation on the samples and then show that the intermediate representation is within tolerance allowances of the arbitrarily fast sampling requirements model.

Once the implementation has been verified in PVS, to prove any specific requirement is correctly implemented by the implementation we only have to

[1] PVS files available at http://www.cas.mcmaster.ca/~lawford/papers/FM08.html

instantiate the PVS theorems with appropriate values and then the verification is reduced to a standard untimed verification on the remainder of the requirement's functionality. We demonstrate this process with a simple *Sensor Lock* example in section 6.

1.1 Related Work

Recent work has begun to address the issue of timing tolerances required to verify implementations of requirements modeled as timed automata with ASAP semantics [4,5]. 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 [5], while [4] 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. They further show 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 our work, response allowance ra, and sample interval ts, correspond most closely to Δ and Δ_L in [4]. 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 [1]. We derive tolerance bounds on ts for the correct detection of a sustained condition.

The temporal logic of actions (TLA) in [6] reasons about safety properties of a real-time system from a concurrent perspective. To specify the timing requirements, Lamport *et al.* use the TLA formula $VTimer(t, A, \delta, v)$ to control the occurrence of the step A until it has been enabled for δ time units and $MaxTime(t)$ to set the upper-bound of the *timer* t. Conjoining these two formulas, one can interpret the timing requirements that a step A (an action from one state to another) must occur if A has been continuously enabled for δ long time.

The *Held for* operator defined in [1] and restated in Section 2 serves a similar purpose to the $|P|$, "since P" operator in the dense time setting of [7] where system actions can occur at any positive rational number. Both operators do not require the introduction of explicit counter variables. Saying "do X when time since P was true is greater than 5 seconds", can be phrased as "do X when $NOTP$ has *Held for* 5 seconds". The $|P|$ operator is more expressive since it returns the exact time since P was last TRUE. We note that in a sampled system this information is not typically available. On the other hand the P *Held for* duration operator returns a value of TRUE when P has been TRUE at all time instances in a duration within tolerance of a nominal value. We provide

conditions under which this requirement is implementable via a sampling implementation. The theory developed in [7] is used to verify invariants for Fischer's mutual exclusion protocol and a railroad crossing example. The definitions described in this paper have been used on safety critical industrial process control applications and we are now adding some mechanized support.

Giotto is an embedded software model that also focuses on the implementation [8] in a manner that is independent of the execution platform, but that is much closer to executable code than a mathematical model. The code generation task of Giotto is partitioned into two steps. In the first step, program generation, a given mathematical model is transformed into an embedded software model, which is entirely independent of any execution platform. In the second step, compilation, the software model is transformed into executable code for a target platform. Program generation specifies only the reactivity of the system relative to a physical environment, while compilation ensures the schedulability of the system in a specific execution environment.

Giotto simplifies the jitter and tolerance by a fixed logical execution time (FLET) assumption in Giotto semantics. In Giotto, the logical execution time of a task is always exactly the period of the task (i.e., the period of the surrounding case block), and the logical execution times of all other activities (switch blocks, data transfer across links, etc.) are always zero. This leads to two important consequences. First, sensors are read only at the beginning of a task's period, and actuators are updated only at the end of a task's period, which minimizes the jitter but also means Giotto does not consider clock sampling jitter and timing tolerances. Second, all inter-task (data) communication happens at period boundaries, which also means the data provided from one task to another will not be refreshed until the source task produces outputs after the current hyperperiod. However, this assumption has the potential to cause "stale data" if the execution time of the task takes too long to output the data.

The assumption of zero-time for computational action in the model language is impossible to ensure on the target platform in the implementation language [9]. Thus the predictable design approach introduced an ϵ-hypothesis to fill the gap between the physical domain and the software domain [10]. This ϵ-hypothesis requires the model and its realization to have the same observable execution sequence. Also, time deviations between activations of corresponding actions in the model and realization should be less than ϵ seconds. In the predictable design approach, the generation tool called Rotalumis takes the model coded with model language POOSL, and automatically generates the executable for the target platform. We note that this hypothesis is very close to what we call "response allowance" (see [1]), one of our *performance timing requirements*, which 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. Our research also covers the tolerances in timing requirements when crossing from the physical domain to the software domain, which is not discussed in [9,10].

