A VERIFICATION SYSTEM FOR INTERVAL-BASED SPECIFICATION LANGUAGES WITH ITS APPLICATION TO SIMULINK

CHEN CHUNQING

(B.Sc. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2009

Acknowledgement

I would like to take this opportunity to express my deep and sincere gratitude to those who helped me, in one way or other, on my Ph.D. study in the last five years.

First and foremost, I am deeply indebted to my supervisor, Dr. DONG Jin Song, for his guidance, insight and encouragement throughout the course of my doctoral program. His careful reading, and constructive criticisms and suggestions of early drafts and other works make this thesis possible.

To my seniors, Dr. SUN Jun and Dr. LI Yuan Fang, and fellow student FENG Yuzhang - for your suggestions and discussions on all aspects of research works and generous sharing of your research experience.

To my former lab mates, Dr. LIANG Hui and Dr. YUAN Ling, and fellow students, LIU Yang and ZHANG Xian - your friendships and funny chit chat helped me go through the long and sometimes rough way of Ph.D. study.

I am grateful to Dr. Abhik ROYCHOUDHURY and Prof. P. S. THIAGARAJAN for the critical comments on this thesis. I am also thankful to the external reviewer and numerous anonymous referees who have reviewed this thesis and previous publications that are parts of this thesis and their valuable comments have contributed to the clarification of many ideas presented in this thesis.

This study was in part funded by the project "Rigorous Design Methods and Tools for Intelligent Autonomous Multi-Agent Systems" supported by Ministry of Education (MOE) of Singapore and the project "Reliable Software Design and Development for Sensor Network Systems" supported by National University of Singapore Academic Research Fund and the project "Formal Design Techniques for Reactive Embedded Systems" supported by Singapore A*STAR Research Grants. The School of Computing also provided the finance for me to present papers in several conferences overseas. Moreover, I have been encouraged by receiving the Dean's Graduate Research Excellence Award 2009. For all this, I am very thankful.

I wish to thank sincerely and deeply my parents Zhenhong and Yimei who have raised me, supported me, and always have belief in me these years.

Contents

1 Introduction and Overview **1**

1.1 Motivation and Goals 1

1.2 Thesis Outline 6

1.3 Publications from the Thesis 7

2 Background **9**

2.1 Timed Interval Calculus (TIC) 9

2.2 Prototype Verification System (PVS) 13

2.3 Duration Calculus (DC) 15

2.4 Simulink 18

3 Encoding TIC in PVS **21**

3.1 Constructing TIC Semantics in PVS 22

3.1.1 Time and Interval Domains 23

3.1.2 Timed Traces and Interval Operators 24

3.1.3 Expressions and Predicates 24

3.1.4 TIC Expressions 26
3.2 Checking TIC Reasoning Rules 28
3.3 Supplementary Rules and Proof Strategies 31
3.4 Summary 33

4 Machine-Assisted Proof Support for TIC **35**
4.1 Translating TIC Models to PVS Specifications 36
4.2 General Proof Procedure 38
4.3 Case Study - A Temperature Control System 40
4.3.1 Specifications of System Properties and Requirements 41
4.3.2 Proofs of Requirements 44
4.3.3 Experimental Results 48
4.4 Summary 50

5 Supporting DC in the Verification System **51**
5.1 Modeling DC Semantics in TIC 52
5.1.1 State Variables 52
5.1.2 State Expressions 54
5.1.3 Temporal Variables 55
5.1.4 Formulas 55
5.2 Validating DC Axioms and Reasoning Rules 57
5.3 Handling DC Proofs 61
5.4 Summary 64

6 Modeling Simulink Library Blocks **65**

6.1 TIC Schemas for Simulink Elementary Blocks 66

6.2 TIC Library Functions for Simulink Library Blocks 69

6.3 Discussions and Discoveries 72

6.4 Summary 76

7 Transforming Simulink Diagrams into TIC Schemas **79**

7.1 Transforming Elementary Blocks 80

7.2 Transforming Wires 83

7.3 Transforming Diagrams 84

7.4 Computing Unspecified Sample Times 85

7.5 Dealing with the *Ports and Subsystems* Category 88

7.5.1 Triggered Subsystems 90

7.5.2 Enabled Subsystems 92

7.6 Summary 96

8 Validation beyond Simulink **97**

8.1 Translating TIC Library Functions 99

8.2 Facilitating TIC Validation of Simulink Diagrams 102

8.3 Implementation and Experimental Study 103

8.3.1 Specifications of System Design and Requirements 104

8.3.2 Validating System Design against Requirements 110

8.4 Summary 114

9 Conclusion **117**

9.1 Main Contributions of the Thesis 117

9.2 Future Work Directions 120

9.2.1 Higher Automation for Verifying TIC Models 121

9.2.2 Further Development for Supporting Simulink Diagrams . . . 123

9.2.3 Expanding the Verification System 124

A Encoding of TIC in PVS **141**

A.1 Basic Definitions 141

A.2 TIC Reasoning Rules 146

A.3 Supplementary Rules 149

A.4 Proof Strategies 152

B Proof of the DC Rule $DC15$ **155**

C Supported Simulink Library Blocks **159**

C.1 Library Blocks Modeled by TIC Library Functions 159

C.2 Library Blocks Handled in Transformation 160

C.3 Commonly Used Simulink Library Blocks in TIC 160

D Handling Conditional Subsystems **167**

D.1 Triggered Subsystems with Discrete Control Inputs 167

D.2 Enabled Subsystems with Continuous Control Inputs 170

Summary

Real-time computing systems usually interact with physical environment, and their computations often involve mathematical functions of time. With their increasing usage in safety-critical situations, it is necessary and important to rigorously validate these system designs associated with the properties of the environment.

Timed Interval Calculus (TIC) is an expressive specification language on modeling and reasoning about real-time systems. It supports differential and integral calculus as well. The formal verification capabilities of TIC are useful to rigorously prove if system designs satisfy functional and non-functional (specifically, timing) requirements.

When real-time computing systems are complex, it is difficult to ensure the correctness of each proof step and to keep track of all proof details in a pencil-and-paper manner. It is thus necessary and important to develop a verification system to make proofs easier. On the other hand, the analysis of these systems usually involves mathematical reasoning such as integral calculus for modeling physical dynamics, and induction mechanisms for dealing with arbitrary intervals and continuous time domain. This thesis presents a systematic way to develop a system to carry out TIC verification at an interval level with a high degree of automation, and further illustrates the extensibility and benefits of our approach.

The verification system is built upon a powerful generic theorem prover, Prototype Verification System (PVS). From our rigorous checking of TIC reasoning rules, subtle flaws in original rules have been discovered. A collection of supplementary rules and proof strategies have also been developed to make proofs more automated. In addition, we have expanded the verification system to handle Duration Calculus (DC) which is another popular interval-based specification language. We can reason about DC axioms and perform DC proofs in a manner similar to manual DC arguments.

