

PROVING IMPLEMENTABILITY OF TIMING
PROPERTIES WITH TOLERANCES

PROVING IMPLEMENTABILITY OF TIMING PROPERTIES WITH TOLERANCES

By

XIAYONG HU, B.Eng., M.Sc.

A Thesis

Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

McMaster University

© Copyright by Xiayong Hu, August 2008

DOCTOR OF PHILOSOPHY (2008)
(Software Engineering)

MCMASTER UNIVERSITY
Hamilton, Ontario

TITLE: Proving Implementability of Timing Properties with
Tolerances

AUTHOR: Xiayong Hu
B.Eng., Shanghai Jiao Tong University, China
M.Sc., McMaster University, Canada

SUPERVISOR: Drs. Mark Lawford and Alan Wassying

NUMBER OF PAGES: xvi, 127

To Jia and Felice

Abstract

Many safety-critical software applications are hard real-time systems. They have stringent timing requirements that have to be met. We present descriptions of timing behaviors that include precise definitions as well as analysis of how functional timing requirements (FTRs) interact with performance timing requirements (PTRs), and how these concepts can be used by software designers. The definitions explicitly show how to specify timing requirements with tolerances on time durations.

This thesis shows the importance of specifying both FTRs and PTRs, by revealing the fact that their interaction directly determines the final implementability of real-time systems. By studying this interaction under three environmental assumptions, we find that the implementability results of the timing properties are different in each environment, but they are closely related. The results allow us to predict the system's implementability without developing or verifying the actual implementation. This also shows that we can sometimes significantly reduce the sampling frequency on the target platform, and still implement the timing requirement correctly.

We present a component-based approach to formalizing common timing requirements and provide a pre-verified implementation of one of these requirements. The verification is performed using the theorem proving tool PVS. This allows domain experts to specify the tolerance in each individual timing requirement precisely. The pre-verified implementation of a timing requirement is demonstrated by applying the method in two examples. These examples show that both the design and verification effort are reduced significantly using a pre-verified template.

A primary focus of this thesis is on how to include tolerances on tim-

ing durations in the specification, implementation and verification of timing behaviors in hard real-time applications.

Acknowledgments

First of all, I shall express my sincere gratitude to my supervisors. Dr. Mark Lawford, my supervisor since my master's program, has been guiding me patiently and supporting me to finish this long run. Dr. Alan Wassyng, who became my co-supervisor in 2005, has dedicated a huge amount of his time to help me make progress in my Ph.D. studies.

Second, I must thank my supervisory committee members. Dr. William M. Farmer has provided invaluable comments on my research work ever since my master's thesis work. Dr. Tom Maibaum provided careful review and so many helpful comments.

Third, I must thank my friends for their knowledge, encouragement and help. Yutong He shared a lot of experience in both thesis writing and defense preparation.

Last and certainly not least, I must thank Jia Yu, my parents and my in-laws. They have always supported my studies and encouraged me to complete them. With my family, I have shared every day of this endeavor.

Contents

Abstract	iv
Acknowledgments	vi
List of Figures	xi
List of Tables	xiv
1 Introduction	1
1.1 Motivation	2
1.2 Related work	4
1.2.1 Summary	4
1.2.2 Two Major Challenges	5
1.2.3 Examples of Requirements Validation	5
1.2.4 Related Work in Implementation Verification	7
1.3 Contributions	10
1.4 Thesis Outline	12
2 Preliminaries	13
2.1 PVS Preliminaries	13
2.1.1 PVS Proofs in Sequent Calculus	14
2.1.2 Unprovable Sequents and Counterexamples	15
2.1.3 PVS Support for Tables	16
2.1.4 Overview of the Naming Convention in PVS	19
2.2 Requirements Refinement and SDV Procedure Overview	21
2.3 The PVS-RT method	23
2.4 Limitations of <i>Held_For</i> operators without tolerance	25

3	Environmental Assumptions and Impact on Implementability	29
3.1	Environmental Assumptions	30
3.2	Functional and Performance Timing Requirements	31
3.3	Implementability of <i>Held_For</i> in a Perfect Clock Environment	39
3.3.1	Manual Analyses of Implementability Results	41
3.3.2	Latest Results for the Perfect Clock Environment	43
3.3.3	PVS Aided Analysis for <i>Case 2</i>	46
3.3.4	Examples of Feasible Sample Interval Ranges	48
3.4	Comparing the feasibility results in different environments	50
3.5	Implementability of <i>Held_For</i> in an Omniscient Environment	53
3.6	Implementability of <i>Held_For</i> in a No Clock Environment	55
3.7	Summary	57
4	Formal Verification of Feasibility Results	59
4.1	Overview of the PVS Theories	60
4.2	PVS Theory for Sample Type	61
4.2.1	Time Theory	61
4.2.2	SampleInstance Theory	61
4.2.3	FeasibilityResults Theory	64
4.3	Roadmap to Prove the Feasibility Theorems	68
4.3.1	Proving Strategy for <i>Case 1</i>	68
4.3.2	Proving Strategy for <i>Case 2</i>	69
4.3.3	Proving Strategy for <i>Case 3</i>	72
4.4	Summary	73
5	Implementing <i>Held_For</i> with Tolerance	75
5.1	Refining Time	76
5.1.1	Timing Model in Physical Domain	76
5.1.2	PVS Theories Based on the Tick Type	77
5.1.3	Difference between Tick and Clock Types	79
5.2	Held_For operator in Physical and Software Domains	80
5.2.1	Definitions	83
5.2.2	Filtered Tick Predicate	87
5.3	A more detailed Implementation Roadmap	89
5.4	Verification of Held_For_I Based on High Level Requirements	91

5.5	Implementation of <i>Held_For</i>	96
5.5.1	Timer Implementation of Held_For_I	96
5.5.2	Verification of the Implementation of Held_For_S	98
5.5.3	Verification of the Implementation of Held_For_I	99
5.6	Summary	100
6	Examples	101
6.1	Example: Sensor Lock System	102
6.1.1	Overview of the System	102
6.1.2	Software Requirement Specification (SRS)	103
6.1.3	Software Design Description (SDD)	104
6.1.4	Implementation Assumptions	105
6.1.5	Formal Verification of the SenLock System	108
6.2	Example: Delayed Trip System	112
6.2.1	Software Requirement Specification (SRS)	113
6.2.2	PVS Software Design Description (SDD)	114
6.2.3	Formal Verification of the Delayed Trip Example	117
6.3	Summary	118
7	Summary	122
7.1	Future Work	127
A	Time Theory	128
B	SampleInstance Theory	129
C	FeasibilityResults Theory	132
D	ClockTick Theory	138
E	SampleInstanceOnTick Theory	140
F	Held_For Theory	142
G	TimerGeneral Theory	148
H	SensorLock Theory	151

List of Figures

2.1	Horizontal Condition Tables	17
2.2	A Labeled Complex Horizontal Condition Table	17
2.3	PVS Code of the sample ELOCK Function	18
2.4	PVS Naming Convention Example	21
2.5	Revised Commutative Diagram for 4 Variable Model	22
2.6	The SRS for the Delayed Trip System Block	26
3.1	Two Valid Implementations of a Sustained Timing Requirement	32
3.2	<i>Held_For</i> Functional Timing Requirement	34
3.3	Formal Definition of “(Condition) <i>Held_For</i> ($d, \delta L, \delta R$)”	35
3.4	A Periodic Functional Timing Requirement	36
3.5	Formal Definition of “ <i>Periodic</i> (Condition, $d, \delta L, \delta R$)”	36
3.6	Synchronized Periodic Functional Timing Requirement	37
3.7	Formal Definition of “ <i>SyncPeriodic</i> ($d, \delta L, \delta R$)”	37
3.8	Timing Resolution for Time Continuous Monitored Variable . . .	38
3.9	Timing Resolution for Time Discrete Monitored Variable	38
3.10	An Example of Sample Instances	40
3.11	An Example of Decision Points	41
3.12	Sample Intervals Required for Sustained Events	42
3.13	Sample Points in the Duration Interval	43
3.14	<i>Case 1</i> of the Perfect Clock Environment	44
3.15	<i>Case 3</i> of the Perfect Clock Environment	45
3.16	<i>Case 2</i> of the Perfect Clock Environment	46
3.17	<i>Case 2</i> of the Perfect Clock Environment (Boundary Example)	48
3.18	Feasible Sample Intervals for Various Durations and Tolerances	49
3.19	<i>Case 2</i> of the Omniscient Environment	54

3.20	<i>Case 1</i> of the No Clock Environment	56
4.1	Overview of PVS Theories and their dependencies	60
4.2	Time Theory in PVS	61
4.3	SampleInstance PVS Theory	62
4.4	PVS Lemmas of Sample Properties	64
4.5	FeasibilityResults PVS Theory	65
4.6	PVS Theorems: Feasibility in Perfect Clock Environment . . .	66
4.7	PVS Theorems: Feasibility in Omniscient Environment	67
4.8	PVS Theorems: Feasibility in No Clock Environment	67
4.9	PVS Theorems: Relationship between Feasible Functions . . .	68
4.10	Proving Strategy for <i>Case 1</i>	69
4.11	PerfectClock_CASE2 and NoClock_CASE2 Break Down	70
4.12	Proving Strategy for <i>Case 2</i>	71
4.13	PVS Theorem TminAndKmax	71
4.14	PVS Theorems for PerfectClock_CASE2A	72
4.15	Proving Strategy for <i>Case 3</i>	73
5.1	ClockTick Theory	78
5.2	SampleInstanceOnTick Theory	80
5.3	Time Model based on Clock Type	81
5.4	Time Model based on Tick Type	81
5.5	Relationship between Held_For_I and High Level SRS	82
5.6	<i>Held_For</i> Versions in Physical and Software Domains	84
5.7	PVS Function Held_For_S	85
5.8	PVS Lemma TICK_BETWEEN_SAMPLE	85
5.9	Left_Sample Definition in PVS	86
5.10	PVS Lemma Left_Sample_PROPERTY3	86
5.11	PVS Function Held_For_I	87
5.12	PVS Function Held_For_P	87
5.13	Demonstration of Filtered Tick Predicate	88
5.14	Definition of FilteredTickPred	89
5.15	A Detailed Roadmap of Implementation	90
5.16	PVS Theorem Held_For_I_TR0	92
5.17	Response Allowance Scenario for PVS Theorem Held_For_I_TR0	93

5.18	PVS Theorem Held_For_I_VERIFY_FTR4	94
5.19	PVS Theorem Held_For_I_VERIFY_FTR2	94
5.20	PVS Theorem Held_For_I_VERIFY_FTR3	95
5.21	Timer_S Function	97
5.22	TimerUpdate Function	97
5.23	Timer_I Function	98
5.24	PVS Theorem TimerGeneral_S	99
5.25	PVS Theorem TimerGeneral_I	99
6.1	Block diagram for real-time SenLock controller	102
6.2	The upgraded version SRS of SenLock System	103
6.3	PVS Definition of Sensor Lock SRS	103
6.4	PVS Definition of Sensor Lock SDD	104
6.5	PVS Definition of ElockUpdate	105
6.6	Example of disagreement on second sample point	106
6.7	Spike behavior between consecutive sample points	108
6.8	SenLock System Proof Obligation	109
6.9	Theorems and Lemmas of the SensorLock Theory	110
6.10	SenLock_SRS_S Function	111
6.11	PVS Theorem SensorLock_Block_S6	112
6.12	PVS Theorem SensorLock_Block_S2	112
6.13	The Upgraded SRS for the Delayed Trip System	113
6.14	PVS Pseudo-SRS of Delayed Trip System	114
6.15	RelayUpdate Function	115
6.16	PVS Code for DTS SDD	116
6.17	PVS Lemma Timer1_Timer	118

List of Tables

2.1	The Updated SRS for the Delayed Trip System block	28
3.1	Comparison of Implementability Results	51
4.1	Comparison of the Proof Work of <i>Case 3</i>	73
4.2	Proof Work of the Feasibility Results	74
4.3	Proof Work of General Theorems	74
6.1	Verification Effort Comparison of Two Examples	120
7.1	Summary of Contributions	123

Glossary

$\mathbb{R}^{\geq 0}$	25	Non-negative real numbers
\mathbb{N}	39	Set of Natural Numbers ($\{0, 1, 2, \dots\}$)
\top	14	Boolean constant <i>TRUE</i>
\perp	14	Boolean constant <i>FALSE</i>
\models	15	Semantic entailment
$g \circ f$	23	Functional composition
d	33	Duration of <i>Held_For</i> operator
δL	33	(Left) Tolerance of duration d
δR	33	(Right) Tolerance of duration d
T_{min}	39	Lower bound of the sample interval
T_{max}	39	Upper bound of the sample interval
K_{max}	44	$\left\lfloor \frac{d - \delta L}{T_{min}} \right\rfloor$
K_{min}	44	$\left\lceil \frac{d - \delta L}{T_{max}} \right\rceil$
δt	76	Arbitrarily small time interval between clock ticks
FTRs	31	Functional Timing Requirements
PTRs	31	Performance Timing Requirements
TR	36	Timing Resolution
RA	38	Response Allowance
SDV	21	Systematic Design Verification
DTS	25	Delayed Trip System
SRS	22	Software Requirements Specification
SDD	22	Software Design Description
<i>Held_For</i>	2	An operator defined for sustained events
<i>Held_For_W</i>	24	<i>Held_For</i> operator <i>without tolerance</i> defined in [9]
<i>Held_For_N</i>	24	<i>Held_For</i> operator <i>without tolerance</i> with a “narrower” window

PVS Glossary

PVS	13	Prototype Verification System
TCC	18	Type Correctness Conditions
Held_For_S	83	A PVS function defined for sustained events at sample level
Held_For_I	11	A PVS function defined as an intermediate operator between the design and high level requirements of <i>Held_For with tolerance</i>
Held_For_P	83	Held_For operator which reflects the sustained result in the physical domain.
Timer_S	97	A PVS function implements the Held_For_S with a timer design
Timer_I	98	A PVS function implements the Held_For_I with a timer design
TimerGeneral_S	99	A PVS theorem pre-verifies Timer_S implements Held_For_S
TimerGeneral_I	99	A PVS theorem pre-verifies Timer_I implements Held_For_I

Chapter 1

Introduction

“A real-time system is one whose correctness depends not only on values of its outputs, but also on the times at which they are produced” [22]. In order to ensure its correct operation in critical contexts, it is important to rigorously demonstrate that a system’s timing requirements are satisfied [7, 25]. In other words, the implementation must not only produce the correct functional outputs but also has to produce them right on time.

Modern real-time systems often have requirements that determine the current outputs based not only on current inputs, but also upon the history of inputs and outputs [15, 34]. An example of such a system is a nuclear reactor shutdown system that has the requirement, “If power and pressure both exceed their maximum allowable limits for 3 seconds, then open the relay for 2 seconds” [14]. Another example is a pacemaker, which is a small device that can be placed under the skin of a patient’s chest, to help control abnormal heart rhythm. If a patient’s heart does not beat or becomes abnormal for a specified duration, the controller must take action to initiate an electrical pulse to prompt the heart to beat at a normal rate. For both examples above, we can see that real-time systems are often safety-critical. Any failure can cause unrecoverable system failures (i.e., a loss of containment of nuclear material or a pace at the wrong time that might end a patient’s life).

Formal methods have been applied in software development in order to provide precise and rigorous specifications, validate the requirements, verify the implementation, etc. However, most formal methods do not scale well

(e.g., consider the state-explosion problem in model checking) or they ignore the hard to deal with real-world details (such as timing tolerances) in order to make the resulting mathematical models tractable for analysis and refinement. In this thesis, we present an approach to precisely define complex timing requirements that include tolerances on all time durations to formally verify the feasibility of implementing those requirements, and finally we use this knowledge to build a pre-verified template that implements one of the most common timing requirements that occurs in real world applications.

1.1 Motivation

A central focus of this thesis is the specification and implementation of sustained events (later described by the *Held_For* operator). Sustained events are extremely common in hard real-time systems. The following is a typical example of such an event:

If there has been no ventricle pace initiated for the automatic interval and no sensed event for the spontaneous interval, then pace for $k_paceWidth$.

In this simple requirement, we can find three “held for” events. The first two are the conditions to trigger the pacing: the condition “no ventricle pace initiation” (from the pacemaker) has “held for” the automatic interval and the condition “no sensed event” (from a patient’s heart) has “held for” the spontaneous interval. The third one is the pacing control, saying if the above conditions are satisfied, then the pacing must start and be “held for” a sustained duration of $k_paceWidth$.

There are other reports of the application of *Held_For* operators in the real-time systems in the industrial world as well (e.g., the Darlington Nuclear Shutdown System [29]). Therefore, we choose the *Held_For* operator as the focus of our interest and work. The following overview of the topics highlights the motivation for this thesis.

Difficulties in formalizing, designing and verifying system timing requirements. Specifying, implementing and verifying real-time requirements for embedded software systems is a difficult and time-consuming task. If a

developer fails to correctly stop, start or reset a timer under conditions that seldom occur, it can result in a design flaw detected late in the development phase, or even worse, a system failure in the field. Hence real-time systems have become an active area of research in the formal methods community. An extensive survey of formal methods for the specification and verification of real-time systems was completed recently by Wang [28] (Section 1.2.1). The survey contains references to well over 200 publications related to the topic. The publications can be categorized by the model of time that they use, either continuous/dense time or discrete time. The overwhelming majority of the referenced publications are dedicated to the specification and validation of real-time requirements within their chosen model of time.

Large gap in requirements and implementation models. Despite a number of comparisons and surveys of real-time formalisms, relatively little work has been done connecting the two types of models when they are used at the levels they best represent: continuous time at the requirements level and discrete time at the implementation level.

Necessary and sufficient conditions for implementation. More researchers are now focusing on using a “platform-independent” layer [8] as a bridge between the abstract high level specification and coding languages like C++. The benefit of using a middle layer is to narrow the large semantic gap between modeling languages and implementation languages. However, the inconsistencies between the design model and realization typically cannot be observed until the final stages. A large amount of work can be saved if we can figure out the necessary and sufficient conditions for implementing the major timing components of the whole system before proceeding to any detailed implementation of the system. For example, the method in [29] shows a way to tell whether further implementation activities are worth it or not as soon as the timing requirements have been specified.

Use pre-verified timing elements. Most real-time systems can be modeled based on basic timing operators at the requirements level, for example

“Since P” [25] and *Held_For* [9, 29]. The serialization, composition and nesting of these fundamental timing elements can be used to model the system requirements in complex real-time systems. A pre-verified timing operator library will benefit real-time system design and verification in the following ways:

1. Developers can pick the timing operator from the library to implement the system and plugin the pre-verified theorem to get the whole system verified on-the-fly.
2. Developers can also provide their own timing operator during the system design, and compare and prove its equivalence to any existing operator or combination of operators in the library, which can speed up the verification of the whole system.

The motivation for this library is that real-time system design and verification is difficult because real-time timing requirements are difficult to implement and verify. We simplify this process by extracting the fundamental timing behaviors of the system, specifying and verifying them using pre-verified timing operators, and thereby saving repetitive work when dealing with systems that fall within the domain for which we have developed a timing operator library.

1.2 Related work

1.2.1 Summary

We took an extensive survey [28] of formal methods for the specification and verification of real-time systems as our roadmap to investigate the related work. Although it contains references to over 200 publications, very few of them are found to be related to our research topic. 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 and implementation verification.

1.2.2 Two Major Challenges

There are two major challenges in designing and implementing real-time systems: requirements validation and implementation verification. Requirements validation, also called “design validation” in [4], is to ensure the correctness of the requirements at the earliest stage possible, while implementation verification is to verify the correctness of the implementation against the formal requirements. For most real-time systems, they are equally important since any mistake in the implementation may directly lead to unrecoverable loss.

In the last two decades, many achievements in requirements validation of real-time systems have been reported [28]. Some of the most referenced modeling techniques are timed automata [2], timed Petri Nets [1] and timed process algebras (e.g., [21]). However, they are not related to our research topic as most of them only focus on modeling the real-time system and validating the correctness of the model. Our objective is to formally model and implement real-time systems with timing tolerances, ensuring the correct timing properties in the implementation using design verification. Since the timed automata formalism is one of the most widely researched modeling techniques introduced, we will use it as an example to explain the differences between our design verification focus and the more common requirements validation focus.

1.2.3 Examples of Requirements Validation

Timed automata modeling and validation

A model describes selected behavior of a system [28]. Modeling is the first task to convert a design into a formalism accepted by a model checking tool [4]. Timed automata were introduced as a formal notation to model the behavior of real-time systems. They provide a general way to annotate state-transition graphs with timing constraints using finitely many clock variables [2].

Temporal logic (with timing extensions) can be used to specify the real-time properties that the design must satisfy. Branching time temporal logics [3] like Computational Tree Logic can, for example, express whether a formula will eventually be true on at least one path or on all paths.

Model checking provides a means for checking whether a model of the design satisfies a given specification. Given the system descriptions as timed automata and the specification temporal logic formulas that describe the properties we want to verify, we can determine whether the models of a given system description satisfies the specification described by Temporal Logic formulas. Some model checking tools can also generate counterexamples, a series of reachable states which do not satisfy the Temporal Logic formula. This helps the verifier to easily identify errors during the validation process. Compared to theorem proving, model checking is almost fully automatic, requiring less user supervision and expertise. However, it is impossible to determine whether the given specification covers all the properties that the system should satisfy [4]. The coverage of the specification formulas usually determines how deeply a design is validated. The specification formulas should be as complete as possible to cover all the important timing properties.

Testing implementation for timed automata

There are mainly two approaches to generate the test cases for timed automata, either online or offline. In online (on-the-fly) approaches, the testing environment is responsible for computing the next test primitive and verifying the current output against the specification on-the-fly. In offline approaches, the complete set of test scenarios are computed before execution [12].

The algorithm for online testing with the timed automata model checker UPPAAL described in [12] randomly offers an input, waits for an output until a timeout, or restarts the testing process in each testing cycle. If UPPAAL observes an output or a time delay, it checks whether this is legal according to the state set. Whenever an input is offered or an output/delay is observed, the state set will be updated. Once UPPAAL detects an illegal occurrence or absence of an output during any test cycle, a fail signal is produced and the testing process is ended. Although the state-space-explosion problem experienced by many offline test generation tools is reduced in UPPAAL, because only a limited part of the state-space needs to be stored at any point in time, very large specifications with an extreme amount of non-deterministic behav-

ior may still need significant time to compute the reachable symbolic state set [18]. Plus, the coverage of this approach is random which has an impact on its error detection capability [12]. Sprinintveld *et al.*, presented the first test generation algorithm [27] that yields a finite and complete set of tests for dense real-time systems. In order to limit the infinite problem size, some strong assumptions are made to restrict the problem size. For example, the system is assumed to be *deterministic* and have *isolated outputs* (for each state, if an output is enabled then no other input/output transition can be enabled). Even with these strong assumptions, the algorithm itself is super exponential and cannot be claimed to be of any practical value (according to the authors) [27].

We have the following conclusions from the above discussion: 1. So far, we still lack a formal and effective method to verify the implementation from a timed automata model of a real-time system. 2. Although researchers have done a lot to minimize or even remove the non-determinism in the timed automata, the problem size is still too big to be of practical use. Tremendous work has been done on validating timed automata, but the verification and testing gap between a timed automata model and its implementation is still huge. Our goal is to introduce a formal approach to perform both requirements validation and implementation verification. In this approach, we will consider timing uncertainty (i.e., outputs and given stimuli classified in an interval of time rather than a time instance), which has not been taken into account in timed automata yet [12, 18].

1.2.4 Related Work in Implementation Verification

The overwhelming majority of the cited works in [28] 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 the timing tolerances when verifying implementations of requirements modeled as timed automata with Almost ASAP (as soon as possible) semantics [35, 36]. 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. 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 [35], while [36] 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. Further it is shown 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 $\Delta > 3\Delta_L + 4\Delta_P$.

Giotto is described as a domain-specific high-level programming language in [8]. The approach provides an intermediate layer between the mathematical model and the code, called the embedded software model, which is independent of the target execution platform, but closer to programming code than a mathematical model. The code generation task of Giotto is partitioned into two steps, *program generation* and *compilation*. During the program generation, a high level mathematical model (e.g., a Simulink model) is transformed into an embedded software model, which is completely independent of any execution platform. Then in the second step, the software model is converted into low level 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 [8]. Therefore the programs in Giotto can be separated into two parts, a timing program which handles the timing concerns (e.g., sampling) and a functionality program which handles the low level implementation. The concept is close to the functional and performance timing requirements that we will introduce in this thesis. The difference is that our approach will predict the impact of their relationship on the implementability of the system in the early stage of the analysis, to avoid unnecessary complex implementation and verification.

The assumption of zero-time for computational action in the model language is impossible to ensure on the target platform in the implementation language [6]. Thus the predictable design approach introduced in [10] an ϵ -hypothesis to fill the gap between the physical domain and the software domain. 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 Rotolumis 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” (Section 3.2), 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 [6, 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 categories: platform-independent and global tolerance. 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 the POOSL model in [6, 10]. In these approaches, whether or not a system can be implemented on a target platform cannot be known 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 [36] 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 and greatly simplifies the problem. However, in most industrial applications, different tolerances are required

for different timing requirements. A global tolerance on all timing durations does not make much sense at the requirements level. In addition, at the implementation level, timing tolerances in sampling intervals (jitter) may be caused in many ways. Unless the executive is a very simple loop, it is likely that different timing functions will exhibit different jitter. Again, a global tolerance seems to be both a simplification and too restrictive. The results we have so far (see Chapter 5) allow one to consider things like jitter associated with such a fast requirement being implemented by repeated calls within a cyclic executive. Our approach, to replace a very conservative global tolerance by including tolerances in each individual timing requirement, may significantly reduce unnecessary load on the target platform. This is illustrated by the Delayed Trip example in Chapter 6.

1.3 Contributions

The main contributions of this thesis include:

1. We present how functional timing requirements (e.g., *Held_For*) and performance timing requirements (e.g., timing resolution) interact with each other to determine feasible conditions of the final system implementation [29]. The results that we have obtained through the feasibility analyses can be used to predict the system’s implementability without carrying out the real implementation or verification work.
2. We have identified three environmental (time) assumptions and their impact on the feasibility results. They are the *Omniscient*, the *Perfect Clock* and the *No Clock* assumptions. For each environmental assumption, we verified the corresponding feasibility results using the theorem proving tool PVS [23]. A feasibility function is defined to indicate whether the timing requirements can be implemented or not under each environmental assumption.
3. We have developed a roadmap of verification, which enables significant reductions in the verification work of the feasibility results, by

applying two general theorems, `NoClock_Implies_PerfectClock` and `PerfectClock_Implies_Omniscient`. These two general theorems, which have been formalized and verified in PVS, reveal the relationships between the feasibility functions across three environments. Using the same general theorems, we also propose an approach to estimate the feasibility results, if a new environment is provided.

4. We demonstrate the approach used to properly formalize functional and performance timing requirements in PVS, through the example of the *Held_For* operator *with tolerance* defined in [29]. We define the `Held_For_I` function which is ready to be applied in the pseudo Software Requirements Specification (pseudo-SRS)¹, as an intermediate step of verifying our design against requirements. This operator is then verified in PVS based on the functional and performance timing requirements we have defined. For example, we verify that the `Held_For_I` operator is in the expected tolerance range as specified in the functional timing requirements, and conforms to the *Response Allowance*, which is one of the performance timing requirements.
5. We present a pre-verified *implementation template* for common pieces of the timing requirements (e.g., *Held_For*) that often appear in real-time systems. These can be reused to guide the design and reduce the associated verification work. We present methods that show how to design and verify a software component (e.g., a timer) that implements the `Held_For_I` operator. The pre-verified component is then used to guide the design of more complex components and to decompose their design verifications into simple inductive proofs.
6. We provide two examples, Sensor Lock and Delayed Trip, to demonstrate our implementation templates and verification approaches. In the Delayed Trip example, we show that our approach allows us to specify different tolerances for each individual timing requirement, which may

¹Imagine 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” [31]

make it possible to implement the real-time systems in a scenario where a global tolerance is not feasible.

1.4 Thesis Outline

The remainder of this thesis consists of the following chapters and appendices.

Chapter 2 introduces the related background and a procedure for software design verification. Our results on the *Held_For* operator *without tolerance* are presented as the preliminary work that motivates this thesis.

Chapter 3 presents three environmental assumptions and discusses the implementability of a new version of the *Held_For* operator *with tolerance* in each environment. Together with the environmental assumptions, interactions between functional and performance timing requirements determine feasibility results. We compare the feasibility results in the three environments and present a possible approach for estimating the feasibility results if a new environment is encountered.

Chapter 4 presents the PVS work to formalize and verify the results that we have in Chapter 3. Following an overview of the PVS theories and their dependencies, we introduce the roadmap of our verification work and how we used the general theorems to reduce duplication of proof efforts.

Chapter 5 introduces the implementation and verification of the *Held_For* operator *with tolerance*. We first refine our time model, which assumes arbitrarily small clock ticks, and formalize the model using PVS theories. We then define the intermediate operator, `Held_For_I`, and verify it against the high level functional and performance timing requirements. A pre-verified timer implementation of `Held_For_I` is introduced to guide the design of more complex components. In Chapter 6 we show how to apply this pre-verified result to simplify the design and verification of two sample applications.

Chapter 7 includes concluding remarks and suggestions for future work.

Chapter 2

Preliminaries

In this chapter we review the essential concepts and notations as well as previous results. Section 2.1 presents the PVS concepts and notation. We provide a brief overview of a Systematic Design Verification (SDV) procedure in Section 2.2. The PVS real-time method [15] and *Held_For* operator *without tolerance*, developed in previous work [9, 15], are introduced in Section 2.3. We revisit the Delayed Trip example in [9] and discuss the limitation of the *Held_For* operator *without tolerance* in Section 2.4. This also becomes our motivation to define a new *Held_For* operator *with tolerance* and implement it in the next chapter.

2.1 PVS Preliminaries

The Prototype Verification System (PVS) was developed by SRI International's Computer Science Lab to provide mechanized support for formal specification and verification. PVS consists of its own specification language, pre-defined theories, a theorem prover, libraries, utilities and documentation with several examples [33]. Currently it supports Mac, Linux and some other UNIX platforms that have Emacs or Xemacs installed. The formalization and verification work was completed on PVS version 4.1 [33] with NASA Langley PVS Libraries [32].

PVS font

In this thesis, all PVS expressions are displayed in **typewriter font** (also called **PVS font**). For example, “ELOCK” is a PVS function name, then it will be displayed in the thesis as **ELOCK**. Section 2.1.4 will provide the naming conventions that we use in the PVS theorem proving.

2.1.1 PVS Proofs in Sequent Calculus

The basic structure of the PVS sequent calculus is a *sequent* [24].

Definition 2.1.1. *Let (Γ, Δ) denote an ordered pair of sets of formulas in higher-order logic. A sequent in the sequent calculus is written $\Gamma \vdash \Delta$. Here \vdash denotes syntactic entailment. Γ is called the antecedent and Δ is called the consequent [26].*

A sequent can be written as $P_1 \wedge P_2 \wedge \dots \wedge P_n \vdash Q_1 \vee Q_2 \vee \dots \vee Q_m$, where P_1, P_2, \dots, P_n are the antecedent formulas, and Q_1, Q_2, \dots, Q_m are the consequent formulas. The goal of the PVS proof process is to determine whether a sequent is *TRUE* by proving that one of its consequents is a logical consequence of its antecedents. A sequent can be proved directly using decision procedures and simplification, or split into a collection of subgoals whose conjunction implies the root sequent.

Let \top and \perp denote boolean constants *TRUE* and *FALSE*. In any of the following cases, a sequent can be discharged [34]:

- (1) ***FALSE*** is one of the antecedents.

$$\frac{\begin{array}{c} \perp \\ \vdots \end{array}}{\vdots} \quad (2.1)$$

- (2) ***TRUE*** is one of the consequents.

$$\frac{\vdots}{\top} \quad (2.2)$$

(3) Formula A is both an antecedent and a consequent.

$$\frac{A \quad \vdots}{A} \quad (2.3)$$

When any one of these three cases appears in the sequent, PVS recognizes them as trivially true and completes the subgoal. When all subgoals are completed, PVS shows “Q.E.D.” at the end of the proof [24, 34].

2.1.2 Unprovable Sequents and Counterexamples

During PVS theorem proving, unprovable sequents can often help to construct counterexamples by examining their characteristic formulas [16]. Suppose we are going to use PVS to check whether the following statement is valid or not:

$$m \vdash p \vee q \quad (2.4)$$

We create the following theorem in the PVS approach to prove this sequent.

EXAMPLE1: THEOREM m IMPLIES p OR q

Applying the (bddsimp) proof command gives us the following unprovable sequent:


```

{1}      m
|-----
{-1}     p
{-2}     q

```

which has the characteristic formula $m \rightarrow (p \vee q)$. This formula is *FALSE* when $m = T$ and $p = q = F$. Therefore in this case, the sequent is unprovable. We can easily verify that this assignment provides a counterexample showing the original formula (2.4) is not a valid formula. One can also generate this counterexample with the **random-test** command that was added in PVS 4.0.

In this case, the counterexample is the conjunction of all the antecedents being *TRUE* and all the consequents being *FALSE*.

2.1.3 PVS Support for Tables

In this section, we briefly present PVS's support for tabular expressions. This section is based on [29].

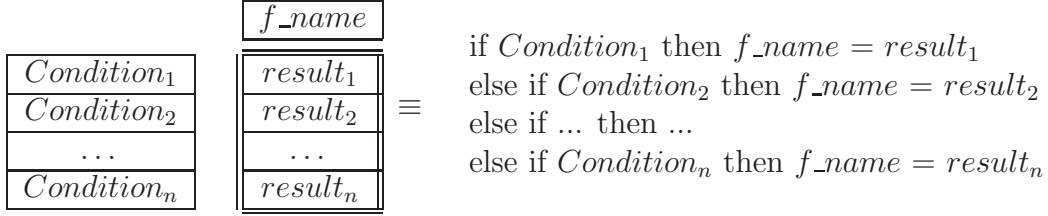
Labeled Complex Horizontal Condition Table

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.

There have been a number of publications on the semantics and usage of tabular expressions (e.g., [11, 30, 31]). The tabular expressions we use most in this thesis are called *Labeled Complex Horizontal Condition Tables*. Fig. 2.1 presents an example table together with its informal semantics. Disjointness and completeness criteria help us to ensure that the functional descriptions are unambiguous and complete [20].

Here we give a practical example shown in Figure 2.2 which arises in [9] in a verification problem. This example has multiple conditions and actions.

When the conjunction of atomic propositions in a given row of the *Condition* columns is *TRUE*, then output variables *Elock* and *lLockDly* are set to the *Result* value for that row, i.e., the first row of Figure 2.2 could be read as:



Disjointness: $Condition_i \wedge Condition_j \Leftrightarrow FALSE, \forall i, j = 1..n, i \neq j$, and

Completeness: $Condition_1 \vee \dots \vee Condition_n \Leftrightarrow TRUE$.

Figure 2.1: Horizontal Condition Tables

<i>Condition</i>			<i>Result</i>	
			Elock	lLockDly
\neg Sensor	Elock	Reset	Good	0
	=Lock	\neg Reset	Lock	0
	Elock \neq Lock		Good	0
Sensor	LTime< ldelay	Elock \neq Lock	Bad	next(LTime)
		Elock=Lock	Lock	next(LTime)
	LTime \geq ldelay		Lock	NoChange

Figure 2.2: A Labeled Complex Horizontal Condition Table

if $\neg Sensor \wedge (Elock = Lock) \wedge Reset$ then $Elock = Good \wedge lLockDly = 0$

The PVS definition of the table in Figure 2.2 is shown in Figure 2.3. To represent the possible *NoChange* status of the result for *lLockDly*, in PVS, we use the **RECURSIVE** statement to define the function. The last statement **MEASURE rank(t)** is a monotonically decreasing function on a well-ordered set, used to prove termination of recursion to ensure that the recursive function is well defined. A *well-ordered set* is a totally ordered set such that every non-empty subset has a least element in its ordering. In this case, **rank(t)** is of the type of natural numbers \mathbb{N} and bounded with least element 0, which allows one to prove the recursion terminates as **rank(t)** represents a monotonically decreasing quantity bounded below that is associated with the recursive function.

```

Lock_State: TYPE = {Good, Bad, lock}
SDD_State: TYPE = [# Elock: Lock_State, lLockDly: clock #]

ELOCK(t,ldelay): RECURSIVE SDD_State =
  IF init(t) THEN
    COND
      NOT sensor(t) -> (# Elock:= lock, lLockDly:= 0 #),
      sensor(t) -> (# Elock:= lock, lLockDly:= next(0) #)
    ENDCOND
  ELSE
    COND
      NOT sensor(t) AND Elock (ELOCK(pre(t),ldelay))=lock
      AND reset(t) -> (# Elock:= Good, lLockDly:= 0 #),
      NOT sensor(t) AND Elock(ELOCK(pre(t),ldelay))=lock
      AND NOT reset(t) -> (# Elock:= lock, lLockDly:= 0 #),
      NOT sensor(t) AND NOT Elock(ELOCK(pre(t),ldelay))=lock ->
      (# Elock:= Good, lLockDly:= 0 #),
      sensor(t) AND lLockDly(ELOCK(pre(t),ldelay))<ldelay ->
      (#Elock:=Elock(ELOCK(pre(t),ldelay)),
      lLockDly:=next(lLockDly(ELOCK(pre(t),ldelay)))#),
      sensor(t) AND lLockDly(ELOCK(pre(t),ldelay))>=ldelay ->
      (#Elock:=lock,lLockDly:= lLockDly(ELOCK(pre(t),ldelay))#)
    ENDCOND
  ENDIF
MEASURE rank(t)

```

Figure 2.3: PVS Code of the sample ELOCK Function

In PVS we can use a record type to define the system outputs or actions. In this case the record type `SDD_State` contains the system outputs *Elock* and *lLockDly*.