The summarized research above is focused on connecting the requirements and implementation. We notice that we can categorize most of the current approaches into two trends: platform-independent approach and global tolerance approach. Most research based on the platform-independent idea will plug in another layer between the high level requirements and coding implementation, e.g, “program generation” in Giotto approach [8] and POOSL model in [9,10]. These approaches cannot tell us whether a system can be implemented on a target platform or not until the final scheduling stage is finished. In the case of the generation of an unimplementable result, the designer has to improve the hardware performance or relax the timing requirements, both of which are problematic.

The approaches with global tolerances (e.g, reaction delay parameter Δ in [4] and ϵ-hypothesis in [10]) all define a global constraint as the constant upper bound of the delay during implementation. The benefit of a single global tolerance is clear. It is easy to analyze since it simplifies the problem. However, in most industrial applications, it is clear that it is advantageous to define different tolerances on different timing components of the system. This may allow us to build implementations on hardware that would not allow us to meet all our timing requirements if we had to meet a global timing requirement. Instead, for more stringent requirements, a piece of code (implementing the stringent timing behaviour) may appear several times in a cyclic loop, while behaviour with less stringent timing requirements would appear just once. The results we have so far (see Section 2.1) allow one to consider things like jitter associated with such a stringent requirement being implemented by repeated calls within a cyclic executive.

The remainder of this paper is organized as follows: Section 2 provides the notation and definitions of terms and the *Held for* operator from [1]. Section 3 describes the different assumptions that can be made about the timing information available to the implementation. The results of the PVS formalization of the *Perfect Clock* implementation assumption are given in Section 4, while Sections 5 and 6 provide an example implementation and its application to meet a specification involving a *functional timing requirement*, respectively. Conclusions and future work are discussed in Section 7.

2 Preliminaries

Parts of this section, including Figs. 1, 2 and 3 are repeated from [1].

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. With-

out tolerances on the time duration, these requirements would be impossible to meet.

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 as shown in Fig. 1. Let us assume that the sample intervals are ts_0, ts_1, ts_2, etc.

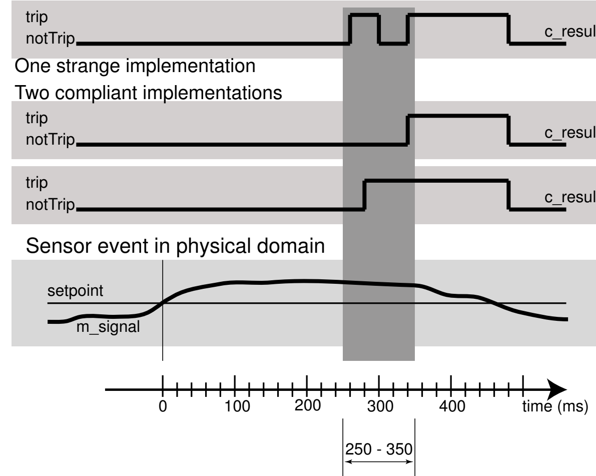

Fig. 1. Two Valid Implementations of a Sustained Timing Requirement

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 can call these limits T_{min} and T_{max}. Once we have these limits, we know that $T_{min} \leq ts_j \leq T_{max}$ for each $j \in \{0..n\}$. Let $\mathit{Sample} : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$ be a sequence of sample times such that $\forall n : T_{min} \leq \mathit{Sample}(n+1) - \mathit{Sample}(n) \leq T_{max}$. We can assume $\mathit{Sample}(0) = 0$ and then for $n \geq 1$, $\mathit{Sample}(n) = \sum_{i=1}^{n} ts_i$ in Figure 2 and $\forall i : T_{min} \leq ts_i \leq T_{max}$.

