Formal Verification of the Implementability of Timing Requirements

Xiayong Hu, 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. 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. In this work we identify three environmental assumptions and compare the implementability of a *Held_For* operator in each of them, formalizing this analysis in PVS. We show how to design a software component that implements the *Held_For* operator and then verify it in PVS. This pre-verified component is then used to guide the design of more complex components and to decompose their design verification into simple inductive proofs as demonstrated through the implementation of a timing requirement for 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 [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.

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.

* Supported by the Natural Sciences and Engineering Research Council of Canada.

D. Cofer and A. Fantechi (Eds.): FMICS 2008, LNCS 5596, pp. 119–134, 2009.

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 [2] and the difficulties and effort involved with the verification of timing requirements on that project, we began studying timing requirements with tolerances. In [3] we justified the 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* (FTRs), and tolerances on *performance timing requirements* (PTRs) that allow for deviation from the idealized behaviour specified by the requirements models. By modeling these requirements, we presented *Implementability Results* which allow some timing requirements to be verifiably implemented at a significantly lower CPU bandwidth than normally assumed.

In this paper we investigate different environmental timing assumptions and present the implementability results for each of them. This can provide detailed answers to questions that are of interest to real-time system engineers. For example, nowadays, with cheap, high performance chips, more and more industry implementations take the "easy" approach, which is to use chips with high sampling rates to achieve the PTR. An obvious question to ask is: *"Is it always necessary to sample at fast sampling rates and is it safe to assume that sampling faster is the best way to implement the system?"* Another important question is: *"The timing environment has been changed, how do I know my implementation will still work for the new timing environmental assumption?"*

In order to formally provide answers to the above questions, we refined the model and formalized the analysis of the *Held_For* operator of [3] in the PVS[1] theorem prover. *Held_For* is an operator that describes "sustained behaviour"' with tolerances on the timing duration. For example, $(signal \geq setpoint)$ *Held_For* $(300 \pm 50ms)$ specifies that the result shall be *true* if $(signal \geq setpoint)$ is *true* for $(300 \pm 50ms)$. Now consider an environmental timing assumption that the software "knows" the exact timing of the sample instances. For this assumption, we provide a full formal proof of necessary and sufficient conditions for when it is possible to construct a discrete implementation of such a requirement, with duration d and tolerances of $[d - \delta L, d + \delta R]$. The implementation may use nonuniformly spaced samples, as long as the intersample spacing is bounded. As a result of the formalization in PVS, we discovered a missing boundary case in the original theorem statement of [3]. The implementability results under two new environments are also formally verified and presented. Also, by comparing the results in different environments we can predict the implementability of real-time timing requirements under new environmental assumptions.

We provide an intermediate representation of the *Held_For* requirement on the implementation's sampled signals, that we use to verify an implementation model of the *Held_For* requirement via a two step process. Once the implementation has

[1] Files available at `http://www.cas.mcmaster.ca/~lawford/papers/FMICS08.html`

been verified in PVS, we can verify the implementation of any specific requirement by simply instantiating the PVS theorems with appropriate values. Thus the verification is reduced to a standard untimed verification on the remainder of the requirement's functionality. We demonstrate this process with a simple *Delayed Trip System (DTS)* example in Section 5.

1.1 Related Work

Recent work addresses the issue of timing tolerances required to verify implementations of requirements modeled as timed automata with ASAP semantics [4,5]. De 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. The implementation (e.g., C code on a BrinkOS platform) can be generated from the controller's automata.

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.

The approaches with global tolerances (e.g, reaction delay parameter Δ in [4] and ϵ-hypothesis in [10]) define a global constraint as the constant upper bound of the delay during implementation. However, in most industrial requirements, it is typical that different timing requirements need different tolerances. Our approach replaces a very conservative global tolerance by including tolerances on each individual timing requirement. We have found that this may significantly reduce unnecessary load on the target platform. This is illustrated by the Delayed Trip System example in Section 5.

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 the Giotto approach [8] and in the POOSL model [9,10]. These approaches cannot determine the feasibility of an implementation on a target platform until the scheduling stage is finalized. 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. In our approach, the implementability of the system is predictable in the first stages of analysis, avoiding unnecessarily complex implementation and verification.

The remainder of this paper is organized as follows: Section 2 presents the preliminary work in [3] and a two step Systematic Design Verification (SDV) procedure. Section 3 introduces four different environmental assumptions and shows the relationship between the implementability results under each of them. An estimation approach is also provided, which allows one to estimate or even precisely predict the implementability of the timing properties in a new environment. Sections 4 and 5 present the approach to refine and implement the high level timing requirements (e.g., *Held_For*), through an *Implementation Template*

(e.g., a pre-verified timer design). An example is provided to demonstrate how the implementation and verification work has been efficiently reduced. Conclusions and future work are discussed in Section 6.