In order to make sure that a table is well-defined, PVS can generate the completeness and disjointness proof obligations which constitute *Type Correctness Conditions (TCCs)* for the function. PVS will first try proving these proof obligations automatically using PVS's built-in proof strategies. If a TCC is too complex for PVS to handle itself, the user can attempt the proof of the TCC interactively. The unprovable sequents of TCCs can often help the user to construct counterexamples to reveal the incompleteness or inconsistency of the specifications. Although one can start proving other theorems by skipping the TCCs, these proofs will not be considered completely proved until all the

TCCs of the dependent definitions, theorems and lemmas are proved [24].

2.1.4 Overview of the Naming Convention in PVS

In this section we go through the major objects in the PVS language and provide an introduction to the naming convention we employ in the thesis.

Theories

Specifications in PVS are built from theories, which contain definitions, lemmas, theorems, etc. PVS theories can be parameterized to provide genericity, reusability, and structuring [19]. We apply the Pascal Case convention¹ to theory names. The Figure 2.4 shows the PVS theory `SampleInstanceOnTick`.

Types

Most of the build-in types in PVS are defined in lower case convention (e.g., `real`). We follow this naming convention when defining mathematical types (e.g., `tick` represents the type of arbitrarily small clock ticks).

Theorems and Lemmas

The names of theorems and lemmas are specified in a way that best describes their purpose. Between each Pascal Case name, we use `_` to make them more understandable to readers. For example: `SensorLock_Block` is the name of the theorem to perform block comparisons between the requirement specification and implementation of the `SensorLock` industry example.

Due to the large number of type check proof obligations generated by PVS, we create lemmas to handle the repetitive proof work. Therefore, we only need to prove them once, and instantiate them everywhere they are used. Most of them are related to the property of the definition, so the naming convention of these lemmas is: identifier followed by `PROPERTY` or `RELATIONSHIP`, separated by `_`. For example, the PVS lemmas `SampleTick_PROPERTY1` and

¹The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized (i.e., `SensorLock`).

`SampleTick_PROPERTY2` state and prove the two trivial properties of the `SampleTick` type in Figure 2.4.

Variables in Lemmas and Theorems

When defining local variables for lemmas and theorems, we keep it as straightforward as possible in PVS. Most of them are one or two letters in lower case convention. In the example shown in Figure 2.4: `n` denotes a natural number variable and `t` denotes a time variable.

Consistency with the Industry Conventions for Important Identifiers

There are quite a few important identifiers in our PVS theories that come from the industry examples or documents [15, 29, 31]. We keep those names as consistent as possible so that the theorems are user-friendly to the domain experts and system implementers. Some examples are: `Held_For` stands for the sustained event operator and `Sample` is used to specify a series of sample instances (see Figure 2.4).

PVS Proving Commands

The PVS proof commands are the commands that we type in to prove the PVS obligations. They need to be in parentheses to be recognized by PVS. For example: `(split)` is the PVS command to split a conjunctive formula in the current goal sequent. The command must be in either upper case or lower case; otherwise it will be considered as an illegal keyword by PVS.

In this section, we reviewed the naming conventions of the PVS work. A glossary on page xvi provides a quick reference to the commonly used PVS names.

```

SampleInstanceOnTick[K: posreal, (IMPORTING Time) TL,
                    TR: {t: time | t < K},
                    delta_t: {tk: posreal | tk < K - TL}]: THEORY
BEGIN

  IMPORTING ClockTick[delta_t]

  IMPORTING SampleInstance[K, TL, TR]

  t: VAR tick
  n: VAR nat

  SampleTick_Type: TYPE =
    {S: Sample_Type | FORALL (n: nat): EXISTS (t: tick): S(n) = t}

  Sample: VAR SampleTick_Type

  n: VAR nat

  SampleTick_PROPERTY1: LEMMA EXISTS (n1: nat): Sample(n) = n1 * delta_t

  SampleTick_PROPERTY2: LEMMA
    pre(Sample(n + 1)) > Sample(n) AND pre(Sample(n + 1)) < Sample(n + 1)
  END SampleInstanceOnTick

```

Figure 2.4: PVS Naming Convention Example

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 in [13]. In the first step, called *Requirement Refinement*, a pseudo-SRS is created and verified as a refinement of the high level requirements. In the second step, called the *pseudo-SDV Procedure*, we verify that the Software Design Description (SDD) is in compliance with the requirements for the behavior as specified in the pseudo-SRS.

When the high level requirements of the real-time systems are created by the domain experts, they can be a combination of formal documentation (e.g., tabular expressions), graphical explanations and example (scenario)

demonstrations. All these efforts are intended to provide as accurate and complete information as possible to the later stages (e.g., requirements refinement, design and implementation). As an example, the *Held_For* operator *with tolerance* that we will present in the next chapter is a functional timing requirement. However, high level requirements are usually not formalized completely and intentionally include under specifications that needs to be further refined in the design and code implementation.

Our approach is to first create the pseudo-SRS as an intermediate step between the high level Software Requirements Specification (SRS) and Software Design Description (SDD). To ensure it is a correct refinement, we need to verify the pseudo-SRS based on all the timing requirements that are specified in the high level requirements. In other words, the behaviors that are specified in the refined pseudo requirements (*pseudo-REQ*) must be a subset of the ones specified by high level requirements (*REQ*). The proof obligation of our first step verification can be formalized as:

$$pseudo-REQ \subseteq REQ$$

In the scope of this thesis, we restrict the pseudo-SRS to be a functional refinement of the high level requirements. The second step of the verification process is then similar to the SDV process presented in [13], as illustrated in the revised 4 variable model shown in Figure 2.5.

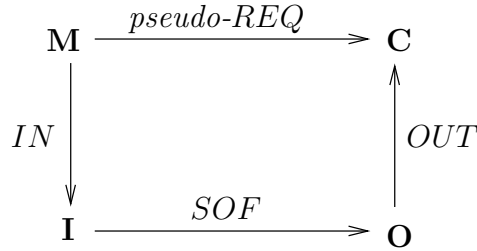


Figure 2.5: Revised Commutative Diagram for 4 Variable Model

In the figure, *pseudo-REQ* represents the pseudo-SRS state transition function mapping the monitored variables represented by **M** to the controlled variables represented by **C**. The other parts are unchanged from [13]. *SOF*

represents the SDD state transition function mapping the behavior of the implementation input variables (represented by statespace **I**) to the behavior of the software output variables (represented by the statespace **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

$$pseudo-REQ = OUT \circ SOF \circ IN \quad (2.5)$$

is illustrated by the solid arrows in the commutative diagram of Figure 2.5, which verifies the functional equivalence of the pseudo-SRS and SDD by comparing their respective one step transition functions [17]. Here \circ is used to denote *functional composition*. Given functions $f : V_1 \rightarrow V_2$ and $g : V_2 \rightarrow V_3$, we use $g \circ f$ to denote *functional composition*, i.e., $g \circ f(v_1) = g(f(v_1))$.

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

2.3 The PVS-RT method

The PVS real-time (PVS-RT) method introduced by Lawford *et al.* in [15], investigates the use of PVS for the specification and verification of the real-time behavior of control systems software. In this section we will review the PVS-RT method and some related preliminary work of [9].

The model of time employed by the proposed method in [9] builds upon a discrete time “Clocks” theory originally defined in [5]. While the model of time put forward in [5] allows for multiple clocks of different frequencies and continuous time functions, we restrict the scope to discrete time functions of a single clock frequency in [9]. We consider time to be the set of non-negative real numbers, denoted by $\mathbb{R}^{\geq 0}$. Then for a positive real number K , we define

a clock of period K , denoted $clock_K$, to be a set of “sample instances”

$$clock_K := \{t_0, t_1, t_2, \dots, t_n, \dots\} = \{0, K, 2K, \dots, nK, \dots\}$$

For a period $K = 5ms$, the clock is simply

$$clock_5 := \{0ms, 5ms, 10ms, 15ms, \dots\}$$

Note that $clock_5$, like all clocks as defined above, “starts” at time $t_0 = 0ms$.

Building on this work in [9, 13], we designed an implementation of timing properties (specified in the real-time system requirement) and verified it using PVS. This result can then be used to guide system design and decompose design verification into simple inductive proofs.

We first specify requirements of a Sensor Lock System using the `Held_For` operator [9] and verified it for an arbitrary value of the system timing parameters. By refining and comparing the related *condition/result* logic embedded in the tabular expressions, we were able to relate the timing behavior of the Software Requirements Specification (SRS) and Software Design Description (SDD), which allowed us to finish the verification of the Sensor Lock System.

In this thesis we rename `Held_For` to *Held_For_W* in order for it not to be confused with the names of the other versions of `Held_For` operators we introduce in this thesis. Therefore, *Held_For_W* stands for the `Held_For` operator in [9], which is *TRUE* while input has been holding for a “wider” window (in the next section we will compare it to another similar definition *Held_For_N*, which is a narrow window version).

We now state a preliminary definition that will aid us in defining the *Held_For* operators. For the $clock_K$, the set of clock predicates, denoted $pred(clock_K)$, is the set of all boolean functions of $clock_K$:

$$pred(clock_K) := \{f \mid f : clock_K \rightarrow \{TRUE, FALSE\}\}$$

The *Held_For_W* operator [9, 13, 15, 34] is formally defined as:

Definition 2.3.1. Let “duration” denote a non-negative real number, and “ P ” represent a clock predicate. *Held_For_W* is an operator, written infix, that

takes a clock predicate as its first argument, a non-negative real number as its second argument and returns a clock predicate:

$$\text{Held_For_}W : \text{pred}(\text{clock}_K) \times \mathbb{R}^{\geq 0} \rightarrow \text{pred}(\text{clock}_K)$$

such that $(P)\text{Held_For_}W(\text{duration})(t_n) = \text{TRUE}$ iff

$$(\exists t_j \in \text{clock}_K)(t_n - t_j \geq \text{duration}) \wedge (\forall t_i \in \text{clock}_K)(t_j \leq t_i \leq t_n \Rightarrow P(t_i))$$

Note that we are supposing the sampling behavior in the physical domain always happens at a constant, fixed interval and we are ignoring inter-sample behavior of the condition P , i.e., the truth value of $\text{Held_For_}W$ is only dependent upon the value of P at the sampling instances corresponding to the clock values. Timing tolerance is also not considered in this version of the $\text{Held_For_}W$ operator. In the next section, we will review the Delayed Trip System we have verified based on the SRS defined with the $\text{Held_For_}W$ operator to demonstrate the limitation of the Held_For operators without timing tolerances.

2.4 Limitations of *Held_For* operators without tolerance

The $\text{Held_For_}W$ operator in [9] does not consider any timing tolerance. As a result, in some scenarios, it might not be able to reflect the exact timing properties that domain specialists want to specify. To illustrate this, we revisit the Delayed Trip System (DTS) [14] which is implemented and verified in [9]. The informal description of the desired input/output relationship for the DTS block is:

Whenever the power exceeds the Power Threshold (PT) and the pressure exceeds the Delayed Trip Set Point (DSP) simultaneously for timeout1 = 3s, then open the relay. The relay must remain open for at least timeout2 = 2s, from the time that the power or pressure no longer satisfy the specified condition.

The DTS block's cycle time is $K = 100ms$, which means that the system reads its inputs and updates its outputs every 0.1 seconds. The formal specification is shown in Figure 2.6:

<i>Condition</i>	<i>Result</i>
$(PP) \text{ Held_For } timeout1$	relay <i>TRUE</i>
$(\neg [(PP) \text{ Held_For } timeout1]) \text{ Held_For } timeout2$	<i>FALSE</i>
$\neg[(PP) \text{ Held_For } timeout1] \wedge \neg([\neg (PP) \text{ Held_For } timeout1]) \text{ Held_For } timeout2)$	No Change

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

Figure 2.6: The SRS for the Delayed Trip System Block

Above is the Software Requirement Specification (SRS) for DTS. PP is a time predicate which is *TRUE* when power exceeds PT and pressure exceeds DSP . The inputs to this SRS function table are *Pressure* and *Power*, both of which are clock predicates. The function output is a boolean variable. When the conjunction of atomic propositions in any give row of the *Condition* column is *TRUE*, *relay* is set to the *Result* of that row. For example, when

$(PP) \text{ Held_For } timeout1$ is *TRUE*,

then $relay = TRUE$.

A complete implementation of the DTS can be found in [9]. However, we also noticed that we cannot perfectly recognize the two sustained events because our assumption is based on an “ideal” constant sampling interval. Recall the *Held_For_W* Definition 2.3.1. The relay may not open until the condition PP has been held for at least 3 seconds in the physical domain. It is still possible that PP has been held *TRUE* for 3.05 seconds but the relay is not open. The domain specialist may find that this is not acceptable and may require the capture of any case in which the condition PP holds over 3 seconds in the physical domain. Thus, in this case, the *Held_For_W* operator, with a larger window, cannot be used to specify the safety requirements correctly. In order to fix this problem, we can make slight changes

and define the following *Held_For_N* operator. The only difference between *Held_For_N* and *Held_For_W* is that *Held_For_N* uses $next(t_n)$ in the duration condition to narrow the held for window for condition P . In [9], we defined $next(t_n) = t_{n+1} = t_n + K$, which returns the next clock value of t_n .

Definition 2.4.1. Let “duration” denote a non-negative real number, and “ P ” represent a clock predicate. *Held_For_N* is an infix operator that takes a clock predicate as its first argument, a non-negative real number as its second argument and returns a clock predicate:

$$Held_For_N : pred(clock_K) \times \mathbb{R}^{\geq 0} \rightarrow pred(clock_K)$$

such that $(P)Held_For_N(duration)(t_n) = TRUE$ iff

$$(\exists t_j \in clock_K)((t_n - t_j \leq duration) \wedge (next(t_n) - t_j > duration)) \wedge$$

$$(\forall t_i \in clock_K)(t_j \leq t_i \leq t_n \Rightarrow P(t_i))$$

In this scenario, *Held_For_N* can help us specify the requirements better than *Held_For_W*, by guaranteeing a smaller window time to trigger the relay. On the other hand, for some safety requirements we really need to utilize the larger window of the *Held_For_W* operator to ensure a long enough sustained event or action. For example, the DTS requires that once the channel gets tripped, then open the relay for at least 2 seconds. If we use the *Held_For_N* operator with a duration of 2 seconds, it can only guarantee a relay signal longer than 1.9 seconds (e.g., 1.95 seconds). From a safety perspective, domain experts might reasonably insist that the relay should open for “at least” 2 seconds. In this case, the *Held_For_W* operator with a duration of 2 seconds is a better choice (to produce a relay for any interval over 2 seconds but no more than 2.1 seconds).

An updated version of the Delayed Trip System specification is shown in Table 2.1. We also used PVS to prove some relationships between the *Held_For_W* and *Held_For_N* operators. In the special case when *duration* is the clock type, not just a positive real value but rather a multiple of the clock period, we can prove that they are equivalent. In this case, both operators check the same history window of the system samples and generate the identical result.

Condition	Result
	relay
$(PP) \text{ Held_For_}N \text{ timeout1}$	TRUE
$(\neg [(PP) \text{ Held_For_}N \text{ timeout1}]) \text{ Held_For_}W \text{ timeout2}$	FALSE
$\neg[(PP) \text{ Held_For_}N \text{ timeout1}] \wedge$ $\neg([\neg (PP) \text{ Held_For_}N \text{ timeout1}]) \text{ Held_For_}W \text{ timeout2})$	No Change

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

Table 2.1: The Updated SRS for the Delayed Trip System block

We have defined the two *Held_For* operators *without tolerance* and applied them to the Delayed Trip example. But domain engineers may still argue about the safety of the gap left by the smaller and larger windows of *Held_For* operators. Worse still, the implicit tolerance or safety margins of these operators is dependent upon the sampling period K . Changing K from 100ms to 200ms effectively doubles the tolerated behaviour. Instead of becoming true when the condition has held for between 1.9 and 2.0 seconds, the *Held_For_N* operator now becomes true when the condition has held for between 1.8 and 2.0 seconds. To provide a better solution, we need a *Held_For* operator *with (explicit) tolerance* and the concepts of timing resolution and response allowance, which will be explained in the next chapter.

Chapter 3

Environmental Assumptions and Impact on Implementability

In this chapter we will present three different environmental assumptions and discuss the implementability of an upgraded version of the *Held_For* operator *with tolerance* in each of them. In order to fully analyze the problem, two types of the timing requirements (functional and performance) are introduced in Section 3.2.

To formalize the investigation, we have defined one feasibility function for each of the environments, which directly indicates the implementability of this *Held_For* operator in that environment. Sections 3.3 - 3.6 show how the relationships of the functional and performance timing requirements become the conditions which satisfy the feasibility function in each environment. By comparing the feasibility results across these three environments, we find an approach to roughly estimate how difficult it will be to implement the *Held_For* operator in a new environment. This chapter provides the basic results for the subsequent chapters that discuss the implementation of the *Held_For* operator with tolerances.

3.1 Environmental Assumptions

We assume that there are 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 to assume the implemented system will refresh the output at each sample point, which is a polling based rather than interrupt driven setting. However, even in an interrupt driven system where the analog signal is fed into a comparator to generate an interrupt when the signal exceeds or falls below setpoint, the comparator output is effectively being sampled at the system clock frequency.

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.

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. Like the *Omniscient* environment, we can take actions on the events only at sample times.

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). To precisely specify this situation, we need to first model the imperfect clock and construct any real-time operators based on it. We leave as future work, the formalization of possible subcases that are associated with different imperfect clock assumptions.

No Clock: In this environmental assumption, our access to the timing of the events becomes very limited. The exact time of a sample instance is not exposed to the software domain. 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 , that will

work under any possible bounded sample spacing and the actual time of occurrence of an event.

We will introduce each of the environments in detail in the later sections when discussing the feasibility results.

3.2 Functional and Performance Timing Requirements

This section is based on [29]. All the figures in this sections are from [29].

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 behavior 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.

We differentiate between *Functional Timing Requirements (FTRs)* and *Performance Timing Requirements (PTRs)*. *Functional timing requirements* are timing requirements that are directly related to the required behavior of the application. *Performance timing requirements* 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.

Functional Timing Requirements

Here we introduce three functional timing requirements that occur frequently in practical applications: sustained, periodic and synchronized periodic timing requirements.

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, the *Held_For_W* operator in [9] is one form of a

sustained timing requirement. However, in the real world, without tolerances on the time duration, these requirements would be impossible to meet. In the previous chapter, we already showed that the *Held_For_W* operator of [9] lacked the ability to handle tolerances, resulting in two versions, each version having to cover two different scenarios.

To further understand the sustained timing requirements with tolerance, we can look into a simple sensor-trip 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.

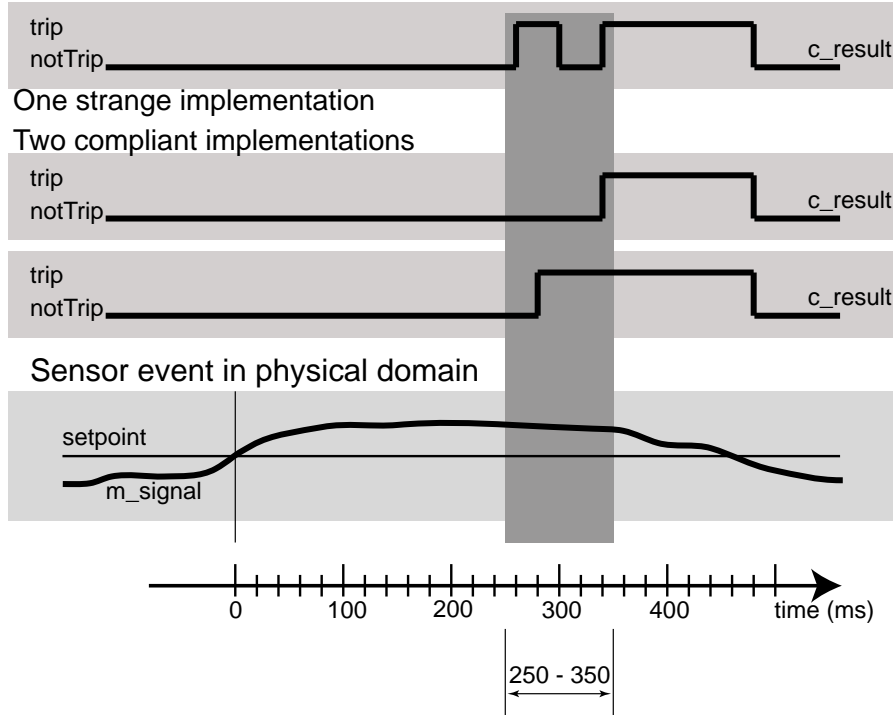


Figure 3.1: Two Valid Implementations of a Sustained Timing Requirement

Figure 3.1 shows an implementation of the behavior specified above for

a controlled variable *c_result* and sustained condition *m_signal* \geq *setpoint*. The strange behavior in the top implementation is almost certainly not what the specifier intended, but it may be compliant with its specification. We need to be able to specify the desired behavior precisely enough to allow many different behaviors that should be permitted, but disallow behaviors that do not comply with the specifier’s intention. So, 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? Figure 3.1 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 behavior 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 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.

To model these sustained events with tolerances, we developed a new version of the infix *Held_For* operator *with tolerance*, (*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* with tolerances is illustrated in Figure 3.2, and is defined formally using tabular expressions in

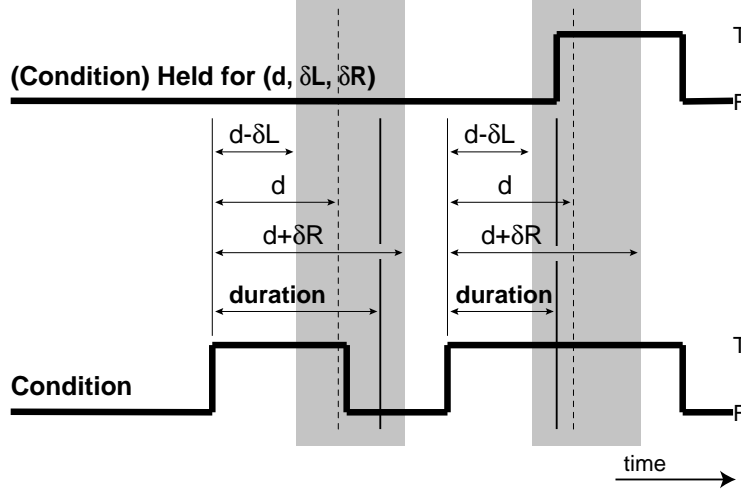


Figure 3.2: *Held_For* Functional Timing Requirement

Figure 3.3. 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. The initiating event in this case is when *Condition* changes from *FALSE* to *TRUE*. The event lasts until *Condition* changes from *TRUE* to *FALSE*. Without this constraint, many different bizarre behaviors are possible, all of them are 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 Figure 3.4, and is defined formally using tabular expressions in Figure 3.5.

A different kind of periodic function is one that is synchronized with an external clock as illustrated in Figure 3.6.

(Condition :bool) **Held_For** (d: $\mathbb{R}^{>0}$, $\delta L, \delta R : \mathbb{R}^{\geq 0}$) :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 = <i>TRUE</i>) & (Condition ₋₁ = <i>FALSE</i>)	Any value in $[d - \delta L, d + \delta R]$	t_{now}
(Condition = <i>FALSE</i>) OR (Condition ₋₁ = <i>TRUE</i>)	No Change	No Change

Held_For	
Condition = <i>TRUE</i>	$t_{now} - \text{Event_start_time} \geq \text{duration}$
	$t_{now} - \text{Event_start_time} < \text{duration}$
Condition = <i>FALSE</i>	

<i>TRUE</i>
<i>FALSE</i>
<i>FALSE</i>

Figure 3.3: Formal Definition of “(Condition) *Held_For* ($d, \delta L, \delta R$)”

If the periodic functional requirement is synchronized with an external clock, definitions equivalent to $t \bmod period = 0$ are useless when the period involves tolerances. The requirement $t \bmod 400 \pm 50 \text{ ms} = 0$ results in millisecond 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 behavior much at all. A practical, formal specification of this periodic functional requirement can be developed from $\forall n : t_n \in [n \times d - \delta L, n \times d + \delta R]$, and is defined using tabular expressions in Figure 3.7. 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.

Performance Timing Requirements

Functional behavior of the application is (typically) described using a model that describes the ideal behavior 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 behavior, the requirements must also specify the performance tolerances that are allowed in meeting functional timing require-

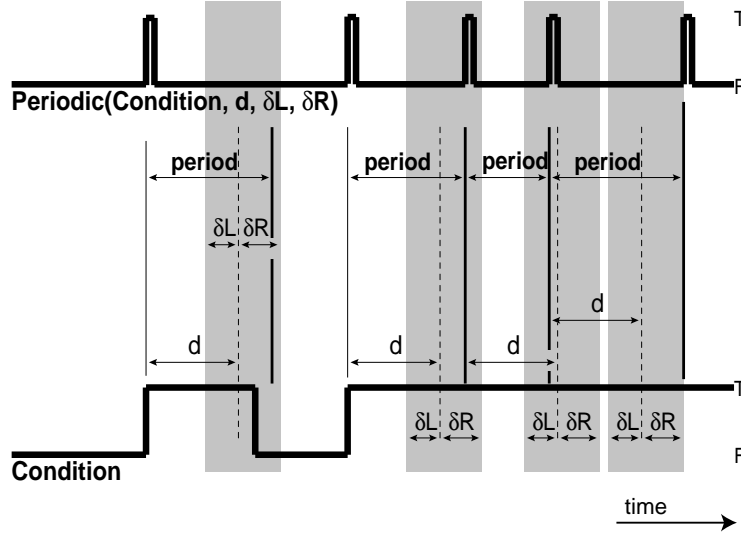


Figure 3.4: A Periodic Functional Timing Requirement

Periodic(Condition :bool, d : $\mathbb{R}^{>0}$, $\delta L, \delta R : \mathbb{R}^{\geq 0}$) :bool

where $\text{period}(\text{Periodic}_{-1} : \text{bool}) : [d - \delta L, d + \delta R]$

$\text{previous_time}(\text{Condition} : \text{bool}) : \mathbb{R}^{\geq 0}$

Initially: $\text{period} = \text{any value in } [0, \delta R]; \text{previous_time}_{-1} = 0; \text{Periodic}_{-1} = \text{FALSE}$

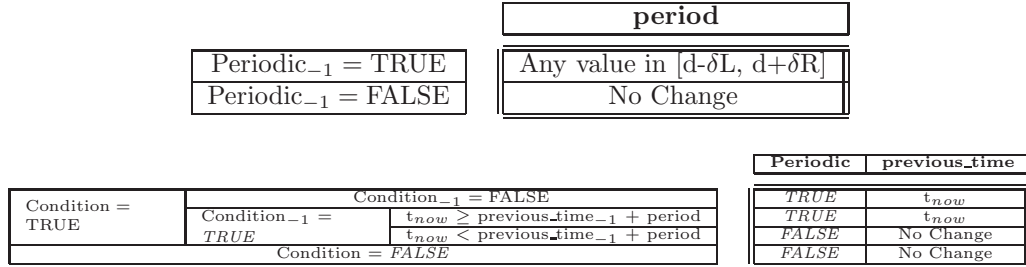


Figure 3.5: Formal Definition of “*Periodic*(Condition, d, $\delta L, \delta R$)”

ments. We have identified two different performance timing requirements, *timing resolution* and *response allowance*. These are defined and discussed below.

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

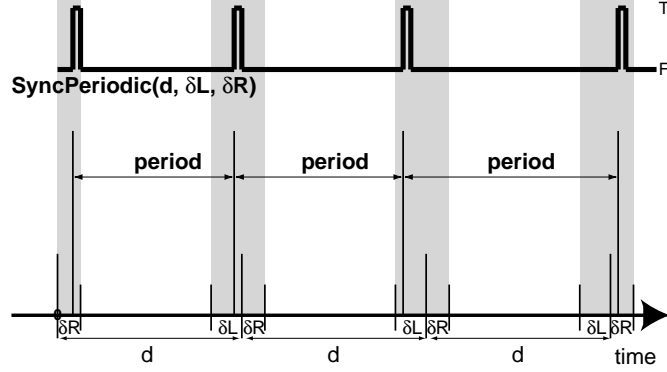


Figure 3.6: Synchronized Periodic Functional Timing Requirement

SyncPeriodic($d : \mathbb{R}^{>0}, \delta L, \delta R : \mathbb{R}^{\geq 0}$) : *bool*

where $n : \mathbb{N}$, and $\Delta : \mathbb{R}$

Initially: $n = 0$; $\Delta = \text{any value in } [0, \delta R]$; $\text{SyncPeriodic}_{-1} = \text{FALSE}$

	Δ	n
$\text{SyncPeriodic}_{-1} = \text{TRUE}$	Any value in $[-\delta L, \delta R]$	$n + 1$
$\text{SyncPeriodic}_{-1} = \text{FALSE}$	No Change	No Change

SyncPeriodic
$t_{\text{now}} \geq n \times d + \Delta$
$t_{\text{now}} < n \times d + \Delta$

Figure 3.7: Formal Definition of “*SyncPeriodic*($d, \delta L, \delta R$)”

(shown in Figure 3.8) 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 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. Figure 3.9 shows this situation.

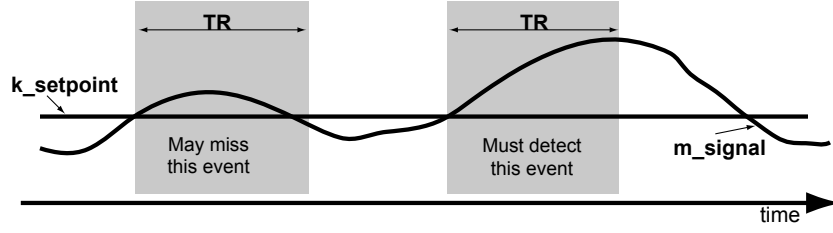


Figure 3.8: Timing Resolution for Time Continuous Monitored Variable

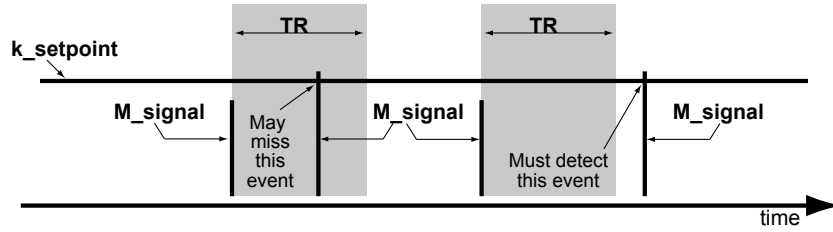


Figure 3.9: Timing Resolution for Time Discrete Monitored Variable

Note that if a monitored variable is used in determining the behavior 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.

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 behavior 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.

3.3 Implementability of *Held_For* in a Perfect Clock Environment

The major objective of this chapter is to find out under what conditions it is possible to implement the *Held_For* operator *with tolerance*, which we refer to as the implementability of *Held_For*. Our manual analysis in [29] was based on the *Perfect Clock* environmental assumption. Thus we pick it as the first environment to illustrate the interactions between functional and performance timing requirements, and its impact on the implementability of the *Held_For* operator.

Definition of the Sample Instances

Let *Sample* be a possible sequence of sample times and *Sample*(*n*) be the time of the (*n* + 1)-th sample (*n* ∈ 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 [9] for *Held_For* without tolerances.

In the example shown in Figure 3.10, 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 $[T_{min}, T_{max}]$, e.g., $Sample(6) - Sample(5) = 85 - 70 = 15$.

Definition of the Feasibility Function in the Perfect Clock Environment

The initiating event of the *Held_For* operator can take place at any time point t between two successive samples, $Sample(n)$ and $Sample(n+1)$. Let $Sample(n_d)$ be a future decision point that works for any possible event time. If the

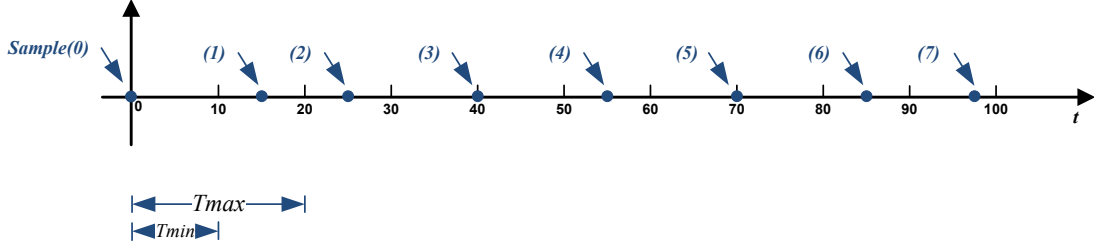


Figure 3.10: An Example of Sample Instances

Held_For requirement is implementable, $Sample(n_d)$ must satisfy the following condition:

$$\begin{aligned} \forall Sample : \forall n : \exists n_d : \forall (t | Sample(n) \leq t \leq Sample(n+1)) : \\ d - \delta L \leq Sample(n_d) - t \leq d + \delta R \end{aligned} \quad (3.1)$$

In the condition above we can think of t as the event start time and n_d is the index of the sample where we will make our decision. It is known from the earlier discussions (Section 3.2) that if the system behavior is specified in the form of (*Condition*) *Held_For* ($d, \delta L, \delta R$), with *duration* $\in [d - \delta L, d + \delta R]$, 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 Figure 3.10. As shown in the Figure 3.11, the initial event (when signal goes above the *setpoint*) occurs between $Sample(1)$ and $Sample(2)$. It is not hard to find that all the sample points before (including) $Sample(5)$ are too early for us to determine the value of the *Held_For* operator, and all the sample points after (including) $Sample(7)$ are too late for us to make decision as well. Only when $t = Sample(6)$, it is the right sample for us to make the decision.

Our research shows that implementability of the *Held_For* operator varies with different environmental assumptions. To properly state the envi-

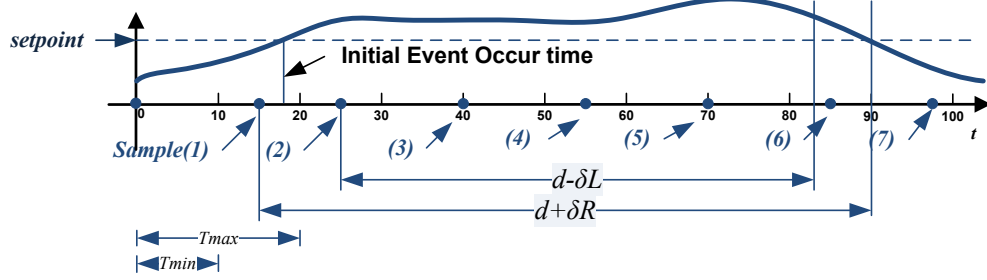


Figure 3.11: An Example of Decision Points

ronment 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 3.3.1.

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

We also assume that duration d should be larger than its tolerance, i.e., $d > \delta L \wedge d > \delta R$. To implement the *Held_For* operator, it is required that the minimum duration $d - \delta L$ should always cover at least two sample points, so that a decision can be made at the second sample point, when the initial event occurs at the first one. Therefore, we also assume $d - \delta L > T_{max}$.