2.1 Results from Earlier Work

Here we summarize the results we need from [1].

Fig. 3 graphically shows the essential components and behaviour of the *Held for* operator.

The formal definition of *Held for* is given in Fig. 4.

Using the formal *Held for*, we can now formally capture the desired behaviour of the m-c pair shown in Fig. 1 by the tabular expression in Fig. 5.

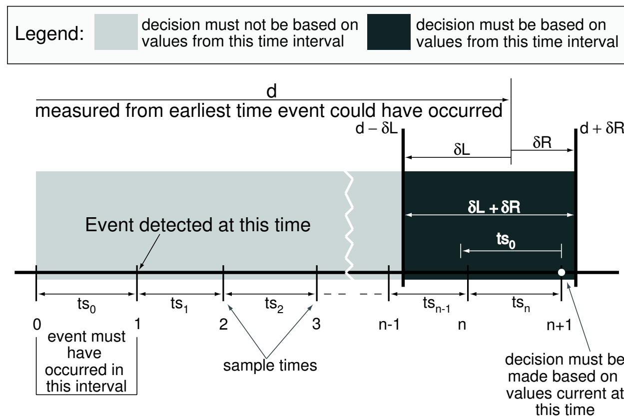

Fig. 2. Sample Intervals Required for Sustained Events

The implementability conditions for *Held for* are:

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

Case 2: $\frac{1}{2}(\delta L + \delta R) < T_{max} \leq (\delta L + \delta R)$ Let $k_{min} = \lfloor \frac{d - \delta L}{T_{max}} \rfloor$, and $k_{max} = \lfloor \frac{d - \delta L}{T_{min}} \rfloor$. Then $k_{min} = k_{max}$ is a necessary condition for a feasible implementation. A sufficient condition when $k = k_{min} = k_{max}$ is that $(k+2) \cdot T_{max} \leq d + \delta R$.

Case 3: $(\delta L + \delta R) < T_{max}$ The sustained event cannot be implemented.

At this stage we should note that in [1] we implicitly considered only the *Perfect Clock* case as described in Section 3.

3 Assumptions about the Implementation Environment

For the purposes of this paper we are ignoring response allowance, the computation time that the system is allowed to take to recognize and respond to a situation. For this work we are merely determining when it is possible to detect a *Held for* requirement with functional timing tolerance if the implementation has non-uniform sampling. In [1] we treated the case of specifying a response allowance separately from functional timing requirements. In any case, if we are unable to even detect a sustained condition within the tolerance specified in the requirements then meeting computational time bounds specified as response allowances is irrelevant. We hope to formalize the relationship between functional

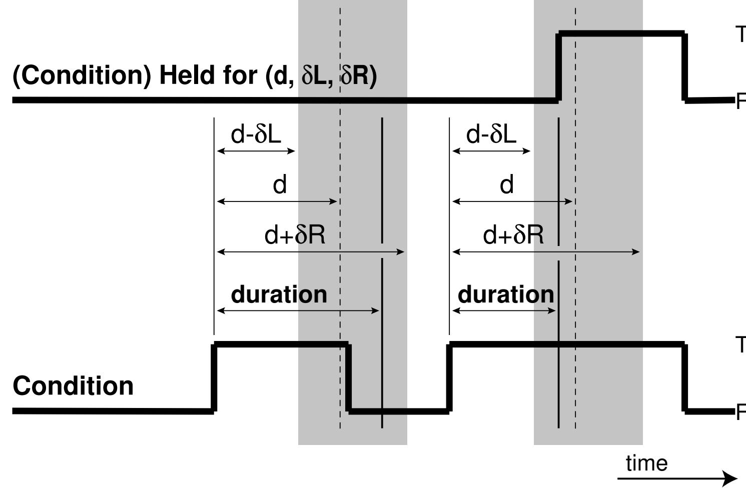

Fig. 3. *Held for* Functional Timing Requirement

timing requirements and response allowance in the future but it is beyond the scope of the current work.