2 Preliminaries

2.1 Functional and Performance Timing Requirements

We differentiate between *Functional Timing Requirements (FTRs)* and *Performance Timing Requirements (PTRs). FTRs* are timing requirements that are directly related to the required behavior of the application. *PTRs* are really timing tolerances that we specify so that the application does not have to adhere to the idealized behavior described by the requirements model. For more background on PTRs and FTRs, readers are referred to [3].

Definition of the *Held_For* operator with tolerance. *Held_For* is a common FTR which specifies a condition must be sustained over a particular time duration. A formal definition of the *Held_For* operator was specified in [3] as shown in Fig. 1.

(Condition) *Held_For* (d: $\mathbb{R}^{>0}$, δL, $\delta R : \mathbb{R}^{\geq 0}$) :bool
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 ≥ duration	*TRUE*
	t_{now} − Event_start_time < duration	*FALSE*
Condition = *FALSE*		*FALSE*

Fig. 1. Formal Definition of "(Condition) *Held_For* (d, δL, δR)"

There are a number of important points to emphasize. i) Duration is measured from when the event started in the physical domain. It does not make sense to define timing requirements with reference to when events are detected. ii) Many different implementations are valid. The behavior between $[d-\delta L, d+\delta R]$ is not deterministic. iii) Even though we have introduced tolerances into the requirement, *Held_For* is a FTR and still describes idealized behavior 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.

As one of the PTRs introduced in [3], the Response Allowance (RA) for a controlled-monitored variable pair specifies an allowable processing delay. 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.

The Timing Resolution (TR) for a monitored-controlled variable pair, can be thought of as the minimum duration event involving those variables that must be detected by the software [3].

2.2 Requirements Refinement and SDV Procedure Overview

In this section we provide an overview of our verification process in a two step approach based on the Systematic Design Verification (SDV) procedure introduced in [11]. In the first step, a pseudo-SRS[2] is created and verified as a refinement of the high level requirements. In the second step, we verify that the Software Design Description (SDD) is in compliance with the requirements for the behavior as specified in the pseudo-SRS.

To ensure the pseudo-SRS is a correct refinement, we must verify the pseudo-SRS based on all the timing requirements that are specified in the high level requirements. Let *pseudo-REQ* and *REQ* denote the pseudo-SRS state transition function and the high level Software Requirements Specification, respectively. The proof obligation of our first verification step can be formalized as:

$$pseudo\text{-}REQ \subseteq REQ$$

We restrict the pseudo-SRS to be a functional refinement of the high level requirements. The second step of the verification process is the SDV procedure based on a modified 4 variable model [11,12].

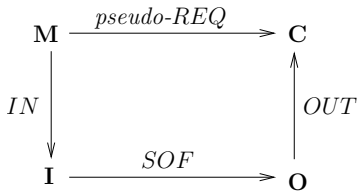

Fig. 2. Modified Commutative Diagram for 4 Variable Model

In Fig. 2, *pseudo-REQ* represents the pseudo-SRS state transition function mapping the monitored variables **M** to the controlled variables represented by **C**. *SOF* represents the SDD state transition function mapping the behavior of

[2] Imagine a version of the Software Requirements Specification (SRS) that is decomposed so that the data flow of the reorganized SRS is the same as that of the software design. This reorganized SRS is known as the "pseudo-SRS" [2].

the implementation input variables (represented by **I**) to the behavior of the software output variables (represented by **O**). The mapping IN models hardware functionality and relates the specification's monitored variables to the implementation's input variables. Similarly, the mapping OUT also models hardware functionality, and relates the implementation's output variables to the specification's controlled variables. The resulting proof obligation:

$$pseudo\text{-}REQ = OUT \circ SOF \circ IN \tag{1}$$

is illustrated by the solid lines in the commutative diagram of Fig. 2, which verifies the functional equivalence of the pseudo-SRS and SDD by comparing their respective one step transition functions [13]. Here $\circ$ is used to denote *functional composition*.

Through this two step SDV procedure, the high level requirements are connected with the low level implementation. In each of the steps, the verification can be formally conducted (e.g., by PVS). In later sections, we demonstrate this approach and provide the reader with an example.