3.3.1 Manual Analyses of Implementability Results

This section is based on [29]. We know from earlier discussion (Section 3.2) that if we specify behaviour in the form of $(Condition) Held_For (d, \delta L, \delta R)$, 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 Figure 3.12. Let us assume that the sample intervals are ts_0, ts_1, ts_2 , etc, where $T_{min} \leq ts_j \leq T_{max}$ for each $j \in$

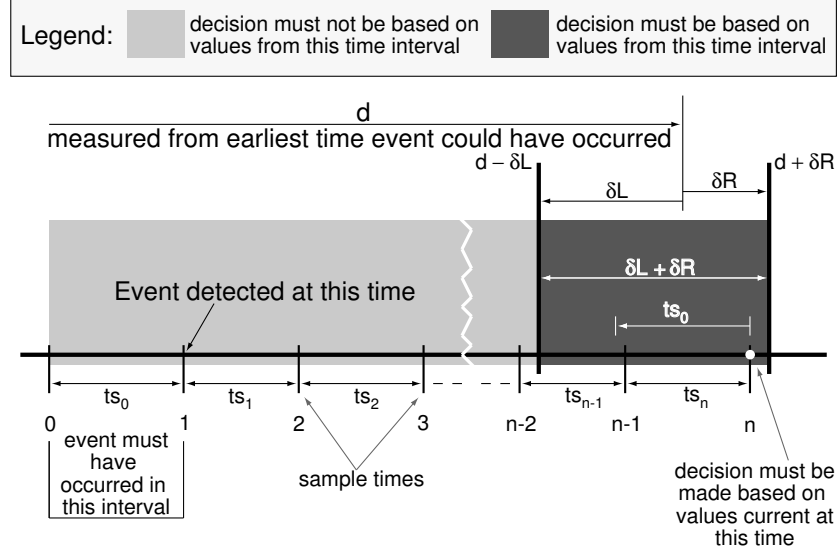


Figure 3.12: Sample Intervals Required for Sustained Events

$\{0..n\}$. 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 Figure 3.12, 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. Based on the analysis presented in [29], there are 3 cases and under each of them the condition determining the feasibility of the implementation is different. Figure 3.13 provides the scenarios for each of them.

Case 1: $0 < T_{max} \leq (\delta L + \delta R)/2$ In *Case 1*, we can guarantee there are always at least two sample points in the time interval $[d - \delta L, d + \delta R]$, based on which we can ensure the implementability of *Held_For*.

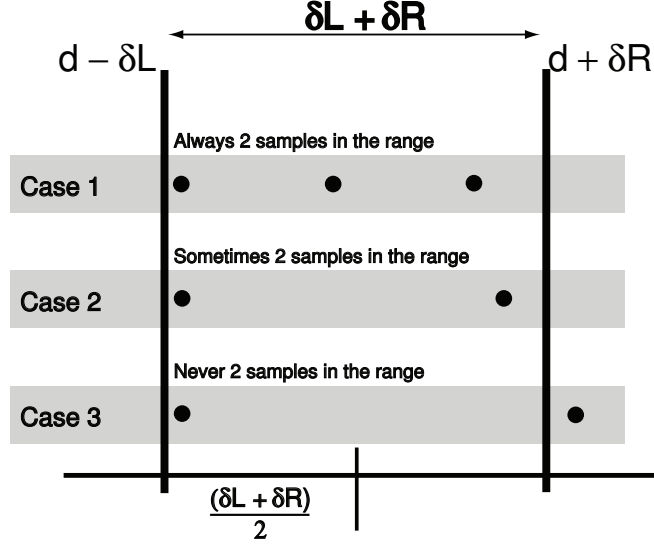


Figure 3.13: Sample Points in the Duration Interval

Case 2: $(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R)$ In *Case 2*, we still find that two sample points will be in the time interval $[d - \delta L, d + \delta R]$ under certain conditions. These will be necessary and sufficient conditions to implement *Held_For*.

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

3.3.2 Latest Results for the Perfect Clock Environment

Based on the manual analysis presented in previous section, there are 3 cases and under each of them the condition determining the feasibility of the implementation is different. For each of these cases we have formalized a theorem and verified it in PVS. In this section, we briefly introduce the latest results and how they are verified in PVS.

Case 1: $0 < T_{max} \leq (\delta L + \delta R)/2$ In *case 1*, it is easy to tell that it is always possible to implement the sustained event. In the example shown

in Figure 3.14, the upper bound on the time between two samples, T_{max} , is $(\delta L + \delta R)/2$, so the event can occur at any time inside the grey area and must be detected at the next sample time. In this example, there are 3 feasible samples and we can make the decision to trip the channel at any of these 3 samples.

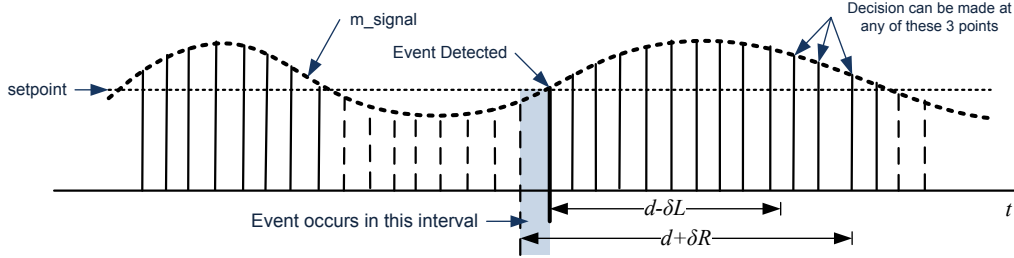


Figure 3.14: *Case 1* of the Perfect Clock Environment

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 performance timing requirement. Theorem 3.3.1 verifies that if the sampling rate is high enough to guarantee $T_{max} \leq (\delta L + \delta R)/2$, it is possible to implement the *Held_For* operator with tolerances.

Theorem 3.3.1.

$$T_{max} \leq (\delta L + \delta R)/2 \implies \text{Feasible_PerfectClock}(d)$$

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. It is still possible to find sample intervals that allow us to implement the sustained event. In this case, it is important to understand the interactions between the functional and performance requirements and their impact on the final feasibility result. Because of the complexity of *Case 2*, we will give the result here first and focus on the detailed analysis in the following section.

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 in *Case 2* is given by the following theorem:

Theorem 3.3.2.

$$\begin{aligned}
 & (\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R) \wedge T_{min} \neq T_{max} \implies \\
 & (T_{min} \geq \frac{d - \delta L}{K_{min} + 1} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R \Leftrightarrow \text{Feasible_PerfectClock}(d))
 \end{aligned}$$

Case 3: $T_{max} > (\delta L + \delta R)$ In *Case 3*, it is not possible to implement the *Held_For* operator. Since the *Feasible_PerfectClock*(d) function is defined to be *TRUE* when all the possible sample series satisfy the condition, one specific sample sequence as our counterexample will be enough to show it will be *FALSE* in *Case 3*. In Figure 3.15, it is assumed that all the intervals of any consecutive sample points are T_{max} , which leads to:

$$\text{Sample}(n) = n \times T_{max}.$$

As shown in Figure 3.15, an event can happen at any time between $(t_0, t_1]$. If there exists a sample point between $[t_1 + d - \delta L, t_0 + d + \delta R]$, then it can be our candidate decision sample point (readers can refer Figure 3.11). However, under the *Case 3*, such sample point does not exist. Because $t_1 - t_0 = T_{max}$, we can get

$$t_0 + d + \delta R = t_1 - T_{max} + d + \delta R < t_1 - (\delta L + \delta R) + d + \delta R.$$

Therefore, we can prove $t_1 + d - \delta L > t_0 + d + \delta R$ as shown in the figure. This means that the interval $[t_1 + d - \delta L, t_0 + d + \delta R]$ to contain decision sample points is empty in *Case 3*.

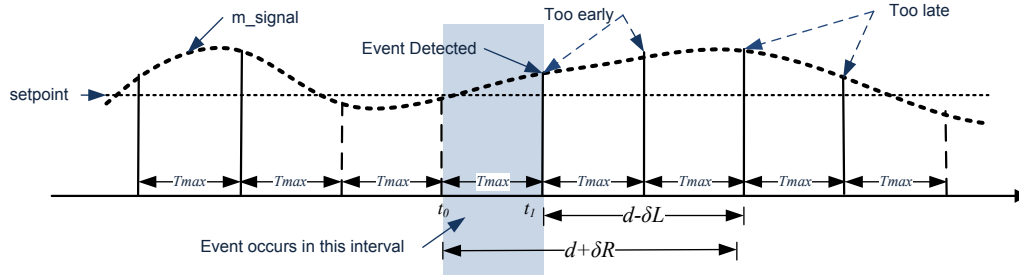


Figure 3.15: *Case 3* of the Perfect Clock Environment

Theorem 3.3.3 formalizes the *Case 3* feasibility result.

Theorem 3.3.3.

$$T_{max} > \delta L + \delta R \implies \neg \text{Feasible_PerfectClock}(d)$$

Theorems 3.3.1, 3.3.2 and 3.3.3 are all verified using the theorem proving tool PVS. During this process, with the help of PVS, we identified some differences from the results presented in [29] for Case 2.

3.3.3 PVS Aided Analysis for *Case 2*

In *Case 2*, the feasibility condition will be more restrictive than for *Case 1*. In the example shown in Figure 3.16, there is only one possible candidate sample point to make the decision, otherwise it will be too early or too late.

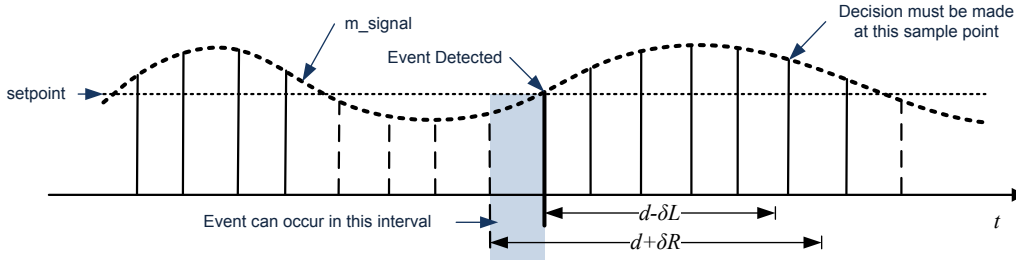


Figure 3.16: *Case 2* of the Perfect Clock Environment

The initial discussion of *Case 2* can be found in [29]. Based on that, our first attempt to formalize and prove the feasibility result for *Case 2* is shown in Conjecture 3.3.4:

Conjecture 3.3.4.

$$(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R) \wedge T_{min} \neq T_{max} \implies \\ (K_{min} = K_{max} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R \Leftrightarrow \text{Feasible_PerfectClock}(d))$$

We were NOT able to prove Conjecture 3.3.4 completely. During our PVS verification process, $K_{min} = K_{max} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R$ was shown to be a sufficient feasible condition (Theorem 3.3.5), but we could not prove that it is necessary. This provided us with the following result.

Theorem 3.3.5.

$$\begin{aligned}
 & (\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R) \wedge T_{min} \neq T_{max} \implies \\
 & (K_{min} = K_{max} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R \implies \text{Feasible_PerfectClock}(d))
 \end{aligned}$$

In Chapter 2, we introduced how PVS can help to debug unprovable obligations. Using the same approach, we identified an interesting boundary condition which is also feasible when $K_{min} \neq K_{max}$. Figure 3.17 shows an example based on this new boundary condition. The scenario constructed is $T_{max}=3.1$, $T_{min}=3.0$, $d=24.09$ and $\delta L = \delta R=3.09$. Then

$$K_{min} = \lfloor \frac{d - \delta L}{T_{max}} \rfloor = 6 \neq 7 = \lfloor \frac{d - \delta L}{T_{min}} \rfloor.$$

The upper part of the figure shows a case that the sample interval is always T_{min} and the lower part shows the case when the sample interval is always T_{max} . Starting from the sample in which the event is detected, it is not hard to find that the 7th sample point (t_7) is the feasible point, using the fact that $t_7 - t_0 \in [21, 21.7]$. Assuming the event in the physical domain occurs right at t_0 , so that $t_0 = t_{event}$, t_7 is the feasible point since $t_7 - t_0 = d - \delta L$. If t_{event} happens earlier than t_0 , again we can conclude t_7 is feasible. Because $t_0 - t_{event} < T_{max}$ and $t_7 - t_0 \leq T_{max} \times 7$, we can get $t_7 - t_{event} < 24.8 < d + \delta R$, where 24.8 is the longest elapsed time that could occur in the case shown in the lower part of the figure.

After verifying this special boundary example in PVS. We formalized our findings in a theorem that states the existence of a feasible implementation when $K_{max} \neq K_{min}$:

Theorem 3.3.6.

$$\begin{aligned}
 & (\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R) \wedge T_{min} \neq T_{max} \implies \\
 & (K_{max} = K_{min} + 1 \wedge K_{max} \times T_{min} = d - \delta L \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R \\
 & \implies \text{Feasible_PerfectClock}(d))
 \end{aligned}$$

With this PVS verified special case of feasibility, we noticed that it only occurs when $K_{max} = K_{min} + 1$ and $K_{max} \times T_{min}$ must be exactly $d - \delta L$.

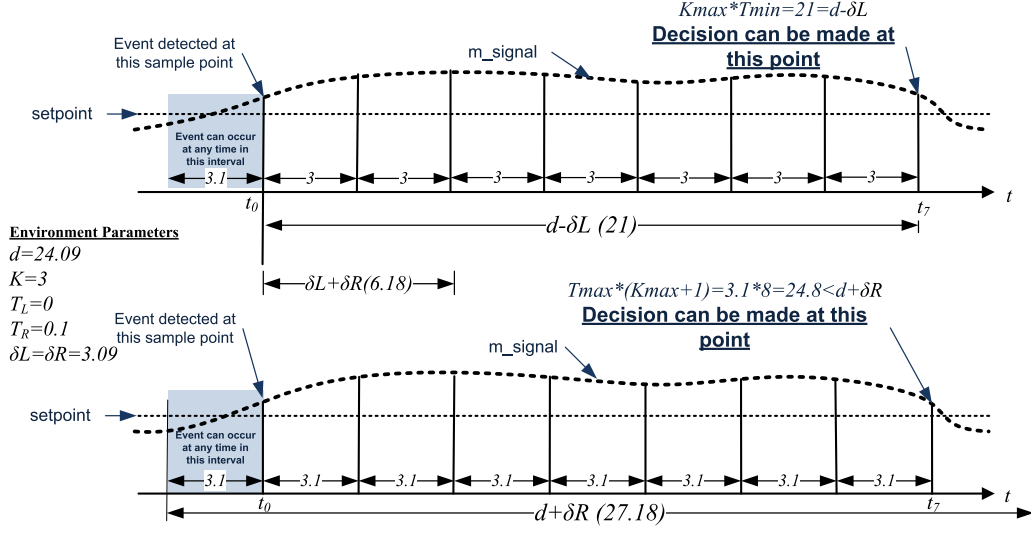


Figure 3.17: *Case 2* of the Perfect Clock Environment (Boundary Example)

This greatly restricts the application of this boundary result in any industrial example. However, for the completeness of our research results, this scenario helps us to conclude the necessary and sufficient condition of the existence of a feasible sample point. Lemma 3.3.7 was developed and proved in PVS to simplify this necessary and sufficient condition:

Lemma 3.3.7.

$$K_{max} = K_{min} \vee K_{max} = K_{min} + 1 \wedge K_{max} \times T_{min} = d - \delta L \Leftrightarrow T_{min} \geq \frac{d - \delta L}{K_{min} + 1}$$

With Lemma 3.3.7, Theorem 3.3.5 and Theorem 3.3.6, it is straightforward to get the final revised version of the *Case 2* result as shown in Theorem 3.3.2.

3.3.4 Examples of Feasible Sample Interval Ranges

This section is based on [29]. 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 (Figure 3.18). Each graph lists $[d - \delta L, d + \delta R]$. It also shows the nominal sample intervals as labels along the x-axis, and lists the deviations as $(-\ell, +r)$. So, for sample interval $T_s=50$, with deviation $(-3, +2)$ we have $T_{min}=47$ and $T_{max}=52$. A deviation of $(-0, +0)$ indicates a constant sample interval.

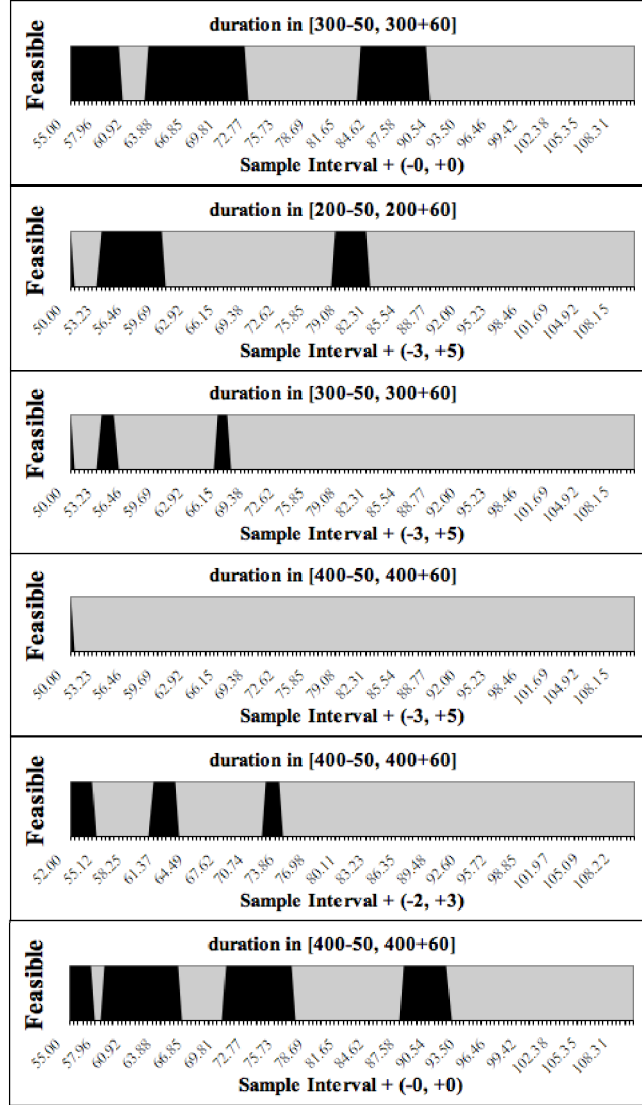


Figure 3.18: Feasible Sample Intervals for Various Durations and Tolerances

Figure 3.18 shows that in the case when $duration \in [400 - 50, 400 + 60]$,

rather than requiring the code to run with every $T_s \leq 50ms$ (a 20Hz or faster task), it is possible to detect the event with every $T_s \in [74-1, 74+2] ms$ (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 T_s , 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 interval (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 Figure 3.18, the available sample times in $[50, 110]$ are first significantly reduced, then completely eliminated.

3.4 Comparing the feasibility results in different environments

Before discussing the detailed results in the *No Clock* and *Omniscient* environments, we first provide an overview and comparison of the feasibility results in the three environments: *Perfect Clock*, *No Clock* and *Omniscient*.

To compare the results, the following two feasibility conditions are introduced.

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

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

Table 3.1 provides the comparison of the feasibility results and other important facts in each environment. The headers of the tables are *Environments*, *Case 1*, *Case 2*, *Case 3*, *Event Visibility* and *Clock Readable*. The *Environments* column lists the three environments (readers can ignore the

Imperfect Clock case at this stage and it will be discussed at the end of this section). Columns *Case 1-3* list the if and only if conditions of the feasibility function for that case. *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.

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

Table 3.1: Comparison of Implementability Results

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* based on Theorem 3.3.2 and Theorem 3.3.3. 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 3.1. 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 3 environmental assumptions for the same reasons noted in Section 3.3.2.

Readers might note that the difficulty of the implementation of the

Held_For operator increases as we progress down Table 3.1. The following theorem states this observation in a more formal way.

Theorem 3.4.1.

$$\begin{aligned} Feasible_NoClock(d) &\implies Feasible_PerfectClock(d) \wedge \\ Feasible_PerfectClock(d) &\implies Feasible_Omniscient(d) \end{aligned}$$

The relationship between the feasibility functions under different environmental assumptions determines the difficulty levels of implementing the *Held_For* in those environments. For example, $Feasible_NoClock(d) \implies Feasible_PerfectClock(d)$, and 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.

To determine the feasibility conditions for a new environment is not an easy task. Usually it is time consuming since we need to explore the relationship of the performance and functional timing requirement elements in that environment very carefully. Now we have found a relatively easy way to estimate the feasibility conditions of a new environment. By identifying the feasibility function and comparing its relationship with the feasibility functions in other environments (e.g. Theorem 3.4.1), it is possible to estimate the range of the conditions under *Cases 1, 2* and *3*. Assume that we only know the results in *Omniscient* and *No Clock* environments, so the *Perfect Clock* environment is a new environment to us. The feasibility condition of *Case 3* can be deduced immediately to be *FALSE*. The feasibility condition for *Case 2* can be expected to be in the range between *TRUE* and *Condition 2*. The same estimation approach can be applied to *Case 1* of the *Perfect Clock* environment.

As another example consider the *Imperfect Clock* case described in Section 3.1. The information available to the implementation 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!

Interested readers can also determine the feasibility function for an imperfect clock case and find the range of the feasibility condition for that class of environmental assumption in *Case 1* (shown as ??? in Table 3.1, *Cases 2* and *3* having already been inferred from the previous results).

3.5 Implementability of *Held_For* in an Omniscient Environment

The *Omniscient* environment provides full read access to the timing of the events as they happened in the physical domain. For example, we know the exact time of the “event” t when the condition becomes true but can only react at sample times. The difference in comparison to the *Perfect Clock* environment is the relaxation of the existence requirement for the decision point. For each 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 3.5.1.

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

We used a similar analysis approach for the *Omniscient* environment and the feasibility results are similar to the *Perfect Clock* environment:

Case 1: $0 < T_{max} \leq (\delta L + \delta R)/2$ In *Case 1*, with Theorems 3.4.1 and 3.3.2, the feasibility result for *Case 1* is obvious.

Theorem 3.5.1.

$$T_{max} \leq (\delta L + \delta R)/2 \implies Feasible_Omniscient(d)$$

Case 2: $(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R)$ In *Case 2*, we can refer to the perfect clock scenario and attempt to prove a similar result. However, the only theorem we have managed to prove in PVS is:

Theorem 3.5.2.

$$(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R) \wedge T_{min} \neq T_{max} \implies \\ (T_{min} \geq \frac{d-\delta L}{K_{min}+1} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R \implies Feasible_Omniscient(d))$$

After investigating the reasons why we could not prove $T_{min} \geq \frac{d-\delta L}{K_{min}+1} \wedge (K_{min}+2) \times T_{max} \leq d+\delta R$ as the necessary condition of $Feasible_Omniscient(d)$ in the *Omniscient* environment, we found that the nature of this environmental assumption provides weaker requirements for feasibility than the other environments. Recalling that in the *Omniscient* environment, the time when the actual event happens (t_{event}) is “visible” to the software domain during the implementation. In this case, it is possible to use t_{event} as the reference value to decide the feasible point. As shown in Figure 3.19, it is always possible to find at least one sample point between $[t_{event} + d - \delta L, t_{event} + d + \delta R]$, since $T_{max} < \delta L + \delta R$.

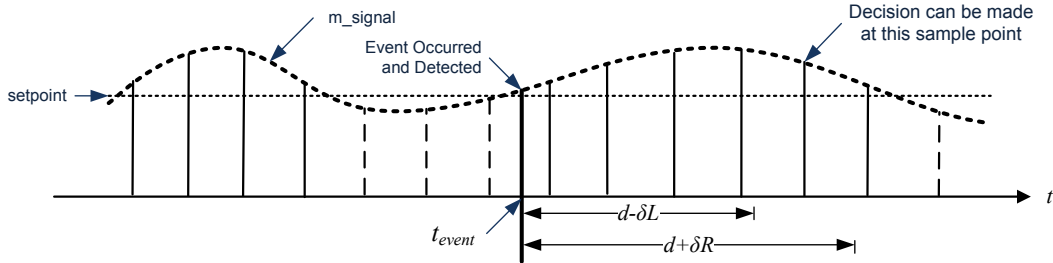


Figure 3.19: *Case 2* of the Omniscient Environment

With the above analysis, we conclude that in the *Omniscient* environment it is always possible to find a feasibility point for *Case 2*. The corresponding theorem is:

Theorem 3.5.3.

$$(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R) \wedge T_{min} \neq T_{max} \implies Feasible_Omniscient(d)$$

We note that Theorem 3.5.3 again proves our assertion that the feasibility condition becomes looser when the environment changes from *Perfect*

Clock to *Omniscient*. Later we will see that *Case 1* of the *No Clock* environment again proves our assumptions. The feasibility condition is always the same or becomes less strict when the environment changes from *No Clock* to *Perfect Clock*, or from *Perfect Clock* to *Omniscient*, as one would expect.

Case 3: $T_{max} > (\delta L + \delta R)$ In *Case 3*, it is not possible to determine a feasibility range. The proof strategy is the same as in the *Perfect Clock* environment case.

Theorem 3.5.4.

$$T_{max} > \delta L + \delta R \implies \neg \text{Feasible_Omniscient}(d)$$

3.6 Implementability of *Held_For* in a No Clock Environment

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 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_d th sample point since $Sample(n)$. Then we have the definition of the feasibility function in the *No Clock* environment as follows:

Definition 3.6.1.

$$\begin{aligned} \text{Feasible_NoClock}(d) : \text{bool} &= \exists n_d : \forall Sample : \forall n : \\ \forall (t | Sample(n) < t \leq Sample(n + 1)) : & d - \delta L \leq Sample(n + n_d) - t \leq d + \delta R \end{aligned}$$

The feasibility analyses of the *No Clock* environment are as follows.

Case 1: $0 < T_{max} \leq (\delta L + \delta R)/2$ The first interesting finding is that our *Case 1* analysis in [29] is no longer applicable under this environmental assumption. Figure 3.20 shows the analysis of the *No Clock* environment, with different sampling rates for the same input signal. To make our counting easy, we start counting at the sample point at which the event is actually detected in the software domain (as the 0th point). In the case of the bottom figure, there are 3 sample points (number 4, 5, 6) which are feasible. If we double the sample rate as shown in the upper figure, there are 6 sample points (number 8, 9, 10, 11, 12, 13) that are feasible.

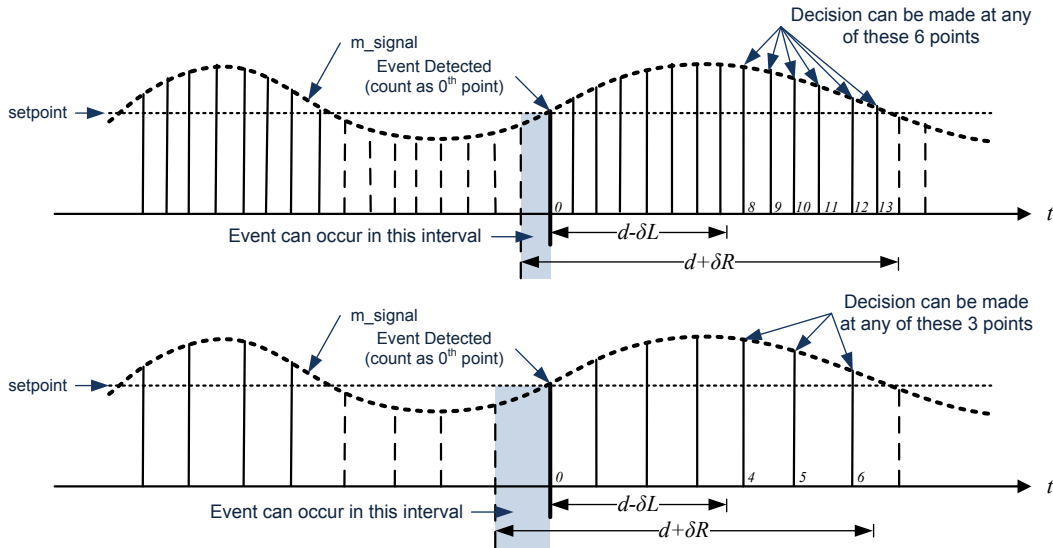


Figure 3.20: *Case 1* of the No Clock Environment

The intersection of set $\{4,5,6\}$ and $\{8,9,10,11,12,13\}$ is empty, so there is no deterministic number we can use to locate the feasible point for these two sample rates. In PVS, we further conclude and prove the theorem:

Theorem 3.6.1.

$$T_{max} \leq (\delta L + \delta R)/2 \implies \left(\left(\left\lceil \frac{d-\delta L}{T_{min}} \right\rceil + 1 \right) \times T_{max} \leq d + \delta R \Leftrightarrow Feasible_NoClock(d) \right)$$

To understand this new condition for *Case 1*, we can consider the two extreme cases, when the sample intervals are always T_{max} or T_{min} . 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 $d + \delta R$.

Case 2: $(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R)$ The feasibility result is the same as the *Perfect Clock* environment in *Case 2*. The analysis is very close to the *Case 2* scenario we have shown in Section 3.3. In Chapter 4, we will demonstrate the proving strategy that uses the result of Theorem 3.4.1 to reduce around 50% of the verification work for the *Case 2* of both environments.

Theorem 3.6.2.

$$(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R) \wedge T_{min} \neq T_{max} \implies \\ (T_{min} \geq \frac{d - \delta L}{K_{min} + 1} \wedge (K_{min} + 2) \times T_{max} \leq d + \delta R \Leftrightarrow Feasible_NoClock(d))$$

Case 3: $T_{max} > (\delta L + \delta R)$ In *Case 3* it is easy to prove the following theorem, based on Theorems 3.5.4 and 3.4.1:

Theorem 3.6.3.

$$T_{max} > \delta L + \delta R \implies \neg Feasible_NoClock(d)$$

3.7 Summary

The main contribution of this chapter is to provide *formal specifications* of a basic real-time sustained condition requirement (the *Held_For* operator *with tolerance*) and *feasibility analyses*, (the necessary and sufficient conditions for *Held_For*'s implementability). The implementability results are determined by both the implementation environment and the interaction of the timing requirements.

We presented three environmental assumptions and provided the different feasibility analyses for each of them. As the indicator of the implementability of *Held_For* in a specific environment, the *feasibility function* plays an important role in helping us complete the analysis. Also, our research shows that the relationship between feasibility functions across different environment

assumptions can help us estimate the implementability results. In Chapter 4, these feasibility analyses will be verified using PVS, and the verification work will be proved to be significantly reduced by the application of Theorem 3.4.1 (the theorem for the relationships between the feasibility functions).

The feasibility analyses show that sampling faster is not always the only option and also not always the correct choice in implementing real-time systems. Once the environmental assumption is provided, the results are determined by the interaction between the functional timing requirements (e.g., duration tolerances of the *Held_For* operator, δL and δR) and performance timing requirements (the upper and lower bound of sample intervals: T_{min} and T_{max}). The feasibility analyses of the three environments that we have studied show that it is still possible to implement the *Held_For* operator *with tolerance* when $T_{max} > (\delta L + \delta R)/2$ (shown as *Case 2*), which provides an alternative solution to the designer of real-time systems, when coping with the limitations of the hardware. On the other side, the results of *Case 1* in the *No Clock* environment show that it is not always safe to assume implementability when $T_{max} \leq (\delta L + \delta R)/2$.

Chapter 4

Formal Verification of Feasibility Results

In Chapter 3, we have precisely defined three different environmental assumptions on timing information available to the implementation and then stated the feasibility results for the *Held_For* operator *with tolerance* under each assumption. Informal justification was provided for each of the results. Our formal proof to these feasibility results are conducted with the PVS theorem proving tool. One of the benefits of using PVS is that it will help us to identify any ambiguities or inconsistencies in an informally stated mathematical theorem and finalize it to a correct version. The boundary feasibility condition in Section 3.3.3 is one such example.

In this chapter, we will describe how PVS is used to verify the results in Chapter 3. Section 4.1 provides an overview of the PVS theories and their dependencies. Section 4.2 presents two basic theories formalizing the time and sample instances. We will then review all the PVS theorems which have been created and proved to verify the feasibility results. Section 4.3 presents a roadmap to prove all the feasibility theorems across the three environments. The proofs are based on the relationships between different feasible functions (Theorem 3.4.1), so that duplicated proof work is saved.

4.1 Overview of the PVS Theories

Before presenting the detailed PVS proofs, we first provide an overview of the PVS theories and their dependencies.

Figure 4.1 provides all the main PVS theories in this thesis. The arrows show the dependencies between the theories. For example, theory **SampleInstance** depends on (i.e., imports) **Time** and **SampleInstanceOnTick** depends on both **ClockTick** and **SampleInstance**. Theory **Time** is a base theory and does not depend on any of the theories in the figure. In this chapter, we will introduce the theories in the left part of the figure. This part will be covered in detail in the next section.

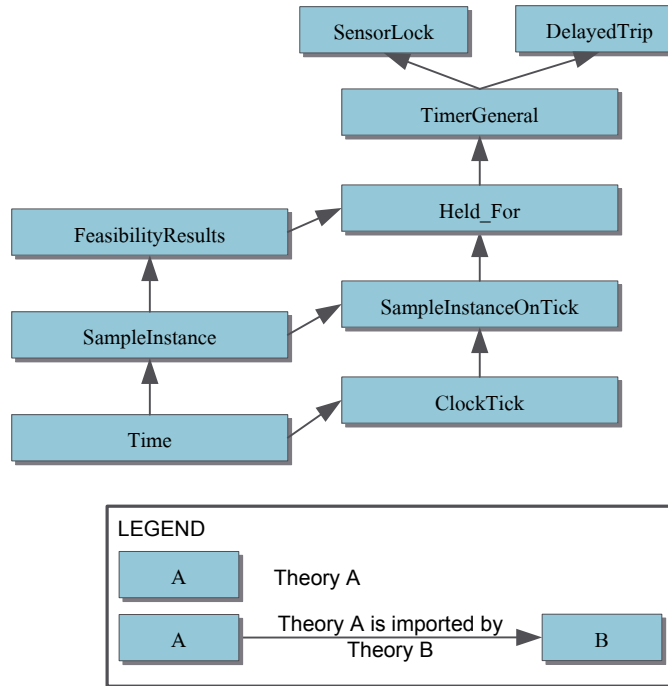


Figure 4.1: Overview of PVS Theories and their dependencies

The PVS theories **Time**, **SampleInstance** and **FeasibilityResults** contain the basic definitions and theorems that will be used in the theories on the right side.

4.2 PVS Theory for Sample Type

In this section, we introduce the PVS theories `Time`, `SampleInstance` and `FeasibilityResults`. Theory `SampleInstance` imports theory `Time`, and theory `FeasibilityResults` imports theory `SampleInstance`.

4.2.1 Time Theory

The PVS theory `Time` is shown in Figure 4.2. It defines `time` to be the built-in PVS `nonneg_real` type for our later real-time system discussion. The `nonneg_real` type is defined in PVS prelude file for the non-negative real numbers. We then define `non_initial_time` as a subtype of `time` for all non-initial time values.

```
Time: THEORY
BEGIN

  time: TYPE+ = nonneg_real

  non_initial_time: TYPE+ = posreal
END Time
```

Figure 4.2: Time Theory in PVS

4.2.2 SampleInstance Theory

The `SampleInstance` theory differs from the `Clocks` theory in [9, 34] in that the `Clocks` theory is based on the assumption of no sampling jitters during the implementation of real-time systems, i.e., the system maintains a constant (non-negative real number) sample interval. In Section 3.3, we considered the case when the interval between any two consecutive sample points is not constant, but bounded with lower and upper bound limits, i.e., $\forall n : \text{Sample}(n+1) - \text{Sample}(n) \in [T_{min}, T_{max}]$. The PVS theory `SampleInstance` is introduced to accommodate this new sample type. In this section, we will also present the PVS lemmas that have been proved regarding the properties of the sample type instances.

When PVS theories are constructed to prove the feasibility results, one of the objectives is to make it as general as possible. Hence, our PVS theory should consider that $Sample(n)$ can occur at any non-negative real number. This allows us to apply the verified feasibility result to a continuous physical domain, or any discrete physical domain (e.g., arbitrarily small clock tick), provided the timing resolution in that environment is a subset of the non-negative real numbers. The PVS theory **SampleInstance** in Figure 4.3 defines the **Sample_Type**, which is the same type of sample instances introduced in Chapter 3.

```

SampleInstance[(IMPORTING Time)
                K:non_initial_time, TL, TR: {t: time | t < K}]: THEORY
BEGIN

  n, n1, n2: VAR nat

  t, t1, t2: VAR time

  Tmin: posreal = K - TL

  Tmax: posreal = K + TR

  init(x: time): bool = (x = 0)

  Sample_Type: TYPE+ =
    {c: [nat -> time] |
      c(0) = 0 AND
      (FORALL n:
        Tmin <= c(n + 1) - c(n) AND c(n + 1) - c(n) <= Tmax)}

  ...
END SampleInstance

```

Figure 4.3: SampleInstance PVS Theory

To be close to industrial practice, **Tmin** and **Tmax** in the theory are defined through an ideal sample interval **K** with the jitter tolerance **TL** and **TR**. Therefore, **K** and **TL** and **TR** can be passed as parameters when domain experts want to reuse this theory. So, **SampleInstance** imports the **Time** Theory before defining them.