There are several different implementation environment assumptions governing how we might have to recognize a sustained event. These are:

Omniscient: We know the exact time of the "event" t when the condition becomes true but can only react at sample times.

In order to implement the *Held for* requirement in this case we must consider every possible sequence of sample times, *Sample*, satisfying the bounded jitter constraint $\forall n : Sample(n+1) - Sample(n) \in [T_{min}, T_{max}]$. The event could take place at any point t between successive samples $Sample(n_0)$ and $Sample(n_0+1)$. Then there must be a sample point in the future, $Sample(n)$, when we can react such that *Held for* has been true for at least $d - \delta_L$ and no more than $d + \delta_R$, i.e.:

$$d - \delta_L \leq Sample(n) - t \leq d + \delta_R.$$

Putting this all together we get the higher order logic formula:

$$\begin{aligned} \forall Sample : \forall n_0 : \forall (t | Sample(n_0) \leq t \leq Sample(n_0+1)) : \exists n : \\ d - \delta_L \leq Sample(n) - t \leq d + \delta_R \end{aligned} \quad (1)$$

Perfect Clock: We know the value of the condition only at sample instances but we know the exact timing of samples (i.e. we can look at a perfect real valued clock).

To determine if we can implement the *Held for* requirement in this case we again need to consider every possible sequence of sample times. The main

(Condition :bool) *Held for* (d: $\mathbb{R}^{>0}$, δL, $\delta R : \mathbb{R}^{\geq 0}$) :bool
where duration(Condition: bool): [d $- \delta L$, 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)"

	c_result
(m_signal $\geq$ setpoint) Held for (300$\pm$50)	trip
NOT [(m_signal $\geq$ setpoint) Held for (300$\pm$50)]	notTrip

Fig. 5. Tabular expression formalizing desired behaviors in Fig. 1

difference now is that our future decision point at *Sample*(n) must work for any possible event time between *Sample*(n_0) and *Sample*($n_0 + 1$), i.e. we exchange the order of the last two quantifiers to obtain:

$$\begin{aligned} &\forall Sample : \forall n_0 : \exists n : \forall (t | Sample(n_0) \leq t \leq Sample(n_0 + 1)) : \\ &\quad d - \delta_L \leq Sample(n) - t \leq d + \delta_R \end{aligned} \tag{2}$$

Imperfect Clock: This situation is the same as in (2) but with access to an imperfect clock (e.g. finite precision, bounded drift, etc).

To precisely specify this situation we need to first model the imperfect clock and then account for it in choosing our decision point. We leave the formalization of the various possible subcases associated with different imperfect clock assumptions for consideration in future work.

No Clock: Knowledge only of T_{min} and T_{max} and the number of samples since the condition became true.

In this case we have no recourse in our implementation but to simply count the number of samples since we first detected the event. In this case we need "count" value n that will work under any possible bounded sample spacing

and actual time of event occurrence. Thus we have:

$$\exists n : \forall Sample : \forall n_0 : \forall (t | Sample(n_0) \leq t \leq Sample(n_0+1)) : \\ d - \delta_L \leq Sample(n_0+n) - t \leq d + \delta_R \tag{3}$$

By formalizing the conditions that must be satisfied to detect the *Held for* with tolerance requirement we have eliminated a significant source of ambiguity in interpreting the results and understanding their applicability to a particular situation. The formalization of the system implementation environment

4 Results for Perfect Clock Case

Below we define the predicate $Feasible(d)$, restating the condition for implementability in the *Perfect Clock* case given in (2) as a function of the sustained condition's nominal duration, d, assuming that the other parameters - δ_L, δ_R, T_{min} and T_{max} are fixed.