2.3 Sample Instances

Let *Sample* be a possible sequence of sample times and $Sample(n)$ be the time of the $(n+1)$-th sample (n $\in \mathbb{N}$). *Sample* is assumed to satisfy the bounded jitter constraint, where T_{min} and T_{max} are the minimum and maximum sample intervals over the complete range of sample intervals, respectively.

$$Sample(0) = 0 \wedge \forall n : Sample(n+1) - Sample(n) \in [T_{min}, T_{max}].$$

We then also assume that the first sample point happens when $t = 0$, which is $Sample(0) = 0$. Note that when T_{max}=T_{min}, the problem is simplified to a fixed sample interval scenario, which is discussed in [14] for *Held_For* without tolerances.

In the example shown in Fig. 3, we assume $T_{min} = 10$ and $T_{max} = 20$. The first sample $Sample(0)$ occurs when $t = 0$, and the interval between any two consecutive sample points is in the range $[10, 20]$, e.g., $Sample(6) - Sample(5) = 85-70 = 15$. Further details on the example in Fig. 3 can be found in Section 3.1.

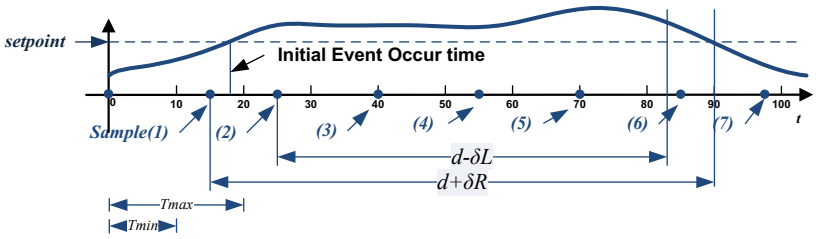

Fig. 3. An Example of Decision Points

3 Environmental Assumptions and Their Impact on Implementability

We have shown in [3] that the implementability results of the *Held_For* operator *with tolerance* are determined by the interaction between the FTRs (e.g., duration tolerances of the *Held_For* operator, δL and δR) and PTRs (the upper and lower bound of sample intervals: T_{min} and T_{max}).

In this section, we show feasibility analyses to answer the questions listed in Section 1. We first formally analyze the implementability results under three different environmental assumptions. By comparing the results across the environments, we develop an estimation approach based on the relationship between the environments. Finally we show that it is possible to estimate the range or even precisely predict the implementability results for a new environment.

3.1 Environmental Assumptions

We consider four different implementation environments which govern how we recognize a sustained event like the *Held_For* operator. They are the *Omniscient*, the *Perfect Clock*, the *Imperfect Clock* and the *No Clock* environments. We limit the scope of the analysis by assuming the implemented system will refresh the output at each sample point, which is a polling based rather than interrupt driven setting.

Perfect Clock: This environment provides the value of the condition only at sample instances and we know the exact timing of samples by using a perfect real valued clock. We can take actions (e.g., produce outputs) on the events only at sample times.

To properly state the environment conditions for implementation, we define the predicate *Feasible(d)* as a function of the sustained condition's nominal duration d and assume that the other parameters, $\delta L, \delta R$, T_{min} and T_{max}, are fixed. The feasibility function of the *Perfect Clock* environment is defined as follows.

Definition 1. $Feasible\_PerfectClock(d) : bool = \forall Sample : \forall n : \exists n_d :$
$\forall (t | Sample(n) < t \leq Sample(n+1)) : d - \delta L \leq Sample(n_d) - t \leq d + \delta R$

In the function above t represents the event start time and n_d is the index of the sample where we will make our decision. It is known from earlier work [3] that if the system behavior is specified in the form of $(Condition) Held\_For$ $(d, \delta L, \delta R)$, the final decision as to whether *Held_For* generates *TRUE* or *FALSE* based on the sampled values, cannot be made until we are sure that a time period with length $d - \delta L$ has elapsed since the event occurred in the physical domain (i.e. $d - \delta L \leq Sample(n_d) - t$). The decision also must be made before $d + \delta R$ has elapsed since the event occurred.

To explain this, we introduce an input signal and the duration with tolerances to the example in Fig. 3. The initial event (when the *signal* goes above the *setpoint*) occurs between *Sample*(1) and *Sample*(2). It is not hard to find that all the sample points up to and including *Sample*(5) are too early for us to determine

the value of the *Held_For* operator, and all the sample points from *Sample*(7) onwards are too late for us to make the decision. Only when $t = Sample(6)$, is it the right sample for us to make the decision.

Omniscient: This environment provides full read access to the timing of the events that happen in the physical domain. In this environment, we know the exact time of each event when the condition becomes *TRUE* or *FALSE*. However, we can only take actions on these events at sample times. The difference in comparison to the *Perfect Clock* environment is the relaxation of the existence requirement for the decision point. For any t between $Sample(n)$ and $Sample(n+1)$, a different decision sample point $Sample(n_d)$ is acceptable. Putting this all together we can find the feasibility function in the *Omniscient* environment.