Properties of Sample Points

Throughout the verification process, we encountered a large number of type check proofs. PVS generates them to ensure that our definitions and theorems are consistent. Due to the extensive Type Correctness Conditions (TCCs) in `FeasibilityResults` and other theories, it is worthwhile to introduce the theorems we have utilized to prove the properties of sample series, which are in the type of `Sample_Type`. To keep consistent with the lemmas we have proved, *Sample* represents a variable of `Sample_Type` type defined in PVS from now on.

Interval between any two sample points: It is obvious that the distance between any two sample points is in the range of $[m \times T_{min}, m \times T_{max}]$, where m is the sequence number difference of the sample points. This property is stated in Theorem 4.2.1 and theorem 4.2.2 as follows.

Theorem 4.2.1.

$$\forall Sample : \forall n, m : Sample(n + m) \geq Sample(n) + m \times T_{min}$$

Theorem 4.2.2.

$$\forall Sample : \forall n, m : Sample(n + m) \leq Sample(n) + m \times T_{max}$$

Sample function is injective. If two sample points have the same value in the time domain, we must be looking at the same sample point. In another word, *Sample* is an injective function. However, it is obvious that *Sample* function is not surjective. Let $n1$ and $n2$ be any two natural numbers, the injective theorem is shown below.

Theorem 4.2.3.

$$\forall Sample : \forall n1, n2 : Sample(n1) = Sample(n2) \implies n1 = n2$$

Relationship between time and sample. For any time point t , it is always between two consecutive samples.

Theorem 4.2.4.

$$\forall \text{Sample} : \forall t : \exists n : \text{Sample}(n) \leq t < \text{Sample}(n + 1)$$

The PVS lemmas corresponding to the above theorems are proved and listed in Figure 4.4.

```

Sample_Interval2: LEMMA
  FORALL (n: nat, m: nat): Sample(n + m) >= Sample(n) + m * Tmin

Sample_Interval3: LEMMA
  FORALL (n: nat, m: nat): Sample(n + m) <= Sample(n) + m * Tmax

Sample_Sequence: LEMMA
  FORALL (n1, n2: nat): Sample(n1) = Sample(n2) IMPLIES n1 = n2

TIME_BETWEEN_SAMPLE: LEMMA
  FORALL (t: time): EXISTS (n: nat): Sample(n) <= t AND Sample(n + 1) > t

```

Figure 4.4: PVS Lemmas of Sample Properties

4.2.3 FeasibilityResults Theory

The PVS theory **FeasibilityResults** contains the definition of the feasibility functions in the *Omniscient*, the *Perfect Clock* and the *No Clock* environments, as well as the PVS theorems that map to the feasibility results in Chapter 3. A complete copy of the PVS specification file is available in Appendix C. Figure 4.5 contains an extract of the theory, listing the feasibility functions corresponding to the three environmental assumptions.

As an advanced theory, **FeasibilityResults** imports the **Time** and **SampleInstance** theories that have been introduced in the previous section. It also defines a new type, **Duration** in PVS. Based on the discussion we have for Definition 3.3.1 in Section 3.3, we define **Duration** as a subtype of **time** and require that a variable of type **Duration** **d** should satisfy **d > delta_R** and **d - delta_L > Tmax**.

We will walk through the PVS theorems of the feasibility results, in the same environment sequence we have discussed in Chapter 3 (first *Perfect Clock*, then *Omniscient* and finally *No Clock*).

```
FeasibilityResults[(IMPORTING Time) K: non_initial_time, TL,
                  TR: {t: time | t < K}, delta_L, delta_R: time]: THEORY
BEGIN

  IMPORTING SampleInstance[K, TL, TR]

%% Variable Declaration
  Duration: TYPE = {du: time | du > delta_R AND du - delta_L > Tmax}
  d: VAR Duration
  n, n0: VAR nat
  t : VAR time
  Sample: VAR Sample_Type

%% Feasibility Functions
  Feasible_Omniscient(d): bool =
    FORALL Sample:
      FORALL n0:
        FORALL (t | t > Sample(n0) AND t <= Sample(n0 + 1)):
          EXISTS n:
            Sample(n) - t >= d - delta_L AND Sample(n) - t <= d + delta_R

  Feasible_PerfectClock(d): bool =
    FORALL Sample:
      FORALL n0:
        EXISTS n:
          FORALL (t | t > Sample(n0) AND t <= Sample(n0 + 1)):
            Sample(n) - t >= d - delta_L AND Sample(n) - t <= d + delta_R

  Feasible_NoClock(d): bool =
    EXISTS n:
      FORALL Sample:
        FORALL n0:
          FORALL (t | t > Sample(n0) AND t <= Sample(n0 + 1)):
            Sample(n + n0) - t >= d - delta_L AND
            Sample(n + n0) - t <= d + delta_R

%% Feasibility Results in each environment
%% .....
END FeasibilityResults
```

Figure 4.5: FeasibilityResults PVS Theory

Perfect Clock

The discussion of the feasibility results in the *Perfect Clock* environment was broken down into 3 cases in Section 3.3.2. To verify the results in PVS, we follow the same approach and create 3 PVS theorems mapping to Theorems 3.3.1, 3.3.2 and 3.3.3. They are shown in Figure 4.6. We merged all of them into one theorem `PerfectClock_ALLCASES` and this theorem will be applied in the Chapter 5 to produce more advanced results.

```

PerfectClock_CASE1: THEOREM
  Tmax <= (delta_L + delta_R) / 2 IMPLIES Feasible_PerfectClock(d)

PerfectClock_CASE2: THEOREM
  (delta_L + delta_R) / 2 < Tmax &
  Tmax <= delta_L + delta_R AND Tmin /= Tmax
  IMPLIES
  (Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
   (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R
   IFF Feasible_PerfectClock(d))

PerfectClock_CASE3: THEOREM
  Tmax > delta_L + delta_R IMPLIES NOT Feasible_PerfectClock(d)

PerfectClock_ALLCASES: THEOREM
  Tmax /= Tmin IMPLIES
  ((Tmax <= (delta_L + delta_R) / 2 OR
   ((delta_L + delta_R) / 2 < Tmax AND
    Tmax <= (delta_L + delta_R) AND
    Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
    (Kmin(d) + 2) * Tmax <= d + delta_R))
   IFF Feasible_PerfectClock(d))
  
```

Figure 4.6: PVS Theorems: Feasibility in Perfect Clock Environment

Omniscient

The discussion of the feasibility results in the *Omniscient* environment has been broken down into 3 cases in Section 3.5. To verify the results in PVS, we follow the same approach and create 3 PVS theorems mapping to Theorems 3.5.1, 3.5.2 and 3.5.4. They are shown in Figure 4.7.

```

Omniscient_CASE1: THEOREM
  Tmax <= (delta_L + delta_R) / 2 IMPLIES Feasible_Omniscient(d)

Omniscient_CASE2: THEOREM
  (delta_L + delta_R) / 2 < Tmax &
  Tmax <= delta_L + delta_R AND Tmin /= Tmax
  IMPLIES Feasible_Omniscient(d)

Omniscient_CASE3: THEOREM
  Tmax > delta_L + delta_R IMPLIES NOT Feasible_Omniscient(d)

```

Figure 4.7: PVS Theorems: Feasibility in Omniscient Environment

No Clock

The discussion of the feasibility results in the *No Clock* environment has been broken down into 3 cases in Section 3.6. To verify the results in PVS, we follow the same approach and create 3 PVS theorems mapping to Theorems 3.6.1, 3.6.2 and 3.6.3. They are shown in Figure 4.8.

```

NoClock_CASE1: THEOREM
  Tmax <= (delta_L + delta_R) / 2 AND Tmin /= Tmax IMPLIES
  ((ceiling((d - delta_L) / Tmin) + 1) * Tmax <= d + delta_R IFF
   Feasible_NoClock(d))

NoClock_CASE2: THEOREM
  (delta_L + delta_R) / 2 < Tmax &
  Tmax <= delta_L + delta_R AND Tmin /= Tmax
  IMPLIES
  (Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
   (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R
   IFF Feasible_NoClock(d))

NoClock_CASE3: THEOREM
  Tmax > delta_L + delta_R IMPLIES NOT Feasible_NoClock(d)

```

Figure 4.8: PVS Theorems: Feasibility in No Clock Environment

4.3 Roadmap to Prove the Feasibility Theorems

PVS Theorems: Relationships of Feasibility Functions

Two PVS theorems shown in Figure 4.9 are created to prove Theorem 3.4.1, which shows the relationship between the feasible functions.

```
%%-----FEASIBLE FUNCTION RELATIONSHIPS---

NoClock_Implies_PerfectClock: LEMMA
Feasible_NoClock(d) IMPLIES Feasible_PerfectClock(d)

PerfectClock_Implies_Omniscient: LEMMA
Feasible_PerfectClock(d) IMPLIES Feasible_Omniscient(d)
```

Figure 4.9: PVS Theorems: Relationship between Feasible Functions

The proving work of the feasibility results can be reduced by around 50%, if we properly instantiate the proved PVS theorems

`NoClock_Implies_PerfectClock` and `PerfectClock_Implies_Omniscient`. To demonstrate this, it is better to review *Case 1*, *Case 2* and *Case 3* of the three environments all together.

4.3.1 Proving Strategy for *Case 1*

Figure 4.10 shows the strategy to prove the *Case 1* theorems in each environments. There are 3 types of the theorems in terms of the PVS proof work: *Base Theorems*, *General Theorems* and *Target Theorems*.

Base Theorems are ones where we have to construct the proof from scratch. In most of the cases, we cannot utilize the existing theorems to easily get them proved. `PerfectClock_CASE1` and `NoClock_CASE1` are this type of theorem. The proof of `PerfectClock_CASE1` takes 31 PVS commands. Comparing to the base theorems of *Case 2*, this theorem's proof is relatively simple. However, it would be nice if we do not need to repeat similar proof work for `Omniscient_CASE1`.

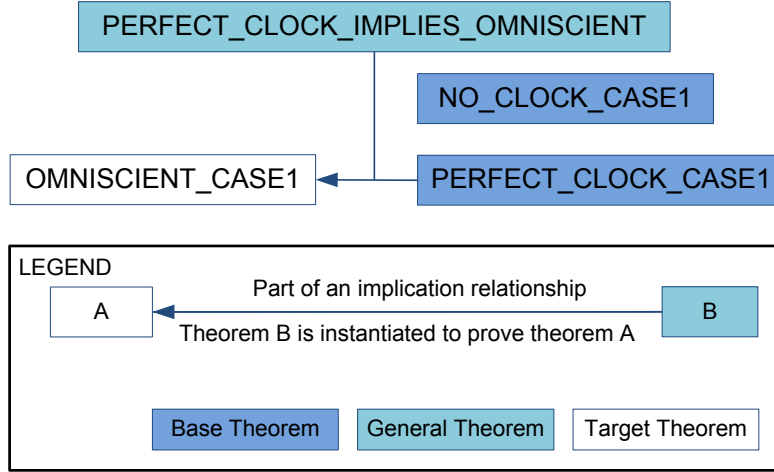


Figure 4.10: Proving Strategy for *Case 1*

General Theorems, such as `PerfectClock_Implies_Omniscient`, contain the general results of the feasibility functions across the three environments. In some cases, they can make our target PVS theorems easy to prove. Considering theorem `Omniscient_CASE1` as a target theorem, its proof work is simplified after instantiating the base theorem `PerfectClock_CASE1` and the general theorem `PerfectClock_Implies_Omniscient` (only takes 6 PVS commands to complete).

4.3.2 Proving Strategy for *Case 2*

The PVS proof of the feasibility theorems in *Case 2* is the most complicated one in the three cases. The approach here is again to reduce the duplicated work as much as possible. We will skip the introduction on how to prove `Omniscient_CASE2` since it is relatively easy. Readers can review the complete proof files from the attached CD. The focus here is on how we prove theorems `PerfectClock_CASE2` and `NoClock_CASE2`.

Each theorem will be broken down into two obligations by PVS. As an example, the two proof obligations of `PerfectClock_CASE2` are listed in Figure 4.11. In the proof of `NoClock_CASE2`, another two proof obligations will be generated as well. If we choose to prove all these 4 obligations from

scratch, there will be quite a lot of work. Again, like in *Case 1*, almost half of the proving work here can be simplified, by instantiating the general theorems `NoClock_Implies_PerfectClock` and `PerfectClock_Implies_Omniscient`.

```

PerfectClock_CASE2A: THEOREM
(delta_L+delta_R)/2 < Tmax & Tmax <= delta_L+delta_R AND Tmin/=Tmax IMPLIES
(Feasible_PerfectClock(d)
IMPLIES Tmin >= (d-delta_L)/(Kmin(d)+1) AND (floor((d-delta_L)/Tmax)+2)*Tmax<=d+delta_R)

PerfectClock_CASE2B: THEOREM
(delta_L+delta_R)/2 < Tmax & Tmax <= delta_L+delta_R AND Tmin/=Tmax IMPLIES
(Tmin >= (d-delta_L)/(Kmin(d)+1) AND
(floor((d-delta_L)/Tmax)+2)*Tmax<=d+delta_R IMPLIES
Feasible_PerfectClock(d))

NoClock_CASE2A: THEOREM
(delta_L+delta_R)/2 < Tmax & Tmax <= delta_L+delta_R AND Tmin/=Tmax IMPLIES
(Feasible_NoClock(d) IMPLIES
Tmin >= (d-delta_L)/(Kmin(d)+1) AND (floor((d-delta_L)/Tmax)+2)*Tmax<=d+delta_R)

NoClock_CASE2B: THEOREM
(delta_L+delta_R)/2 < Tmax & Tmax <= delta_L+delta_R AND Tmin/=Tmax IMPLIES
(Tmin >= (d-delta_L)/(Kmin(d)+1) AND (floor((d-delta_L)/Tmax)+2)*Tmax<=d+delta_R
IMPLIES Feasible_NoClock(d))

```

Figure 4.11: `PerfectClock_CASE2` and `NoClock_CASE2` Break Down

In *Case 2*, only two obligations, `PerfectClock_CASE2A` and `NoClock_CASE2B`, are necessary to us in the roadmap of proving. The roadmap is shown in Figure 4.12. We can treat the two obligations as our new target theorems, because once they are proved, the rest of the work is trivial.

Based on our analysis in Section 3.3.3, there is a special boundary case when $K_{max} = K_{min} + 1$ and $K_{max} * T_{min} = d - \delta_L$. Accordingly, each proof obligation is further split into two branches¹, which correspond to two boundary cases, $K_{max} = K_{min} + 1 \wedge K_{max} * T_{min} = d - \delta_L$ (i.e., Theorem 3.3.6) and $K_{max} = K_{min}$ (i.e., Theorem 3.3.5), respectively. Lemma 3.3.7 unifies these two cases into one single condition $T_{min} \geq \frac{d-\delta_L}{K_{min}+1}$, which is corresponding to `TminAndKmax` lemma in PVS (shown in Figure 4.13).

In PVS, our verification path is close to the above analysis. The major piece of the proof work is covered by PVS theorems `PerfectClock_CASE2A_1` and `PerfectClock_CASE2A_2` (shown in the Figure 4.14) for these two branches,

¹Readers can consider branches as the sub proof obligations in PVS.

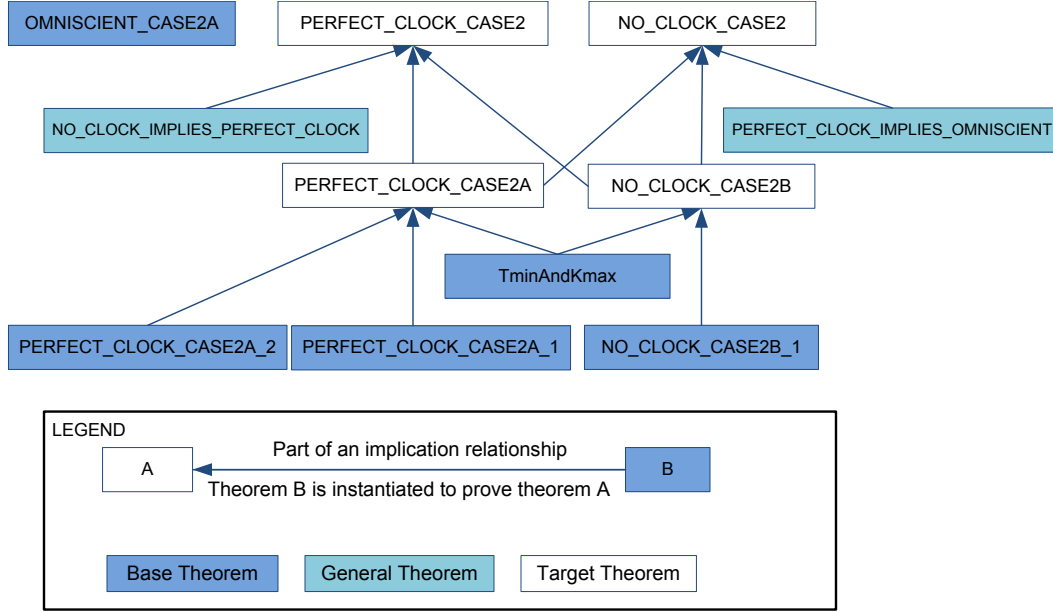


Figure 4.12: Proving Strategy for *Case 2*

TminAndKmax: LEMMA
 $(K_{\max}(d) = K_{\min}(d) \text{ OR } (K_{\max}(d) = K_{\min}(d) + 1 \ \& \ K_{\max}(d) * T_{\min} = d - \delta_L))$
 IFF $T_{\min} \geq (d - \delta_L) / (K_{\min}(d) + 1)$

Figure 4.13: PVS Theorem TminAndKmax

respectively. Then lemma `TminAndKmax` helps to merge the results to prove the target theorem `PerfectClock_CASE2A`. Neither of the theorems in Figure 4.14 is an easy task to prove in PVS.

```
PerfectClock_CASE2A_1:LEMMA
(delta_L+delta_R)/2 < Tmax & Tmax <= delta_L+delta_R IMPLIES
(Feasible_PerfectClock(d) AND floor((d-delta_L)/Tmin)*Tmin/=d-delta_L
IMPLIES FORALL (t:time|t<=Tmax AND t>=Tmin):floor((d-delta_L)/t)*t<d-delta_L)

PerfectClock_CASE2A_2: LEMMA
(delta_L+delta_R)/2 < Tmax & Tmax <= delta_L+delta_R
AND floor((d-delta_L)/Tmin)*Tmin=d-delta_L AND Tmax/=Tmin IMPLIES
(Feasible_PerfectClock(d) IMPLIES Kmax(d)=Kmin(d)+1)
```

Figure 4.14: PVS Theorems for `PerfectClock_CASE2A`

Due to the complexity of the *Case 2*, our roadmap just shows the major proving path we have taken to prove our target theorems. There are around 50 lemmas and TCCs to support the main theorems introduced above. Interested readers can review all of them in the source code available in the CD.

4.3.3 Proving Strategy for *Case 3*

Compared with *Case 1* and *Case 2*, *Case 3* is a scenario where we can utilize both the general theorems to reduce almost 2/3 of the proof work. The roadmap is shown in the Figure 4.15. We only need to prove theorem `Omniscient_CASE3`, and then instantiate the general theorem `PerfectClock_Implies_Omniscient` to prove theorem `PerfectClock_CASE3`. Taking the same approach, `NoClock_CASE3` can be proved trivially by instantiating theorems `NoClock_Implies_PerfectClock` and `PerfectClock_CASE3`.

To most readers, any theorems in *Case 3* should be very trivial. However, the formalized PVS theorem proving process is not that simple, because PVS will do all the type checks and create related proof obligations. Table 4.1 shows that it takes 81 PVS commands to prove `Omniscient_CASE3`. With the help of general theorems, the proof works of `PerfectClock_CASE3` and `NoClock_CASE3` have been reduced to 6 commands each.

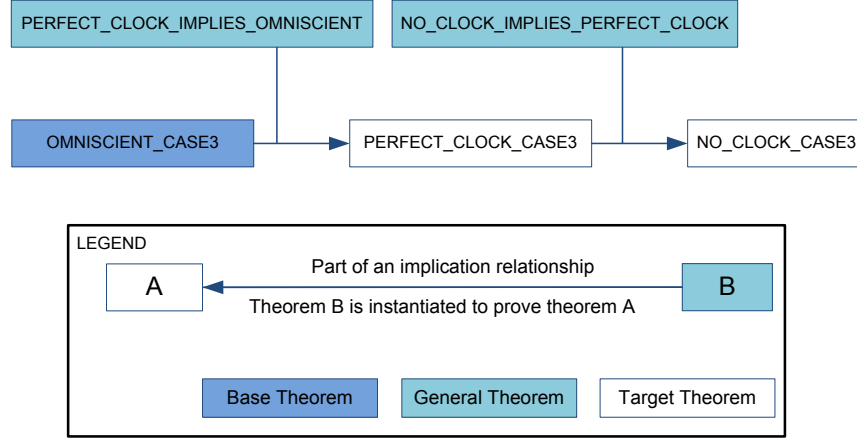


Figure 4.15: Proving Strategy for *Case 3*

<i>PVS Theorem Name</i>	<i>Number of the proof commands</i>
Omniscient_CASE3	81
PerfectClock_CASE3	6
NoClock_CASE3	6

Table 4.1: Comparison of the Proof Work of *Case 3*

4.4 Summary

In this chapter we formalized three environmental assumptions and verified the feasibility results in each of them using PVS. The verification strategy we took was to prove the general theorems that reveal the relationships of these three environments first, then applied them to the proof to reduce the verification work. Based on our rough calculation, we saved from 30% to 60% of the required work, based on the cases we explored.

Table 4.2 provides a summary of the verification cost of *Cases 1, 2* and *3* of each environment.² In the table, the number of the PVS commands to prove the theorem is listed in normal font size and the subscript number beside it is the effort of the theorem which is instantiated, that is to simplify the proof, together with general theorems. For example, in *Case 2A* of the *Per-*

²The statistics data does not include the effort of the lemmas, theorems and TCC proofs of imported theories.

fect Clock environment, the number of the PVS commands to prove theorem **PerfectClock_CASE2A** is 897. Then this theorem, and the two general theorems, are instantiated to prove *Case 2A* of the *No Clock* environment. This reduces the work of the same case in *No Clock* to only 13 PVS commands. Therefore, in the figure, the effort of *Case 2A* in the *No Clock* case is listed as 13₈₉₇. It is obvious from the table that with the proper application of the general theorems, the amount of the verification work has been significantly reduced.

<i>Environmental Assumptions</i>	<i>Case 1</i>	<i>Case 2A</i>	<i>Case 2B</i>	<i>Case 3</i>
<i>Omniscient</i>	6 ₃₁	15		81
<i>Perfect Clock</i>	31	897	17 ₉₉	6 ₈₁
<i>No Clock</i>	159	13 ₈₉₇	99	6 ₈₁

Table 4.2: Proof Work of the Feasibility Results

Readers may be interested in the effort needed to prove the general theorems. The verification costs of the two general theorems are listed in Table 4.3, which shows the effort is close to trivial.

<i>PVS Theorem Name</i>	<i>Number of proof commands</i>
NoClock_Implies_PerfectClock	9
PerfectClock_Implies_Omniscient	12

Table 4.3: Proof Work of General Theorems

To summarize, to perform feasibility analyses is complex and timing-consuming. However, the relationships between the feasibility functions across the environments can possibly simplify some of the verification work for us. The relatively tiny amount of the effort to verify these general theorems provides a simplified and feasible way for us to possibly estimate the effort of a new environment as well. An environmental assumptions library could be established in this way for engineers to quickly estimate the implementability of the real-time timing properties, without having to carry out the actual implementation or verification.

Chapter 5

Implementing *Held_For* with Tolerance

In the previous chapters, we provided the definition of the *Held_For* operator *with tolerance* and presented the implementability results for three different environmental assumptions. In this chapter, we present how to implement the *Held_For* operator *with tolerance* in the *Perfect Clock* environment. We will 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.

In Section 5.1, we define the PVS type `tick` and refine the sample instances type based on this `tick` type. Section 5.2 presents the PVS functions `Held_For_I` and `Held_For_S` which are used in intermediate “pseudo requirements” to allow an implementation defined at sample points to be verified against the arbitrarily fast clock tick model of the Software Requirement Specification (SRS) via a two step process demonstrated in Section 2.2. Section 5.3 presents an implementation roadmap which shows how the implementation work of Chapters 5 and 6 is connected. Section 5.4 will continue to verify `Held_For_I` based on the functional and performance timing requirements of the *Held_For* operator in Section 3.2. In Section 5.5 we show how to design a software component, `Timer_I`, that implements `Held_For_I` and how to verify it in PVS. This pre-verified `Timer_I` component can then be used to

guide the design of more complex components and to decompose their design verifications into simple inductive proofs. Chapter 6 will demonstrate how to apply this pre-verified result to simplify the implementation and verification of timing requirements through two example applications.

5.1 Refining Time

5.1.1 Timing Model in Physical Domain

Consider a *discrete-time* model in the physical domain, in which the time sequence is a set of “clock ticks” with the period δt , denoted as

$$tick := \{t_0, t_1, t_2, \dots, t_n, \dots\} = \{0, \delta t, 2\delta t, \dots, n\delta t, \dots\}.$$

Here δt is an arbitrarily small positive real number, representing the step between two consecutive clock ticks. As δt approaches 0, the defined system approaches the continuous time model.

Let $t_0 = 0$ and $t_n = n\delta t$ denote the initial and the $(n+1)th$ clock ticks, respectively. To identify the initial clock tick and thereby specify the initial system state, we define the predicate

$$init(t_n) := \begin{cases} TRUE, & n = 0 \\ FALSE, & \text{otherwise} \end{cases}$$

which is *TRUE* only at t_0 .

By identifying the initial clock tick, we are able to define the system state at any clock tick value in terms of the system state at the previous tick value. To formalize the notion of “previous clock tick value”, we define the *rank* of t_n as $rank : tick \rightarrow \mathbb{N}$ where $t_n \mapsto n$. This rank will be used in proving the termination properties of the recursive functions defined over *tick*.

We also define the *next* and *pre* operators on the elements of *tick* as

follows.

$$\begin{aligned} pre(t_n) &:= \begin{cases} t_{n-1}, & n \geq 1 \\ \text{undefined}, & \text{otherwise} \end{cases} \\ next(t_n) &:= t_{n+1} \end{aligned}$$

PVS requires that all functions are total (i.e., a function is defined at every value in its domain). To meet this requirement, we use the subtype

$$noninit_elem := \{t_n \in tick \mid \neg init(t_n)\}$$

as the $pre()$ operator's domain.

The way we introduce the *tick* definition is reproduced based on the *Clock* theory in [34]. However, there is a significant difference between our time model and the ones used in [5, 34]. We will discuss the difference in Section 5.1.3.

5.1.2 PVS Theories Based on the Tick Type

Overview of the PVS theories

Now let us revisit the roadmap shown in Figure 4.1 which appeared in the previous chapter. The left part was introduced in Chapter 4. The right part provides the theory dependency tree from the basic `tick` type (in the `ClockTick` theory), up to the implementation and verification of two examples (in the `SensorLock` and `DelayedTrip` theories). The `ClockTick` theory defines the PVS type `tick`, for the *tick* type introduced in Section 5.1.1. The `SampleInstanceOnTick` theory defines a new type, `SampleTick_Type`, based on the `tick` and `SampleInstance` theories. With these two fundamental theories, an intermediate operator `Held_For_I` for the Software Requirement Specification (SRS) is defined in the `Held_For` theory (see Sections 5.2 and 5.4). Then a software component `Timer_I` that implements `Held_For_I` is formalized in PVS and completely verified in the `TimerGeneral` theory (see Section 5.5). Finally, based on all these results, we are able to demonstrate two sample applications in the `SensorLock` and `DelayedTrip` theories, respectively (see Sections 6.1 and 6.2).

Now we will review these theories one-by-one, following the roadmap.

ClockTick Theory

Figure 5.1 shows the PVS source code of the `ClockTick` theory. The `ClockTick` theory imports the `Time` theory, and defines the `tick` type as a subtype of the `time` type. PVS type `tick` corresponds to *tick*. Functions `pre`, `next` and `rank` are also defined.

Note that `delta_t` is passed as a parameter to the theory, so that it can be instantiated when the theory is reused. For more information on parameterized PVS theories, the user can refer to PVS Language Reference [19].

```
ClockTick[delta_t: posreal]: THEORY
BEGIN

  IMPORTING Time

  n: VAR nat

  tick: TYPE = {t: time | EXISTS (n: nat): t = n * delta_t}

  x: VAR tick

  init(x): bool = (x = 0)

  noninit_elem: TYPE = {x | NOT init(x)}

  y: VAR noninit_elem

  pre(y): tick = y - delta_t

  next(x): tick = x + delta_t

  rank(x): nat = x / delta_t

  time_induct: LEMMA
    FORALL (P: pred[tick]):
      (FORALL x, n: rank(x) = n IMPLIES P(x)) IMPLIES (FORALL x: P(x))

  time_induction: PROPOSITION
    FORALL (P: pred[tick]):
      (FORALL (t: tick): init(t) IMPLIES P(t)) AND
      (FORALL (t: noninit_elem): P(pre(t)) IMPLIES P(t))
      IMPLIES (FORALL (t: tick): P(t))
END ClockTick
```

Figure 5.1: ClockTick Theory

To help define the timing operators in the remaining sections, we define a set of tick predicates, denoted as $\text{pred}(\text{tick})$, to be the set of all boolean functions of tick . That is

$$\text{pred}(\text{tick}) := \{f \mid f : \text{tick} \rightarrow \{\text{TRUE}, \text{FALSE}\}\}.$$

The set $\text{pred}(\text{tick})$ is formalized as the `pred[tick]` type in PVS.

The `time_induction` proposition is a simple statement allowing us to apply induction over tick values. It says that for a tick predicate P , if (i) $P(t_0)$ is *TRUE*, and (ii) for any $n > 0$, $P(t_{n-1})$ is *TRUE* implies that $P(t_n)$ is *TRUE*, then $P(t_n)$ is *TRUE* for all t_n in tick . We will use this proposition to prove that an SRS function and an SDD function are equivalent at all tick values or sample instances or that they deviate within acceptable tolerances, in particular response allowance.

SampleInstanceOnTick Theory

Since the time in the physical domain is now being modeled as the discrete `tick` type in PVS, the sample instances should also be of `tick` type as well. Therefore, we define the PVS subtype `SampleTick_Type`, as a predicate subtype of the `Sample_Type` by requiring sample times to be of type `tick`. As a result, Figure 5.2 shows the `SampleInstanceOnTick` theory, which imports both the `SampleInstance` and the `ClockTick` theories.

5.1.3 Difference between Tick and Clock Types

The definition of type `tick` appears similar to that of type `clock` in [13], however, they have different interpretations. The `clock` type represents the sample intervals without tolerance, while for the `tick` type the time between successive instances is arbitrarily smaller than the fixed sample intervals.

The `clock` type in [9] considers a single clock frequency and each sample instance occurs at a clock value. This restriction means that the clock frequency and sample frequency must be the same as shown in Figure 5.3, and both the sample instances and the clock values form the same sequence: $0, K, 2K, \dots, nK$.


```

SampleInstanceOnTick[(IMPORTING Time) K: non_initial_time, TL,
                    TR: {t: time | t < K},
                    delta_t: {tk: non_initial_time | tk < K - TL
                              AND tk < TR + TL}]: THEORY

BEGIN

  IMPORTING ClockTick[delta_t]
  IMPORTING SampleInstance[K, TL, TR]

  t: VAR tick

  SampleTick_Type: TYPE+ =
    {S: Sample_Type | FORALL (n: nat): EXISTS (t: tick): S(n) = t}
  Sample: VAR SampleTick_Type
  ...
END SampleInstanceOnTick

```

Figure 5.2: SampleInstanceOnTick Theory

Our `tick` type considers the fact that in the real world the sampling frequency is usually different from the clock frequency. The clock tick value should be significantly smaller than the sample interval. Therefore, in the physical domain, the input signal has a timing resolution of δt , which is an arbitrarily small clock tick (see Figure 5.4). In the software domain, the sample instances are of `tick` type, but occur at a (typically) much lower rate bounded by $\frac{1}{T_{max}}$ and $\frac{1}{T_{min}}$. The interval of any two consecutive sample points may be arbitrarily larger than δt , i.e., $Sample(n+1) - Sample(n) \gg \delta t$.

5.2 Held_For operator in Physical and Software Domains

The *Held_For* operator defined in Figure 3.3 specifies the tolerance on the duration of the sustained window, and it leads to indeterminism in the implementation of the system. If the *Condition* has been sustained for the interval that is in the range $[d - \delta L, d + \delta R]$, the behavior of the *Held_For* operator is then not deterministic. For example, if the event has only been sustained for $300ms$ in Figure 3.2 (assuming $d = 300ms$ and $\delta L = \delta R = 50ms$), it is possible

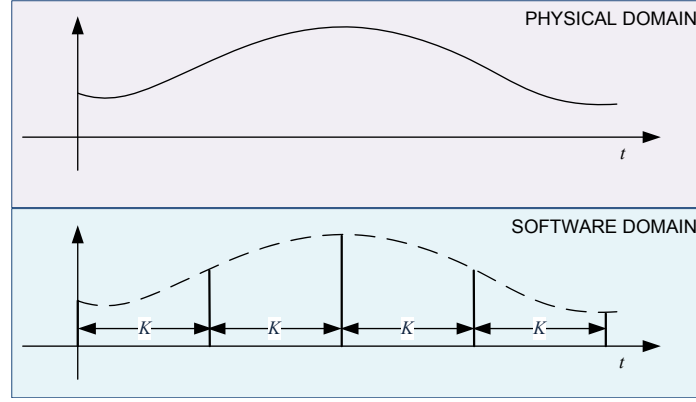


Figure 5.3: Time Model based on Clock Type

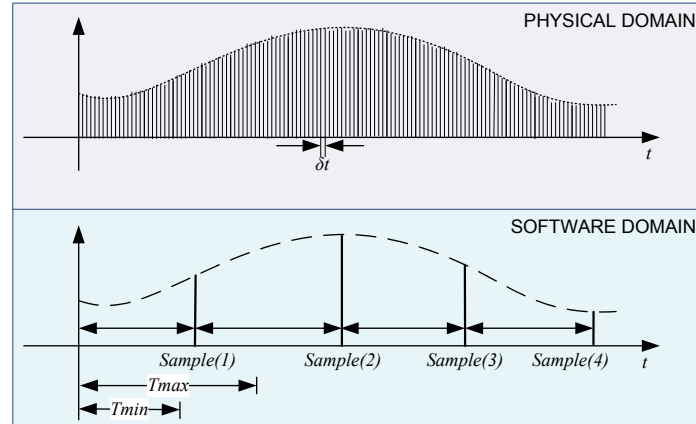


Figure 5.4: Time Model based on Tick Type

that the output is *FALSE* in one implementation, but *TRUE* in another implementation with the same sample instances. However, both implementations are reasonable for this *Held_For* definition.

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. Our objective is first to define an intermediate operator, *Held_For_I*, which is ready to be applied in the pseudo-requirements in PVS. There are some considerations here when we define this intermediate operator for the pseudo-SRS. Since the pseudo-SRS has the same data flow as the SDD, we need to consider other timing requirements as well when producing this intermediate operator for the pseudo-SRS. In other words, *Held_For_I* should be verified based on all the FTRs and PTRs (including timing resolution and response allowance). This means that both the environmental assumption and the interaction between the FTRs and PTRs need to be considered, as shown in Figure 5.5. Again it shows the importance of the *feasibility analyses* we have completed in Chapter 3. In Section 5.4, we present the verification of *Held_For_I*.

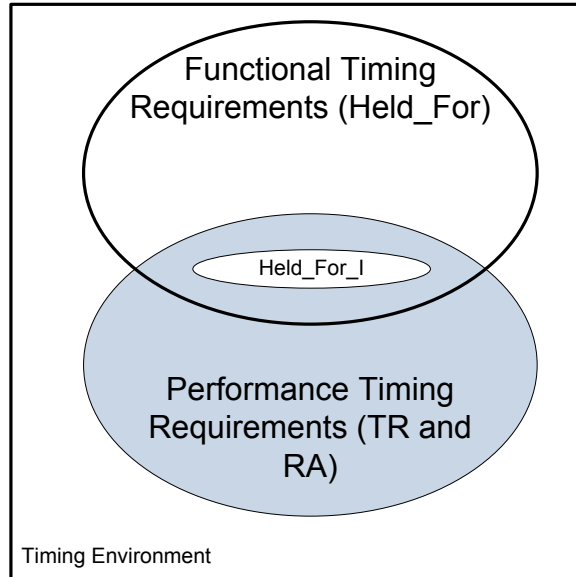


Figure 5.5: Relationship between *Held_For_I* and High Level SRS

5.2.1 Definitions

Based on the *Perfect Clock* environmental assumption, the implemented system can only refresh its output at each sample point and it maintains the same output value until the next sample arrives. To model this, we have defined two operators in PVS, `Held_For_S` and `Held_For_I`. `Held_For_S` is the basic operator that calculates and produces an output at each sample point. However, it is not defined at any clock ticks which are not at the sample points. `Held_For_I` is similar to `Held_For_S` except that it is defined at the clock tick level. It is a lifting of the `Held_For_S` operator to the tick level that for a given sampling sequence updates its value at sample points and holds it constant in between.