Based on the TIC-PVS verification system, a novel framework is proposed to explore the usage of formal methods on improving industrial tools, for example, Simulink which graphically models and simulates embedded systems. However, Simulink is deficient in checking (timing) requirements of high-level assurance, and this is where formals methods have strengths. Our framework can formally capture functional and timing aspects of Simulink diagrams, enlarge the design space of Simulink, and rigorously conduct validation of Simulink diagrams. Furthermore, semantic incompleteness and a bug in Simulink library blocks have been discovered.

Key words: **Real-time Computing Systems, Interval-Based Specification languages, Formal Verification, PVS, Simulink**

List of Figures

2.1 A system *calculation* in Simulink graphical and textual formats . . . 19

4.1 The abstract syntax tree of the requirement *SimpleReq* 38

6.1 An incorrect simulation result of the *Dead Zone* library block 74

6.2 A wrong simulation result of the *Interval Test* library block. 75

6.3 A correct simulation result of the *Interval Test* library block. 75

7.1 A system *calculation* in Simulink graphical and textual contents . . . 80

7.2 A diagram with specified sample times for blocks *Delay* and *IC*. . . . 85

7.3 A triggered subsystem with a continuous control input 91

7.4 An enable subsystem *open* in a system *tank*. 94

8.1 The framework structure to model and validate Simulink diagrams . . 104

8.2 The brake control system in Simulink 105

D.1 A triggered subsystem controlled by a discrete input 168

D.2 An enabled subsystem controlled by a continuous input 171

List of Tables

4.1 Validation results of the temperature control system 49

6.1 The initial relay state of the *Relay* library block in different cases . . 73

8.1 Experimental results of the validation of the brake control system . . 114

Chapter 1

Introduction and Overview

1.1 Motivation and Goals

Real-time computing systems are computer systems which need to meet real-time constraints: They react to events within a certain time interval, to produce output before a prescribed delay has elapsed, etc. These systems usually interact with the physical environment and involve mathematical functions of time. With their increasing usage in safety-critical situations such as fly-by-wire aircraft, and in vast production areas such as automobiles, their high-level assurance is required and timing analysis becomes necessary [108]. Moreover, it is important to model both discrete computer system behavior and continuous physical processes in order to rigorously validate these embedded real-time computing systems at an early development stage [34].

Formal methods [38, 68, 112] have been suggested as a systematic way of improving the quality of system designs in general. In the past decade, they have received more attention on developing embedded real-time systems [47, 67, 132]. The focus is to exploit mathematically well-founded notations to model system designs and require-

ments, and to rigorously verify whether designs satisfy desired requirements with the assistance of computers. Two well-established approaches to verification are model checking [37] and theorem proving [113]. Model checking is the principle of building a finite model of a system and checking that a desired property (usually formulated in temporal logic [83]) holds in that model. Theorem proving is the technique which consists of specifying both designs and requirements in some mathematical logic and proving the existence of the implication relationship between designs and requirements by logic inference.

Formal specification languages for real-time systems can be divided into two broad groups [6]: those based on time points [3, 5, 39, 69, 83, 100] and those based on time intervals [50, 82, 88, 93, 95, 137]. Point-based specification languages express system behavior with respect to certain time points, which are determined by a specific system state and by the occurrence of events marking state transitions. On the other hand, interval-based specification languages have a higher level view of time; they can avoid mentioning time points and express system behavior over a given period of time points. The latter is regarded as more appropriate and concise than the former because constraints on intervals rather than time points occur frequently in real-time systems, especially in control engineering [44, 75] (for example, the use of an integral equation to constrain a function within an interval).

Timed Interval Calculus (TIC) [50] is a highly expressive specification language for modeling and reasoning rigorously about real-time systems in terms of intervals. It is based on *set* theory and extends the well-known Z notation [123, 136]. TIC has been applied to specify and verify various systems [48, 81, 82, 131, 134] against functional and non-functional (for instance, timing) requirements, by the well-defined reasoning rules and the strong support of mathematical analysis. The modeling features, especially the continuous-time domain and the support of integral and differential cal-

culus [49], make TIC an excellent formalism that can potentially support the widely used industrial tool Simulink [85] because Simulink adopts continuous-time semantics [70].

When real-time computing systems are complex, analyzing their formal models in TIC becomes error-prone and difficult in a paper-and-pencil manner. It is thus necessary and important to develop a verification system to provide machine-assisted proof support for TIC with a high degree of automation. Nevertheless, the analysis of these systems usually involves mathematical reasoning such as integral calculus used to model physical dynamics, and induction mechanisms for dealing with arbitrary intervals and the continuous time domain. These characteristics are not well supported by model checking which usually applies a discrete abstraction for infinite state spaces [14, 36, 65]. This may lead in some cases to a discrepancy between the real physical system and its model resulting in unreliable analysis and inconsistent conclusions [96, 97]. In contrast, theorem proving [25, 62, 113] can directly handle infinite state spaces and support expressive specifications.

Instead of building a specialized theorem prover for TIC from scratch, we choose one of the powerful generic theorem provers, Prototype Verification System (PVS) [101], because of its highly integrated environment for writing formal specifications and developing rigorous verification. The PVS specification language is based on higher-order logic associated with a rich type system [114]. Its interactive theorem prover offers powerful automatic reasoning techniques at low levels such as arithmetic of *real numbers* and decision procedures for *sets*. Users can directly control proof development at a high level, for example, selecting proper user-defined proof strategies [102]. The NASA PVS library [19, 46] has included the formalization and validation of elementary calculus covering integration and differentiation. The library has been successfully applied to verify a practical aircraft traffic control system [20]. The

above strengths of PVS are useful to achieve our goal of developing mechanical proof support for TIC.

To develop a verification system using PVS for TIC, we firstly encode the TIC denotational semantics in PVS. Secondly, we formalize and validate all TIC reasoning rules based on the semantic encoding. From our rigorous validation process, we have discovered subtle flaws in some original reasoning rules which are often used to check important requirements, for example, safety requirements. Thirdly, we implement a tool to automatically translate TIC models to PVS specifications. Last but not least, to make the proving process more automated, we define a set of supplementary reasoning rules dedicated to certain domain-specific features, and a collection of PVS proof strategies to capture repetitive patterns of proof commands. These proof strategies also hide the detailed encoding of TIC from users. With our verification system, we can systematically validate TIC models at the interval level by applying the encoded TIC reasoning and supplementary rules. Proofs at low levels, such as propositional logic and real numbers, can be automatically discharged by exploiting the PVS reasoning power. Furthermore, we can cope with proofs which involve continuous dynamics and arbitrary (infinite) intervals.

By supporting the highly expressive TIC, the verification system is generic to handle other interval-based specification languages, such as Duration Calculus (DC) [137], Temporal-Interval Logic with Compositional Operators (TILCO) [88], Real-Time Graphical Interval Logic (RTGIL) [93], etc. In this thesis, we extend the verification system to support DC which is based on interval temporal logic [94], since TIC and DC offer similar operators and capabilities. Specifically, the time domain of both notations is continuous (while in TILCO, time is discrete), and both are well-suited for model and reason about *accumulative* behavior (which is not supported by RTGIL). Although TIC [82] and DC [139] were independently developed at around the

same time, there was unfortunately no comparative study in the literature. We investigate in this thesis the differences between TIC and DC, and elaborately model DC semantics using TIC. The modeling enables us to rigorously conduct DC proofs in a manner similar to manual DC arguments in our verification system. Besides, we have identified an improper proof step in a common DC case study.