Definition 2. $Feasible\_Omniscient(d) : bool = \forall Sample : \forall n :$
$\forall (t | Sample(n) < t \leq Sample(n+1)) : \exists n_d : d - \delta L \leq Sample(n_d) - t \leq d + \delta R$

Imperfect Clock: This environment is the same as in the *Perfect Clock* environment but with access to an imperfect clock (e.g. finite precision, bounded drift, etc). We leave as future work, the formalization of possible subcases that are associated with different imperfect clock assumptions. At the end of this section we will apply our estimation approach to this environment, which allows us to predict the implementability without having to perform complicated feasibility analyses and verification.

No Clock: Under this environmental assumption, our access to the timing of the events becomes very limited. The exact time of samples is not exposed even in the software domain. Our knowledge is only that each sample interval is between T_{min} and T_{max} and we also know 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 a "count" value n_d that will work under any possible bounded sample spacing and actual time of occurrence of the event. Let $Sample(n+n_d)$ be the decision sample point, which is the n_dth sample point since $Sample(n)$. Then we have the definition of the feasibility function in the *No Clock* environment as follows:

Definition 3. $Feasible\_NoClock(d) : bool = \exists n_d : \forall Sample : \forall n :$
$\forall (t | Sample(n) < t \leq Sample(n+1)) : d - \delta L \leq Sample(n + n_d) - t \leq d + \delta R$

3.2 Latest Environment Based Feasibility Analyses

Manual analysis in [3] shows that the only way that we can ensure the feasibility is to make sure that we have at least two sample points inside that interval $[d - \delta L, d + \delta R]$. After the recent PVS formal verification work, it turns out this is a necessary condition, but it is not sufficient. In this section, we will demonstrate how manual analysis with Fig. 4 provides a neat roadmap to guide us to the major results and how PVS formal verification captured a missing case in the manual analysis.

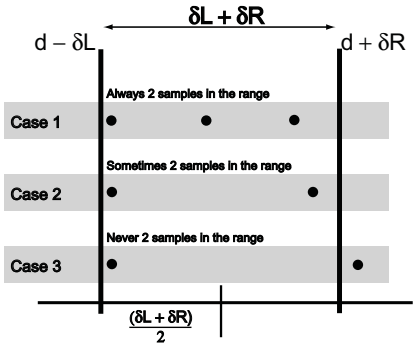

Fig. 4. Sample Points in the Duration Interval

Case 1: $0 < T_{max} \leq (\delta L + \delta R)/2$. In this case, we can guarantee there are always at least two sample points in the time interval $[d - \delta L, d + \delta R]$ in Fig. 4, based on which we can ensure the implementability of *Held_For*. Theorem 1 is proved for both the *Perfect Clock* and *Omniscient* environments.

Theorem 1. *Assume* $T_{max} \leq (\delta L + \delta R)/2$. *Then*

$$\mathit{Feasible\_PerfectClock}(d) \wedge \mathit{Feasible\_Omniscient}(d)$$

In the *No Clock* environment, implementability cannot be assumed. To understand this new result, shown in Theorem 2, we consider the two extreme cases, when the sample intervals are always T_{min} or T_{max}. For the T_{min} case, the first sample that is guaranteed to be on the right side of $d - \delta L$ is $\left\lceil \frac{d-\delta L}{T_{min}} \right\rceil + 1$. If $k = \left\lceil \frac{d-\delta L}{T_{min}} \right\rceil + 1$, then it is obvious that for feasibility in the T_{max} case, we must have that $k \times T_{max}$ cannot be to the right of $d + \delta R$.

Theorem 2. *Assume* $T_{max} < (\delta L + \delta R)/2$. *Then*

$$\left(\left(\left\lceil \frac{d - \delta L}{T_{min}} \right\rceil + 1 \right) \times T_{max} \leq d + \delta R \Leftrightarrow \mathit{Feasible\_NoClock}(d) \right)$$

Case 2: $(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R)$. 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*. Alternatively, we might be interested in operating at a slower sample rate in order to conserve power. In *Case 2*, two sample points will be in the time interval $[d - \delta L, d + \delta R]$ under certain conditions, which guides us to identify the necessary and sufficient conditions to implement *Held_For*.