Before we can use the `Held_For_I` operator as an intermediate (pseudo-requirement) to replace a requirement in the SRS using the *Held_For* operator *with tolerance*, it is necessary to verify that `Held_For_I` is a refinement of *Held_For* and it meets the functional and performance requirements discussed in Section 3.2. Then we can use the `Held_For_I` operator in the pseudo-SRS as the intermediate step for the implementation verification. One may ask, for example, does the `Held_For_I` operator return *TRUE* if *m_signal* has been above the *setpoint* for longer than $d - \delta L$? To verify this using PVS, we need to formalize the condition “if *m_signal* has been above the *setpoint* for longer than $d - \delta L$ ”. For this purpose, we define the `Held_For_P` function which precisely returns the result of the sustained event in the physical domain.

Figure 5.6 shows the comparison of the three operators, P, S, I, in the physical and software domains, respectively. The physical domain part shows the mapping from the input *m_signal* to the predicate *P*. In our example, we can define the tick predicate

$$P(t : tick) : bool = m_signal(t) \geq setpoint$$

In the physical domain of Figure 5.6, `Held_For_P` with a sustained duration $d - \delta L$ will help us specify “*m_signal* has been above the *setpoint* for longer than $d - \delta L$ ”. In the software domain, the `Held_For_S` operator with a sustained duration $d - \delta L$ produces *TRUE* when the sample instances have all been *TRUE* for a period longer than $d - \delta L$. However, `Held_For_S` is not

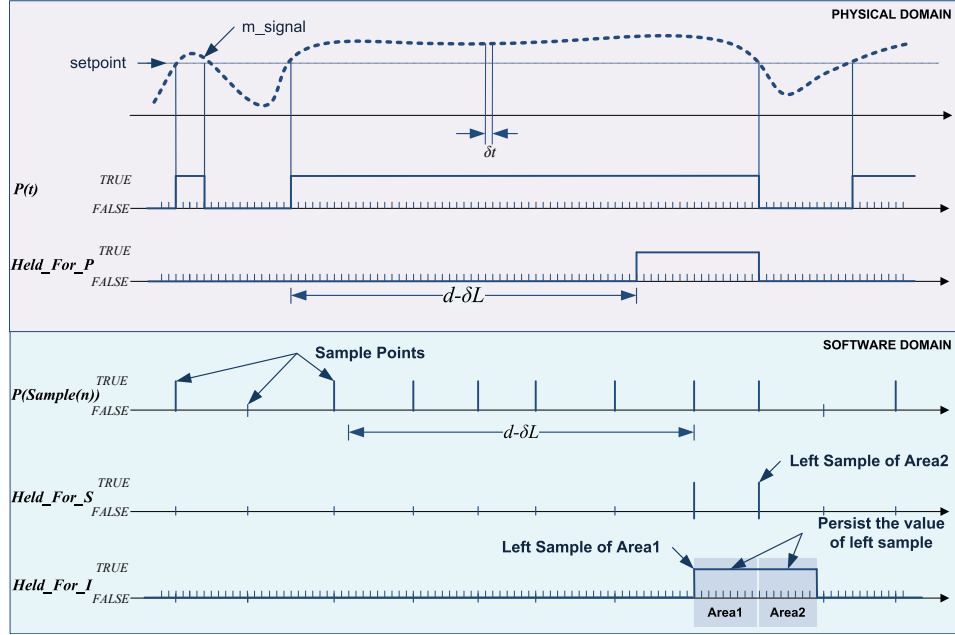


Figure 5.6: *Held_For* Versions in Physical and Software Domains

defined for any clock tick in-between two consecutive sample points. In some complicated industrial scenarios (e.g. the Delayed Trip example introduced later in Section 6.2), there are situations in which we need to model nested sustained events. In other words, the output of one *Held_For_S* operator will be the input of another *Held_For_S* operator. This is not achievable because the range of *Held_For_S* is of `pred[nat]` type, while the function domain is of `pred[tick]` type.

The solution to this problem is the *Held_For_I* operator. It produces the same output as *Held_For_S* right on any sample point. Based on that, it will maintain the output at any clock tick until the next sample point (as shown in Figure 5.6). Therefore, *Held_For_I* has the same behavior at sample points as *Held_For_S* but its range and domain allow us to specify the nested scenarios.

As to the naming convention, *Held_For_P* stands for the *Held_For* operator in the physical domain. In the software domain, *Held_For_S* is used because it produces output only at each sample point. *Held_For_I* means

that this operator is defined as an intermediate operator which is ready to be used in the pseudo code version requirements in PVS, once it has been verified against the original functional and performance timing requirements specified in Section 3.2.

Held_For_S

Figure 5.7 shows the definition of the `Held_For_S` operator in PVS. The tick predicate `P` is of type `pred[tick]` (as introduced in Section 5.1.2). The variable `duration` is of `non_initial_time` type (as introduced in Section 4.2). `Held_For_S` takes the natural number `ne` as a parameter and returns *TRUE* if there exists a sample point `Sample(n0)` in the sample history, such that the condition

$$\text{Sample}(\text{ne}) - \text{Sample}(n0) \geq \text{duration}$$

holds and the tick predicate `P` is always *TRUE* for all the sample points in between.

```
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))
```

Figure 5.7: PVS Function `Held_For_S`

Left_Sample function

For any tick value `t`, there exists a natural number `n`, such that `Sample(n) <= t` and `t < Sample(n+1)`. The `TICK_BETWEEN_SAMPLE` lemma shown in Figure 5.8 verifies this result in PVS.

```
TICK_BETWEEN_SAMPLE: LEMMA
  FORALL (t: tick): EXISTS (n: nat): Sample(n) <= t AND t < Sample(n + 1)
```

Figure 5.8: PVS Lemma `TICK_BETWEEN_SAMPLE`

In order to define the `Held_For_I` operator, we need to locate the natural number `n` that appears in the `TICK_BETWEEN_SAMPLE` lemma for the

clock tick t , which is the largest sequence number of the sample points that are on the left side of t in the timeline. For this purpose, we define the PVS function `Left_Sample`, shown in Figure 5.9.

```
Left_Sample(Sample, t): {n: nat | Sample(n) <= t AND t < Sample(n + 1)} =
  sup(LAMBDA (n: nat): Sample(n) <= t)
```

Figure 5.9: `Left_Sample` Definition in PVS

The `Left_Sample` function takes the sample instances `Sample` and the time t as input parameters, and returns the index number of the sample immediately to the left of t . Let us revisit Figure 3.10 as an example. If $t=80$, then `Left_Sample(Sample, t)=5`, since `Sample(5)=70` is the closest “left sample” in this scenario. The lemma `Left_Sample_PROPERTY3` shown in Figure 5.10 verifies that `Left_Sample(Sample, Sample(n))=n`. As a special case, `Left_Sample(Sample, 0)=0` since `Sample(0)=0`.

```
Left_Sample_PROPERTY3: LEMMA Left_Sample(Sample, Sample(n)) = n
```

Figure 5.10: PVS Lemma `Left_Sample_PROPERTY3`

PVS allows the definition of dependent types in functions. In other words, some components of a function may depend on the components defined earlier [19]. For example, the range of the function `Left_Sample` needs to be determined by the first and second parameters of the function.

We also utilize the `sup` function in the NASA Langley PVS Libraries. The NASA Langley PVS Libraries [32] are powerful extensions to the PVS base libraries. The `sup` function returns the unique least upper bound of a set. In this case, it returns exactly the result we require for `Left_Sample`.

Held_For_I

The `Held_For_I` function is defined based on the sample output of the `Held_For_S` operator. It uses the `Left_Sample` function to align the output of `Held_For_I` and `Held_For_S` at sample points and maintains the output of `Held_For_I` at that value until the next sample point. Therefore, the

`Held_For_I` function can be considered as a “lift” of the `Held_For_S` function from the sample points level to the clock tick level. Figure 5.11 shows the PVS definition of `Held_For_I`.

```
Held_For_I(P, duration, Sample)(t): bool =
  Held_For_S(P, duration, Sample)(Left_Sample(Sample, t))
```

Figure 5.11: PVS Function `Held_For_I`

`Held_For_P`

Figure 5.12 shows the definition of the PVS function `Held_For_P`. Let `t_n` and `t_j` be tick type variables, `Held_For_P(P,duration)(t_n)` is *TRUE* if and only if there exists a `t_j` such that the condition

$$t_n - t_j \geq \text{duration}$$

holds and the tick predicate `P` is always *TRUE* for all the ticks in between.

```
Held_For_P(P, duration): pred[tick] =
  LAMBDA (t_n):
    EXISTS (t_j):
      (t_n - t_j >= duration) AND
      (FORALL (t: tick | t >= t_j & t <= t_n): P(t))
```

Figure 5.12: PVS Function `Held_For_P`

5.2.2 Filtered Tick Predicate

In this section we will introduce an important assumption on the input signal `P`. The scenario shown in Figure 5.13 provides us with the motivation for this assumption. Between two consecutive sample instances, there is a chance that the input signal `P` varies rapidly and creates a “spike” whose time duration is less than the relevant timing resolution. It is impossible to detect this kind of behavior in the software domain if it occurs in-between two sample instances. As highlighted in the figure, this will cause a difference between the `Held_For_P` and `Held_For_I` operators.

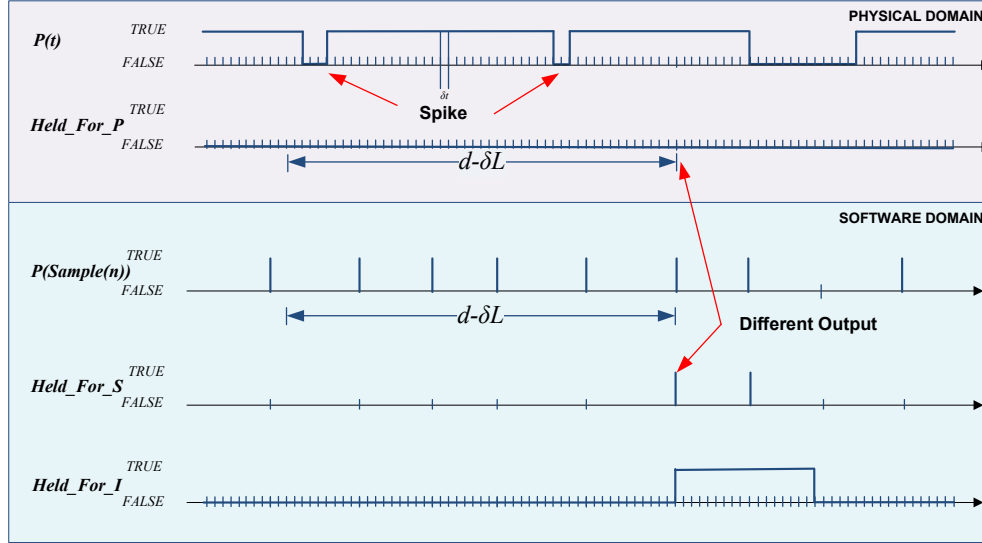


Figure 5.13: Demonstration of Filtered Tick Predicate

Therefore, one way of implementing the system is to filter out the “spike” in the input P (e.g., by a low-pass filter). However, a low pass filter introduces a phase lag resulting in a time delay from filter input to output. Thus a low-pass filter can lead to additional delay for event detection. In this case, to meet specified response allowance, one needs to consider the delay introduced by the filter in addition to the sampling delay (the maximum time between samples). Together these times gives us the worst case scenario for the delay from time the event actually occurred in physical domain, until the event is detected in software domain. We then also have to consider the response time (the time from the sample detecting the event, until the value of the controlled variable is generated and crosses the application boundary into the physical domain). If domain experts predict “spikes” of duration less than the timing resolution that need to be considered as valid events, then the upper bound of the sample interval T_{max} needs to be decreased to guarantee their capture.

In this thesis, we will assume that the input signal can be filtered as discussed above and the associated delays have been taken into account in the response allowance requirements. As a result we simplify our model

by using a *filtered tick predicate*. Figure 5.14 shows the definition of the `FilteredTickPred` type in PVS.

```
FilteredTickPred?(P: PRED[tick]): bool =
  (FORALL t0:
    P(t0) /= P(next(t0)) =>
      (FORALL (t: tick[delta_t] | t0 < t AND t <= t0 + Tmax):
        P(next(t0)) = P(t)))
  AND (FORALL (t: tick[delta_t] | t <= Tmax): P(t) = P(0))

FilteredTickPred: TYPE+ = (FilteredTickPred?)

Pf: VAR FilteredTickPred
```

Figure 5.14: Definition of `FilteredTickPred`

The definition of `FilteredTickPred` ensures that the spike scenario described in Figure 5.13 will not occur. It also restricts the input value so that it cannot change before the second sample point occurs, using the condition

$$(\text{FORALL}(t:\text{tick}[\text{delta_t}] \mid t \leq \text{Tmax}): P(t) = P(0)).$$

In this way, it is guaranteed that during the implementation process, we can capture all the critical changes that occur in the physical domain, through the defined sample instances. The variable `Pf`, of type `FilteredTickPred`, will be used in verifying the `Held_For_I` operator.

5.3 A more detailed Implementation Roadmap

The rest of Chapter 5 and Chapter 6 cover the PVS theories `Held_For`, `TimerGeneral`, `SensorLock` and `DelayedTrip` in sequence. These PVS theories contain over 200 lemmas, theorems and TCCs. Here we provide the readers with a roadmap which highlights the major theorems that connect the PVS theories in Figure 5.15.

The lower part of the figure is covered by Sections 5.4 and 5.5. Section 5.4 presents the `Held_For` theory which includes the theorems that verify `Held_For_I` against the high level requirements (FTRs and PTRs) presented

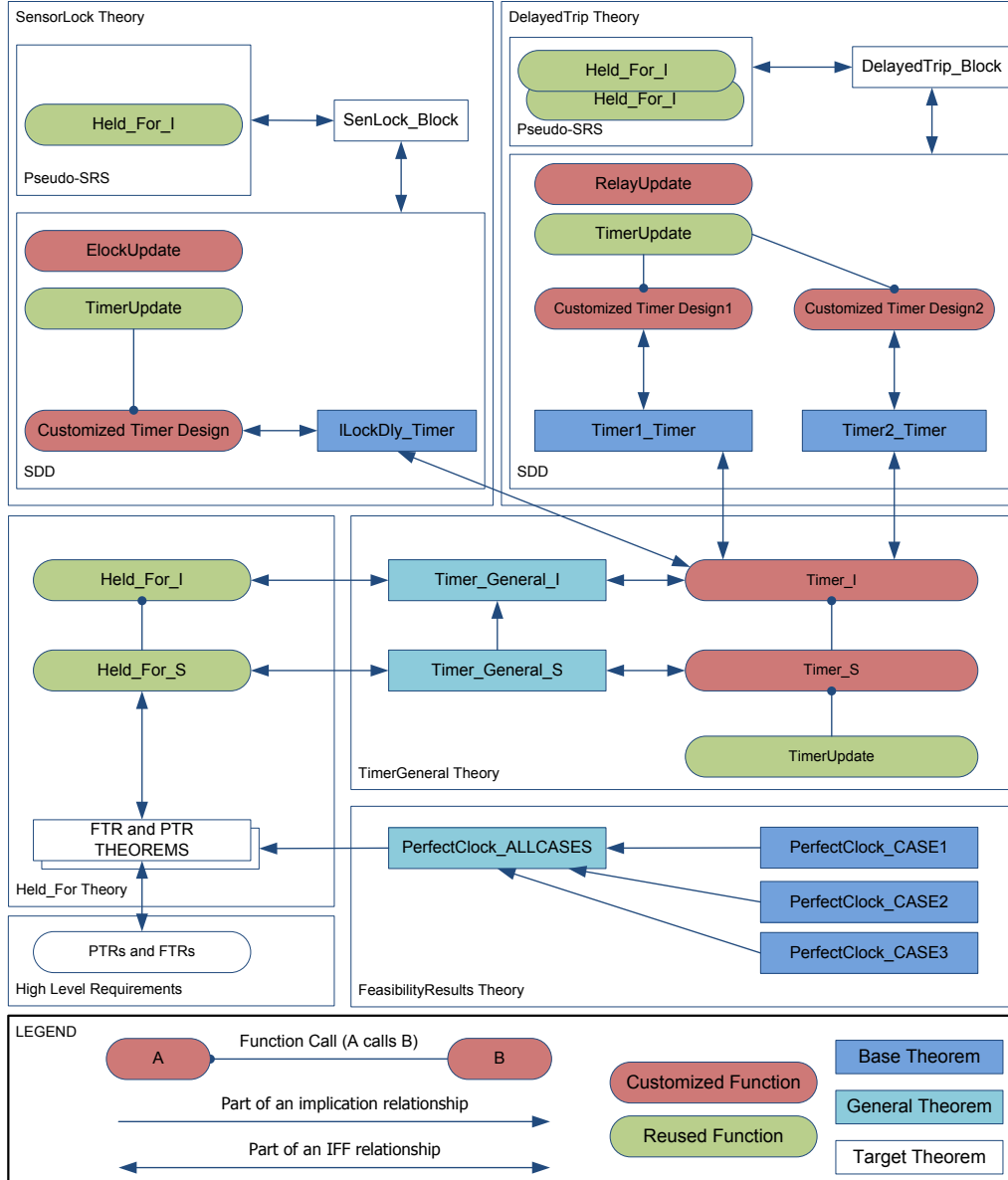


Figure 5.15: A Detailed Roadmap of Implementation

in Section 3.2. This ensures that we can apply the **Held_For_I** function in the “pseudo” version of the requirements for the two examples in PVS. Section 5.5 presents a **Timer_S** function which implements **Held_For_S**. Based on the **Timer_S** function, the **Timer_I** function is defined to implement **Held_For_I**. Two important general theorems **TimerGeneral_S** and **TimerGeneral_I** which

verify these implementation results are also provided.

In the top part of Figure 5.15, we show the pseudo-SRS and SDD breakdown diagrams of the Sensor Lock and Delayed Trip examples. Both of them use the `Held_For_I` function to specify the timing requirements in the “pseudo” version of the SRS, and customize the `Timer_I` design by reusing the function `TimerUpdate`. In the Sensor Lock System example (shown in Section 6.1), we use the `lLockDly_Timer` theorem to show the equivalence between the customized timer design and the original `Timer_I` design. The pseudo-SRS of the Delayed Trip System in PVS (shown in Section 6.2) is specified with the nested `Held_For_I` functions, which requires two customized timers to implement the system. Therefore, two PVS theorems `Timer1_Timer` and `Timer2_Timer` are created to show their equivalence to the original `Timer_I` design. We use these theorems and the general theorem `TimerGeneral_I` to provide the linkage between the `Held_For_I` in the pseudo-SRS and customized timer design in the SDD, which greatly reduces the effort for us to verify both of the examples. The effort required to verify the `TimerGeneral_I` theorem takes over 500 PVS commands to complete. This part of the work has been successfully reused in the verification of the target theorems `SenLock_Block` and `DelayedTrip_Block` of both examples.

At this stage we provide only an overview of our major results. In the following sections, we will explain Figure 5.15 in detail, step by step.

5.4 Verification of `Held_For_I` Based on High Level Requirements

Figure 5.15 shows high level requirements we introduced in Section 3.2 and the pseudo Software Requirement Specifications (SRS) of Sensor Lock and Delayed Trip Systems, in which we would like to apply the `Held_For_I` function as an intermediate step linking the our design and the requirements. In this section, we present the `Held_For` theory which contains the theorems that connect the `Held_For_I` function back to the high level requirements, based on the step one of the verification process in Section 2.2. To achieve this we have

to verify that the `Held_For_I` function (with a sustained duration of $d - \delta L$) conforms to the functional timing requirements (FTRs) and performance timing requirements (PTRs). Let Pf denote a filter tick predicate variable and P denote a tick predicate variable, we go through the following propositions to verify `Held_For_I`.

Proposition 5.4.1. *(Pf) Held_For(d, δL , δR) is TRUE, if and only if there exists a duration x ($d - \delta L \leq x \leq d + \delta R$), such that Pf is TRUE for x .*

Proposition 5.4.1 is trivial based on the functional timing requirements in Section 3.2, but it is critical to verify the `Held_For_I` operator against it. We create the PVS theorem shown in Figure 5.16, to specify that there should always be a duration x that is within the range of tolerance $[d - \delta L, d + \delta R]$, such that

$$\text{Held_For_I}(Pf, d - \delta L, \text{Sample})(t) = \text{Held_For_P}(Pf, x)(t).$$

Once again, the filtered tick predicate Pf needs to be applied here to prove $\text{Held_For_I}(Pf, d, \text{Sample})(t) \text{ IMPLIES } \text{Held_For_P}(Pf, x)(t)$, as shown in Section 5.2.2.

```
Held_For_I_VERIFY_TR0: THEOREM
  Tmax /= Tmin AND Feasible_PerfectClock(d) IMPLIES
  (FORALL (Sample: SampleTick_Type, t: tick):
    (EXISTS (x: time | x >= d - delta_L AND x <= d + delta_R):
      Held_For_I(Pf, d - delta_L, Sample)(t) = Held_For_P(Pf, x)(t))
    OR
    (NOT Pf(t) AND
     Pf(Sample(Left_Sample(Sample, t))) AND
     Sample(Left_Sample(Sample, t)) >= t - Tmax))
```

Figure 5.16: PVS Theorem `Held_For_I-TR0`

There is one scenario shown in Figure 5.17 where $\text{Held_For_P}(Pf, x)(t)$ and $\text{Held_For_I}(Pf, d - \delta L, \text{Sample})(t)$ do not agree. This only happens when t is not a sample point, $Pf(t)$ is *FALSE*, and `Held_For_I` and `Held_For_P` are both *TRUE* at the last sample point. In this case, the input Pf has just changed since the last sample point $\text{Sample}(\text{Left_Sample}(\text{Sample}, t))$,

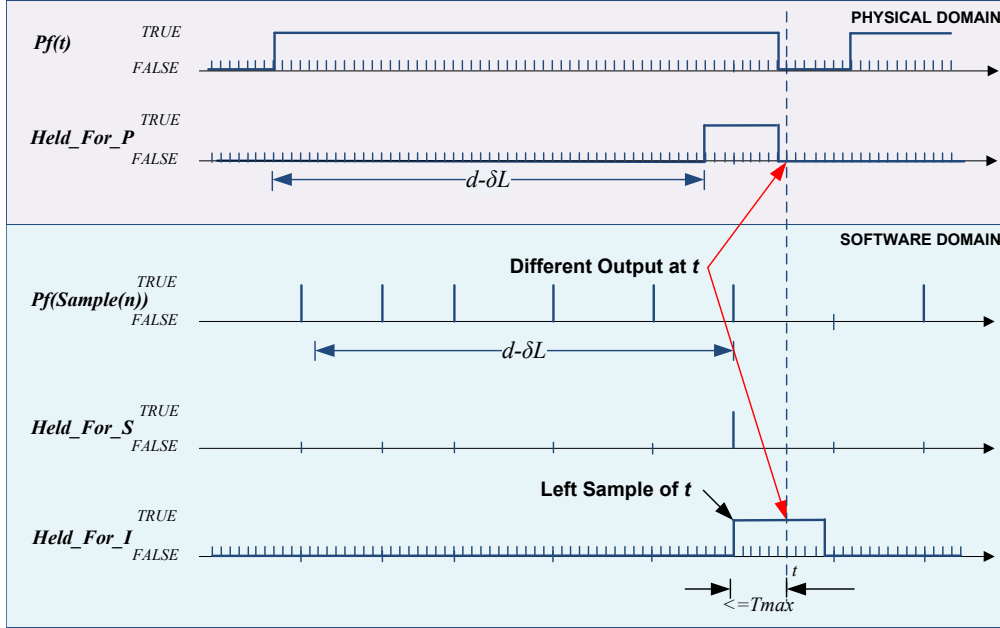


Figure 5.17: Response Allowance Scenario for PVS Theorem Held_For_I_TR0

and we can prove that the time elapsed is still within the *Response Allowance*, which is T_{max} .

Note that in the theorem we need to add `Feasible_PerfectClock(d)` as one of the premises in order to get this proposition proved in PVS. This means that only in the *Perfect Clock* environment assumption can we use `Held_For_I(Pf, d-delta_L, Sample)` to specify the high level requirement $(Pf) \text{ Held_For}(d, \delta L, \delta R)$. During the course of the PVS verification work, it was also necessary to instantiate the feasibility result in Chapter 4, the general theorem `PerfectClock_ALLCASES`. For the complete proof, readers are referred to the `Held_For.prf` file in the attached CD.

Proposition 5.4.2. $(P) \text{ Held_For}(d, \delta L, \delta R)$ is *TRUE*, if the condition P has been *TRUE* for $d + \delta R$ or longer.

As shown in the example in Figure 3.1, when the condition $m_signal \geq \text{setpoint}$ has been *TRUE* for $d + \delta R$, c_result must always trigger and equal *TRUE*. In PVS, we formalize the requirement as theorem `Held_For_I_VERIFY_FTR4`, as shown in Figure 5.18. It states that when the

tick predicate P has been *TRUE* for a period of $d + \delta R$, the *Held_For_I* operator must be *TRUE*.

```
Held_For_I_VERIFY_FTR4: THEOREM
  Tmax /= Tmin AND Feasible_PerfectClock(d) IMPLIES
  (FORALL (Sample: SampleTick_Type, t: tick):
    (Held_For_P(P, d + delta_R)(t) IMPLIES
      (FORALL (t1: tick
        | t1 >= Sample(Left_Sample(Sample, t)) AND
          t1 < Sample(Left_Sample(Sample, t) + 1)):
        Held_For_I(P, d - delta_L, Sample)(t1))))
```

Figure 5.18: PVS Theorem Held_For_I_VERIFY_FTR4

The timing condition “ $m_signal \geq setpoint$ being *TRUE* for $d + \delta R$ ” is formalized as *Held_For_P*($P, d + \delta R$)(t) in PVS. The verified consequent is that the operator *Held_For_I*($P, d - \delta L, Sample$) produces a *TRUE* value at all the clock ticks between two sample points, *Sample*(*Left_Sample*(*Sample*, t)) and *Sample*(*Left_Sample*(*Sample*, t) + 1).

Proposition 5.4.3. (P) *Held_For* ($d, \delta L, \delta R$) is *FALSE*, if the condition P has been *TRUE* for less than $d - \delta L$.

Again, based on the Figure 5.18, this functional timing requirement specifies that at any sample point, if the predicate P has continuously been *TRUE* only for an interval less than the period $d - \delta L$, the *Held_For_I* operator must not produce *TRUE*. Figure 5.19 shows the corresponding theorem formalized in PVS. The timing condition “if the predicate P has continuously been *TRUE* only for an interval less than the period $d - \delta L$ ” is formalized as *NOT Held_For_P*($P_f, d - \delta L$)(*Sample*(n)) in PVS.

```
Held_For_I_VERIFY_FTR2: THEOREM
  FORALL (Sample: SampleTick_Type, n: nat):
    (NOT Held_For_P(Pf, d - delta_L)(Sample(n)) IMPLIES
      (FORALL (t: tick | t >= Sample(n) AND t < Sample(n + 1)):
        NOT Held_For_I(Pf, d - delta_L, Sample)(t)))
```

Figure 5.19: PVS Theorem Held_For_I_VERIFY_FTR2

As shown in Figure 5.13, the input signal P could potentially include a “spike” and the sample instances may miss this behavior and result in a

difference between `Held_For_P` and `Held_For_I`. Therefore, it is important to use the filtered tick predicate variable `Pf` instead of an unrestricted tick predicate `P`. Based on this assumption, we have proved that the operator `Held_For_I(Pf, d-delta_L, Sample)` is *FALSE* at all the clock ticks from that sample point, until the next one.

Proposition 5.4.4. *Once (P) `Held_For` $(d, \delta L, \delta R)$ becomes *TRUE*, it must stay *TRUE* if the condition P stays *TRUE*.*

In PVS, we need to verify that if the `Held_For_I` operator is *TRUE* at a certain clock tick, it must maintain the value *TRUE* at the next sample, provided the predicate $P(t)$ is *TRUE* at that time. Correspondingly, we created and proved the PVS theorem `Held_For_I_VERIFY_FTR3`, as shown in Figure 5.20.

```
Held_For_I_VERIFY_FTR3: THEOREM
  Held_For_I(P, d - delta_L, Sample)(Sample(n)) AND P(Sample(n + 1))
  IMPLIES
    (FORALL (t: tick | t >= Sample(n + 1) AND t < Sample(n + 2)):
      Held_For_I(P, d - delta_L, Sample)(t))
```

Figure 5.20: PVS Theorem `Held_For_I_VERIFY_FTR3`

The verification of the performance timing requirement *Timing Resolution* becomes trivial based on the *Sample* definition that

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

This guarantees that `Held_For_I` is sampled within the *Timing Resolution* (T_{max}). The *Response Allowance* verification is covered by `Held_For_I_TR0` theorem.

Summary

In this section we have verified the `Held_For_I` operator based on the high level timing requirements. Most of our theorems can only be proved under the assumption `Feasible_PerfectClock(d)`, which emphasizes the importance of the *feasibility analyses* we developed in the previous chapters. In other words,

only when the environment satisfies the *Perfect Clock* feasible conditions, can we apply `Held_For_I` (with the duration $d - \delta L$) as an intermediate operator in the “pseudo” requirements in PVS.

5.5 Implementation of *Held_For*

To implement and verify a real-time system is not an easy task. Usually, the whole process includes requirements gathering, formalization, design and implementation. When both functional and performance timing requirements are involved, the requirements specification and design implementation will become more complicated. In [9], a modularized approach is introduced to specify and verify the timing behaviors of a real-time system. The approach is to first design a software component that implements the *Held_For* operator *without tolerance*, and verify it in PVS. Then this pre-verified component can be used as a *Implementation Template* in both designing more complex components and decomposing their design verifications.

In this section, we will present the `TimerGeneral` theory shown in the Figure 5.15. We first design a `Timer_S` to implement the `Held_For_S` function, both of them produce the results only at each sample points. As shown in the Figure 5.15, `Timer_I` is defined to implement the `Held_For_I` function. Both `Timer_S` and `Timer_I` are close to industrial practices and pseudo code language, which can be easily converted into implementation language (e.g., C++) [9]. Subsequently in Chapter 6, we will apply the `Timer_I` implementation in two more complicated examples: Sensor Lock and Delayed Trip.

5.5.1 Timer Implementation of `Held_For_I`

As a refinement of the *Held_For* operator with tolerance, the `Held_For_I` operator can only be considered as “implementation ready”, but not as a true implementation. This is because to determine the value of `Held_For_I` requires the history of the sample instances of the input $P(t)$. If the duration $d - \delta L$ is infinitely large, infinitely large memory is required to store the sample history in order to calculate the value of `Held_For_I` at the current

clock tick. Obviously this kind of implementation is not ideal.

Timer_S and TimerUpdate Functions

To implement the `Held_For_I` operator, we can first design a timer that updates at every sample instance. In Figure 5.21, the `Timer_S` function updates the 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 previous 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.

```

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

```

Figure 5.21: Timer_S Function

```

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

```

Figure 5.22: TimerUpdate Function

In our design, the values of the clock predicate at the current and last sample instance, `P(Sample(ne))` and `P(Sample(ne-1))`, are passed to `TimerUpdate` as the first and second parameters, `CurrentP` and `PreviousP`. 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.

The `Timer_S` function determines the current timer step based on two different conditions and passes it to the `TimerUpdate` function. When `ne=0` we are at the first sample instance and the timer should be set as 0. Therefore, the timer step `step=0` is passed to `TimerUpdate`. When `ne>0`, the time that has elapsed between the current sample instance and previous one needs to be passed to `TimerUpdate` as the latest timer step. Then `TimerUpdate` will determine whether this step is a valid increment for the timer or not, based on the values of `CurrentP` and `PreviousP`.

Timer_I Function

The `Timer_I` function shown in Figure 5.23 returns a tick predicate type. It will take on the value of the `Timer_S` function at the sample point and keep the same output at any clock tick going forward until the next sample point. The definition is based on `Timer_S` in a way that is similar to how `Held_For_I` depends on `Held_For_S`. The `Timer_S` and `Timer_I` functions will implement `Held_For_S` and `Held_For_I` at the sample point and tick levels, respectively.

```
Timer_I(P, Sample, TimeOut)(t): tick
= Timer_S(P, Sample, TimeOut)(Left_Sample(Sample, t))
```

Figure 5.23: `Timer_I` Function

5.5.2 Verification of the Implementation of `Held_For_S`

The verification is done by proving the `TimerGeneral_S` theorem shown in Figure 5.24. The theorem states that `Held_For_S` is *TRUE* at the current sample point if and only if the timer has timed out. Here `timeout` is of `Duration` type.

```
TimerGeneral_S: THEOREM
  Held_For_S(P, timeout - delta_L, Sample)(n) IFF
    Timer_S(P, Sample, timeout - delta_L)(n) >= timeout - delta_L
```

Figure 5.24: PVS Theorem TimerGeneral_S

Proving this theorem is more complicated than the apparently similar theorem `TimerGeneral` [9] (which is the `Timer` implementation of the *Held_For* operator *without tolerance*). We have constructed over 16 lemmas and theorems to support the proof of this theorem. Readers are referred to the complete proof code in the attached CD.

5.5.3 Verification of the Implementation of Held_For_I

The `TimerGeneral_S` theorem shown in Figure 5.24 proves that the outputs of the timer design `Timer_S` and the `Held_For_S` operator agree at the sample level. Based on this important result, it is not difficult to prove that the timer design `Timer_I` and the `Held_For_I` operator agree at the tick level. This result is shown in Figure 5.25.

```
TimerGeneral_I: THEOREM
  Held_For_I(P, timeout - delta_L, Sample)(t) IFF
    Timer_I(P, Sample, timeout - delta_L)(t) >= timeout - delta_L
```

Figure 5.25: PVS Theorem TimerGeneral_I

The proof work of this theorem is simplified after instantiating the `TimerGeneral_S` theorem plus two lemmas `Timer_RELATIONSHIP1` and `Held_For_RELATIONSHIP2A`. The `Timer_RELATIONSHIP1` lemma reveals the relationships between `Timer_S` and `Timer_I`. The `Held_For_RELATIONSHIP2A` lemma reveals the relationships between `Held_For_S` and `Held_For_I`. They are both available in the source code attached in the CD.

5.6 Summary

In this chapter, we demonstrated how to implement the real-time operator *Held_For with tolerance* under the *Perfect Clock* environmental assumption. We defined the function `Held_For_I` as an intermediate operator that can be applied to the pseudo-requirements in PVS. This function is first verified based on the high level functional and performance timing requirements as defined in Section 3.2. Note that only under the feasibility conditions of the *Perfect Clock* environment can we prove that `Held_For_I` conforms to all the high level requirements we have specified for the *Held_For* operator *with tolerance*. As the concluding theorem of the *Perfect Clock* environment, `PerfectClock_ALLCASES` is the fundamental theorem of the verification. This result shows the importance of our *feasibility analyses* in Chapter 3 and 4.

We then presented the PVS function `Timer_I` as an implementation of the `Held_For_I` function and verified this result in PVS through the general theorem `TimerGeneral_I`. The `Timer_I` design yields a relatively easy implementation of the `Held_For_I` operator. Other equivalent implementations can be defined, and, in practice, there could be a lot of 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_I`, so that designers can customize the `Timer_I` design based on different situations. In the next chapter, we present two examples, both of which are not as obvious as one might first believe, and our designs vary from the original `Timer_I` design. By utilizing the general theorem `TimerGeneral_I`, a large amount of the verification work is saved by proving the equivalence of the customized timer implementation and the original `Timer_I` implementation. As a pre-verified result, the PVS general theorem `TimerGeneral_I` is reused in both examples to reduce the required verification effort.