Developing a verification system for TIC and DC has its own merits. It is also important if the verification system can make a useful contribution to practical tools, such as Simulink [85] which is used widely in many applications [12, 51, 92] for graphically specifying and simulating dynamic systems. By means of simulation, Simulink can illustrate system behavior under particular circumstances, such as specific parameter values and simulation periods. However, simulation is deficient in checking system behavior for large parameter values over possibly infinite simulation periods. In addition, *open* systems, whose exact input functions are usually unknown, are not analyzable in Simulink because simulation is inapplicable. Moreover, Simulink lacks timing analysis. We propose a framework, based upon the verification system for TIC, to complement Simulink. Existing work [1, 10, 21, 22, 89, 128, 129] focuses on specifying discrete behavior. Our approach differs from them in that we are the first to model Simulink diagrams in terms of continuous time. This difference allows our framework to cover a wider range of Simulink library blocks (for instance, the *Integrator* library block). The framework can enlarge the design space of Simulink by precisely specifying environmental properties of open systems and important (timing) requirements, and rigorous validate Simulink diagrams with a high level of machine-assisted proof support. In addition, we have discovered incomplete semantics and also a bug in the original documentation of Simulink library blocks.

1.2 Thesis Outline

The main contributions of our work consist of the development of a verification system for interval-based specification languages and the application of formal methods for assisting in the development of embedded real-time system designs. This section gives an overview of the structure of the thesis.

Chapter 2 introduces background information on specification languages and tools used in the presented work. We firstly review TIC and DC with their respective syntax and semantics. Next we briefly describe PVS modeling features and its interactive prover. Lastly we present Simulink.

Encoding the TIC semantics in PVS is demonstrated in Chapter 3. The basic constructs of TIC are modeled in the PVS typed, high-order logic specification language. Based on the encoding, we formalize and validate all TIC reasoning rules in PVS. A collection of supplementary rules and proof strategies are defined to simplify the reasoning process and hide detailed encoding of TIC to users.

Chapter 4 illustrates the advantages of the verification system. A translator implemented in Java translates TIC models to PVS specifications. A general proof procedure is proposed to systematically reason about TIC models. With a case study of a hybrid application, we show that verification can be rigorously carried out directly at the interval level, and the proof obligations at low levels can be automatically discharged using our developed system.

In Chapter 5, we extend the verification system to support DC. We firstly model DC constructs using TIC, and then formalize and check the correctness of DC axioms as well as reasoning rules. A common DC case study is used to indicate that the resulting verification system is able to handle DC proofs in a closed manner of the corresponding manual DC arguments.

Chapters 6, 7 and 8 are devoted to applying TIC to complement Simulink. We build up a framework to formally represent and rigorously validate Simulink diagrams. In Chapter 6, we elaborately construct a set of TIC library functions to model Simulink library blocks by capturing the time-dependent relationships. In Chapter 7, we describe a systematic way to transform various Simulink diagrams to TIC models. The translation preserves the functional and timing aspects, and it takes into account conditionally executed subsystem in Simulink. In Chapter 8, we demonstrate the advantages and benefits of the framework, such as enlarging design space and supporting validation beyond Simulink.

Lastly, Chapter 9 concludes this thesis, summarizes the main contributions and discusses possible future work directions.

1.3 Publications from the Thesis

Most of the work presented in this thesis has been published/accepted in international conference proceedings or journals.

The work on developing the verification system for TIC (Chapters 3 and 4) has been published in *The Thirtieth International Conference on Software Engineering* (ICSE'08, May 2008, Leipzig) [32]. The work on extending the verification system (Chapters 3, 4 and 5) has been accepted by the *ACM Transactions on Software Engineering and Methodology* [33]. The work on applying TIC to model Simulink diagrams (Chapters 6 and 7) has been published in *The Eighth International Conference on Formal Engineering Methods* (ICFEM'06, November, Macau) [28]. The work on developing machine-assisted proof support for validation beyond Simulink (Chapter 8) has been published in *The Ninth International Conference on Formal Engineering Methods* (ICFEM'07, November, Boca Raton) [31]. The comprehensive work on the

framework for supporting Simulink based on the verification system (Chapters 6, 7 and 8) has been accepted for publication in *Formal Aspects of Computing* [29].

Besides, the preliminary work on proposing the formal framework for Simulink has been presented in *The Doctoral Symposium of The Fourteenth International Symposium on Formal Methods* (FM'06, August, Hamilton) [26].

A full list of the publications on which the thesis is based is given below, and they are also included in the bibliography.

[26] C. Chen. A continuous-time approach to modeling and validating Simulink models. In *the Doctoral Symposium of the 14th International Symposium on Formal Methods*, Hamilton, Canada, 2006.

[28] C. Chen and J. S. Dong. Applying Timed Interval Calculus to Simulink diagrams. In *ICFEM'06: Proceedings of the 8th International Conference on Formal Engineering Methods*, pages 74-93, Macau, China, 2006.

[29] C. Chen, J. S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*. Accepted.

[31] C. Chen, J. S. Dong, and J. Sun. Machine-assisted proof support for validation beyond Simulink. In *ICFEM'07: Proceedings of the 9th International Conference on Formal Engineering Methods*, pages 96-115, Boca Raton, USA, 2007.

[32] C. Chen, J. S. Dong, and J. Sun. A verification system for Timed Interval Calculus. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 271-280, Leipzig, Germany, 2008.

[33] C. Chen, J. S. Dong, J. Sun, and A. Martin. A verification system for interval-based specification languages. *ACM Transactions on Software Engineering and Methodology*. Accepted.

Chapter 2

Background

This chapter presents background information on the notations, techniques and tools which are employed in this thesis. It is divided into four parts. In Section 2.1, we introduce TIC with its modeling features and verification capability. Section 2.2 is devoted to PVS, in particular its specification language and interactive prover. The following section describes DC and the differences between DC and TIC are highlighted. Finally, Simulink is briefly described in Section 2.4.

2.1 Timed Interval Calculus (TIC)

Timed Interval Calculus (TIC) is based on set theory and reuses the Z notation [123, 136]. It adopts total functions of continuous time to model system behavior [82], and defines *interval brackets* for concisely expressing properties in terms of intervals [50]. Interval endpoints can be explicitly accessed, and hence TIC can model behavior over special intervals by constraining interval endpoints.