Let $K_{min} = \left\lfloor \frac{d-\delta L}{T_{max}} \right\rfloor$ and $K_{max} = \left\lfloor \frac{d-\delta L}{T_{min}} \right\rfloor$, then the feasibility result for *Case 2* is given by the following theorem:

Theorem 3. *Assume* $(\delta L + \delta R)/2 < T_{max} \leq \delta L + \delta R \wedge T_{min} \neq T_{max}$. *Then*

$$T_{min} \geq \frac{d - \delta L}{K_{min} + 1} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R \Leftrightarrow \mathit{Feasible\_PerfectClock}(d)$$

We note that in [3] the conjunct $T_{min} \geq \frac{d-\delta L}{K_{min}+1}$ was incorrectly stated as $K_{min} = K_{max}$. During the course of formalizing the results of [3] in PVS, we identified a missing boundary condition which is also feasible under *Case 2*. The missing boundary case resulting in the new statement shown in Theorem 3 is when $K_{max} = K_{min}+1$ and $K_{max} \times T_{min} = d - \delta L$. This is when equality holds in the new conjunct (i.e. $T_{min} = \frac{d-\delta L}{K_{min}+1}$). While the latter condition restricts the application of this boundary case in practice, for the completeness of our results, this scenario needs to be considered to obtain the correct necessary and sufficient conditions for *Case 2* under the *Perfect Clock* and *No Clock* environmental assumptions.

Case 3: $T_{max} > (\delta L + \delta R)$. In *Case 3*, there is at most one sample point in the range of $[d - \delta L, d + \delta R]$ (shown in Fig. 4). Therefore, it is not possible to implement the *Held_For* operator.

Theorem 4. *Assume* $T_{max} > \delta L + \delta R$. *Then*

$$\neg \mathit{Feasible\_PerfectClock}(d) \wedge \neg \mathit{Feasible\_NoClock}(d) \wedge \neg \mathit{Feasible\_Omniscient}(d)$$

3.3 Comparing the Feasibility Results in Different Environments

In this section, we provide an overview and comparison of the feasibility results in the three environments: *Perfect Clock*, *No Clock* and *Omniscient*.

To compare the results we introduce the following two feasibility conditions.

$$\mathit{Condition\ 1}: \left(\left\lceil \frac{d - \delta L}{T_{min}} \right\rceil + 1 \right) \times T_{max} \leq d + \delta R$$

$$\mathit{Condition\ 2}: T_{min} \geq \frac{d - \delta L}{K_{min} + 1} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R$$

Table 1 provides the comparison of the feasibility results and other important facts in each environment. The column headings of the table are *Environments*, *Case 1*, *Case 2*, *Case 3*, *Event Visibility* and *Clock Readable*. The *Environments* column lists the environments. The *Imperfect Clock* case will be discussed at the end of this section. Columns *Case 1-3* list the necessary and sufficient conditions of the feasibility function for that case in the different environments. *Event Visibility* specifies in which domain we can access the timing of any physical event. The final column of the table is *Clock Readable*, indicating whether the clock is accessible in the environment. Taking the *Perfect Clock* environment as an example, here is the approach we used to fill in the values in this comparison table.

Table 1. Comparison of Implementability Results

Environments	*Case 1*	*Case 2*	*Case 3*	*Event Visibility*	*Clock Readable*
Omniscient	*TRUE*	*TRUE*	*FALSE*	Physical Domain	*YES*
Perfect Clock	*TRUE*	*Condition 2*	*FALSE*	Software Domain	*YES*
Imperfect Clock	*???*	*Condition 2*	*FALSE*	Software Domain	*YES*
No Clock	*Condition 1*	*Condition 2*	*FALSE*	Software Domain	*NO*

We filled in *TRUE* for *Case 1* because we do not require any additional condition to attain feasibility for *Case 1*. For *Case 2* and *Case 3*, the values are *Condition 2* and *FALSE* respectively, based on Theorem 3 and Theorem 4. We set *Event Visibility* to "Software Domain" because we will not be able to observe any event in the physical domain until the next sample point occurs in the software domain. Based on our discussion in Section 3.1, the value for *Clock Readable* should be YES.

We can now discuss the comparisons contained in Table 1. At one extreme, the *Omniscient* environment assumes that the time of the event is instantaneously reported to the software domain and the controller can calculate and produce the output simultaneously. The idealization embodied by this assumption allows us to design the implementation of the *Held_For* operator in a simpler way than any practical capability will allow. In *Case 2*, this environment does not require any feasibility condition. On the other hand, the *No Clock* assumption completely forbids access to the clock during the implementation process, which increases the difficulty of the implementation. Therefore, even in *Case 1*, an implementation is not always feasible. In *Case 3*, the *Held_For* operator is not implementable under any of the three environmental assumptions.

Note that the difficulty of the implementation of the *Held_For* operator increases as we progress down Table 1. The following states this more formally.