$$Feasible(d) : bool = \forall Sample : \forall n_0 : \exists n : \forall (t | Sample(n_0) \leq t \leq Sample(n_0+1)) : \\ d - \delta_L \leq Sample(n) - t <= d + \delta_R$$

We assumed that we would be able to prove the implementability results from Section 2.1 in PVS once we had formalized the implementation environment. We were able to prove Case 1 and Case 3 with relative ease but in the most interesting case, Case 2, when $\frac{1}{2}(\delta L+\delta R) < T_{max} \leq (\delta L+\delta R)$, we were unable to prove that $Feasible(d)$ was equivalent to $K_{min} = K_{max}$. In particular, the proof failed at the boundary case when T_{min} is reduced to the point where K_{max} first becomes greater than K_{min}, that is when T_{min} is an integer multiple of $d - \delta_L$, i.e.

$$\lfloor \frac{d - \delta_L}{T_{min}} \rfloor * T_{min} = d - \delta_L$$

In this case

$$T_{min} = \frac{d - \delta_L}{K_{min} + 1}. \tag{4}$$

In all cases when $K_{min} = K_{max}$ then

$$\begin{aligned} \frac{d - \delta_L}{K_{min} + 1} &= \frac{d - \delta_L}{K_{max} + 1} \\ &= \frac{d - \delta_L}{\lfloor \frac{d - \delta_L}{T_{min}} \rfloor + 1} \\ &< \frac{d - \delta_L}{\frac{d - \delta_L}{T_{min}}}, \text{ since } \lfloor \frac{d - \delta_L}{T_{min} + 1} \rfloor > \frac{d - \delta_L}{T_{min}} \\ &= T_{min} \end{aligned}$$

Combining (4) with the above provides us with a new condition that combines the $K_{min} = K_{max}$ case with the missing boundary case, giving:

$$T_{min} \geq \frac{d - \delta_L}{K_{min} + 1} \tag{5}$$

A counter example corresponding to this missing boundary case is illustrated in Fig. 6. For this particular example we take $T_{min} = 10, T_{max} = 12, d = 25, \delta_L =$

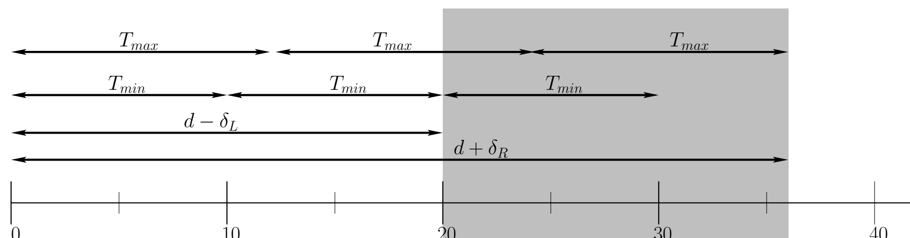

Fig. 6. Example of boundary case missed in [1]

$5, \delta_R = 11$. Then $d - \delta_L = 20, d + \delta_R = 36$ and

$$K_{min} = \lfloor (d - delta_L)/T_{max} \rfloor = \lfloor 20/12 \rfloor = 1$$
$$K_{max} = \lfloor (d - delta_L)/T_{min} \rfloor = \lfloor 20/10 \rfloor = 2$$

Thus $K_{min} \neq K_{max}$ but *Feasible*(d) can be shown to be true in PVS. We can intuitively see from Fig. 6 why this might be the case.

If the condition becomes true between 0-10 time units, we can always wait until the third sample to make our decision. The top sequence shows the maximum spacing for samples while the second row show shows the minimum spacing. All other sampling sequences would fall somewhere in between these two extremes. But even if the condition became true at some time point t just before 10 time units $30 - t \geq 30 - 10 = d - \delta_L$, so elapsed time is not less than $d - \delta_L$. On the other hand, if t occurred just after time 0, then $36 - t \leq 36 - 0 = d + \delta_R$, so elapsed time is not greater than $d + \delta_R$.