The time domain $\mathbb{T}$ is the set of all non-negative real numbers. An interval is a

range of continuous time points. Intervals are classified into four basic types based on the inclusion of interval endpoints. For example, the operator $[\_ \ldots \_]$ defined below denotes a *both-closed* interval in the Z *axiomatic definition* style, where the expression $\mathbb{P}\,\mathbb{T}$ denotes a set of time points. Three other operators, $(\_ \ldots \_), (\_ \ldots \_]$, and $[\_ \ldots \_)$ for *both-open* intervals, *left-open and right-closed* intervals, and *left-closed and right-open* intervals, are defined similarly.

$$[\_ \ldots \_] : \mathbb{T} \times \mathbb{T} \rightarrow \mathbb{P}\,\mathbb{T}$$

$$\forall\, x, y : \mathbb{R} \bullet [x \ldots y] = \{z : \mathbb{T} \mid x \leq z \leq y\}$$

There are three types of elements which compose TIC models as shown below.

- *Constants.* A constant is independent of time points and intervals. For example, a maximum temperature *MaxTmp* is a real number, namely, $MaxTmp : \mathbb{R}$.
- *Timed traces.* These represent variables which are functions of time. Specifically, a timed trace is a *total* function from the time domain to the type of a variable, and the variable type can be either continuous or discrete. For instance, temperature in a room can be modeled by a timed trace *Tmp* with real numbers $\mathbb{R}$, $(V : \mathbb{T} \rightarrow \mathbb{R})$, while an alarm signal can be depicted by a timed trace *alarm* whose range consists of two values, 0 and 1, $(alarm : \mathbb{T} \rightarrow \{0, 1\})$.
- *Interval operators.* Distinct from a timed trace, an interval operator is a function from intervals to the type of a variable. There are three predefined interval operators in TIC, namely, α, ω, and δ. They have the same type, $\mathbb{I} \rightarrow \mathbb{T}$, where the symbol $\mathbb{I}$ denotes all nonempty intervals, and return the starting point, ending point and length of an interval respectively.

A key construction of TIC is interval brackets. A pair of interval brackets with a predicate enclosed by the pair forms a *TIC expression* which denotes a set of intervals

where the predicate holds everywhere. A predicate is usually expressed in the first-order logic and contains timed traces and interval operators. All references to the time domain and intervals can be elided in the predicate. For example, the TIC expression $[\![Tmp(\alpha) \leq Tmp]\!]$, where $[\![\ ]\!]$ is pair of interval brackets[1], represents a set of *both-closed* intervals, and in each interval temperature Tmp is not less than its value at the starting point of the interval. Without using $[\![\ ]\!]$, we need to explicitly associate the timed trace Tmp and the interval operator α with their corresponding domains, as shown in the following expanded set construction of the TIC expression.

$$\begin{aligned} & [\![Tmp(\alpha) \leq Tmp]\!] \\ = & \{x, y : \mathbb{T} \mid (\forall\, t : [x \ldots y] \bullet Tmp(\alpha([x \ldots y])) \leq Tmp(t)) \bullet [x \ldots y]\} \end{aligned}$$

Conventional set operators such as $\cup$ and $\cap$ can be applied to compose new TIC expressions. To depict sequential behavior over intervals, the concatenation operator $\frown$ is defined below to connect intervals end-to-end. Specifically, $\frown$ takes two sets of intervals X and Y as arguments, and returns a set of intervals each of which is composed by an interval x from the left-hand set X and an interval y from the right-hand set Y, provided (1) x occurs strictly before y and (2) x and y meet exactly, with no overlap and no gap. Note that this operator is a partial function, indicated by the symbol $\pfun$, as the concatenation of any two sets of intervals may return the empty set.

$$\begin{array}{|l} \_ \frown \_ : \mathbb{P}\,\mathbb{I} \times \mathbb{P}\,\mathbb{I} \pfun \mathbb{P}\,\mathbb{I} \\ \hline \forall\, X, Y : \mathbb{I} \bullet X \frown Y = \\ \qquad \{z : \mathbb{I} \mid \exists\, x : X;\ y : Y \bullet z = x \cup y \wedge (\forall\, t1 : x;\ t2 : y \bullet t1 < t2)\} \end{array}$$

TIC predicates specify system properties and requirements at the interval level by imposing constraints on TIC expressions. For instance, the following TIC predicate

[1]Three other basic types of interval brackets are $(\!)$, $(\!]$, and $[\!)$ for *both-open* intervals, *left-open and right-closed* intervals, and *left-closed and right-open* intervals.

as a subset relation over two sets of intervals specifies a periodic behavior that a sensor stores the temperature *Tmp_in* every k time units.

$$[\![\exists\, i : \mathbb{N} \bullet \alpha = i * k \wedge \omega = (i+1) * k]\!] \subseteq [\![store = Tmp\_in(\alpha)]\!]$$

In the above TIC predicate, the TIC expression at the left side of $\subseteq$ decomposes the time domain into a sequence of *left-closed and right-open* intervals where $\mathbb{N}$ denotes the set of all natural numbers; and each interval lasts k time units. The TIC expression at the right side depicts the periodic update of the stored temperature.

To manage TIC models in a structural manner, we adopt the Z schema structure to group a list of variables in its *declaration part* and specify the constraints over these variables in its *predicate part*. The following schema represents the above sensor, where the declaration $Tmp\_in : \mathbb{T} \nrightarrow \mathbb{R}$ captures the continuity feature that *Tmp_in* is continuous in any non-empty interval by the symbol $\nrightarrow$ defined in [49].

Sensor

$Tmp\_in : \mathbb{T} \nrightarrow \mathbb{R};\ store : \mathbb{T} \to \mathbb{R}$ [Declaration]

$[\![\exists\, i : \mathbb{N} \bullet \alpha = i * k \wedge \omega = (i+1) * k]\!] \subseteq [\![store = Tmp\_in(\alpha)]\!]$ [Predicate]

TIC defines a collection of reasoning rules for capturing timing properties of sets of intervals and their concatenations. These rules allow TIC verification to be carried out at the interval level. For instance, given a predicate which is independent of interval operators, the following rule can decompose a non-point interval in which the predicate holds in two concatenated intervals, both of which satisfy the predicate. Note that an implicitly necessary condition is that the time domain is continuous, as intervals over the discrete-time domain, such as an interval whose endpoints are a pair of consecutive discrete time points, cannot be decomposed.

if a predicate P is independent on α, ω, and δ, then we have
$[\![P \wedge \delta > 0]\!] = [\![P]\!] \frown [\![P]\!]$

In the above specification, the interval brackets $[\![\ ]\!]$ denote a union of four basic types of interval brackets, specifically, $[\![P]\!] == (P) \cup (P] \cup [P) \cup [P]$. This operator is often used when predicates are independent of interval endpoints.

Using TIC, we can formally specify system designs and important requirements such as *safety* and *bounded liveness*, and rigorously prove whether system designs imply requirements by deduction. A proof is usually divided into several sub-proofs, and each sub-proof concentrates on a simple requirement of a subsystem. Each deductive step in a proof is reached by rigorously applying a hypothesis (as an axiom), a TIC reasoning rule, a mathematical law, or a proved requirement from a sub-proof.