Theorem 5. $Feasible\_NoClock(d) \implies Feasible\_PerfectClock(d) \wedge$
$Feasible\_PerfectClock(d) \implies Feasible\_Omniscient(d)$

The relationship between the feasibility functions under different environmental assumptions determines the difficulty of implementing *Held_For* in those environments. For example, $Feasible\_NoClock(d) \implies Feasible\_PerfectClock(d)$, so we find that for any of the *Cases 1-3*, the condition to implement *Held_For* is always equivalent or more restricted under the *No Clock* environmental assumption than under the *Perfect Clock* environmental assumption. Now we consider the possible *Imperfect Clock*s described in Section 3.1. The information available to the implementation in this environment falls in between the *Perfect Clock* and *No Clock* cases. Since the latter two scenarios have the same necessary and sufficient conditions in *Case 2*, we can conclude that *Condition 2* is necessary and sufficient for any imperfect clock environment! For *Case 1* the *Imperfect Clock* case shows *???* since the precise feasibility function may depend upon what clock imperfections are considered, but it should be no stronger than *Condition 1*.

4 Implementation of the Held_For Operator

In this section, we briefly describe an implementation of the *Held_For* operator *with tolerance* in the *Perfect Clock* environment and refer the reader to [17] for a more detailed account. We refine our model of time to a discrete time model that assumes arbitrarily small clock ticks, which allows us to apply a straightforward inductive proving approach to verify the implementation of the *Held_For* operator.

The *Held_For* operator defined in Fig. 1 specifies the tolerance on the duration of the sustained window, and it leads to indeterminism in the implementation of the system. In particular, if the *Condition* has been sustained for an interval that is in the range $[d - \delta L, d + \delta R)$, then the current value of the *Held_For* operator can be either *TRUE* or *FALSE*. Based on the first step in Section 2.2, we can refine the requirements of the *Held_For* operator to a deterministic subset of the high level requirements that matches our implementation's behaviour at the sampling points. This is done by defining the `Held_For_S` operator as follows:

```
Held_For_S(P,duration,Sample)(ne):bool=
  EXISTS(n0|Sample(ne)-Sample(n0) >= duration):
    FORALL (n: nat | n0 <= n AND n <= ne): P(Sample(n))
```

When this operator defined on sample indexes is lifted to the arbitrarily fast clock tick level of the requirements in the natural way, it can be shown to be a refinement of the original *Held_For* operator under an appropriate PTR assumption [17].

4.1 Timer Implementation of Held_For_S

Timer_S and TimerUpdate Functions. To implement the `Held_For_S` operator, we can design a timer that updates its value at every sample instance. In Fig. 5, the `Timer_S` PVS function updates its value through a `TimerUpdate` function, by passing the following information: the condition at both the current and last sample instances, the pre-set timeout value, the current value of the timer and the elapsed time since the last update of the timer.

Then the `TimerUpdate` function will update the timer by returning the latest value.

In our design, we pass the values of the current and last sample instances, `P(Sample(ne))` and `P(Sample(ne-1))`, to `TimerUpdate` as the first and second parameters, `CurrentPP` and `PreviousPP`. The `TimerUpdate` function will reset the `Timer` to 0 when any of them is *FALSE*. When both of them are *TRUE*, `TimerUpdate` will update the `Timer` function by adding the elapsed time (step) to the previous `Timer` value. If the previous value has exceed the `TimeOut` value, the `TimerUpdate` function will do nothing but return the previous value to avoid an eventual overflow error.

We can then verify that an appropriately defined predicate on the current input value and the PVS function `Timer_S` is an implementation of the `Held_For_S` function.

```
TimerUpdate(CurrentPP, PreviousPP, TimeOut, PreviousTimerValue, step): tick =
TABLE
                            %+-----------------------------+-----------------------------++
                            |[ PreviousTimerValue < TimeOut| PreviousTimerValue >= TimeOut]|
%---------------------------+-----------------------------+-----------------------------++
| CurrentPP AND PreviousPP    | PreviousTimerValue + step   | PreviousTimerValue           ||
%---------------------------+-----------------------------+-----------------------------++
|NOT(CurrentPP AND PreviousPP)| 0                           | 0                            ||
%---------------------------+-----------------------------+-----------------------------++
ENDTABLE

Timer_S(P, Sample, TimeOut)(ne): RECURSIVE tick =
 TABLE
 %--------+-------------------------------------------------------------------------------++
 | ne = 0 | TimerUpdate(P(Sample(ne)), FALSE, TimeOut, 0, 0)
 %--------+-------------------------------------------------------------------------------++
 | ne > 0 | TimerUpdate(P(Sample(ne)), P(Sample(ne - 1)),
           TimeOut, Timer_S(P, Sample, TimeOut)(ne - 1), Sample(ne) - Sample(ne - 1)) ||
 %--------+-------------------------------------------------------------------------------++
 ENDTABLE
MEASURE ne
```