Replacing the $K_{min} = K_{max}$ condition in Case 2 of the main result of [1] with (5), we obtain a new result which has been proven in PVS.

Theorem 1. *Assume that* $T_{min} < T_{max}$*,* $\delta_L, \delta_R > 0$*, and* $d > \max(\delta_R, T_{max} + \delta_L)$*. Let* $K_{min} = \lfloor (d - delta_L)/T_{max} \rfloor$*. Then*

Case 1 *: If* $T_{max} \leq \frac{\delta_L + \delta_R}{2}$ *then Feasible*(d)
Case 2 *:* $\frac{\delta_L + \delta_R}{2} < T_{max} \leq \delta_L + \delta_R$ *then*

$$T_{min} \geq \frac{d - \delta_L}{K_{min} + 1} \wedge (K_{min} + 2) * T_{max} <= d + \delta_R \Leftrightarrow \mathit{Feasible}(d)$$

Case 3 *:* $T_{max} > \delta_L + \delta_R$ *then* $\neg$*Feasible*(d)

5 One Implementation of *Held for* with Tolerance

In the implementation environment corresponding to the *Perfect Clock* case of Section 3, a simple way to implement the *Held for* operator is to have a timer that you update by the time between the last sample and the current sample when the input condition CurrentP is true until it exceeds $d - \delta_L$. We can then define a Timer() function that calls the TimerUpdate function at each sample instance with appropriate arguments.

TimerUpdate(CurrentP, TimeOut, previous, step): tick =
TABLE

	previous < TimeOut	previous ≥ TimeOut
CurrentP	previous + step	previous
¬ CurrentP	0	0

ENDTABLE

Held_For_I(P, duration, Sample)(ne): bool =
$\exists$ (n_0: nat | Sample(ne) − Sample(n_0) $\geq$ duration − delta_L):
$\forall$ (n: nat | $n_0 \leq n \wedge n \leq$ ne): P(Sample(n))

Timer_General: THEOREM
Held_For_I(P, timeout, Sample)($n+1$) $\equiv$
Timer(P, Sample, timeout $- \delta_L$)(n) + Sample($n+1$) − Sample(n) $\geq$ timeout $- \delta_L$

Fig. 7. Timer update function to implement *Held for*

5.1 Verifying the Implementation's Correctness

We perform the verification via a two step process. First we show that the module is equivalent to the following intermediate representation of the *Held for* requirement that only considers the values of the condition at sample times. We then show that under a mild filtering condition on the sampled input signal, the intermediate *Held for* representation is equivalent to the original requirements level *Held for* operator with its arbitrarily fast clock tick.

The Timer_General theorem in Fig. 7 represents the first part of the proof. Below we provide the filtering assumption that is required to connect the sampled intermediate representation of *Held for* with the arbitrarily small clock "tick" version of the requirements given in Section 2.1. For this we make use of predicate subtypes.

$$\text{FilteredTickPred} : \text{TYPE}+ = \{P : [tick \to bool] | \forall t_j, t, t_n :$$
$$t_j < t < t_n \land P(t_j) \neq P(t) \Rightarrow$$
$$P(t) = P(t_n) \lor t_n - t_j > T_{max}\}$$

Pf: VAR FilteredTickPred

We can then show that there exists a duration in the tolerance range such that requirements *Held for* is true when the intermediate version Held_For_I is true.

6 Example

The Sensor Lock System is a watchdog control system. As shown in Figure 8, it monitors the plant parameter *Sensor* and reacts to send the output "lock" to shutdown the system only if anomalous behavior is observed for plant parameter *Sensor* for an extended period of time [11,12]. Once the Sensor Lock System produces a "channel lock" to force the shutdown of the plant, the channel will not be "unlocked" until the manual reset button is pushed.