2.2 Prototype Verification System (PVS)

Prototype Verification System (PVS) [101] is an integrated environment for formal specification and formal verification. It builds on over 25 years experience at SRI in developing and using tools to support formal methods. The specification language of PVS is based on classic typed, higher-order logic. Built-in types include *Boolean* (`bool`), *real numbers* (`real`), *natural numbers* (`nat`) and so on. Standard logic and arithmetic operators are also defined.

Entities of PVS are introduced by means of *declarations*, which are the main constituent of PVS specifications. Declarations are used to define variables, constants, formulas, and so on. *Variable declarations* introduce new variables with their types. In addition, variables are local when they are defined in binding expressions which may involve keywords such as `FORALL` denoting the universal quantifier $\forall$ or `LAMBDA` denoting the symbol λ used in lambda abstractions. *Constant declarations* introduce new constants which can be functions, relations or the usual (0-ary) constants; for example, the declaration `f: [nat -> nat]` defines a total function (indicated by

the symbol `->`) whose domain and range are natural numbers. *Formula declarations* can introduce axioms by the keyword `AXIOM` and theorems by the keyword `LEMMA`. Axioms can be referenced by the proof command `lemma` during proofs.

New types can be defined from the built-in types using *type constructors*. Two frequently used constructors are *predicate subtypes* and *record types*. A predicate subtype denotes a subset of individuals in a type satisfying a given predicate; for instance, nonzero real numbers are written as {`x: real | x /= 0`}. Note that types in PVS are modeled as *sets*. A record type combines the types of its *record accessors*. For example, a record type `r` specified by `[# x: bool, y: real #]` is composed by the Boolean type and the type of real-numbers. A component of a record type can be accessed by its accessor name; the x-component of `r` is accessed by `r‘x`.

The *name overloading* technique is supported in PVS. This technique allows declaration identifiers to have the same names, even when the declarations are of different types. That is to say, functions can have the same name as long as they have different argument types, and a function and an axiom can have the same time although their types are different. PVS specifications are organized in *theories*. A theory usually contains type definitions, variable or constant declarations, axioms and theorems. Simple theories can be reused to construct complex ones by using the *importing* clause.

The PVS prover is based on sequent calculus and proofs are constructed interactively by developing a *proof tree*. The objective is to construct a complete proof tree in which all leaves are trivially true. Each node in a proof tree is a *proof goal* which is a sequent that consists of a list of formulas named *antecedents* and a list of formulas called *consequents*. The intuitive interpretation of a proof goal is that the conjunction of the antecedents implies the disjunction of the consequents.

The prover provides a collection of primitive proof commands such as expanding definitions (`expand`) and eliminating quantifies (`skosimp`), to manipulate proof trees. A

frequently used powerful proof command is `grind`, which does skolemization, instantiation, simplification, rewriting and applying decision procedures. Users can introduce more powerful proof strategies [9, 102] which combine basic proof commands so as to enhance the automation of verification in PVS.

PVS contains many built-in theories about logics, sets, numbers, and so on. These theories offer the mathematics needed to support specification and verification in PVS. For instance, set theory provides common definitions, such as `emptyset` representing $\varnothing$ and `subset?` for the subset relation. A recently developed NASA PVS library [19] has formalized the definitions of *limits*, *derivatives*, *continuity* and *integration*, and also validated a number of theorems of these definitions. The library has been successfully applied to analyze practical systems which involve continuous dynamics [20, 97].

2.3 Duration Calculus (DC)

Duration Calculus (DC) [137] is a logic-based approach to formal design of real-time systems. The basic calculus of DC [139] and its extensions, including Mean Value Calculus [140] and Extended Duration Calculus [141], are founded on the interval temporal logic [94] and integral calculus. We consider the basic DC in this article. It axiomatizes state durations for the Boolean state model, namely, integrals of Boolean-valued functions. Other extensions are introduced by adding to the basic DC extra axioms, which formalize the extended models and also their interrelations with the Boolean state model.

In the basic calculus of DC (abbreviated as DC henceforth), *state variables* are the basic type to model system states. A state variable P is a function from time to Boolean values $\{0, 1\}$, namely, $P : \mathbb{T} \to \{0, 1\}$. Furthermore, DC assumes that

state variables satisfy the *finite variability* property, which stipulates that a state variable can only change its value finitely many times in any bounded interval. This assumption ensures that state variables are integrable on every interval.

State expressions are formed by applying propositional logic operators over state variables, following the abstract syntax: $S ::= 0 \mid 1 \mid P \mid \neg S_1 \mid S_1 \wedge S_2$ where S, S_1, and S_2 are state expressions. Semantically, a state expression returns a value 0 or 1 at a time point. For example, two state variables *Gas* and *Flame* are introduced in a gas burner system to characterize the flowing and burning of gas. Specifically, $Gas(t) = 1$ means that gas is flowing and $Flame(t) = 1$ means that flame is burning. Hence, $Gas \wedge \neg\ Flame$ is the state expression specifying the leaking of gas, and it is interpreted with respect to a given time point t in the following way where $(\neg\ Flame)(t) = 1 - Flame(t)$.

$$(Gas \wedge \neg\ Flame)(t) = \begin{cases} 1 & \text{if } Gas(t) = 1 \text{ and } (\neg\ Flame)(t) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Temporal variables in DC which are real-valued functions of intervals can have a structure $\int S$ to denote the *duration* of a state expression S over a closed time interval $[b, e]$ where $b \leq e$. The duration is the accumulated presence time of S in the interval, namely, $(\int S)([b, e]) = \int_b^e S(t)dt$. Another predefined temporal variable in DC is ℓ which denotes the interval length, namely, $\ell([b, e]) = e - b$. We remark that intervals considered in DC are *both-closed*.

DC *terms* are built upon temporal variables or constants using mathematical operators, a summation for instance. DC *formulas* are composed by constraining DC terms or sub-formulas. Besides the conventional predicate logic operators such as the disjunction $\vee$ and the universal quantifier $\forall$, DC also adopts the chop operator $\frown$ to construct new formulas. Namely, the formula $\phi \frown \psi$, where ϕ and ψ are formulas, is satisfied by an interval if and only if the interval can be chopped into

two adjacent *both-closed* subintervals such that the first subinterval satisfies ϕ and the second satisfies ψ. Based on the chop operator, two commonly used operators over subintervals, specifically, $\Diamond$ (eventually) and $\Box$ (always), are defined as follows: $\Diamond\phi == (\text{true}^\frown\phi)^\frown \text{true}$ and $\Box\phi == \neg \Diamond(\neg \phi)$. For example, an interval $[b, e]$ satisfies a DC formula $\Diamond\phi$ provided there exist c and d such that $b \leq c \leq d \leq e$ and the interval $[c, d]$ satisfies ϕ.

A formula is *valid* in DC if and only if it holds in all intervals. For instance, a design property in a gas burner is that any leak represented by a state variable *Leak* should not last longer than 1 time unit, and this can be represented by the formula, $\Box(\lceil\!\lceil Leak \rceil\!\rceil \Rightarrow \ell \leq 1)$, where $\lceil\!\lceil Leak \rceil\!\rceil$ is an abbreviation of the formula $\int Leak = \ell \wedge \ell > 0$. Note that $\Box$ indicates that the property holds in any interval.