Fig. 5. TimerUpdate and Timer_S Functions

```
TimerGeneral_S1: THEOREM Held_For_S(P, timeout - delta_L, Sample)(n+1)
  IFF (P(Sample(n + 1)) AND Timer_S(P,Sample,timeout-delta_L)(n) +
                            Sample(n+1)-Sample(n)>=timeout-delta_L)
```

The `Timer_S` design yields a relatively easy implementation of the `Held_For_S` operator. Other equivalent implementations can be defined, and, in practice, there could be many similar implementations using the same design pattern. Our objective here is not to create a strict formula for software designers to follow, but to provide a generic design pattern like `Timer_S`, so that designers can customize the `Timer_S` design based on different situations. In the next section, we present the DTS example. By utilizing the general theorem `TimerGeneral_S1`, a large amount of the verification work is saved by proving the equivalence of the customized timer implementation to the original `Timer_S` implementation.

5 Example: Delayed Trip System with Tolerances

We now revisit the Delayed Trip System (DTS) [16] which was implemented and verified in [14] - but without explicitly considering timing tolerances. A modified Software Requirement Specification (SRS) for the DTS with explicit tolerances is shown at the top of Fig. 6. In this version, the requirements are specified with the *Held_For* operator *with tolerances*. If the condition *PP* has held for *timeout1*, the relay must be open. When the power drops below *PT* or the pressure becomes lower than *DSP*, the relay must not close until after another time period of *timeout2*. The bottom portion of Fig. 6 presents part of the PVS for the Software Design Description (SDD) of the DTS, the function `RelayUpdate`. With the help of the pre-verified `TimerGeneral_S1` theorem, we

Condition	*Result* ***relay***
(PP) *Held_For*$(timeout1,\delta L1,\delta R1)$	*TRUE*
$(\neg\ [(PP)$ *Held_For* $(timeout1,\ \delta L1,\ \delta R1)])$ *Held_For* $(timeout2,\ \delta L2,\ \delta R2)$	*FALSE*
$\neg\ (PP)$ *Held_For*$(timeout1,\delta L1,\delta R1)\ \wedge$ $\neg\ (\neg\ [(PP)$ *Held_For* $(timeout1,\ \delta L1,\ \delta R1)])$ *Held_For* $(timeout2,\ \delta L2,\ \delta R2)$	No Change

$$where\ PP(t) = Power(t) \geq PT \wedge Pressure(t) \geq DSP$$

```
SDD_State: TYPE =
      [# Relay: Relay_State, Timer1: tick, Timer2: tick,
         PreviousInput1: bool, PreviousInput2: bool #]

RelayUpdate(timeout1, timeout2, CurrentPP, S, step): Relay_State =
TABLE
%-------------------------------------------------------------------+-------++
| CurrentPP&(Timer1(S)+step>=timeout1)                             |OPEN   ||
%-------------------------------------------------------------------+-------++
| NOT(CurrentPP&Timer1(S)+step>=timeout1)&Timer2(S)+step>=timeout2|CLOSED||
%-------------------------------------------------------------------+-------++
| NOT(CurrentPP&Timer1(S)+step>=timeout1)&
          NOT (Timer2(S)+step>=timeout2)                          |Relay(S)||
%-------------------------------------------------------------------+--------++
ENDTABLE
```

Fig. 6. The SRS (top) and SDD (bottom) for the DTS with Tolerances

can clearly identify the functional behaviour of the DTS mapping from SRS to SDD tables. For example, the *(PP)Held_For(timeout1,...)* in the first row is implemented by `Timer1` and current condition `CurrentPP` (as shown in the first lines of both tables). Similarly, `Timer2` (with its current condition) implements the *Held_For* (with *timeout2*), as shown in the seconds line of both tables.

Both of the `Timer` functions call the `TimerUpdate` function to update themselves. Function `RelayUpdate` updates the current output of the relay, based on the *timeout1* and *timeout2*, the current condition `PP` and `step` (as introduced for `TimerUpdate`). Here the variable `S` is of record type `SDD_State`. It stores the system state - the status of the `Relay`, values of `Timer1` and `Timer2` and the input conditions of each timer at the previous sample time, `PreviousInput1` and `PreviousInput2`. These last two fields are passed to the `TimerUpdate` function applications as `PreviousPP` parameters, in order to determine whether the timer should add a step increment or perform a reset.