Chapter 6

Examples

In this chapter, we provide two examples, Sensor Lock and Delayed Trip, to illustrate the application of the *Held_For* operator and the use of the timer design and general theorem in both implementation and verification. In the Sensor Lock example (shown in Section 6.1), we will experience the difference between the SRS and SDD because they update their outputs at tick and sample level respectively. This again emphasizes the importance of the filtered tick predicate as the implementation assumption. The Delayed Trip System (DTS) example is illustrated in Section 6.2, which provides us with a case where the requirements need to be specified in a more precise level than applying a single global tolerance. The SRS of this example is specified with nested *Held_For* operators with a different tolerance for each of them. The implementation and verification process demonstrates the flexibility of our approach to handle the tolerance precisely for each of the timing properties of the system, in contrast with the global timing tolerance approach.

In both of the examples, the `Timer_I` design has been a pre-verified component to guide the implementation, and the general theorem `TimerGeneral_I` is instantiated to reduce the verification work. In Section 6.3, we provide the summary of the effort of these two examples and show that around 39% and 51% of the verification work has been saved, respectively, by the pre-verified general theorem.

6.1 Example: Sensor Lock System

The Sensor Lock System, also called *SenLock* System, appeared first in [15] as an industrial example for the PVS real-time (PVS-RT) method. The problem was redefined to be able to handle arbitrary values of a fixed sample interval in [9]. In both of the previous attempts, timing tolerance was not considered during the implementations. In this section, we will introduce the timing tolerance explicitly in the requirements and verify the design implementation.

6.1.1 Overview of the System

The *SenLock* System is a watchdog control system. As shown in Figure 6.1, it monitors the plant parameter *Sensor* and reacts to send the output “lock” to shutdown the system, if anomalous behavior is observed for the parameter *Sensor* for an extended period of time [15, 34]. Once the system produces a “channel lock” output to force the shutdown of the plant, the channel will not be “unlocked” until the manual reset button is pushed.

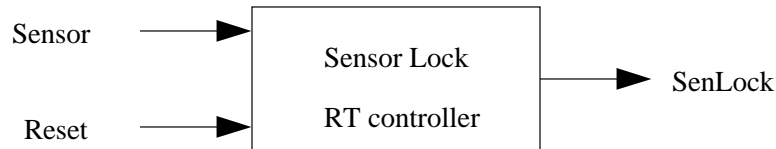


Figure 6.1: Block diagram for real-time SenLock controller

As shown in Figure 6.1, the *SenLock* real-time controller takes two boolean valued inputs, *Sensor* and *Reset*, and produces a single boolean valued output *SenLock* that is updated at every clock tick. When the value of *Sensor* is continuously *TRUE* for a number of 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 setting *Reset* = *TRUE* [15, 34]. The initial state of *SenLock* should be *TRUE*.

6.1.2 Software Requirement Specification (SRS)

Figure 6.2 provides an upgraded version of the Software Requirement Specification (SRS) of the system. This formal tabular definition is based on [9], except that the *Held_For* operator has been upgraded to the version *with tolerance* we have discussed in Section 3.2. The first row of the SRS table indicates that *SenLock* should be *TRUE* if sensor has been *TRUE* for *ldelay* units or longer. The remaining three rows indicate that only a manual reset in a safe sensor input situation can make *SenLock* *FALSE*.

Condition			Result
			<i>SenLock</i>
$(Sensor) Held_For (ldelay, \delta L, \delta R)$			<i>TRUE</i>
$\neg [(Sensor) Held_For (ldelay, \delta L, \delta R)]$	<i>Reset</i>	$\neg Sensor$	<i>FALSE</i>
		<i>Sensor</i>	No Change
	$\neg Reset$		No Change

Figure 6.2: The upgraded version SRS of SenLock System

Figure 6.3 shows the SRS of the *SenLock* System in PVS.

```

Sample: SampleTick_Type
t: VAR tick
ldelay: VAR Duration
sensor, reset: VAR PRED[tick]

SenLock_SRS(sensor, reset, ldelay)(t): RECURSIVE bool =
  IF init(t) THEN TRUE
  ELSE COND Held_For_I(sensor, ldelay - delta_L, Sample)(t) -> TRUE,
    NOT Held_For_I(sensor, ldelay - delta_L, Sample)(t) AND
    reset(t) AND sensor(t)
    -> SenLock_SRS(sensor, reset, ldelay)(pre(t)),
    NOT Held_For_I(sensor, ldelay - delta_L, Sample)(t) AND
    reset(t) AND NOT sensor(t)
    -> FALSE,
    NOT Held_For_I(sensor, ldelay - delta_L, Sample)(t) AND
    NOT reset(t)
    -> SenLock_SRS(sensor, reset, ldelay)(pre(t))
  ENDCOND
ENDIF
MEASURE rank(t)

```

Figure 6.3: PVS Definition of Sensor Lock SRS

6.1.3 Software Design Description (SDD)

In the SDD of the *SenLock* System, we define the PVS function `ELOCK` which customizes the `Timer_I` design. This function implements the timer design by calling the same `TimerUpdate` function. The output of the `ELOCK` function is defined as `SDD_State` record type, which contains three fields `Elock`, `lLockDly` and `PreviousInput`.

```

Lock_State: TYPE = {Good, Bad, Lock}
SDD_State: TYPE = [# Elock: Lock_State, lLockDly: tick, PreviousInput:bool #]
S: VAR SDD_State
sensor_now, reset_now: VAR bool

ELOCK(sensor: PRED[tick], reset: PRED[tick], ldelay: non_initial_time)
  (t): RECURSIVE
    SDD_State =
    IF init(t)
      THEN (# Elock := Lock, lLockDly := 0, PreviousInput := sensor(0) #)
    ELSE IF t = Sample(Left_Sample(Sample, t))
      THEN (# Elock
              := ElockUpdate(sensor(t),
                             reset(t),
                             ELOCK(sensor, reset, ldelay)(pre(t)),
                             ldelay,
                             t - Sample(Left_Sample(Sample, t) - 1)),
            lLockDly
              := TimerUpdate(sensor(t),
                             PreviousInput
                               (ELOCK(sensor, reset, ldelay)(pre(t))),
                             ldelay,
                             lLockDly
                               (ELOCK(sensor, reset, ldelay)(pre(t))),
                             t - Sample(Left_Sample(Sample, t) - 1)),
            PreviousInput := sensor(t) #)
    ELSE (# Elock := Elock(ELOCK(sensor, reset, ldelay)(pre(t))),
          lLockDly
            := lLockDly(ELOCK(sensor, reset, ldelay)(pre(t))),
          PreviousInput
            := PreviousInput(ELOCK
                             (sensor, reset, ldelay)(pre(t))) #)
    ENDIF
  ENDIF
  MEASURE rank(t)

```

Figure 6.4: PVS Definition of Sensor Lock SDD

As shown in Figure 6.4, our approach is to utilize the existing `TimerUpdate` function to update `lLockDly` and create the `ElockUpdate` function to update `Elock`. `PreviousInput` is updated at each sample point, which stores the value of `sensor`. Thus this field can be used to represent the value of `sensor` at

the previous sample point, which is passed to the `PreviousP` parameter of the `TimerUpdate` function to update the timer. This design consideration covers the fact that usually only the current sampled input `sensor(t)` is available for the implementation, while the previous sampled input has to be stored in the register.

The `ELOCK` function's output is refined to three states: `Good`, `Bad` and `Lock`. Among them, the `Lock` state is expected to match the SRS output `TRUE`. Just as the `TimerUpdate` function handles the timing properties, the `ElockUpdate` function (shown in Figure 6.5) is used to update the output of the *SenLock* System. The `ELOCK` function updates its timer status and output by calling both `TimerUpdate` and `ElockUpdate` functions in a modularized and flexible way. Figure 5.15 also provide a high level graphic explanation of this design consideration.

```
ElockUpdate(sensor_now: bool, reset_now: bool, S: SDD_State,
            ldelay: non_initial_time, step: time):
    Lock_State =
    TABLE
    %-----+-----+
    |NOT sensor_now AND Elock(S) = Lock AND reset_now          |Good||
    %-----+-----+
    |NOT sensor_now AND Elock(S) = Lock AND NOT reset_now      |Lock||
    %-----+-----+
    |NOT sensor_now AND NOT Elock(S) = Lock                    |Good||
    %-----+-----+
    |sensor_now AND (NOT Elock(S) = Lock AND lLockDly(S) + step < ldelay)|Bad ||
    %-----+-----+
    |sensor_now AND (Elock(S) = Lock OR lLockDly(S) + step >= ldelay)  |Lock||
    %-----+-----+
    ENDTABLE
```

Figure 6.5: PVS Definition of `ElockUpdate`

6.1.4 Implementation Assumptions

During the course of verifying the *SenLock* System, we have encountered unprovable obligations. By debugging them in PVS, we have uncovered that the behaviors of the *Reset* and *Sensor* inputs can also cause an inconsistency between the SRS and the SDD. In order to implement the *SenLock* System, we have to make following implementation assumptions:

Assumption 6.1.1. Consider $\neg \text{Sensor} \wedge \text{Reset}$ to be the combined input to the SenLock System. This input should not change its value before the second sample point.

Figure 6.6 demonstrates the unprovable obligations that we have encountered in PVS, which can help us to understand the necessity of this assumption. In the figure, the input $\text{Reset} \wedge \neg \text{Sensor}$ is *TRUE* starting from $t = 0$, and turns to *FALSE* before *Sample(1)*. The SRS updates its status to be *FALSE* when $t = \delta t$, right after it passes initial time. The SDD needs to wait till the next sample point *Sample(1)* to update its status. In this case, SRS and SDD disagree at *Sample(1)*, because the input $\text{Reset} \wedge \neg \text{Sensor}$ changes too early before SDD can respond in the software domain.

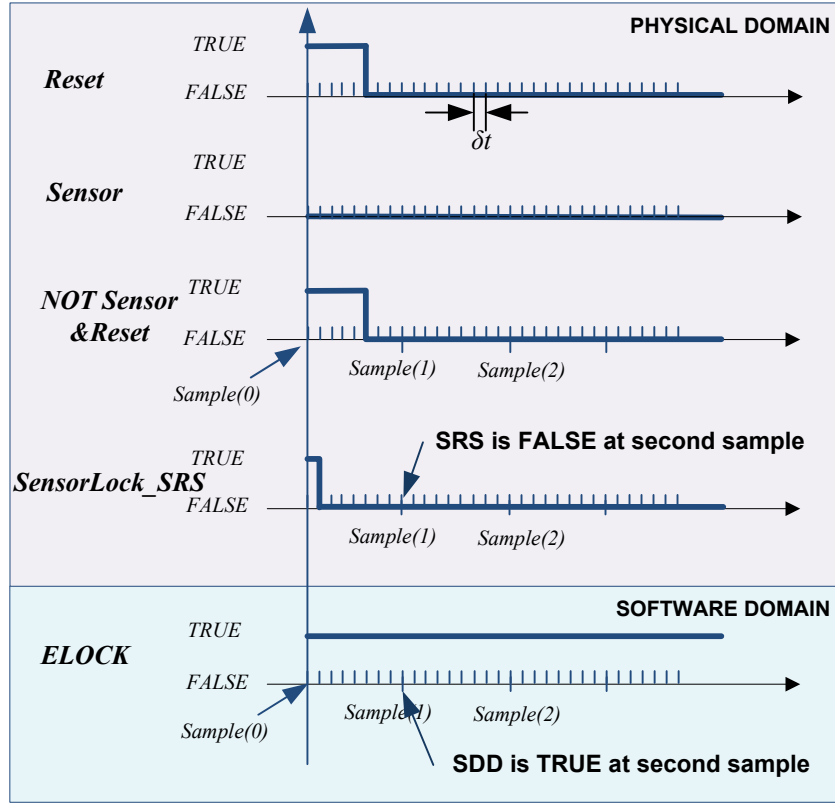


Figure 6.6: Example of disagreement on second sample point

Therefore, it is necessary to assume that the input $\text{Reset} \wedge \neg \text{Sensor}$

should not change before the second sample point $Sample(1)$ occurs. Considering $Sample(0) = 0$ and the condition $Sample(1) - Sample(0) \leq T_{max}$, it is safe to assume that the input $Reset \wedge \neg Sensor$ does not change before T_{max} .

Assumption 6.1.2. *Consider $\neg Sensor \wedge Reset$ to be the combined input to the SenLock System. This input should not produce “spike” behavior between any consecutive sample instances.*

To deal with the “spike” behavior, it was originally assumed that both the $Reset$ and $Sensor$ inputs should be of filtered tick predicate type. However, this assumption is not correct after we have discovered some PVS unprovable obligations. Figure 6.7 shows an example of “spike” behavior of the input $Reset \wedge \neg Sensor$. In this example, we can easily identify that $Reset$ and $Sensor$ are both filtered tick predicate (by assuming both of them keep $FALSE$ after changing their values), however, the input $Reset \wedge \neg Sensor$ creates a “spike” between $Sample(1)$ and $Sample(2)$. Because of it, the SRS is reset to $FALSE$ immediately but the SDD “missed” it in the software domain.

Summary of Implementation Assumptions

Now we can conclude that in order to implement the *SenLock* System, the input $Reset \wedge \neg Sensor$ is required to be filtered tick predicate type. However, it is not mandatory that both inputs, $Reset$ and $Sensor$, must be filtered tick predicate separately. So the Assumptions 6.1.1 and 6.1.2 can be summarized as:

Assumption 6.1.3. *Consider $\neg Sensor \wedge Reset$ to be the combined input to the SenLock System. This input should be a filtered tick predicate for the SenLock System to be implemented correctly.*

In the next section, we will formalize this assumption in PVS and present the PVS theorem to verify the *SenLock* System.

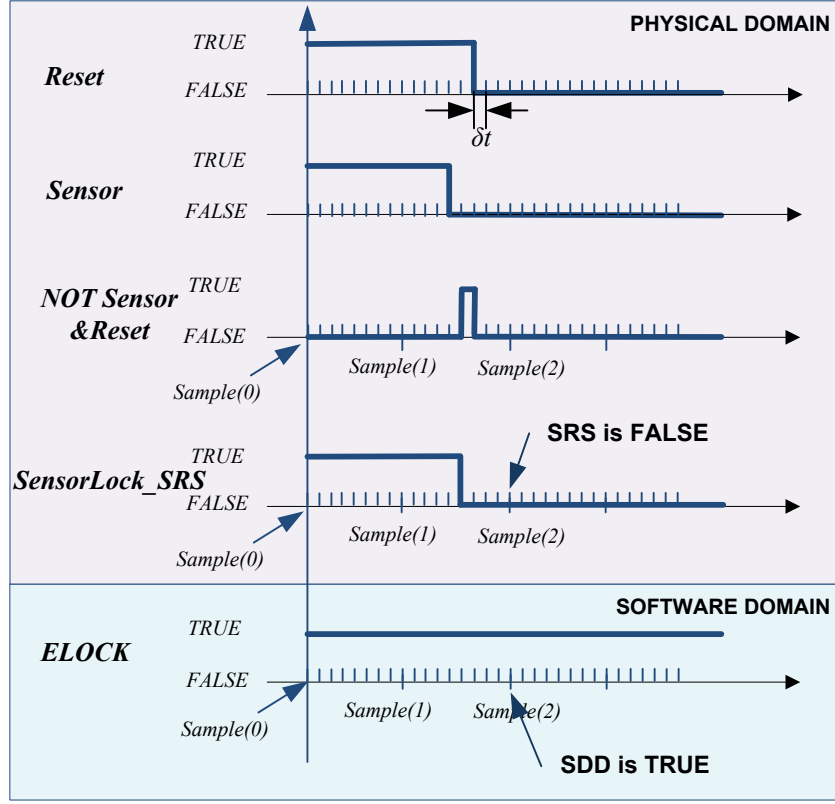


Figure 6.7: Spike behavior between consecutive sample points

6.1.5 Formal Verification of the SenLock System

PVS Proof Obligation for the SenLock System

For the SenLock System, we can consider a simplified discrete-time 4-variable model that represents a digital control system’s periodic sampling of inputs and updating of outputs. In this case, each of the four “variables”, \mathbf{M} , \mathbf{I} , \mathbf{O} , and \mathbf{C} , is a set of “time series vectors” or *dataflows*. For example, with a series of sample instances *Sample*, the element $m \in \mathbf{M}$ will be the dataflow of observations on the monitored variables at times $t = \text{Sample}(0), \text{Sample}(1), \text{Sample}(2), \dots$

To ensure that the SDD implements the SRS, we have to discharge the main block comparison theorem

$$Abst_C \circ REQ = SOF_{req} \circ Abst_M$$

which is described in Section 2.2. This results in the following PVS theorem as shown in Figure 6.8.

```

delay: VAR Duration

SensorLock_Block: THEOREM
  FilteredTickPred?(LAMBDA (t: tick): reset(t) AND NOT sensor(t)) IMPLIES
    SenLock_SRS(sensor, reset, delay)(Sample(n)) =
      Lock?(Elock(ELOCK(sensor, reset, delay - delta_L)(Sample(n))))

```

Figure 6.8: SenLock System Proof Obligation

The Assumption 6.1.3 that we concluded in the previous section is formalized to `FilteredTickPred?(LAMBDA (t: tick): reset(t) AND NOT sensor(t))` in PVS, as the premise of the block comparison theorem. The theorem is verified in PVS and the source code is available in the attached CD. In the next section we will provide an overview of how we verify the `SensorLock_Block` theorem in PVS.

Overview of the Verification Work

This section provides readers an overview of the major piece of the verification work, which will help the reader to understand the attached verification source code. Figure 6.9 provides the insights of our verification approach, with main theorems and components listed.

Figure 6.9 provides a more detailed view of the *SenLock* System, compared to the implementation roadmap shown in Figure 5.15. We will take 4 steps to introduce our work.

1. Define SenLock_SRS_S Function The proof obligation of the *SenLock* System, `SensorLock_Block`, is to verify that the `SenLock_SRS` and `ELOCK` agree at each sample point. The two functions update their output in different frequencies: `SenLock_SRS` can refresh and change the output at any clock tick, while `ELOCK` can only update the output at each sample point. Therefore, it will be helpful to define another version of the SRS function `SenLock_SRS_S`, which has the same behavior as the `SenLock_SRS` at the sample points. The

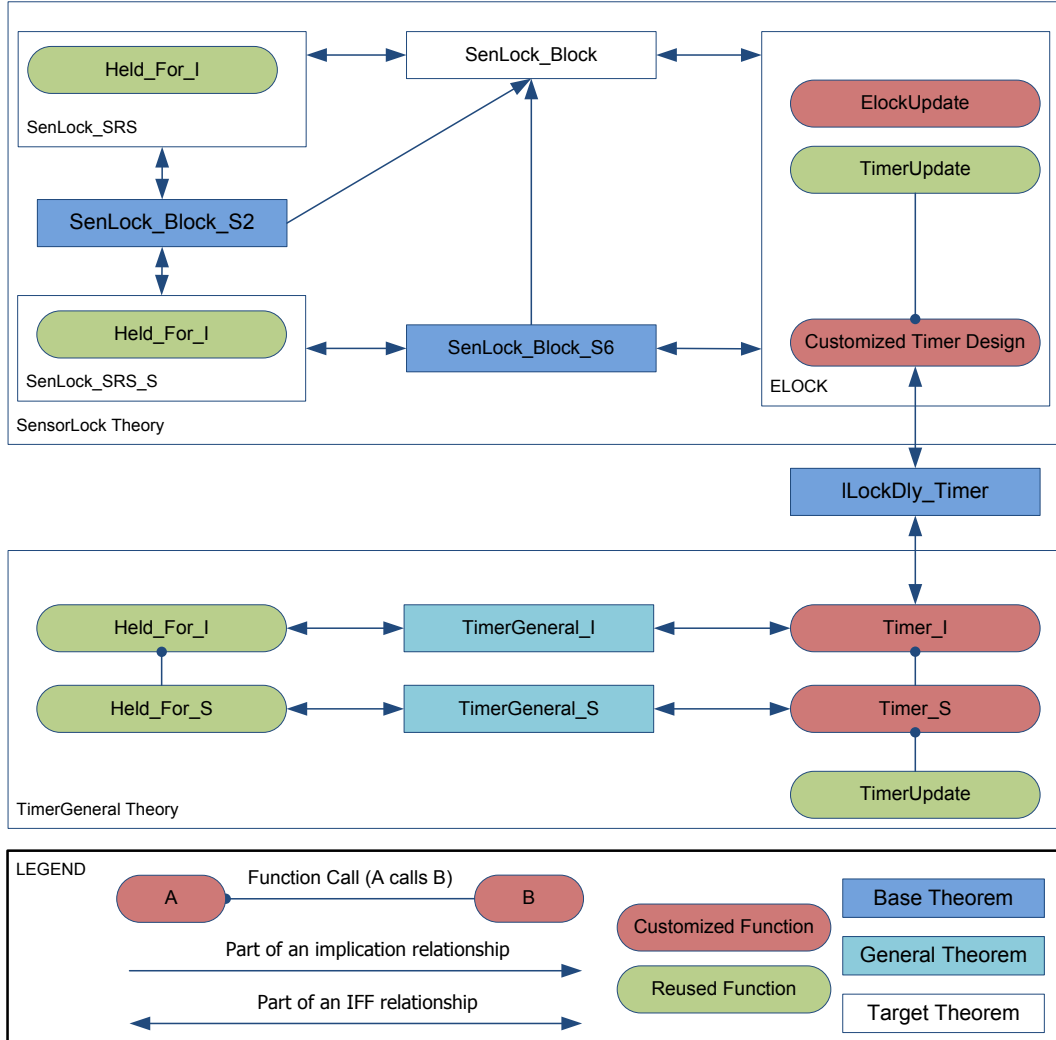


Figure 6.9: Theorems and Lemmas of the SensorLock Theory

`SenLock_SRS_S` is shown in Figure 6.10. This function is only defined at each sample point, so we name it as `SenLock_SRS_S`.

```

SenLock_SRS_S(sensor, reset, ldelay)(n): RECURSIVE bool =
  IF n = 0 THEN TRUE
  ELSE COND Held_For_S(sensor, ldelay - delta_L, Sample)(n) -> TRUE,
    NOT Held_For_S(sensor, ldelay - delta_L, Sample)(n) AND
      reset(Sample(n)) AND sensor(Sample(n))
      -> SenLock_SRS_S(sensor, reset, ldelay)(n - 1),
    NOT Held_For_S(sensor, ldelay - delta_L, Sample)(n) AND
      reset(Sample(n)) AND NOT sensor(Sample(n))
      -> FALSE,
    NOT Held_For_S(sensor, ldelay - delta_L, Sample)(n) AND
      NOT reset(Sample(n))
      -> SenLock_SRS_S(sensor, reset, ldelay)(n - 1)
  ENDCOND
ENDIF
MEASURE n

```

Figure 6.10: `SenLock_SRS_S` Function

2. Create and Verify Theorem `SenLock_Block_S6` We create the PVS theorem `SenLock_Block_S6` in order to verify the equivalence of `SenLock_SRS_S` and the SDD `ELock` at all the sample points. The verification of this theorem is completed with ease, by instantiating theorems `TimerGeneral_I` and `lLockDly_Timer`. For the `SenLock` example, `TimerGeneral_I` is the reusable result in the `TimerGeneral` Theory. In this case, we only need to prove the `lLockDly_Timer` theorem, which shows the equivalence between the `Timer_I` design and the customized timer design in *SenLock* System. The design of these two components is close, so the verification of `lLockDly_Timer` takes much less effort than proving the equivalence between `Held_For_I` and customized timer design from scratch.

3. Create and Verify Theorem `SenLock_Block_S2` We create the PVS theorem `SenLock_Block_S2` in order to verify the equivalence of `SenLock_SRS_S` and the original SRS function `SenLock_SRS`. We mark this part of work as a significant piece of the whole *SenLock* System verification, because a lot


```
SensorLock_Block_S6: THEOREM
  SenLock_SRS_S(sensor, reset, delay)(n) =
    Lock?(Elock(ELock(sensor, reset, delay - delta_L)(Sample(n))))
```

Figure 6.11: PVS Theorem SensorLock_Block_S6

of intersample behavior scenarios need to be verified one by one in PVS. In total, we have created 7 additional SRS_PROPERTY lemmas (containing over 200 PVS commands) to verify theorem `SenLock_Block_S2`. The proof of `SenLock_Block_S2` itself takes 332 PVS commands to complete.

```
SensorLock_Block_S2: THEOREM
  FilteredTickPred?(LAMBDA (t: tick): reset(t) AND NOT sensor(t)) IMPLIES
  SenLock_SRS_S(sensor, reset, delay)(n) =
    SenLock_SRS(sensor, reset, delay)(Sample(n))
```

Figure 6.12: PVS Theorem SensorLock_Block_S2

4. Verify SenLock_Block Theorem Based on the result of step 2 and 3, we can now easily use the results of theorems `SenLock_Block_S6` and `SenLock_Block_S2` to connect the `SenLock_SRS` and `ELock` together. The proof of the target theorem `SenLock_Block` only takes 7 PVS commands to complete, by just instantiating these two theorems.

6.2 Example: Delayed Trip System

The Delayed Trip System (DTS) has been introduced in Chapter 2. In this section we will upgrade the requirement specification with the *Held_For* operator *with tolerance*, based on which we construct the pseudo version of the SRS in PVS. In the design and verification of the Delayed Trip System, the `Timer_I` design and the general theorem `TimerGeneral_I` are involved to simplify both the implementation and verification work.

6.2.1 Software Requirement Specification (SRS)

The upgraded Software Requirement Specification (SRS) for DTS is shown in Figure 6.13. In this version, the requirements are specified with the *Held_For* operators *with tolerances*.

Condition	Result <i>relay</i>
$(PP) \text{ Held_For}(\text{timeout1}, \delta L1, \delta R1)$	TRUE
$(\neg [(PP) \text{ Held_For}(\text{timeout1}, \delta L1, \delta R1)]) \text{ Held_For}(\text{timeout2}, \delta L2, \delta R2)$	FALSE
$\neg (PP) \text{ Held_For}(\text{timeout1}, \delta L1, \delta R1) \wedge$ $\neg (\neg [(PP) \text{ Held_For}(\text{timeout1}, \delta L1, \delta R1)]) \text{ Held_For}(\text{timeout2}, \delta L2, \delta R2)$	No Change

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

Figure 6.13: The Upgraded SRS for the Delayed Trip System

If the condition PP holds longer than timeout1 , the relay should keep the open status. When the power drops below PT or the pressure becomes lower than DSP , the relay should not close until after another time period of timeout2 . Note that *Held_For* operator with the duration timeout1 has the tolerance settings $\delta L1$ and $\delta R1$ and another *Held_For* operator with the duration timeout2 has its own tolerance settings $\delta L2$ and $\delta R2$ defined. This will better fit the real-world engineering specification, where the timeout1 and timeout2 have huge difference and they are not possible to share a global tolerance. For example, $\text{timeout1}=30$ seconds and $\text{timeout2}=2$ seconds can lead to the tolerance settings $\delta L1=\delta R1=1$ second and $\delta L2=\delta R2=0.1$ seconds. It also shows the *Held_For* operator *with tolerance* allows us to specify different tolerances for the each duration, respectively.

PVS Pseudo-SRS

Based on the formal Software Requirement Specification (SRS) of the Delayed Trip System shown in Figure 6.13, we can construct the pseudo-SRS in PVS using *Held_For_I* function (as shown in Figure 6.14). The *Held_For_I* function returns a tick predicate and can be nested like in the SRS function.

In the definition of *DelayedTrip_SRS* function, the tick predicate P is passed as a more generic condition type to handle the situation where “both

```

DelayedTrip_SRS(P: Condition_Type,
               timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
               timeout2: Duration[K, TL, TR, delta_L2, delta_R2])
(t): RECURSIVE

  bool =
  IF init(t) THEN FALSE
  ELSE TABLE
    %-----+-----++
    | Held_For_I(P, timeout1 - delta_L1, Sample)(t)          | TRUE      ||
    %-----+-----++
    | Held_For_I(LAMBDA (t1: tick):
                  NOT Held_For_I(P, timeout1 - delta_L1, Sample)
                      (t1),
                  timeout2 - delta_L2, Sample)(t)          | FALSE      ||
    %-----+-----++
    | NOT Held_For_I(P, timeout1 - delta_L1, Sample)(t) AND
      (NOT Held_For_I(LAMBDA (t1: tick):
                      NOT Held_For_I
                        (P, timeout1 - delta_L1, Sample)(t1),
                      timeout2 - delta_L2, Sample)
                        (t)) | DelayedTrip_SRS(P, timeout1, timeout2)(pre(t)) ||
    %-----+-----++
  ENDTABLE
  ENDIF
  MEASURE rank(t)

```

Figure 6.14: PVS Pseudo-SRS of Delayed Trip System

the power exceeds the Power Threshold (PT) and the pressure exceeds the Delayed Trip Set Point (DSP) simultaneously”. In this way the implementer can instantiate P with any specific tick predicate type function. As shown in the figure, $\text{Held_For_I}(P, \text{timeout1} - \text{delta_L1}, \text{Sample})(t)$ formalizes the situation where both the power exceeds the PT and the pressure exceeds the DSP simultaneously for the duration timeout1 . In this case, the relay is opened for the next period timeout2 .

6.2.2 PVS Software Design Description (SDD)

The Delayed Trip System is implemented based on the `Timer_I` design. We will use two timer variables, `Timer1` and `Timer2`, to implement the first and second `Held_For_I` operators in the pseudo-SRS respectively. According to the SRS, if P has been *TRUE* for timeout1 (with a tolerance setting $\delta L1$ and $\delta R1$), the relay will be open for the next timeout2 time units (with a tolerance setting $\delta L2$ and $\delta R2$). So we will use one timer for measuring timeout1 , and

the other timer for measuring `timeout2`.

We can define the system state as a record type which has a relay status field, two timer fields and two fields for storing the conditions at the previous sample (see Figure 6.15). Here the variable `S` is of type `SDD_State` and represents the system's previous state. Function `RelayUpdate` updates the current output of the relay, according to the previous values of `Timer1` and `Timer2`, and the current condition `P`. `PreviousInput1` and `PreviousInput2` are the fields defined to store the conditions of `Timer1` and `Timer2` at the previous sample point, for a similar purpose that `PreviousInput` is defined to be passed as the `PreviousP` parameter of `TimerUpdate` in the Sensor Lock example.

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

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

Figure 6.15: RelayUpdate Function

Since the `RelayUpdate` function only updates the output `Relay`, we need a function to update the system's two timers. The function `TimerUpdate` defined in Section 5.5 can be used for this purpose. By using the `TimerUpdate` and `RelayUpdate` functions together, we can implement the SDD for the DTS block (as shown in Figure 6.16).

We define the return type of the `DelayedTrip_SDD` function to be `SDD_State`. It should only be able to refresh the output at the sample points.

```

timeout1, timeout2: VAR non_initial_time

DelayedTrip_SDD(P, timeout1, timeout2)(t): RECURSIVE SDD_State =
  IF t = Sample(0)
    THEN (# Relay := CLOSED,
          Timer1 := 0,
          Timer2 := 0,
          PreviousInput1 := P(Sample(0)),
          PreviousInput2 := TRUE #)
  ELSIF t = Sample(Left_Sample(Sample, t))
    THEN (# Relay
          := RelayUpdate(timeout1, timeout2, P(t),
                          DelayedTrip_SDD(P, timeout1, timeout2)(pre(t)),
                          t - Sample(Left_Sample(Sample, t) - 1)),
          Timer1
          := TimerUpdate(P(t),
                          PreviousInput1(DelayedTrip_SDD
                                           (P, timeout1, timeout2)
                                           (Sample
                                            (Left_Sample(Sample, t) - 1))),
                          timeout1,
                          Timer1(DelayedTrip_SDD
                                   (P, timeout1, timeout2)(pre(t))),
                          t - Sample(Left_Sample(Sample, t) - 1)),
          Timer2
          := TimerUpdate(NOT (P(t)
                              &
                              Timer1
                              (DelayedTrip_SDD
                               (P, timeout1, timeout2)
                               (Sample(Left_Sample(Sample, t) - 1)))
                              + t
                              - Sample(Left_Sample(Sample, t) - 1)
                              >= timeout1),
                          PreviousInput2(DelayedTrip_SDD
                                           (P, timeout1, timeout2)
                                           (Sample
                                            (Left_Sample(Sample, t) - 1))),
                          timeout2,
                          Timer2(DelayedTrip_SDD
                                   (P, timeout1, timeout2)(pre(t))),
                          t - Sample(Left_Sample(Sample, t) - 1)),
          PreviousInput1 := P(t),
          PreviousInput2
          := NOT (P(t) &
                  Timer1(DelayedTrip_SDD(P, timeout1, timeout2)
                          (Sample
                           (Left_Sample(Sample, t) - 1)))
                  + t - Sample(Left_Sample(Sample, t) - 1) >= timeout1) #)
  ELSE (# Relay := Relay(DelayedTrip_SDD(P, timeout1, timeout2)(pre(t))),
        Timer1
        := Timer1(DelayedTrip_SDD(P, timeout1, timeout2)(pre(t))),
        Timer2
        := Timer2(DelayedTrip_SDD(P, timeout1, timeout2)(pre(t))),
        PreviousInput1
        := PreviousInput1(DelayedTrip_SDD(P, timeout1, timeout2)(pre(t))),
        PreviousInput2
        := PreviousInput2(DelayedTrip_SDD(P, timeout1, timeout2)
                              (pre(t))) #)
  ENDIF
  MEASURE rank(t)

```

Figure 6.16: PVS Code for DTS SDD

Therefore, in the `DelayedTrip_SDD` function the condition `t=Sample(LeftSample(Sample,t))` is used to check whether the current tick value is right on a sample point or not. The system will then update the related timer. If the current time `t` is not a sample point, the system should maintain the previous status (as shown in the last `ELSE` statement block of Figure 6.16).

6.2.3 Formal Verification of the Delayed Trip Example

PVS Proof Obligation for Delayed Trip System

We follow the same approach of the *SenLock* System to create the PVS proof obligation for DTS, based on the 4-variable model.

```
DelayedTrip_Block: THEOREM
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
          timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    DelayedTrip_SRS(PP, timeout1, timeout2)(Sample(n)) =
      OPEN?(Relay(DelayedTrip_SDD(PP, timeout1 - delta_L1,
                                timeout2 - delta_L2)(Sample(n))))
```

We can define tick predicate `PP` in PVS as shown below:

```
PP(t):bool=Power(t)>=PT AND Pressure(t)>=DSP
```

Then we can instantiate this specific tick predicate `PP` in the final proof obligation of DTS. The theorem `DelayedTrip_Block` specifies that the SRS specification `DelayedTrip_SRS` should be *TRUE* if and only if the `Relay` field of the `DelayedTrip_SDD` is `OPEN`.

Overview of the Verification Work

We revisit Figure 5.15 to provide reader an overview of our verification strategy. We will take 2 steps to introduce our work.

1. Create and Verify the Timer_Timer lemmas We create the PVS theorem `Timer1_Timer` and `Timer2_Timer` to show that the customized `Timer1` and `Timer2` designs are equivalent to the general `Timer_I` design we present in Section 5.5. For example, `Timer1_Timer` shown in the Figure 6.17 verifies the `Timer1` of the `DelayedTrip_SDD` function behave same as the `Timer_I` function.

```
Timer1_Timer: LEMMA
  FORALL (timeout1, timeout2: non_initial_time):
    Timer_I(P, Sample, timeout1)(t) =
      Timer1(DelayedTrip_SDD(P, timeout1, timeout2)(t))
```

Figure 6.17: PVS Lemma `Timer1_Timer`

2. Verify DelayedTrip_Block Theorem As shown in the Figure 5.15, we can show the equivalence between the two nested `Held_For_I` in the SRS and the two timers, based on the two `Timer_Timer` lemmas and `TimerGeneral_I` theorem. These results help us to simplify the proof work of the final theorem `DelayedTrip_Block`.

We also developed another version of the pseudo-SRS in PVS, named as `DelayedTrip_SRS1`. Then we show the two SRS functions are equivalent with the lemma `DelayedTrip_EQUAL`. Finally we can verify that our `DelayedTrip_SDD` function conforms to both of the SRS functions we have defined with ease. Interested reader can review these results in the complete source code.

6.3 Summary

In the previous sections of this chapter, we demonstrated how the pre-verified implementation template (e.g., `Timer_I`) and the general theorem (e.g., `TimerGeneral_I`) can be used to guide and simplify both the implementation and verification, through two examples: Sensor Lock and Delayed Trip. In both of the examples, customized timer designs have been created based

on the guidance of the pre-verified `Timer_I` implementation (for the *Held_For* operator *with tolerance*). The general theorem `TimerGeneral_I` that verifies the equivalence relationship between `Held_For_I` and `Timer_I` is instantiated in the verification of both of the examples, which significantly reduces the amount of repetitive work.