Properties of state durations are declared as axioms in DC. These axioms are important for deriving DC reasoning rules in DC proofs. Taking the axiom **DCA5** from [137] as an example, the axiom as shown below captures the relationship between the duration length (where x and y are nonnegative real numbers) of a state expression S and the chop operator.

DCA5 $(\int S = x)^\frown(\int S = y) \Rightarrow \int S = x + y,$

As we will show in Chapter 5.2, the DC axioms are declared as lemmas and they can be formally validated using our verification system.

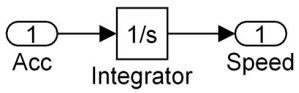
Although DC and TIC possess similar capabilities, their basis is different. TIC is based on the set theory and models system behavior by constraining the intervals on which predicates hold everywhere, while DC is based on interval temporal logic and models system behavior by accumulating state variables over *both-closed* intervals. Furthermore, TIC supports explicit references to interval endpoints, which can specify properties over special intervals with particular endpoints.

2.4 Simulink

Simulink [85] is a graphical toolkit that enables users to model and simulate dynamic systems whose outputs change over time. It has been widely used in industry covering electronics [92], automotive [51] and aerospace [12] application areas. A Simulink diagram made up of blocks and wires represents a set of time-dependent mathematical relationships which model a dynamic system. A block can be an *elementary block*, which is the basic structure unit of Simulink diagrams and denotes a primitive mathematical relationship between its inputs and outputs, such as calculating the sum of two inputs. An elementary block is created by assigning specific values to the parameters of a Simulink *library block* [84]. This parameterization technique enables a library block to create elementary blocks with different functionalities. A block can also be a diagram composed of other sub-diagrams and elementary blocks so as to support hierarchical modeling using Simulink. A *wire* is a directed edge which indicates a dependent relationship between two connected blocks; the input of the destination block depends on the output of the source block.

Every elementary block is considered to have a *sample time* as its execution rate. A sample time of an elementary block can be explicitly specified via the *SampleTime* block parameter, determined by the block type (for instance, elementary blocks generated by the *Integrator* library block have continuous sample time), or derived from blocks which connect to the block inputs. Simulink adopts continuous-time semantics [21, 70], and discrete systems are treated to be a special case of continuous systems: their behavior is piecewise constantly continuous. Moreover, Simulink supports *conditionally executed subsystems* whose executions depend on their control input values instead of time. For example, an *enabled* subsystem is active when its control input is positive.

A Simulink diagram is represented textually in a model file [84], which describes diagrams by *keywords* and *parameter-value pairs*. Parameter-value pairs denote the contents of diagrams such as block sample times by associating particular values with relevant parameters. Keywords followed by a pair of brackets models components at the same hierarchical layer of diagrams. Taking Figure 2.1 as an example, the left part graphically depicts a simple system which outputs speed as the integration of the acceleration from input port *Acc* to output port *Speed*, and the right part shows the corresponding textual representation.




```
1  System{ Name             "calculation"
2    Block{ BlockType       Inport
3           Name            "Acc"      }
4    Block{ BlockType       Integrator
5           Name            "Integrator"
6           InitialCondition "4"       }
7    Line{ SrcBlock         "Integrator"
8           DstBlock        "Speed"  }
9    Line{ SrcBlock         "Acc"
10          DstBlock        "Integrator" }}
```

Figure 2.1: A system *calculation* in Simulink graphical and textual formats

In the above context (from line 4 to line 6) of elementary block *Integrator*, its mathematical function *integration* is not explicitly specified but indicated by the value of the *BlockType* block parameter. Moreover, the initial value 4 of *Integrator* is not visually available in the diagram. In other words, model files contain all information of systems denoted in Simulink, and they are the source of our approach which captures (mathematical) functional and timing (namely, sample times) aspects and the structure of Simulink diagrams.

Chapter 3

Encoding TIC in PVS

As stated in Chapter 2.1, TIC is expressive enough to model various real-time computing systems covering continuous, discrete and hybrid systems. The powerful expressiveness on the other hand makes machine-assisted proof support for TIC challenging, as the verification of TIC models usually takes into account the continuous time domain and mathematics such as integration. To cope with these TIC characteristics, we apply theorem proving, a popular mechanism in formal verification, which can directly handle infinite state spaces and support mathematical reasoning.

A number of theorem provers exist including ACL2 [74] (a first-order logic prover), PVS [101], HOL [53] and Isabelle [99] (the latter three are higher-order logic proof tools). All provers have particular applications that they are especially suited for. Comparisons among provers have been intensively studied. For example, [55] compared PVS and Isabelle/HOL, and a comprehensive investigation [133] surveyed seventeen popular theorem provers. It is not our primary objective in this thesis to judge strengths of different provers. Our main objective is to exploit generic theorem provers to achieve machine-assisted proof support for TIC.

PVS attracts our attention because of its large mathematical standard library, in particular the NASA PVS library as mentioned in Chapter 2.2. Moreover, its specification language which is based on typed higher order logic can encode other expressive specification languages in principle. We describe here the semantics of the source logic, which in this case TIC, by using the base logic of the tool which in this case is PVS. This technique is known as the *semantic encoding* [16, 17, 130] technique. Furthermore, we choose to apply a *deep* rather than *shallow* semantic encoding, where a shallow encoding does not express the syntax of the source logic in the base logic. The advantage of deep encodings, which express both syntax and semantics in the theorem prover, is that theorems *about* the encoded language can be proved. That is to say, by adopting a deep encoding, not only can we reason about concrete TIC models, we can also check the correctness of TIC reasoning rules.

In the following of this chapter, we firstly model TIC denotational semantics in PVS. Next we formalize and validate all TIC reasoning rules based on the semantic encoding, and illustrate the subtle flaws discovered in original rules. Last but not least, we define a collection of supplementary reasoning rules dedicated to domain-specific properties, and a set of proof strategies to make proofs more automated. A comparison with other related works is presented at the end.

3.1 Constructing TIC Semantics in PVS

The construction of TIC semantics in PVS forms a foundation from which we formalize the TIC reasoning rules and carry out the verification of TIC models. An important requirement is that the resulting PVS specifications should be close to the structure of the TIC models, so any diagnostic information obtained at the level of PVS can be easily reflected back to the level of TIC. The PVS theories of the TIC semantics

are formed in a bottom-up manner, and each subsection below corresponds to a PVS theory. Simple theories are hence used to compose complex ones. All PVS theories for encoding TIC are available in Appendix A.1. To avoid the problem of sub-goal explosion arisen in reasoning procedures, we model TIC constructs, especially the interval brackets and concatenation operator, in a hierarchical manner. Moreover, the flexible style of type declaration in PVS reduces the size of the PVS specifications.