Note that the *Held_For* operator with duration *timeout1* has tolerance settings $\delta L1$ and $\delta R1$ and another *Held_For* operator with duration *timeout2* has its own tolerance settings $\delta L2$ and $\delta R2$. This may better fit a real-world engineering specification, where *timeout1* and *timeout2* may differ by more than an order of magnitude. In this case it typically would not make sense for the timing requirements to share a single global tolerance. For example, we may want *timeout1*=300±2 seconds and *timeout2*=2±0.1 seconds. This provides an example

in which the requirements of the system do not fit into a global tolerance model (e.g, the reaction delay parameter Δ of the Almost ASAP semantics in [4] and ϵ-hypothesis in [10]). Instead of performing a scheduling check in the final stage [8,10], our approach can also determine whether further work on an implementation is worthwhile as soon as the timing requirements have been specified (based on the *feasibility analyses* discussed in Section 3.2).

We have shown (in the complete PVS source code) how to reuse this result to reduce the verification work of the customized timer components, through the `TimerGeneral_S1` theorem. This general theorem requires more than 16 lemmas and 600 PVS commands to complete. This one time effort can benefit other going forward. In the DTS example, 51% of the total PVS prover commands used in the verification are eliminated by repeated instantiations of this theorem. More details on the DTS example are available in [17].

6 Summary

In this paper, we expand the scope of our *feasibility analyses* to explicitly include environmental assumptions. Our latest results show that implementability of timing requirements (e.g., *Held_For* operator) is determined by both the implementation environment and the interaction of the timing requirements. Now we are in a position to answer the questions we proposed in Section 1.

"Is it always necessary to sample at fast sampling rates and is it safe to assume that sampling faster is the best way to implement the system?" The latest feasibility analyses show that sampling faster is not always the only option and also not always the correct choice in implementing real-time systems. The feasibility analyses show that it is still possible to implement the *Held_For* operator when $T_{max} > (\delta L + \delta R)/2$, which provides an alternative solution to the designer of real-time systems, when coping with hardware limitations. On the other hand, the results of *Case 1* in the *No Clock* environment show that it is not always safe to assume implementability when $T_{max} \leq (\delta L + \delta R)/2$.

"The timing environment has been changed, how do I know my implementation will still work for the new timing environmental assumption?" This could be easily determined since we have the *implementability results* for different environments. Further, if the target environment is altered for a particular timing requirement, the relationships between feasibility functions under different environmental assumptions can help us estimate the implementability results for the new environment.

We have introduced a pre-verified *Implementation Template*, which benefits real-time software in two respects. First, it allows domain experts to specify different tolerances for each functional timing requirement, instead of a global tolerance for the timing behavior on the target system. Second, it helps to simplify and reduce the effort required in both the implementation and verification stages.

References

1. Wang, F.: Formal verification of timed systems: A survey and perspective. Proceedings of the IEEE 92(8), 1283–1307 (2004)
2. 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. LNCS, vol. 2805, pp. 133–153. Springer, Heidelberg (2003)
3. Wassyng, A., Lawford, M., Hu, X.: Timing tolerances in safety-critical software. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 157–172. Springer, Heidelberg (2005)
4. De Wulf, M., Doyen, L., Raskin, J.F.: Almost asap semantics: From timed models to timed implementations. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 296–310. Springer, Heidelberg (2004)
5. De Wulf, M., Doyen, L., Markey, N., Raskin, J.F.: Robustness and implementability of timed automata. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 118–133. Springer, Heidelberg (2004)
6. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Transactions on Programming Languages and Systems 16(5), 1543–1571 (1994)
7. Shankar, N.: Verification of real-time systems using PVS. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 280–291. Springer, Heidelberg (1993)
8. Henzinger, T.A., Kirsch, C.M., Sanvido, M.A., Pree, W.: A Giotto-based helicopter control system. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 46–60. Springer, Heidelberg (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, pp. 228–239 (2004)
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, pp. 123–139. Kluwer Academic Publishers, Dordrecht (2005)
11. Lawford, M., Hu, X.: Right on time: Pre-verified software components for constructuion of real-time systems. Technical Report 8, Software Quality Research Lab, McMaster University, Hamilton, ON, Canada (2002)
12. Parnas, D.L., Madey, J.: Functional documents for computer systems. Science of Computer Programming 25(1), 41–61 (1995)
13. Lawford, M., McDougall, J., Froebel, P., Moum, G.: Practical application of functional and relational methods for the specification and verification of safety critical software. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 73–88. Springer, Heidelberg (2000)
14. Hu, X.: Proving real-time properties of embedded software systems. M.Sc., Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada (2002)
15. Website, N.L.P.L.O.: `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`
16. Lawford, M., Wonham, W.: Equivalence preserving transformations of timed transition models. IEEE Trans. Automatic Control 40(7), 1167–1179 (1995)
17. Hu, X.: Proving Implementability of Timing Properties with Tolerance. Ph.D., Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada (2008)