The Sensor Lock example provides us a scenario where the SRS and SDD will not match when SRS function is changed due to an intersample behavior with “spike” in a logical combination of the system input signals, that the SDD function fails to observe it with the samples. This shows the importance and limitations of the filtered tick predicate assumption. Our analysis of the PVS verification results shows simply assuming each input variable to be filtered tick predicate might still not be sufficient to implement the system.

The requirement specification of the Delayed Trip example is composed with two nested *Held_For* operators, with different durations and different timing tolerances for each of them. Thus this example provide us with a case where 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 [36] and the ϵ -hypothesis in [10]) where in each timing component of the system specification the timing tolerances are different. The Delayed Trip example demonstrated the ability of our approach to specify and implement the timing requirements *with tolerances* of a system precisely, unrestricted by a global tolerance. For a system with multiple timing requirements, the intersection of the feasible condition sets might result in some implementations that are feasible even when a global tolerance would make us think they are not feasible. In this case, our approach, to replace a very conservative global tolerance by including tolerances on each individual timing requirement, may significantly reduce unnecessary load on the target platform. Instead of performing a scheduling check in the final stage [8, 10], our approach can also determine whether further implementation effort is worth it or not as soon as the timing requirements have been specified (based on *feasibility analyses*).

Table 6.1 shows a statistical analysis of the verification work performed in these two examples.

Sensor Lock			Delayed Trip		
TimerGeneral_I(20)	624	39%	TimerGeneral_I(20)	624	51%
ldelay_Timer	69	4%	Timer1_Timer(1)	116	9%
SensorLock_Block_S2(7)	706	44%	Timer2_Timer(2)	258	21%
SensorLock_Block_S6(2)	199	12%	Other lemmas(3)	24	2%
SensorLock_Block	7	1%	DelayedTrip_Block(1)	209	17%
Total	1605	100%	Total	1231	100%

Table 6.1: Verification Effort Comparison of Two Examples

For each of the examples shown in the figure, we list the main theorems, number of PVS commands to complete the theorem, and the percentage of the total work required to complete each of the main theorems. The number in parentheses besides the theorem name indicates the total number of lemmas and theorems that help to verify the main theorem¹. For example: **TimerGeneral_I** takes 20 theorems and lemmas and 624 PVS commands to be proved. Theorems **llockDly_Timer**, **Timer1_Timer** and **Timer2_Timer** show the equivalence between the customized timers of the SDD functions and the general **Timer_I** design. They are the necessary effort in order to connect the timer components in the examples with the **Held_For_I** in the pseudo requirements. In most cases, this part of the work is relatively trivial. For example, **llockDly_Timer** and **Timer1_Timer** take only 4% and 9% of the total work. However, to verify **Timer2_Timer** takes 258 PVS commands (21%) because of the complexity of nested functions. Both of the Sensor Lock and Delayed Trip examples reuse the general theorem **TimerGeneral_I**. This important pre-verified result helps us to save 39% and 51% of the verification work respectively.

The results we have shown will not only benefit the domain expert, who will have great flexibility to specify real-world performance timing requirements (e.g., timing resolution and response allowance), but also give a re-usable and reliable result for the designer and developer, allowing them to apply the implementation of this timing requirement to real-time systems cor-

¹The statistics data does not include the effort of the lemmas, theorems and TCC proofs of imported theories, since they are one time cost that can be reused in any example.

rectly. Overall, we believe this implementation approach feasible and reusable, both in the design and verification phases. The next chapter presents a summary of the thesis, and describes possible future work in this area.

Chapter 7

Summary

In this thesis, we investigated how to deal with timing properties of real-time systems. Our approach results in precise definitions and analyses of how functional timing requirements interact with performance timing requirements, and covers most of the activities of the software engineering life-cycle of real-time systems.

Using three common functional timing requirements, *Held_For*, *Periodic*, and *Sync_Periodic*, we demonstrated:

- *Formal Specifications*: Tabular specifications of functional timing requirements that include tolerances on the time durations.

Using *Held_For* as an example, we also demonstrated:

- *Feasibility Analyses*: Conditions under which such timing requirements can be implemented.
- *Implementation Templates*: Pre-verified implementation templates for common pieces of the timing requirements that often appear in real-time systems. These can be reused to guide the design and reduce the associated verification work.

Table 7.1 highlights the contributions of this thesis to these three areas.

<i>Areas</i>	<i>Contributions</i>
Formal Specification	Based on the formal specifications that arose out of the Darlington Shutdown development [31, 29]. The formalization of the definition of <i>Held_For</i> in PVS is a contribution of this thesis.
Feasibility Analyses	The manual analysis was presented in [29]. The contribution in this thesis is its formalization in PVS for three environmental assumptions, the proof that the primary aspects of the manual analysis in [29] are true for practical environmental assumptions, and the interesting discovery of an additional mathematically feasible, but practically unlikely case. Discovered the relationships between the environments and discussed the estimation approach for a new environment. Revealed the importance of the general theorems to the verification work and conducted all the actual work.
Implementation Templates	This entire aspect is described for the first time in this thesis. The idea was suggested in [13, 29].

Table 7.1: Summary of Contributions

Formal Specifications The functional behaviour of an application is typically specified in the system requirements in an “ideal” way within the (continuous) physical domain. 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. We have presented the precise definitions of functional timing requirements (e.g., the *Held_For* operator *with timing tolerances*) and performance timing requirements (e.g., the upper and lower bound of sample intervals: T_{min} and T_{max}), which allow us to determine the implementability of real-time systems without incurring the complex implementation and verification work described in Chapter 3.

In most industrial applications, tolerances are defined (or should be

defined) clearly on the major timing components of a real-time system. The definition we provided allows a domain specialist to specify the requirements very precisely and this sets the stage for the feasibility analyses and implementation templates summarized in the following sections.

Feasibility Analyses We have provided the necessary and sufficient conditions for the implementability of the *Held_For* operator *with tolerance*, which are determined by both the environmental assumptions and the interaction of the timing requirements.

We also presented three environmental assumptions, *Omniscient*, *Perfect Clock* and *No Clock*, and the feasibility analyses show that in each of the environments, the implementability of the *Held_For* operator *with tolerance* varies. In each environment, the interaction between the functional timing requirements (FTRs) and performance timing requirements (PTRs) determine whether the implementation is feasible or not. Timing tolerances are introduced in both the FTRs (e.g., duration tolerances of the *Held_For* operator, δL and δR) and PTRs (e.g., the intervals between the samples are bounded by $[T_{min}, T_{max}]$).

For each environment, the feasibility of the implementation is discussed via three cases. *Case 1* is when $T_{max} \leq (\delta L + \delta R)/2$. Comparing with the other two cases, the implementation condition is relatively simple. However, it is not correct to assume the *Held_For* operator can always be implemented in this case. In the *No Clock* environment, the implementability of *Held_For* is not always possible. This shows that sampling fast is not always the solution. *Case 2* is when $(\delta L + \delta R)/2 < T_{max} \leq (\delta L + \delta R)$. We discovered that under certain conditions the *Held_For* operator *with tolerance* is still implementable, which provides an alternative solution to the designer of the real-time system, when facing the limitation of the hardware. *Case 3* is when $T_{max} > (\delta L + \delta R)$. The implementation of the *Held_For* operator does not exist (based on our analyses of this case in all three environments). Section 3.4 presented the table comparison of the results and detailed discussions.

For each of the environments, we defined a *feasible function* which determines the implementability of the *Held_For* in that environment. The

function not only is a key factor in the feasibility analysis of each specific environment, but also helps us to connect the feasibility results across the three environments to create a more general view of the feasibility conditions. As an important result, Theorem 3.4.1 reveals the relationships between feasibility functions across the environmental assumptions. First, it contributes to our verification work of the feasibility results using the PVS theorem proving approach. The verification strategies (shown in Section 4.3) that we have created based on this theorem saved at least 40% of the total amount of the theorem proving work. Another contribution of this theorem (in different environments), is that it can help one to estimate (or even precisely predict) the feasibility of an implementation under a new environmental assumption. Our summary report shows the work to verify this general theorem itself in PVS takes less than 1% of the total verification effort. This shows that the trivial effort to verify this theorem first can possibly benefit the complex analysis and proof of the overall feasibility to a significant degree.

Implementation Templates Based on the *feasibility analyses*, we selected the *Perfect Clock* environment to demonstrate how to reduce the work required to implement the real-time operator *Held_For* with tolerance. The proposed implementation uses a template approach as presented in a two step approach based on the verification process described in Section 2.2.

The `Held_For_I` function is applied in the pseudo version of the requirements (pseudo-SRS) in PVS. This function is first verified based on the high level functional and performance timing requirements through the PVS theorem proving methodology. The *feasibility analyses* provide essential guidance during this verification work. Only under the feasible conditions of the *Perfect Clock* environment can we prove that `Held_For_I` (with a duration of $d - \delta L$) conforms to all the high level requirements we have specified for the *Held_For* operator *with tolerance*.

We then presented a timer design (the function `Timer_I` in PVS) as an implementation of `Held_For_I` and verified this result through the general theorem `TimerGeneral_I`. Thus `Timer_I` can be used as a pre-verified template to implement the *Held_For* operator in a straight-forward manner.

It can be easily customized when dealing with different industrial examples. The general theorem `TimerGeneral_I` can be reused to reduce the amount of verification work in future implementations. To illustrate this approach, we provided two examples, Sensor Lock and Delayed Trip, to demonstrate our implementation strategy.

The Sensor Lock example provides us a scenario where the SRS and SDD will not match when SRS function is changed due to an intersample behavior with “spike” in a logical combination of the system input signals, while the SDD function fails to observe it with the samples. This shows the importance and limitations of the filtered tick predicate assumption. Our analysis of the PVS verification results shows that simply assuming each input variable to be of filtered tick predicate type might still not be sufficient to implement the system. The design and implementation is guided by the general template `Timer_I` function, at a level that the designer does not need deep knowledge of our feasibility analyses.

The requirement specification of the Delayed Trip example is a composition of two nested *Held_For* operators, with different durations and different timing tolerances for each of them. This provides an example in which the requirement of the system does not fit into a global tolerance model (e.g., the reaction delay parameter Δ of the Almost ASAP semantics in [36] and ϵ -hypothesis in [10]), because, in each timing component of the system specification the timing tolerances are different. For a system with multiple timing requirements, the intersection of the feasible condition sets might result in some implementations that are feasible even when a global tolerance would make us think they are not feasible. In this case, our approach, to replace a very conservative global tolerance by including tolerances in each individual timing requirement, may significantly reduce unnecessary load on the target platform. Instead of performing a scheduling check in the final stage [8, 10], our approach can also determine whether further implementation is worth it or not as soon as the timing requirements have been specified (based on the *feasibility analyses*).

The general theorem `TimerGeneral_I` takes over 16 lemmas and over 600 PVS commands to complete. We have shown how to reuse this result

to reduce the verification work of the customized timer components, in both of the examples. In the Sensor Lock example, the theorem `llockDly_Timer` verifies the equivalence between the customized design in the SDD function and the template `Timer_I`. In the Delayed Trip example, `Timer1_Timer` and `Timer2_Timer` show that each customized timer behaves in the same way as does `Timer_I`. Together with the general theorem `TimerGeneral_I`, these theorems connect the timing requirements in the SRS and customized implementation components in the SDD, and verify the implementation of the system in a component based approach. Through this approach, 39% and 51% of the verification work was saved for Sensor Lock and Delayed Trip, respectively.

7.1 Future Work

The list below is suggested future work arising from this thesis.

- It will be of interest to formalize the *Imperfect Clock* environmental assumption and complete its feasibility analyses. Further formal analyses and verification of *Case 1* can be completed to finalize the implementability results of *Imperfect Clock* environment.
- It should be possible to expand the idea of the PVS pre-verified implementation template for *Held_For*, to create a real-time operator library, which can cover all the common timing requirements that appear in real-time systems. Each of the operators would then have an implementation template that is relatively easy for a designer to follow, and a general theorem (with a guideline) on how to reuse the template to reduce the effort required to implement and verify the specified timing behavior.

Appendix A

Time Theory

This appendix contains the PVS input files for **Time** theory of Chapter 4. The complete PVS dump files are available from the attached CD.

```
Time: THEORY
BEGIN

  time: TYPE+ = nonneg_real

  non_initial_time: TYPE+ = posreal
END Time
```

Appendix B

SampleInstance Theory

This appendix contains the PVS input files for `SampleInstance` theory of Chapter 4. The complete PVS dump files are available from the attached CD.

```
SampleInstance[(IMPORTING Time) K: non_initial_time, TL,
               TR: {t: time | t < K}]: THEORY
BEGIN

  n, n1, n2: VAR nat

  t, t1, t2: VAR time

  Tmin: posreal = K - TL

  Tmax: posreal = K + TR

  init(x: time): bool = (x = 0)

  Sample_Type: TYPE+ =
    {c: [nat -> time] |
      c(0) = 0 AND
      (FORALL n:
        Tmin <= c(n + 1) - c(n) AND c(n + 1) - c(n) <= Tmax)}

  Sample: VAR Sample_Type

  Sample_PROPERTY1: LEMMA n2 > n1 IMPLIES Sample(n2) > Sample(n1)

  Sample_PROPERTY2: LEMMA n2 >= n1 IMPLIES Sample(n2) >= Sample(n1)

  Sample_PROPERTY3: LEMMA Sample(n2) > Sample(n1) IMPLIES n2 > n1

  Sample_PROPERTY4: THEOREM n2 > n1 IFF Sample(n2) > Sample(n1)

  Sample_PROPERTY5: LEMMA Sample(n) = 0 IFF n = 0
```

```
Sample_Interval: LEMMA
  FORALL (n: nat, duration: time):
    Sample(n + floor(duration / Tmin) + 1) >= Sample(n) + duration

Sample_Interval2: LEMMA
  FORALL (n: nat, m: nat): Sample(n + m) >= Sample(n) + m * Tmin

Sample_Interval3: LEMMA
  FORALL (n: nat, m: nat): Sample(n + m) <= Sample(n) + m * Tmax

Sample_Interval4: LEMMA
  FORALL (n, k: nat):
    FORALL (t: real):
      Sample(n) <= t AND Sample(n + 1) > t AND Sample(k) <= t IMPLIES
        k <= n

Sample_Interval7: LEMMA
  FORALL (n: nat, k: nat | k >= 1):
    Sample(n) >= k * Tmax IMPLIES n >= k - 1

Sample_Interval5: LEMMA
  FORALL (n: nat, k: nat | k >= 1): Sample(n) > k * Tmax IMPLIES n >= k

Sample_Compare: LEMMA
  FORALL (n1, n2: nat): n2 >= n1 IMPLIES Sample(n2) >= Sample(n1)

Sample_Compare1: LEMMA
  FORALL (n: nat, k: nat): Sample(n) >= Sample(k) IFF n >= k

Sample_Sequence: LEMMA
  FORALL (n1, n2: nat): Sample(n1) = Sample(n2) IMPLIES n1 = n2

Sample_Value_PROPERTY1: LEMMA Sample(n + 1) > 0

TClock_2: LEMMA
  FORALL (t: time | t > Tmax):
    EXISTS (n: nat, j: nat): Sample(n) <= t AND Sample(n + j) > t

TClock_4: LEMMA
  FORALL (t: time | t >= Sample(0)):
    EXISTS (n: nat, j: nat): Sample(n) <= t AND Sample(n + j) > t

TClock_1: LEMMA
  FORALL (t: time | t > Tmax):
    EXISTS (n: nat): Sample(n) <= t AND Sample(n + 1) > t

TClock_3: LEMMA
  FORALL (t: time): EXISTS (n: nat): Sample(n) <= t AND Sample(n + 1) > t
```

```
TIME_BETWEEN_SAMPLE: LEMMA
  FORALL (t: time): EXISTS (n: nat): Sample(n) <= t AND Sample(n + 1) > t
END SampleInstance
```

Appendix C

FeasibilityResults Theory

This appendix contains the PVS input files for `FeasibilityResults` theory of Chapter 4. The complete PVS dump files are available from the attached CD.

```
FeasibilityResults[(IMPORTING Time) K: non_initial_time, TL,
                  TR: {t: time | t < K}, delta_L, delta_R: time]: THEORY
BEGIN

  IMPORTING SampleInstance[K, TL, TR]

  P: VAR pred[time]

  Duration: TYPE = {du: time | du > delta_R AND du - delta_L > Tmax}

  d, duration: VAR Duration

  n, n0: VAR nat

  t, t1, t2, t_now, t_n, t_j: VAR time

  t3: VAR posreal

  Sample: VAR Sample_Type

  Kmin(d): nat = floor((d - delta_L) / Tmax)

  Kmax(d): nat = floor((d - delta_L) / Tmin)

  Feasible_Omniscient(d): bool =
    FORALL Sample:
      FORALL n0:
        FORALL (t | t > Sample(n0) AND t <= Sample(n0 + 1)):
          EXISTS n:
            Sample(n) - t >= d - delta_L AND Sample(n) - t <= d + delta_R
```

```

Feasible_PerfectClock(d): bool =
  FORALL Sample:
    FORALL n0:
      EXISTS n:
        FORALL (t | t > Sample(n0) AND t <= Sample(n0 + 1)):
          Sample(n) - t >= d - delta_L AND Sample(n) - t <= d + delta_R

Feasible_NoClock(d): bool =
  EXISTS n:
    FORALL Sample:
      FORALL n0:
        FORALL (t | t > Sample(n0) AND t <= Sample(n0 + 1)):
          Sample(n + n0) - t >= d - delta_L AND
          Sample(n + n0) - t <= d + delta_R

sampleExists: LEMMA
  t2 - t1 > Tmax => (EXISTS n: t1 < Sample(n) & Sample(n) < t2)

Sample_Exists1: LEMMA
  FORALL (ns: nat | ns > 1):
    FORALL (t: time | t <= Tmax AND t >= Tmin):
      FORALL (t1: time | t1 <= Tmax AND t1 >= Tmin):
        EXISTS (Sample: Sample_Type):
          Sample(0) = 0 AND
          Sample(1) = Tmax AND
          (FORALL (n: nat | n > 1 AND n <= ns):
            Sample(n) = (n - 1) * t + Tmax)
          AND
          Sample(ns + 1) = (ns - 1) * t + Tmax + t1 AND
          (FORALL (n: nat | n > ns + 1):
            (Sample(n) =
              (ns - 1) * t + Tmax + t1 + (n - ns - 1) * Tmax))

FLOOR_REAL1: LEMMA FORALL (a, b: time): a > b IMPLIES floor(a) >= floor(b)

FLOOR_REAL2: LEMMA
  FORALL (a, b: time): floor(a) > floor(b) IMPLIES floor(a) >= b

FLOOR_REAL3: LEMMA
  FORALL (a, c: posreal, b: time): a >= b / c IMPLIES b / a <= c

FLOOR_REAL4: LEMMA
  FORALL (a, c: posreal, b: time): a <= b / c IMPLIES b / a >= c

FLOOR_TRUTH: LEMMA floor((duration - delta_L) / Tmax) >= 0

FLOOR_TRUTH1: LEMMA floor((duration - delta_L) / Tmin) >= 0

```

```

FLOOR_TRUTH2: LEMMA
  (FORALL (t: time | t <= Tmax AND t >= Tmin):
    floor((d - delta_L) / t) < (d - delta_L) / t)
  IMPLIES floor((d - delta_L) / Tmin) = floor((d - delta_L) / Tmax)

FLOOR_COMMON: LEMMA
  (FORALL (t: time | t <= Tmax AND t >= Tmin):
    floor((duration - delta_L) / t) > 0 AND
    floor((duration + delta_R) / t) > 0)

CEILING_COMMON: LEMMA
  (FORALL (t: time | t <= Tmax AND t >= Tmin):
    ceiling((duration - delta_L) / t) > 0 AND
    ceiling((duration + delta_R) / t) > 0)

LT_LEQ_PROP: LEMMA
  (FORALL (x: {y: nnreal | y < Tmax}): t + x <= t1) => t + Tmax <= t1

GT_LEQ_PROP1: LEMMA
  (FORALL (x: {y: real | y > 0 AND y <= t3}): t - x <= t1) IMPLIES t <= t1

Duration_PROPERTY: LEMMA (d - delta_L) / Tmin > 1

TminAndKmax: THEOREM
  (Kmax(d) = Kmin(d) OR
   (Kmax(d) = Kmin(d) + 1 & Kmax(d) * Tmin = d - delta_L))
  IFF Tmin >= (d - delta_L) / (Kmin(d) + 1)

Feasible_PerfectClockAnddMinusDeltaL: LEMMA
  Feasible_PerfectClock(d) &
  (EXISTS n: d - delta_L = n * Tmax) & Tmin /= Tmax
  => delta_L + delta_R >= 2 * Tmax

PerfectClock_CASE2A_1: LEMMA
  (delta_L + delta_R) / 2 < Tmax & Tmax <= delta_L + delta_R IMPLIES
  (Feasible_PerfectClock(d) AND
   floor((d - delta_L) / Tmin) * Tmin /= d - delta_L
   IMPLIES
   (FORALL (t: time | t <= Tmax AND t >= Tmin):
     floor((d - delta_L) / t) * t < d - delta_L))

PerfectClock_CASE2A_2: LEMMA
  (delta_L + delta_R) / 2 < Tmax &
  Tmax <= delta_L + delta_R AND
  floor((d - delta_L) / Tmin) * Tmin = d - delta_L AND Tmax /= Tmin
  IMPLIES (Feasible_PerfectClock(d) IMPLIES Kmax(d) = Kmin(d) + 1)

PerfectClock_CASE2A: THEOREM
  (delta_L + delta_R) / 2 < Tmax &

```

```
Tmax <= delta_L + delta_R AND Tmin /= Tmax
IMPLIES
(Feasible_PerfectClock(d) IMPLIES
  Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
  (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R)

NoClock_CASE2B_1: LEMMA
(delta_L + delta_R) / 2 <= Tmax & Tmax <= delta_L + delta_R IMPLIES
(floor((d - delta_L) / Tmax) = floor((d - delta_L) / Tmin) AND
  (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R
  IMPLIES Feasible_NoClock(d))

NoClock_CASE2B: THEOREM
(delta_L + delta_R) / 2 < Tmax &
Tmax <= delta_L + delta_R AND Tmin /= Tmax
IMPLIES
(Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
  (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R
  IMPLIES Feasible_NoClock(d))

PerfectClock_CASE1B: LEMMA
Tmax <= (delta_L + delta_R) / 2 IMPLIES Feasible_PerfectClock(d)

NoClock_Implies_PerfectClock: THEOREM
Feasible_NoClock(d) IMPLIES Feasible_PerfectClock(d)

PerfectClock_Implies_Omniscient: THEOREM
Feasible_PerfectClock(d) IMPLIES Feasible_Omniscient(d)

NoClock_CASE1: THEOREM
Tmax <= (delta_L + delta_R) / 2 AND Tmin /= Tmax IMPLIES
((ceiling((d - delta_L) / Tmin) + 1) * Tmax <= d + delta_R IFF
  Feasible_NoClock(d))

PerfectClock_CASE1: THEOREM
Tmax <= (delta_L + delta_R) / 2 IMPLIES Feasible_PerfectClock(d)

Omniscient_CASE1: THEOREM
Tmax <= (delta_L + delta_R) / 2 IMPLIES Feasible_Omniscient(d)

NoClock_CASE2: THEOREM
(delta_L + delta_R) / 2 < Tmax &
Tmax <= delta_L + delta_R AND Tmin /= Tmax
IMPLIES
(Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
  (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R
  IFF Feasible_NoClock(d))

PerfectClock_CASE2: THEOREM
```



```

(delta_L + delta_R) / 2 < Tmax &
Tmax <= delta_L + delta_R AND Tmin /= Tmax
IMPLIES
(Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
 (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R
 IFF Feasible_PerfectClock(d))

Omniscient_CASE2: THEOREM
(delta_L + delta_R) / 2 < Tmax &
Tmax <= delta_L + delta_R AND Tmin /= Tmax
IMPLIES Feasible_Omniscient(d)

Omniscient_CASE3: THEOREM
Tmax > delta_L + delta_R IMPLIES NOT Feasible_Omniscient(d)

PerfectClock_CASE3: THEOREM
Tmax > delta_L + delta_R IMPLIES NOT Feasible_PerfectClock(d)

NoClock_CASE3: THEOREM
Tmax > delta_L + delta_R IMPLIES NOT Feasible_NoClock(d)

FeasiblePoint(d): bool =
  FORALL (Sample: Sample_Type):
    FORALL (n0: nat):
      FORALL (t | t > Sample(n0) AND t <= Sample(n0 + 1)):
        Sample(Kmin(d) + 2 + n0) - t >= d - delta_L AND
        Sample(Kmin(d) + 2 + n0) - t <= d + delta_R

FeasiblePoint_CASE2_1: LEMMA
(delta_L + delta_R) / 2 <= Tmax & Tmax <= delta_L + delta_R IMPLIES
(floor((d - delta_L) / Tmax) = floor((d - delta_L) / Tmin) AND
 (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R
 IMPLIES FeasiblePoint(d))

FeasiblePoint_CASE2: LEMMA
(delta_L + delta_R) / 2 < Tmax &
Tmax <= delta_L + delta_R AND Tmin /= Tmax
IMPLIES
(Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
 (floor((d - delta_L) / Tmax) + 2) * Tmax <= d + delta_R
 IMPLIES FeasiblePoint(d))

PerfectClock_ALLCASES: THEOREM
Tmax /= Tmin IMPLIES
((Tmax <= (delta_L + delta_R) / 2 OR
 ((delta_L + delta_R) / 2 < Tmax AND
  Tmax <= (delta_L + delta_R) AND
  Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
  (Kmin(d) + 2) * Tmax <= d + delta_R))

```

```
      IFF Feasible_PerfectClock(d))  
END FeasibilityResults
```

Appendix D

ClockTick Theory

This appendix contains the PVS input files for ClockTick theory of Section 5.1.2. The complete PVS dump files are available from the attached CD.

```
ClockTick[delta_t: posreal]: THEORY
BEGIN

  IMPORTING Time

  n: VAR nat

  tick: TYPE = {t: time | EXISTS (n: nat): t = n * delta_t}

  x: VAR tick

  init(x): bool = (x = 0)

  noninit_elem: TYPE = {x | NOT init(x)}

  y: VAR noninit_elem

  pre(y): tick = y - delta_t

  next(x): tick = x + delta_t

  rank(x): nat = x / delta_t

  time_induct: LEMMA
    FORALL (P: pred[tick]):
      (FORALL x, n: rank(x) = n IMPLIES P(x)) IMPLIES (FORALL x: P(x))

  time_induction: PROPOSITION
    FORALL (P: pred[tick]):
      (FORALL (t: tick): init(t) IMPLIES P(t)) AND
```

```
(FORALL (t: noninit_elem): P(pre(t)) IMPLIES P(t))
IMPLIES (FORALL (t: tick): P(t))

tick_PROPERTY0: LEMMA
  FORALL (n1, n2: nat):
    n1 * delta_t > n2 * delta_t IFF n1 * delta_t - delta_t >= n2 * delta_t

tick_PROPERTY1: LEMMA FORALL (t: tick | t > 0): t > x IFF pre(t) >= x
END ClockTick
```

Appendix E

SampleInstanceOnTick Theory

This appendix contains the PVS input files for `SampleInstanceOnTick` theory of Section 5.1.2. The complete PVS dump files are available from the attached CD.

```
SampleInstanceOnTick[(IMPORTING Time) K: non_initial_time, TL,
                     TR: {t: time | t < K},
                     delta_t: {tk: non_initial_time |
                               tk < K - TL
                               AND
                               tk < TR + TL}]: THEORY

BEGIN

  IMPORTING ClockTick[delta_t]

  IMPORTING SampleInstance[K, TL, TR]

  t: VAR tick

  Delta_t_Tmax_Tmin: THEOREM Tmin <= floor(Tmax / delta_t) * delta_t

  SampleTick_Type: TYPE+ =
    {S: Sample_Type | FORALL (n: nat): EXISTS (t: tick): S(n) = t}

  Sample: VAR SampleTick_Type

  n: VAR nat

  SampleTick_PROPERTY1: LEMMA EXISTS (n1: nat): Sample(n) = n1 * delta_t

  SampleTick_PROPERTY2: LEMMA
    pre(Sample(n + 1)) > Sample(n) AND pre(Sample(n + 1)) < Sample(n + 1)

  SampleTick_PROPERTY3: LEMMA
    NOT init(t) IMPLIES (Sample(n) > pre(t) IFF Sample(n) >= t)
```

END SampleInstanceOnTick

Appendix F

Held_For Theory

This appendix contains the PVS input files for Held_For theory of Section 5.2. The complete PVS dump files are available from the attached CD.

```
Held_For[(IMPORTING Time) K: non_initial_time, TL, TR: {t: time | t < K},
          delta_t: {tk: posreal | tk < K - TL AND tk < TR + TL}, delta_L,
          delta_R: time]: THEORY
BEGIN

  IMPORTING SampleInstanceOnTick[K, TL, TR, delta_t]

  IMPORTING FeasibilityResults[K, TL, TR, delta_L, delta_R]

  IMPORTING reals@bounded_reals

  Condition_Type: TYPE = pred[tick]

  P: VAR Condition_Type

  t, t0, t_n, t_j: VAR tick

  ne, n0, n: VAR nat

  d: VAR Duration

  Sample: VAR SampleTick_Type

  duration: VAR non_initial_time

  Sup_FACT1: LEMMA
    sup(LAMBDA (n: nat): Sample(n) <= t) >= 0 AND
    integer?(sup(LAMBDA (n: nat): Sample(n) <= t))

  Left_Sample(Sample, t): {n: nat | Sample(n) <= t AND t < Sample(n + 1)} =
    sup(LAMBDA (n: nat): Sample(n) <= t)
```

```

Held_For_P(P, duration): pred[tick] =
  LAMBDA (t_n):
    EXISTS (t_j):
      (t_n - t_j >= duration) AND
      (FORALL (t: tick | t >= t_j & t <= t_n): P(t))

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))

Held_For_I(P, duration, Sample)(t): bool =
  Held_For_S(P, duration, Sample)(Left_Sample(Sample, t))

Held_For_S_OLD(P, d, Sample)(ne): bool =
  EXISTS (n0 | Sample(ne) - Sample(n0) >= d - delta_L):
    FORALL (n: nat | n0 <= n AND n <= ne): P(Sample(n))

Held_For_I_OLD(P, d, Sample)(t): bool =
  Held_For_S_OLD(P, d, Sample)(Left_Sample(Sample, t))

FilteredTickPred?(P: PRED[tick]): bool =
  (FORALL t0:
    P(t0) /= P(next(t0)) =>
      (FORALL (t | t0 < t AND t <= t0 + Tmax): P(next(t0)) = P(t)))
  AND (FORALL (t | t <= Tmax): P(t) = P(0))

FilteredTickPred: TYPE+ = (FilteredTickPred?)

Pf: VAR FilteredTickPred

Left_Sample_PROPERTY0: THEOREM
  t >= Sample(n) IMPLIES Left_Sample(Sample, t) >= n

Left_Sample_PROPERTY1: THEOREM
  t >= Sample(0) IMPLIES Left_Sample(Sample, t) >= 0

Left_Sample_PROPERTY2: THEOREM
  t >= Sample(1) IMPLIES Left_Sample(Sample, t) >= 1

Left_Sample_PROPERTY3: THEOREM Left_Sample(Sample, Sample(n)) = n

Left_Sample_PROPERTY4: THEOREM
  t < Sample(n + 1) AND t >= Sample(n) IMPLIES Left_Sample(Sample, t) = n

Left_Sample_PROPERTY5: THEOREM
  t > 0 AND NOT t = Sample(Left_Sample(Sample, t)) IMPLIES
    Sample(Left_Sample(Sample, pre(t))) = Sample(Left_Sample(Sample, t))

```


Left_Sample_PROPERTY6: THEOREM $t - \text{Sample}(\text{Left_Sample}(\text{Sample}, t)) < T_{\max}$

Left_Sample_PROPERTY7: THEOREM
 $t \geq \text{Sample}(0) \text{ AND } n \leq \text{Left_Sample}(\text{Sample}, t) \text{ IMPLIES } \text{Sample}(n) \leq t$

Left_Sample_PROPERTY8: THEOREM
 $\text{NOT } t = \text{Sample}(\text{Left_Sample}(\text{Sample}, t)) \text{ IMPLIES } \text{Sample}(\text{Left_Sample}(\text{Sample}, \text{pre}(t))) = \text{Sample}(\text{Left_Sample}(\text{Sample}, t))$

Left_Sample_PROPERTY9: THEOREM $\text{Left_Sample}(\text{Sample}, 0) = 0$

Left_Sample_PROPERTY10: THEOREM
 $t = \text{Sample}(\text{Left_Sample}(\text{Sample}, t)) \text{ AND } \text{NOT } t = \text{Sample}(0) \text{ IMPLIES } \text{Left_Sample}(\text{Sample}, t) - 1 \geq 0$

Held_For_S_PROPERTY1: THEOREM
 $\text{Held_For_S}(P, \text{duration}, \text{Sample})(n) \text{ IMPLIES } P(\text{Sample}(n))$

Held_For_RELATIONSHIP5: THEOREM
 $\text{Held_For_S}(P, d, \text{Sample})(n) \text{ IMPLIES } (\text{FORALL } (t: \text{tick} \mid \text{Sample}(n) \leq t \text{ AND } t < \text{Sample}(n + 1)): \text{Held_For_I}(P, d, \text{Sample})(t))$

Held_For_RELATIONSHIP6: THEOREM
 $\text{FORALL } (t: \text{tick} \mid \text{Sample}(n) \leq t \text{ AND } t < \text{Sample}(n + 1)): \text{Held_For_I}(P, \text{duration}, \text{Sample})(t) = \text{Held_For_S}(P, \text{duration}, \text{Sample})(n)$

FILTER_TRUTH1: LEMMA
 $\text{Pf}(\text{Sample}(n)) = \text{Pf}(\text{Sample}(n + 1)) \text{ IMPLIES } (\text{FORALL } (t \mid t > \text{Sample}(n) \text{ AND } t < \text{Sample}(n + 1)): \text{Pf}(t) = \text{Pf}(\text{Sample}(n)))$

FILTER_TRUTH2: LEMMA
 $\text{FORALL } (t: \text{tick} \mid \text{Sample}(n) < t \text{ AND } t < \text{Sample}(n + 1)): \text{NOT } \text{Pf}(\text{Sample}(n)) = \text{Pf}(t) \text{ IMPLIES } \text{Pf}(t) = \text{Pf}(\text{Sample}(n + 1))$

FILTER_TRUTH3: LEMMA
 $(\text{FORALL } (ne \mid ne \geq n_0 \text{ AND } ne \leq n_0 + n): \text{Pf}(\text{Sample}(ne))) \text{ IMPLIES } (\text{FORALL } (t \mid t \geq \text{Sample}(n_0) \text{ AND } t \leq \text{Sample}(n_0 + n)): \text{Pf}(t) = \text{Pf}(\text{Sample}(n_0)))$

FILTER_TRUTH4: LEMMA $\text{FORALL } (t: \text{tick} \mid t \leq T_{\max}): \text{Pf}(t) = \text{Pf}(0)$

FILTER_TRUTH5: LEMMA
 $\text{FORALL } (t: \text{tick} \mid t \geq \text{Sample}(0)): \text{Pf}(t) = \text{Pf}(\text{Sample}(\text{Left_Sample}(\text{Sample}, t))) \text{ IMPLIES } (\text{FORALL } (t_1: \text{tick} \mid t_1 \leq t \text{ AND } t_1 \geq \text{Sample}(\text{Left_Sample}(\text{Sample}, t))):$

```

    Pf(t1) = Pf(t))

Held_For_RELATIONSHIP1: THEOREM
  Held_For_I(P, d - delta_L, Sample)(t) = Held_For_I_OLD(P, d, Sample)(t)

Held_For_RELATIONSHIP2A: THEOREM
  Held_For_I(P, duration, Sample)(t) =
    Held_For_S(P, duration, Sample)(Left_Sample(Sample, t))

Held_For_RELATIONSHIP2C: THEOREM
  t < Sample(1) IMPLIES NOT Held_For_I(P, duration, Sample)(t)

Held_For_RELATIONSHIP3: THEOREM
  FORALL (n: nat):
    Held_For_I(P, duration, Sample)(Sample(n)) =
      Held_For_S(P, duration, Sample)(n)

Held_For_RELATIONSHIP4: THEOREM
  t >= Sample(1) AND NOT t = Sample(Left_Sample(Sample, t)) IMPLIES
    Held_For_I(P, duration, Sample)(pre(t)) =
      Held_For_I(P, duration, Sample)(t)

Held_For_RELATIONSHIP7: THEOREM
  t > Sample(0) AND NOT t = Sample(Left_Sample(Sample, t)) IMPLIES
    Held_For_I(P, duration, Sample)(pre(t)) =
      Held_For_I(P, duration, Sample)(t)

Held_For_I_PROPERTY1: THEOREM
  FORALL (t: tick | Sample(n) <= t AND t < Sample(n + 1)):
    Held_For_I(P, duration, Sample)(t) =
      Held_For_I(P, duration, Sample)(Sample(n))

ceiling_delta_t: LEMMA
  FORALL (t: time): ceiling(t / delta_t) * delta_t >= t

ceiling_tick: LEMMA
  FORALL (tk: tick):
    FORALL (t: time): tk < ceiling(t / delta_t) * delta_t IMPLIES tk <= t

TICK_BETWEEN_SAMPLE: LEMMA
  FORALL (t: tick): EXISTS (n: nat): Sample(n) <= t AND t < Sample(n + 1)

EXISTS_SAMPLE_BETWEEN_TIME: LEMMA
  FORALL (t: time):
    EXISTS (n: nat): Sample(n) < t AND Sample(n + 1) >= t OR Sample(0) = t

Held_For_S_VERIFY_FTR1: THEOREM
  Tmax /= Tmin AND
    (Tmax <= (delta_L + delta_R) / 2 OR