3.1.1 Time and Interval Domains

The time domain is represented by the PVS built-in type `nnreal` as a set of non-negative real numbers.

```
Time: TYPE = nnreal;
```

An interval is modeled as a tuple and its type is `GenInterVal` as shown below: the first element (`invt`) indicates the interval type, (for example, `CO` indicating that the interval is *left-closed and right-open*); the second element is also a tuple which consists of the starting point (`stp`) and the ending points (`etp`).

```
Interval_Type: TYPE = {OO, OC, CO, CC};
GenInterval: TYPE = [invt: Interval_Type, {stp, etp: Time | stp <= etp}];
```

The following type `II` denotes all non-empty intervals, and the constraints of interval endpoints with respect to interval types are captured. For example, the predicate `i‘1 = CC and i‘2‘1 <= i‘2‘2` specifies that the ending point can be equal to the starting point if the interval is *both-closed* (indicated by `CC`), where the apostrophe ‘ is the PVS projection operator to refer to components in a tuple. By using the predicate subtype technique in PVS, specific interval types are easily constructed based on `II`. For instance, `COInteral` which represents *left-closed and right-open* intervals restricts the interval type to be `CO`.

```
II: TYPE = {i: GenInterval | (i`1 = CC and i`2`1 <= i`2`2)
                   or ((i`1 = OO or i`1 = OC or i`1 = CO) and i`2`1 < i`2`2)};
COInterval: TYPE = {i: II | i`1 = CO};
```

3.1.2 Timed Traces and Interval Operators

A timed trace (`Trace`) is a function from time to the real numbers. We further model *discrete* timed traces (`BTrace`) whose ranges consist of two values, 0 and 1.

```
Trace: TYPE = [Time -> real];
BTrace: TYPE = [Time -> {x:real | x = 0 or x = 1}];
```

Interval operators are functions of intervals. They are independent from the inclusion/exclusion of interval endpoints. That is to say, we only need to define their functionalities with respect to `II` without respectively listing those of specific interval types (for example, `COInterval`). The following PVS specifications correspond to three predefined TIC interval operators, namely, α, ω, and δ.

```
ALPHA(i: II): Time = i`2`1;
OMEGA(i: II): Time = i`2`2;
DELTA(i: II): Time = OMEGA(i) - ALPHA(i);
```

3.1.3 Expressions and Predicates

As a modeling feature of TIC, references to the time domain and interval domain are elided in expressions and predicates within a pair of interval brackets. However, it is necessary for these references to be explicitly shown for the correct interpretation of expressions and predicates. We declare expressions (`TExp`) and predicates (`TPred`) to be functions in PVS where time and intervals compose the domain.

```
TExp: TYPE = [Time, II -> real];
TPred: TYPE = [Time, II -> bool];
```

Primitive elements of TIC form expressions and in turn predicates. An element is a constant, a timed trace, or an interval operator. By the overloading mechanism of PVS, the following function `LIFT` performs different functionalities according to the type of its first argument[1]. To be specific, `LIFT` returns the value at a time point `t` for a timed trace, and returns the value with respect to an interval `i` for an interval operator.

```
LIFT(c)(t, i): real = c;          % c: real, t: Time, i: II
LIFT(tr)(t, i): real = tr(t);     % tr: Trace
LIFT(tm)(t, i): real = tm(i);     % tm: Term
```

When interpreting an expression of TIC, we pass its parameters denoting the time domain and interval domain to its constituent expressions. This propagation repeats until all constituent expressions are primitive elements. For instance, a subtraction of TIC is interpreted below in PVS, where the pair `(t, i)` is passed to the component expressions `el` and `er`. A similar approach is used to handle predicates (a disjunction of TIC is provided as an example).

```
-(el, er)(t, i): real = el(t, i) - er(t, i);      % el, er: TExp
or(pl, pr)(t, i): bool = pl(t, i) OR pr(t, i);    % pl, pr: TPred
```

TIC also supports elementary calculus including integration and differentiation. We adopt the NASA PVS library to model the calculus. For example, the expression $\int_{\alpha(i)}^{\omega(i)} tr$ is represented by the following PVS function `TICIntegral` which invokes function `Integral` defined in the NASA PVS library.

```
TICIntegral(tr)(t, i): real = Integral(ALPHA(i), OMEGA(i), tr)
```

[1]Characters following the symbol '%' are comments in PVS.

3.1.4 TIC Expressions

A TIC expression denotes a set of intervals. The basic structure of TIC expressions is a pair of interval brackets which encloses a predicate. Common set operators can be applied to form complex TIC expressions. Here we demonstrate how to encode the TIC expressions with interval brackets and the special set operator of TIC, namely the concatenation operator. Other types of TIC expressions can be constructed by the built-in functions in the PVS set theory.

A pair of interval brackets enclosing a predicate represents a set of intervals, and in each interval the predicate holds *everywhere*, namely, at all time points of the interval. In the following PVS specifications, function `t_in_i` detects whether a time point is within an interval according to the interval type. Note that there are four basic types of interval brackets. Based on `t_in_i`, we define the function `Everywhere?` to check if a predicate holds in an interval. TIC expressions containing general interval brackets $[\![\ ]\!]$ are thus modeled by function `AllS`. In addition, TIC expressions of basic types of interval brackets are easily specified by applying the predicate subtype mechanism (for example, function `COS` representing $[\ )$).

```
t_in_i(t, i): bool = (i`1 = OO and t > i`2`1 and t < i`2`2) or
                     (i`1 = OC and t > i`2`1 and t <= i`2`2) or
                     (i`1 = CO and t >= i`2`1 and t < i`2`2) or
                     (i`1 = CC and t >= i`2`1 and t <= i`2`2);
Everywhere?(pl, i): bool = forall t: t_in_i(t, i) => pl(t, i);
AllS(pl): setof[II] =  {i | Everywhere?(pl, i)};
COS(pl): setof[COInterval] = {i: COInterval | Everywhere?(pl, i)};
```

A concatenation in TIC requires that two connected intervals must meet *exactly*, namely, no overlap and no gap. There are thus eight correct ways of concatenations from four basic types of intervals. Instead of modeling each one individually, we represent all eight cases together by the following function `concat`. This function

takes two sets of intervals as parameters (namely, `iisl` and `iisr`) which may contain any type of intervals, and each interval in the returned set is composed by two adjacent intervals respectively from two parameters.

```
ConcatType(l, r, re: II): bool =
    (re`1 = OO AND ((l`1 = OC AND r`1 = OO) OR (l`1 = OO AND r`1 = CO)))
 OR (re`1 = CO AND ((l`1 = CC AND r`1 = OO) OR (l`1 = CO AND r`1 = CO)))
 OR (re`1 = OC AND ((l`1 = OO AND r`1 = CC) OR (l`1 = OC AND r`1 = OC)))
 OR (re`1 = CC AND ((l`1 = CO AND r`1 = CC) OR (l`1 = CC AND r`1 = OC)));
concat(iisl, iisr: PII): PII = {i | exists (i1, i2: II):
    ConcatType(i1, i2, i) AND member(i1, iisl) AND member(i2, iisr) AND
    OMEGA(i1) = ALPHA(i2) AND ALPHA(i1) = ALPHA(i) AND OMEGA(i2) = OMEGA(i)};
```

In the above PVS specifications, function `ConcatType` constrains the types of concatenated intervals. The constraints cover all eight cases. Being different from other constraints in `concat` (for example, `OMEGA(i1) = ALPHA(i2)` indicates that a concatenated interval's ending point is equal to the starting point of the other), the application of `ConcatType` in `concat` encapsulates the predicates of interval types at a lower level. That is to say, we create a hierarchical structure. We remark that this structure is useful to avoid the problem of sub-goal explosion which is often encountered during reasoning procedures in PVS. That is, the PVS prover automatically splits a proof goal into a number of sub-goals at a proof step, although the split is unnecessary at that step since there are many repetitive proof commands used to discharge those sub-goals. For instance, if we directly specify eight constraints of interval types in `concat`, the prover would automatically split one proof goal into eight sub-goals when expanding the concatenation definition in PVS, although these sub-goals can be proved by applying many repetitive proof commands.