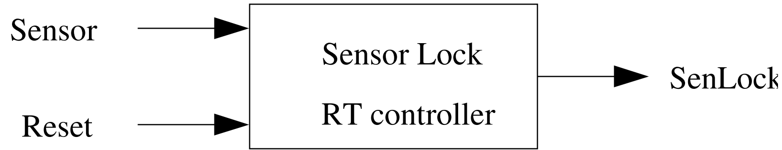

Fig. 8. Block diagram for real-time Sensor lock controller

When the value of *Sensor* is continuously $TRUE$ for *ldelay* time units or longer, the sensor is "locked" and *SenLock* is set to $TRUE$. Once the sensor is "locked", it stays locked until the system is manually reset by making $Reset = TRUE$. As a step further to the work in [11,12], the tolerance δL and δR are introduced to the variable *ldelay*.

The Held_For_I operator defined in section 2.1 is used to construct the real-time system requirements. The Software Requirement Specification (SRS) of the Sensor Lock System is shown in Fig. 9. The first row of the SRS table indicates that SenLock should be $TRUE$ if sensor has been $TRUE$ for *ldelay* time units or longer. The remaining three rows indicate that only a manual reset in a safe sensor input situation can make SenLock $FALSE$.

The Software Design Description (SDD) is shown in Fig. 10. We applied ElockUpdate function to handle the control logic and the TimerUpdate function in section 5 to plug in the timing behaviour implementation of Held_For_I operator (the function TimerUpdate is listed with parameters *previous* and *step*

<table>
<tr><td colspan="3"></td><td>Result</td></tr>
<tr><td colspan="3">Condition</td><td>SenLock</td></tr>
<tr><td colspan="3">(Sensor) Held_For_I (ldelay,δ_L,δ_R)</td><td>True</td></tr>
<tr><td rowspan="3">NOT [(Sensor) Held _For_I (ldelay,δ_L,δ_R)]</td><td rowspan="2">Reset</td><td>¬Sensor</td><td>False</td></tr>
<tr><td>Sensor</td><td>No Change</td></tr>
<tr><td colspan="2">¬Reset</td><td>No Change</td></tr>
</table>

Fig. 9. The SRS of SenLock System (with tolerance)

due to the space). The variable $lastSamplePeriod$ indicates the time frame between the current sample and previous sample. With the Timer_General theorem pre-verified (section 5), the final verification is completed painlessly.

<table>
<tr><td colspan="2"></td><td colspan="2">Results</td></tr>
<tr><td colspan="2">Condition</td><td>Elock</td><td>lLockDly</td></tr>
<tr><td colspan="2">n=0</td><td>Lock</td><td>TimerUpdate(0,0)</td></tr>
<tr><td rowspan="2">¬ n=0</td><td>¬pre(sensor)</td><td>ElockUpdate(sensor,reset)</td><td>TimerUpdate(pre(lLockDly),0)</td></tr>
<tr><td>pre(sensor)</td><td>ElockUpdate(sensor,reset)</td><td>TimerUpdate(pre(lLockDly),lastSamplePeriod)</td></tr>
</table>

Fig. 10. The SDD of SenLock System (using Timer and Elock Update functions)

7 Conclusions

Timing behaviour is clearly of crucial concern in hard real-time systems. Real, industrial systems require that we explicitly consider tolerances in this behaviour. There are currently few attempts to include tolerances in all aspects of timing behaviour in hard real-time systems.

Within the timed automata formalism, [13] provides a sufficient condition for implementability of the timed automata in terms of a global upper bound on the system latency.

In our previous work [1], we laid the foundations for a comprehensive approach to this problem. In comparison with [13], we provide a necessary and sufficient condition for the implementability of the basic real-time sustained condition requirement in terms of tolerance on the duration, the sample rate and the jitter. Our theory allows for per requirement tolerances to be specified and verified rather than requiring all parts of the implementation meet some global minimum response time.