```

```

((delta_L + delta_R) / 2 < Tmax AND
Tmax <= (delta_L + delta_R) AND
Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
(Kmin(d) + 2) * Tmax <= d + delta_R))
IMPLIES
(FORALL (Sample: SampleTick_Type, t: tick):
(Held_For_P(P, d + delta_R)(t) IMPLIES
Held_For_S(P, d - delta_L, Sample)(Left_Sample(Sample, t))))

Held_For_S_VERIFY_FTR2: THEOREM
FORALL (Sample: SampleTick_Type, n: nat):
(NOT Held_For_P(Pf, d - delta_L)(Sample(n)) IMPLIES
NOT Held_For_S(Pf, d - delta_L, Sample)(n))

Held_For_S_VERIFY_FTR3: THEOREM
Held_For_S(P, duration, Sample)(n) AND P(Sample(n + 1)) IMPLIES
Held_For_S(P, duration, Sample)(n + 1)

Held_For_S_VERIFY_PTR2: THEOREM
FORALL (Sample: SampleTick_Type, n: nat,
t: tick | t >= Sample(n) AND t <= Sample(n + 1)):
Held_For_S(Pf, duration, Sample)(n) AND NOT Pf(t) IMPLIES
NOT Held_For_S(Pf, duration, Sample)(n + 1)

Held_For_S_VERIFY_PTR3: THEOREM
FORALL (Sample: SampleTick_Type):
NOT Pf(t) IMPLIES
NOT Held_For_S(Pf, d, Sample)(Left_Sample(Sample, t) + 1)

Held_For_S_VERIFY_PTR4: THEOREM
FORALL (Sample: SampleTick_Type):
t >= Sample(0) IMPLIES Sample(Left_Sample(Sample, t) + 1) <= t + Tmax

Held_For_I_VERIFY_FTR1: THEOREM
Tmax /= Tmin AND
(Tmax <= (delta_L + delta_R) / 2 OR
((delta_L + delta_R) / 2 < Tmax AND
Tmax <= (delta_L + delta_R) AND
Tmin >= (d - delta_L) / (Kmin(d) + 1) AND
(Kmin(d) + 2) * Tmax <= d + delta_R))
IMPLIES
(FORALL (Sample: SampleTick_Type, t: tick):
(Held_For_P(P, d + delta_R)(t) IMPLIES
(FORALL (t1: tick
| t1 >= Sample(Left_Sample(Sample, t)) AND
t1 < Sample(Left_Sample(Sample, t) + 1)):
Held_For_I(P, d - delta_L, Sample)(t1))))

Held_For_I_VERIFY_FTR2: THEOREM

```

```
FORALL (Sample: SampleTick_Type, n: nat):
  (NOT Held_For_P(Pf, d - delta_L)(Sample(n)) IMPLIES
    (FORALL (t: tick | t >= Sample(n) AND t < Sample(n + 1)):
      NOT Held_For_I(Pf, d - delta_L, Sample)(t)))

Held_For_I_VERIFY_FTR3: THEOREM
  Held_For_I(P, d - delta_L, Sample)(Sample(n)) AND P(Sample(n + 1))
  IMPLIES
    (FORALL (t: tick | t >= Sample(n + 1) AND t < Sample(n + 2)):
      Held_For_I(P, d - delta_L, Sample)(t))

Held_For_I_VERIFY_FTR4: THEOREM
  Tmax /= Tmin AND Feasible_PerfectClock(d) IMPLIES
    (FORALL (Sample: SampleTick_Type, t: tick):
      (Held_For_P(P, d + delta_R)(t) IMPLIES
        (FORALL (t1: tick
          | t1 >= Sample(Left_Sample(Sample, t)) AND
            t1 < Sample(Left_Sample(Sample, t) + 1)):
          Held_For_I(P, d - delta_L, Sample)(t1))))

Held_For_I_VERIFY_TR0: THEOREM
  Tmax /= Tmin AND Feasible_PerfectClock(d) IMPLIES
    (FORALL (Sample: SampleTick_Type, t: tick):
      (EXISTS (x: time | x >= d - delta_L AND x <= d + delta_R):
        Held_For_I(Pf, d - delta_L, Sample)(t) = Held_For_P(Pf, x)(t))
      OR
      (NOT Pf(t) AND
        Pf(Sample(Left_Sample(Sample, t))) AND
        Sample(Left_Sample(Sample, t)) >= t - Tmax))
END Held_For
```

Appendix G

TimerGeneral Theory

This appendix contains the PVS input files for **TimerGeneral** theory of Section 5.5. The complete PVS dump files are available from the attached CD.

```
TimerGeneral[(IMPORTING Time) K: non_initial_time, TL,
              TR: {t: time | t < K},
              delta_t: {tk: non_initial_time | tk < K - TL AND tk < TR + TL},
              delta_L, delta_R: time]: THEORY

BEGIN

  IMPORTING Held_For[K, TL, TR, delta_t, delta_L, delta_R]

  P: VAR Condition_Type

  t, PreviousTimerValue: VAR tick

  Sample: VAR SampleTick_Type

  timeout: VAR Duration

  TimeOut: VAR non_initial_time

  CurrentP, PreviousP: VAR bool

  ne, n0, n: VAR nat

  step: VAR tick

  TimerUpdate(CurrentP, PreviousP, TimeOut, PreviousTimerValue, step): tick =
  TABLE
    %+-----+-----+-----+
    %| [ PreviousTimerValue < TimeOut | PreviousTimerValue >= TimeOut ] |
    %-----+-----+-----+
    %| CurrentP AND PreviousP      | PreviousTimerValue + step | PreviousTimerValue      ||
    %-----+-----+-----+
    %| NOT (CurrentP AND PreviousP) | 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

Timer_I(P, Sample, TimeOut)(t): tick =
    Timer_S(P, Sample, TimeOut)(Left_Sample(Sample, t))

Timer_S_PROPERTY1: LEMMA
    NOT P(Sample(n)) IMPLIES Timer_S(P, Sample, TimeOut)(n) = 0

Timer_RELATIONSHIP1: THEOREM
    Timer_S(P, Sample, TimeOut)(n) = Timer_I(P, Sample, TimeOut)(Sample(n))

Timer_RELATIONSHIP4: THEOREM
    t >= Sample(1) AND t = Sample(Left_Sample(Sample, t)) IMPLIES
    Timer_I(P, Sample, TimeOut)(Sample(Left_Sample(Sample, t) - 1)) =
    Timer_I(P, Sample, TimeOut)(pre(t))

Timer_RELATIONSHIP2: THEOREM
    t >= Sample(1) AND t = Sample(Left_Sample(Sample, t)) IMPLIES
    Timer_S(P, Sample, TimeOut)(Left_Sample(Sample, t) - 1) =
    Timer_I(P, Sample, TimeOut)(pre(t))

Timer_RELATIONSHIP2A: THEOREM
    FORALL (t: tick | t >= Sample(1)):
        t = Sample(Left_Sample(Sample, t)) IMPLIES
        Timer_S(P, Sample, TimeOut)(Left_Sample(Sample, t) - 1) =
        Timer_I(P, Sample, TimeOut)(pre(t))

Timer_RELATIONSHIP2B: THEOREM
    FORALL (t: tick):
        NOT t = Sample(Left_Sample(Sample, t)) IMPLIES
        Timer_S(P, Sample, TimeOut)(Left_Sample(Sample, t)) =
        Timer_I(P, Sample, TimeOut)(pre(t))

Timer_I_PROPERTY1: LEMMA
    FORALL (t: tick | t > Sample(n) AND t < Sample(n + 1)):
        Timer_I(P, Sample, TimeOut)(t) =
        Timer_I(P, Sample, TimeOut)(Sample(n))

Timer_Lemma2: LEMMA
    Timer_S(P, Sample, TimeOut)(ne) > 0 IMPLIES P(Sample(ne))

Timer_Lemma3: LEMMA Timer_S(P, Sample, timeout)(ne) >= 0

Timer_Lemma4: LEMMA
    Timer_S(P, Sample, TimeOut)(ne) = 0 IMPLIES
    Timer_S(P, Sample, TimeOut)(ne + n) <= Sample(ne + n) - Sample(ne)

Timer_Lemma6: LEMMA
    Sample(n0 + n) - Sample(n0) > 0 IMPLIES
    Timer_S(P, Sample, TimeOut)(n0 + n) >= Sample(n0 + n) - Sample(n0)
    IMPLIES (FORALL (ne: nat | n0 <= ne AND ne <= n + n0): P(Sample(ne)))

Timer_Lemma7: LEMMA
    Timer_S(P, Sample, TimeOut)(n) <= Sample(n) - Sample(0)

Timer_Lemma8: LEMMA

```

```

FORALL (TimeOut: time | TimeOut > Tmax):
  Timer_S(P, Sample, TimeOut)(n) > 0 IMPLIES
    Timer_S(P, Sample, TimeOut)(n) >= Sample(n) - Sample(n - 1)

TimerGeneral_S1: THEOREM
  FORALL (TimeOut: time | TimeOut > Tmax):
    Held_For_S(P, TimeOut, Sample)(n) IMPLIES
      Timer_S(P, Sample, TimeOut)(n) >= TimeOut

TimerGeneral_S11: THEOREM
  FORALL (TimeOut: time | TimeOut > Tmax):
    Held_For_S(P, TimeOut, Sample)(n) IMPLIES
      Timer_S(P, Sample, TimeOut)(n) >= TimeOut

TimerGeneral_S2: THEOREM
  FORALL (TimeOut: time | TimeOut > Tmax):
    Timer_S(P, Sample, TimeOut)(n) >= TimeOut IMPLIES
      Held_For_S(P, TimeOut, Sample)(n)

TimerGeneral_SC: THEOREM
  FORALL (TimeOut: time | TimeOut > Tmax):
    Held_For_S(P, TimeOut, Sample)(n) IFF
      Timer_S(P, Sample, TimeOut)(n) >= TimeOut

TimerGeneral_S: THEOREM
  Held_For_S(P, timeout - delta_L, Sample)(n) IFF
    Timer_S(P, Sample, timeout - delta_L)(n) >= timeout - delta_L

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

TimerGeneral_IC: THEOREM
  FORALL (TimeOut: time | TimeOut > Tmax):
    Held_For_I(P, TimeOut, Sample)(t) IFF
      Timer_I(P, Sample, TimeOut)(t) >= TimeOut

TimerGeneral_I: THEOREM
  Held_For_I(P, timeout - delta_L, Sample)(t) IFF
    Timer_I(P, Sample, timeout - delta_L)(t) >= timeout - delta_L

Timer_S_Eqv1: THEOREM
  FORALL (TimeOut: time | TimeOut > Tmax):
    Timer_S(P, Sample, TimeOut)(n + 1) >= TimeOut IFF
      (Timer_S(P, Sample, TimeOut)(n) + Sample(n + 1) - Sample(n) >=
        TimeOut
        AND P(Sample(n + 1)))

Timer_I_Eqv: THEOREM
  FORALL (TimeOut: time | TimeOut > Tmax):
    Timer_I(P, Sample, TimeOut)(Sample(n + 1)) >= TimeOut IFF
      (Timer_I(P, Sample, TimeOut)(Sample(n)) + Sample(n + 1) - Sample(n)
        >= TimeOut
        AND P(Sample(n + 1)))
END TimerGeneral

```

Appendix H

SensorLock Theory

This appendix contains the PVS input files for **SensorLock** theory of Section 6.1. The complete PVS dump files are available from the attached CD.

```
SensorLock[(IMPORTING Time) K: non_initial_time, TL, TR: {t: time | t < K},
           delta_t: {tk: non_initial_time | tk < K - TL AND tk < TR + TL},
           delta_L, delta_R: time]: THEORY
BEGIN

  IMPORTING TimerGeneral[K, TL, TR, delta_t, delta_L, delta_R]

  t: VAR tick

  ldelay, step: VAR Duration

  sensor: VAR PRED[tick]

  b: VAR FilteredTickPred

  reset: VAR PRED[tick]

  Sample: SampleTick_Type

  ne, n0, n: VAR nat

  delay: VAR Duration

  SenLock_SRS(sensor, reset, ldelay)(t): RECURSIVE bool =
    IF init(t) THEN TRUE
    ELSE COND Held_For_I(sensor, ldelay - delta_L, Sample)(t) -> TRUE,
              NOT Held_For_I(sensor, ldelay - delta_L, Sample)(t) AND
                reset(t) AND sensor(t)
                -> SenLock_SRS(sensor, reset, ldelay)(pre(t)),
              NOT Held_For_I(sensor, ldelay - delta_L, Sample)(t) AND
                reset(t) AND NOT sensor(t)
```



```

        -> FALSE,
        NOT Held_For_I(sensor, ldelay - delta_L, Sample)(t) AND
        NOT reset(t)
        -> SenLock_SRS(sensor, reset, ldelay)(pre(t))
    ENDCOND
ENDIF
MEASURE rank(t)

SenLock_SRS_S(sensor, reset, ldelay)(n): RECURSIVE bool =
    IF n = 0 THEN TRUE
    ELSE COND Held_For_S(sensor, ldelay - delta_L, Sample)(n) -> TRUE,
        NOT Held_For_S(sensor, ldelay - delta_L, Sample)(n) AND
        reset(Sample(n)) AND sensor(Sample(n))
        -> SenLock_SRS_S(sensor, reset, ldelay)(n - 1),
        NOT Held_For_S(sensor, ldelay - delta_L, Sample)(n) AND
        reset(Sample(n)) AND NOT sensor(Sample(n))
        -> FALSE,
        NOT Held_For_S(sensor, ldelay - delta_L, Sample)(n) AND
        NOT reset(Sample(n))
        -> SenLock_SRS_S(sensor, reset, ldelay)(n - 1)
    ENDCOND
ENDIF
MEASURE n

SRS_PROPERTY0: LEMMA
    NOT SenLock_SRS(sensor, reset, delay)(Sample(n)) IMPLIES
    NOT Held_For_I(sensor, delay - delta_L, Sample)(Sample(n))

SRS_PROPERTY7: LEMMA
    Held_For_I(sensor, delay - delta_L, Sample)(Sample(n)) IMPLIES
    (FORALL (t: tick | t >= Sample(n) AND t < Sample(n + 1)):
        SenLock_SRS(sensor, reset, delay)(Sample(n)) =
        SenLock_SRS(sensor, reset, delay)(t))

SRS_PROPERTY2: LEMMA
    NOT SenLock_SRS(sensor, reset, delay)(Sample(n)) AND
    NOT Held_For_I(sensor, delay - delta_L, Sample)(Sample(n + 1))
    IMPLIES NOT SenLock_SRS(sensor, reset, delay)(Sample(n + 1))

SRS_PROPERTY3: LEMMA
    FilteredTickPred?(LAMBDA (t: tick): reset(t) AND NOT sensor(t)) AND
    SenLock_SRS(sensor, reset, delay)(Sample(n)) AND
    NOT Held_For_I(sensor, delay - delta_L, Sample)(Sample(n)) AND
    NOT Held_For_I(sensor, delay - delta_L, Sample)(Sample(n + 1)) AND
    (EXISTS (t: tick | Sample(n) < t AND t < Sample(n + 1)):
        (reset(t) AND NOT sensor(t)))
    IMPLIES NOT SenLock_SRS(sensor, reset, delay)(Sample(n + 1))

SRS_PROPERTY4: LEMMA

```

```

FilteredTickPred?(LAMBDA (t: tick): reset(t) AND NOT sensor(t)) IMPLIES
  SenLock_SRS(sensor, reset, delay)(Sample(0)) =
    SenLock_SRS_S(sensor, reset, delay)(0)

SRS_PROPERTY5: LEMMA
  NOT SenLock_SRS(sensor, reset, delay)(Sample(n)) AND
  NOT Held_For_I(sensor, delay - delta_L, Sample)(Sample(n + 1))
  IMPLIES NOT SenLock_SRS(sensor, reset, delay)(Sample(n + 1))

SRS_PROPERTY6: LEMMA
  (FORALL (t: tick | t > Sample(n) AND t <= Sample(n + 1)):
    NOT (reset(t) AND NOT sensor(t)))
  AND NOT Held_For_I(sensor, delay - delta_L, Sample)(Sample(n + 1))
  IMPLIES
    SenLock_SRS(sensor, reset, delay)(Sample(n + 1)) =
      SenLock_SRS(sensor, reset, delay)(Sample(n))

Lock_State: TYPE = {Good, Bad, Lock}

SDD_State: TYPE =
  [# Elock: Lock_State, lLockDly: tick, PreviousInput: bool #]

sensor_now, reset_now: VAR bool

ElockUpdate(sensor_now: bool, reset_now: bool, S: SDD_State,
  ldelay: non_initial_time, step: time):
  Lock_State =
    TABLE
%-----+-----+
|NOT sensor_now AND Elock(S) = Lock AND reset_now          |Good||
%-----+-----+
|NOT sensor_now AND Elock(S) = Lock AND NOT reset_now       |Lock||
%-----+-----+
|NOT sensor_now AND NOT Elock(S) = Lock                     |Good||
%-----+-----+
|sensor_now AND (NOT Elock(S) = Lock AND lLockDly(S) + step < ldelay)|Bad ||
%-----+-----+
|sensor_now AND (Elock(S) = Lock OR lLockDly(S) + step >= ldelay) |Lock||
%-----+-----+
  ENDTABLE

S: VAR SDD_State

ELOCK(sensor: PRED[tick], reset: PRED[tick], ldelay: non_initial_time)
  (t): RECURSIVE
    SDD_State =
  IF init(t)
    THEN (# Elock := Lock, lLockDly := 0, PreviousInput := sensor(0) #)
    ELSE IF t = Sample(Left_Sample(Sample, t))

```

```

    THEN (# Elock
        := ElockUpdate(sensor(t),
            reset(t),
            ELOCK(sensor, reset, ldelay)(pre(t)),
            ldelay,
            t
            -
            Sample(Left_Sample(Sample, t) - 1)),
        lLockDly
        := TimerUpdate(sensor(t),
            PreviousInput
            (ELOCK(sensor, reset, ldelay)(pre(t))),
            ldelay,
            lLockDly
            (ELOCK(sensor, reset, ldelay)(pre(t))),
            t
            -
            Sample(Left_Sample(Sample, t) - 1)),
        PreviousInput := sensor(t) #)
    ELSE (# Elock := Elock(ELOCK(sensor, reset, ldelay)(pre(t))),
        lLockDly
        := lLockDly(ELOCK(sensor, reset, ldelay)(pre(t))),
        PreviousInput
        := PreviousInput(ELOCK
            (sensor, reset, ldelay)(pre(t))) #)
    ENDIF
ENDIF
MEASURE rank(t)

ELOCK_PROPERTY1: LEMMA
    FORALL (t: tick | t > Sample(n) AND t < Sample(n + 1),
        ldelay: non_initial_time):
        ELOCK(sensor, reset, ldelay)(Sample(n)) =
        ELOCK(sensor, reset, ldelay)(t)

ELOCK_PROPERTY2: LEMMA
    FORALL (t: tick | t > Sample(n) AND t < Sample(n + 1),
        ldelay: non_initial_time):
        PreviousInput(ELOCK(sensor, reset, ldelay)(t)) = sensor(Sample(n))

lLockDly_Timer: THEOREM
    FORALL (ldelay: non_initial_time):
        lLockDly(ELOCK(sensor, reset, ldelay)(t)) =
        Timer_I(sensor, Sample, ldelay)(t)

SensorLock_Block_S2: THEOREM
    FilteredTickPred?(LAMBDA (t: tick): reset(t) AND NOT sensor(t)) IMPLIES
    SenLock_SRS_S(sensor, reset, delay)(n) =
    SenLock_SRS(sensor, reset, delay)(Sample(n))

```

```
SensorLock_Block_S6: THEOREM
  SenLock_SRS_S(sensor, reset, delay)(n) =
    Lock?(Elock(ELOCK(sensor, reset, delay - delta_L)(Sample(n))))

SensorLock_Block: THEOREM
  FilteredTickPred?(LAMBDA (t: tick): reset(t) AND NOT sensor(t)) IMPLIES
    SenLock_SRS(sensor, reset, delay)(Sample(n)) =
      Lock?(Elock(ELOCK(sensor, reset, delay - delta_L)(Sample(n))))
END SensorLock
```

Appendix I

DelayedTrip Theory

This appendix contains the PVS input files for DelayedTrip theory of Section 6.2. The complete PVS dump files are available from the attached CD.

```
DelayedTrip[(IMPORTING Time) K: non_initial_time, TL,
            TR: {t: time | t < K},
            delta_t: {tk: non_initial_time | tk < K - TL AND tk < TR + TL},
            delta_L1, delta_L2, delta_R1, delta_R2: time]: THEORY
BEGIN

  IMPORTING TimerGeneral[K, TL, TR, delta_t, delta_L1, delta_R1]

  IMPORTING FeasibilityResults[K, TL, TR, delta_L2, delta_R2]

  timed_real: TYPE = [time -> real]

  Relay_State: TYPE = {OPEN, CLOSED}

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

  n: VAR nat

  t: VAR tick

  timeout1, timeout2: VAR non_initial_time

  step: VAR time

  P: VAR pred[tick]

  Power, Pressure: timed_real

  PT, DSP: posreal

  CurrentP, previous1, previous2: VAR bool

  previous: VAR time
```

```

PP(t): bool = Power(t) >= PT AND Pressure(t) >= DSP

Sample: SampleTick_Type

DelayedTrip_SRS(P: Condition_Type,
                timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
                timeout2: Duration[K, TL, TR, delta_L2, delta_R2])
(t): RECURSIVE

    bool =
    IF init(t) THEN FALSE
    ELSE TABLE
        %-----+-----++
        | Held_For_I(P, timeout1 - delta_L1, Sample)(t)          | TRUE      ||
        %-----+-----++
        | Held_For_I(LAMBDA (t1: tick):
                    NOT Held_For_I(P, timeout1 - delta_L1, Sample)
                        (t1),
                    timeout2 - delta_L2, Sample)(t)          | FALSE      ||
        %-----+-----++
        | NOT Held_For_I(P, timeout1 - delta_L1, Sample)(t) AND
          (NOT Held_For_I(LAMBDA (t1: tick):
                        NOT Held_For_I
                            (P, timeout1 - delta_L1, Sample)(t1),
                        timeout2 - delta_L2, Sample)
                    (t)) | DelayedTrip_SRS(P, timeout1, timeout2)(pre(t)) ||
        %-----+-----++
    ENDTABLE
    ENDIF
    MEASURE rank(t)

NOT_Held_For_I(P: Condition_Type, d: non_initial_time,
              Sample: SampleTick_Type)
(t):
    bool = NOT Held_For_I(P, d, Sample)(t)

DelayedTrip_SRS1(P: Condition_Type,
                 timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
                 timeout2: Duration[K, TL, TR, delta_L2, delta_R2])
(t): RECURSIVE

    bool =
    IF init(t) THEN FALSE
    ELSE TABLE
        %-----+-----++
        | Held_For_I(P, timeout1 - delta_L1, Sample)(t)          | TRUE      ||
        %-----+-----++
        | Held_For_I(NOT_Held_For_I(P, timeout1 - delta_L1, Sample),
                    timeout2 - delta_L2, Sample)(t) | FALSE      ||
        %-----+-----++
        | else                  | DelayedTrip_SRS1(P, timeout1, timeout2)(pre(t)) ||
        %-----+-----++
    ENDTABLE
    ENDIF
    MEASURE rank(t)

DelayedTrip_EQUAL: LEMMA
FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
        timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    DelayedTrip_SRS(P, timeout1, timeout2)(t) =
    DelayedTrip_SRS1(P, timeout1, timeout2)(t)

S: VAR SDD_State

```

```

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

DelayedTrip_SDD(P, timeout1, timeout2)(t): RECURSIVE SDD_State =
  IF t = Sample(0)
    THEN (# Relay := CLOSED,
           Timer1 := 0,
           Timer2 := 0,
           PreviousInput1 := P(Sample(0)),
           PreviousInput2 := TRUE #)
  ELSIF t = Sample(Left_Sample(Sample, t))
    THEN (# Relay
           := RelayUpdate(timeout1, timeout2, P(t),
                          DelayedTrip_SDD(P, timeout1, timeout2)
                          (pre(t)),
                          t - Sample(Left_Sample(Sample, t) - 1)),
           Timer1
           := TimerUpdate(P(t),
                          PreviousInput1(DelayedTrip_SDD
                                           (P, timeout1, timeout2)
                                           (Sample
                                            (Left_Sample(Sample, t)
                                             -
                                              1))),
                          timeout1,
                          Timer1(DelayedTrip_SDD
                                  (P, timeout1, timeout2)(pre(t))),
                          t - Sample(Left_Sample(Sample, t) - 1)),
           Timer2
           := TimerUpdate(NOT (P(t)
                               &
                               Timer1
                               (DelayedTrip_SDD
                                (P, timeout1, timeout2)
                                (Sample(Left_Sample(Sample, t) - 1)))
                               +
                               t
                               -
                               Sample(Left_Sample(Sample, t) - 1)
                               >=
                               timeout1),
                          PreviousInput2(DelayedTrip_SDD
                                           (P, timeout1, timeout2)
                                           (Sample
                                            (Left_Sample(Sample, t)
                                             -
                                              1))),
                          timeout2,
                          Timer2(DelayedTrip_SDD
                                  (P, timeout1, timeout2)(pre(t))),
                          t - Sample(Left_Sample(Sample, t) - 1)),
           PreviousInput1 := P(t),
           PreviousInput2

```

```

:= NOT (P(t) &
      Timer1(DelayedTrip_SDD(P, timeout1, timeout2)
            (Sample
              (Left_Sample(Sample, t)
                -
                1)))
      + t
      - Sample(Left_Sample(Sample, t) - 1)
      >= timeout1) #)
ELSE (# Relay := Relay(DelayedTrip_SDD(P, timeout1, timeout2)(pre(t))),
      Timer1
      := Timer1(DelayedTrip_SDD(P, timeout1, timeout2)(pre(t))),
      Timer2
      := Timer2(DelayedTrip_SDD(P, timeout1, timeout2)(pre(t))),
      PreviousInput1
      := PreviousInput1(DelayedTrip_SDD(P, timeout1, timeout2)
            (pre(t))),
      PreviousInput2
      := PreviousInput2(DelayedTrip_SDD(P, timeout1, timeout2)
            (pre(t))) #)
ENDIF
MEASURE rank(t)

SRS1_PROPERTY: LEMMA
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
          timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    t > Sample(0) AND NOT t = Sample(Left_Sample(Sample, t)) IMPLIES
      DelayedTrip_SRS1(P, timeout1, timeout2)(pre(t)) =
      DelayedTrip_SRS1(P, timeout1, timeout2)(t)

SRS1_PROPERTY1: LEMMA
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
          timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    t > Sample(0) AND NOT t = Sample(Left_Sample(Sample, t)) IMPLIES
      DelayedTrip_SRS1(P, timeout1, timeout2)(pre(t)) =
      DelayedTrip_SRS1(P, timeout1, timeout2)(t)

SDD_PROPERTY1: LEMMA
  FORALL (timeout1, timeout2: non_initial_time):
    P(Sample(n)) =
      PreviousInput1(DelayedTrip_SDD(P, timeout1, timeout2)(Sample(n)))

Timer2_PROPERTY1: LEMMA
  t > Sample(0) AND t = Sample(Left_Sample(Sample, t)) IMPLIES
    Timer2(DelayedTrip_SDD(P, timeout1, timeout2)(pre(t))) =
    Timer2(DelayedTrip_SDD(P, timeout1, timeout2)
          (Sample(Left_Sample(Sample, t) - 1)))

Timer1_Timer: LEMMA
  FORALL (timeout1, timeout2: non_initial_time):
    Timer_I(P, Sample, timeout1)(t) =
      Timer1(DelayedTrip_SDD(P, timeout1, timeout2)(t))

SDD_PROPERTY2: LEMMA
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
          timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    PreviousInput2(DelayedTrip_SDD(P,
          timeout1 - delta_L1,
          timeout2 - delta_L2)
          (Sample(n)))
    = NOT_Held_For_I(P, timeout1 - delta_L1, Sample)(Sample(n))

```



```

SDD_PROPERTY3: LEMMA
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
    timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    PreviousInput2(DelayedTrip_SDD(P,
      timeout1 - delta_L1,
      timeout2 - delta_L2)
      (Sample(n)))
    = NOT_Held_For_I(P, timeout1 - delta_L1, Sample)(Sample(n))

Timer2_Timer: LEMMA
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
    timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    Timer_I(NOT_Held_For_I(P, timeout1 - delta_L1, Sample), Sample,
      timeout2 - delta_L2)
      (t)
    =
    Timer2(DelayedTrip_SDD(P, timeout1 - delta_L1, timeout2 - delta_L2)
      (t))

DelayedTrip_Block1: THEOREM
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
    timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    DelayedTrip_SRS1(P, timeout1, timeout2)(t) =
    OPEN?(Relay(DelayedTrip_SDD(P, timeout1 - delta_L1,
      timeout2 - delta_L2)
      (t)))

DelayedTrip_Block_Tick: THEOREM
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
    timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    DelayedTrip_SRS(PP, timeout1, timeout2)(t) =
    OPEN?(Relay(DelayedTrip_SDD(PP, timeout1 - delta_L1,
      timeout2 - delta_L2)
      (t)))

DelayedTrip_Block: THEOREM
  FORALL (timeout1: Duration[K, TL, TR, delta_L1, delta_R1],
    timeout2: Duration[K, TL, TR, delta_L2, delta_R2]):
    DelayedTrip_SRS(PP, timeout1, timeout2)(Sample(n)) =
    OPEN?(Relay(DelayedTrip_SDD(PP, timeout1 - delta_L1,
      timeout2 - delta_L2)
      (Sample(n))))

END DelayedTrip

```

Bibliography

- [1] Parosh Aziz Abdulla and Aletta Nylen. Timed petri nets and bqos. In *ICATPN '01: Proceedings of the 22nd International Conference on Application and Theory of Petri Nets*, pages 53–70, London, UK, Springer-Verlag, 2001.
- [2] Rajeev Alur. Timed automata. In *Computer Aided Verification*, pages 8–22, 1999.
- [3] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, Springer-Verlag, 1982.
- [4] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.
- [5] Bruno Dutertre and Victoria Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5):267–278, May 1997.
- [6] Oana Florescu, Jeroen Voeten, Jinfeng Huang, and Henk corporaal. Error estimation in model-driven development for real-time software. In *Forum on specification and Design Languages*, pages 228–239, 2004.
- [7] Richard Gerber and Insup Lee. A layered approach to automating the verification of real-time systems. *IEEE Transactions on Software Engineering*, 18(9):768–784, September 1992.
- [8] Thomas A. Henzinger, Christoph M. Kirsch, Marco A.A. Sanvido, and Wolfgang Pree. 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] Xiayong Hu. Proving real-time properties of embedded software systems. M.sc., Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada, September 2002.

- [10] Jinfeng Huang, Jeroen Voeten, Oana Florescu, Piet Van Der Putten, and Henk Corporaal. Predictability in real-time system development. In *Advances in Design and Specification Languages for SoCs*, pages 123–139. Kluwer Academic Publishers, 2005.
- [11] Ryszard Janicki and Ridha Khédri. On a formal semantics of tabular expressions. *Science of Computer Programming*, 39(2-3):189–213, 2001.
- [12] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal: Status and future work. In Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.
- [13] Mark Lawford and Xiayong Hu. Right on time: Pre-verified software components for construction of real-time systems. Technical Report 8, Software Quality Research Lab, McMaster University, Hamilton, ON, Canada, 2002.
- [14] Mark Lawford and Murray Wonham. Equivalence preserving transformations of timed transition models. 40:1167–1179, July 1995.
- [15] Mark Lawford and HongYu Wu. Verification of real-time control software using pvs. In P. Ramadge and S. Verdu, editors, *Proceedings of the 2000 Conference on Information Sciences and Systems*, volume 2, pages TP1–13–TP1–17, Princeton, NJ, March 2000. Dept. of Electrical Engineering, Princeton University.
- [16] Mark Lawford. Lecture notes of course real-time system verification. <http://www.cas.mcmaster.ca/~lawford>.
- [17] Mark Lawford, Jeff McDougall, Peter Froebel, and Greg Moum. Practical application of functional and relational methods for the specification and verification of safety critical software. In Teodor Rus, editor, *Algebraic Methodology and Software Technology, AMAST 2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 73–88, Iowa City, IA, May 2000. Springer-Verlag.
- [18] Marius Mikucionis, Brian Nielsen and Kim G. Larsen. Real-time system testing on-the-fly. In *the 15th Nordic Workshop on Programming Theory*, number 34 in B, pages 36–38, Turku, Finland, October 29–31 2003. Åbo Akademi, Department of Computer Science, Finland. Abstracts.

- [19] Sam Owre, Natarajan Shankar, John Rushby, and David Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
- [20] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.
- [21] George M. Reed and Bill Roscoe. A timed model for communicating sequential processes. In *ICALP '86: Proceedings of the 13th International Colloquium on Automata, Languages and Programming*, pages 314–323, London, UK, 1986. Springer-Verlag.
- [22] John Rushby. Formal methods for dependable real-time systems. In *International Symposium on Real-Time Embedded Processing for Space Applications*, pages 355–366, Les Saintes-Maries-de-la-Mer, France, November 1992. CNES, the French Space Agency. Published by Cépaduès-Éditions, Toulouse, France.
- [23] Natarajan Shankar, Sam Owre, and John Rushby. A tutorial on specification and verification using PVS. Technical report, 1993.
- [24] Natarajan Shankar, Sam Owre, John Rushby, and David Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
- [25] Natarajan Shankar. Verification of real-time systems using PVS. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, Elounda, Greece, June/July 1993. Springer-Verlag.
- [26] Jens U. Skakkebak and Natarajan Shankar. Towards a Duration Calculus proof assistant in PVS. In H. Langmaack, W.-P. de Roever, and J. Vytöpil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 660–679, Lübeck, Germany, September 1994. Springer-Verlag.
- [27] Jan Springintveld, Frits Vaandrager, and Pedro R. D’Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1–2):225–257, 2001.
- [28] Farn Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283 – 1307, August 2004.
- [29] Alan Wasssyng, Mark Lawford, and Xiayong Hu. Timing tolerances in safety-critical software. In J. Fitzgerald, I.J. Hayes, and A. Tarlecki,

- editors, *FM 2005: Formal Methods: International Symposium of Formal Methods Europe Proceedings*, volume 3582 of *LNCS*, pages 157 – 172, Newcastle, UK, July 2005. Springer-Verlag.
- [30] Alan Wassyng and Ryszard Janicki. Using tabular expressions. In *Proceedings of International Conference on Software and Systems Engineering and their Applications*, volume 4, pages 1 – 17, Paris, December 2003.
- [31] Alan Wassyng and Mark Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: International Symposium of Formal Methods Europe Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 133–153, Pisa, Italy, August 2003. Springer-Verlag.
- [32] NASA Langley PVS Libraries Official Website. <http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html>.
- [33] PVS Official Website. <http://pvs.csl.sri.com>.
- [34] HongYu Wu. Formal verification of real-time software. Master’s thesis, McMaster University, February 2001.
- [35] Martin De Wulf, Laurent Doyen, Nicolas Markey, and Jean-Francois Raskin. Robustness and implementability of timed automata. In *Proc. of FORMATS04*, volume 3253 of *Lecture Notes in Computer Science*, pages 152–166, Grenoble, 2004.
- [36] Martin De Wulf, Laurent Doyen, and Jean-Francois Raskin. Almost asap semantics: From timed models to timed implementations. In *Proc. of HSCC04*, volume 2993 of *Lecture Notes in Computer Science*, pages 296 – 310, 2004.