This current paper is both a verification of that work and an extension of it. Using PVS we have verified that the feasibility conditions that we developed for implementation of a *Held for* requirement are correct. An important and interesting side-effect of that verification is that we discovered an example that

shows just how useful a theorem prover like PVS can be. We could not confirm that our conditions were both necessary and sufficient - for the simple reason that they were not. We had missed a feasible range that was uncovered by PVS. This range is not practically useful, but is mathematically viable. PVS made sure we did not miss it. We have managed to use the ideas from our PVS proofs to construct a PVS template that provides a pre-verified implementation of a *Held for* requirement.

7.1 Future Work

We plan to complete the work of formalizing the feasibility condition of the *Held for* operator with tolerance under different imperfect clock implementation assumptions. We will then extend the analysis of Section 4 to other clock information assumptions. We already have preliminary results for the "No Clock" case. Next we will work on formally incorporating response allowance by explicitly considering non-zero computation time. The variants of the periodic operator of [1] and other fundamental timing operators can be treated in a similar manner. With the analysis of the operators in place we can then work on using them to develop a library of pre-verified real-time components that can be validated and refined by using them on further application examples.

References

1. Wassyng, A., Lawford, M., Hu, X.: Timing tolerances in safety-critical software. In Fitzgerald, J., Hayes, I., Tarlecki, A., eds.: FM 2005: Formal Methods: International Symposium of Formal Methods Europe Proceedings. Volume 3582 of LNCS., Newcastle, UK, Springer-Verlag (2005) 157 – 172
2. Wang, F.: Formal verification of timed systems: A survey and perspective. Proceedings of the IEEE **92**(8) (2004) 1283 – 1307
3. 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: International Symposium of Formal Methods Europe Proceedings. Volume 2805 of Lecture Notes in Computer Science., Pisa, Italy, Springer-Verlag (2003) 133–153
4. De Wulf, M., Doyen, L., Raskin, J.F.: Almost asap semantics: From timed models to timed implementations. In: In the Proc. of HSCC04. Volume 2993 of Lecture Notes in Computer Science. (2004) 296 – 310
5. De Wulf, M., Doyen, L., Markey, N., Raskin, J.F.: Robustness and implementability of timed automata. In: Proc. of FORMATS04,. Volume 3253 of Lecture Notes in Computer Science., Grenoble (2004) 152–166
6. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Transactions on Programming Languages and Systems **16**(5) (1994) 1543–1571
7. Shankar, N.: Verification of real-time systems using PVS. In Courcoubetis, C., ed.: Computer-Aided Verification, CAV '93. Volume 697 of Lecture Notes in Computer Science., Elounda, Greece, Springer-Verlag (1993) 280–291
8. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A., Pree, W.: From control models to real-time code using giotto. In: Proceedings of the Second International Workshop on Embedded Software, Lecture Notes in Computer Science, Springer-Verlag (2002)

9. Florescu, O., Voeten, J., Huang, J., corporaal, H.: Error estimation in model-driven development for real-time software. In: Forum on specification and Design Languages. (2004) 228–239
10. Huang, J., Voeten, J., Florescu, O., van der Putten, P., Corporaal, H.: Predictability in real-time system development. In: Advances in Design and Specification Languages for SoCs, Kluwer Academic Publishers (2005)
11. Lawford, M., Wu, H.: Verification of real-time control software using pvs. In Ramadge, P., Verdu, S., eds.: Proceedings of the 2000 Conference on Information Sciences and Systems. Volume 2., Princeton, NJ, Dept. of Electrical Engineering, Princeton University (2000) TP1–13–TP1–17
12. Wu, H.: Formal verification of real-time software. Master's thesis, McMaster University (2001)
13. De Wulf, M., Doyen, L., Raskin, J.F.: Systematic implementation of real-time models. In Fitzgerald, J., Hayes, I., Tarlecki, A., eds.: FM 2005: Formal Methods: International Symposium of Formal Methods Europe Proceedings. Volume 3582 of LNCS., Newcastle, UK, Springer-Verlag (2005) 139 – 156