So far, we have faithfully formalized the TIC constructs in PVS, while the way of handling TIC schemas and TIC predicates will be presented in Chapter 4.1. During the encoding, the overloading mechanism has assisted us to define the function `LIFT`

with different functionalities, and the higher-order logic of the PVS specification language has facilitated the interpretation of expressions and predicates of TIC in the bottom-up manner. These PVS theories of the TIC semantics form a base from which to validate the TIC reasoning rules and support mechanical verification of TIC models as we will show in the following chapters.

3.2 Checking TIC Reasoning Rules

TIC reasoning rules capture the properties of sets of intervals. They are used to verify TIC models at the interval level. Guaranteeing their correctness is thus necessary and important. We firstly describe the challenge of the validation. Next, we demonstrate flaws discovered from our rigorous checking process and provide remedies.

Checking TIC reasoning rules is not trivial. Though some of them can be automatically proved by the PVS prover, others require complicated analysis which covers different interval types (for example, there are sixteen cases analyzing a concatenation operation over three sets of intervals) and various types of predicates (for example, a predicate can depend on interval operators, time points, or both). Taking the rule introduced in Chapter 2.1 as an example, its PVS specification is given below based on the encoding presented in the previous section, where function `No_Term?` returns true when a predicate `pl` is independent from interval operators.

```
CONC_CONC: LEMMA No_Term?(pl) =>
    AllS(pl AND LIFT(DELTA) > LIFT(0)) = concat(AllS(pl),AllS(pl));
```

To validate the above rule, we need to consider the concatenation of two sets of all types of intervals. Therefore, there are eight cases. In the reasoning process, human interactions are helpful to increase efficiency. A simplified proof goal below aims to show that there exist two concatenated intervals, which form an interval `x!1` and

satisfy the hypotheses depicted by three antecedents (prefixed by negative integers). For instance, the antecedent at `[-1]` restricts the type of `x!1` to be *left-closed and right-open*. To prove the goal, we select the middle point of `x!1` as the connecting point, namely, `(ALPHA(x!1) + OMEGA(x!1))/2`, and then instantiate the requested intervals by applying our defined proof strategy `assignconcat`.

```
[-1]  TypeOf(x!1) = CO
[-2]  AllS(pl1!1 AND LIFT(DELTA) > LIFT(0))(x!1)
[-3]  No_Term?(pl1!1)
|-------
[1]   concat(AllS(pl1!1), AllS(pl1!1))(x!1)
Rule? (assignconcat 1 "(CO, (ALPHA(x!1), (ALPHA(x!1) + OMEGA(x!1))/2))"
                      "(CO, ((ALPHA(x!1) + OMEGA(x!1))/2, OMEGA(x!1)))")
```

The PVS prover always checks the correctness of assignments, so we are required to show that the above user-specified intervals satisfy the concatenation definition indicated by the function `concat`. Doing so can thus prevent mistakes by users such as assigning two concatenated intervals with the *both-open* interval type.

During our rigorous validation of all TIC reasoning rules, two subtle flaws in the original reasoning rules have been discovered. Below we describe the problematic rules with counterexamples, followed by their corresponding solutions which have been validated in PVS.

- The *True and False* rule is frequently used to reason about safety requirements. The original rule states that a predicate P is true in all intervals if and only if its negation is true nowhere. That is, $[\![P]\!] = \mathbb{I} \Leftrightarrow [\![\neg\ P]\!] = \varnothing$. However, the implication, $[\![\neg\ P]\!] = \varnothing \Rightarrow [\![P]\!] = \mathbb{I}$, does not hold in certain circumstances.

  For example, let x be a timed trace having the value 1 from time points 5 to 7 and the value 0 elsewhere. It is obvious to see that the predicate $\neg\ P == x =$

$1 \wedge \delta = 3$ fails everywhere, although its negation $P == x \neq 1 \vee \delta \neq 3$ is false in some intervals such as the interval $[5 \ldots 8]$.

To solve the problem, a stronger hypothesis is needed. The predicate within interval brackets should be independent of interval characteristics, specifically, the starting point, ending point and length of an interval. The modified rule is expressed in PVS below, where PVS keywords `emptyset` and `fullset` denote the empty set and the set of all intervals respectively.

```
Emp_to_All: LEMMA No_Term?(pl) =>
    AllS(not pl) = emptyset => AllS(pl) = fullset;
```

- The *Concatenation Duration* rule is useful to deal with proofs involving concatenation. Using the rule, a set of intervals can be decomposed into two concatenated sets of intervals with specified interval lengths. So, given a predicate P where interval operators do not occur, if we have $r, s : \mathbb{T}$ and $r > 0 \vee s > 0$, then we can deduce $[\![P \wedge \delta = r + s]\!] = [\![P \wedge \delta = r]\!] \frown [\![P \wedge \delta = s]\!]$.

  However, the above equality in terms of sets of intervals does not always hold. For example, if $r = 0$, then any interval of $[\![P \wedge \delta = r]\!]$ must be *both-closed* according to the interval definition. However, it is possible that $[\![P \wedge \delta = r+s]\!]$ contains intervals which are *left-open*, and hence type conflict occurs. The conflict can be removed by a stronger assumption, namely, $r > 0 \wedge s > 0$.

We remark that this is the first time that the first flaw has been discovered (while the second flaw has also been observed by Dawson and Goré [40]). These discoveries demonstrate the benefits of exploiting a theorem prover for rigorous verification.

Based on the lemma `Emp_to_All`, we further derive a new rule `EmpCC_to_All` to reduce the proof complexity. When applying `Emp_to_All`, we have to show that the proof goal can be discharged respectively with respect to four basic interval types,

although usually each sub-proof follows a similar reasoning process. In contrast, the new rule expressed below allows us to focus on just one interval type, namely, *both-closed* intervals (as indicated by `CCS`).

```
EmpCC_to_All: LEMMA No_Term?(pl) =>
    CCS(not pl) = emptyset => AllS(pl) =  fullset;
```

Currently, we have formalized and validated all TIC reasoning rules in PVS. Their PVS specifications are available in Appendix A.2. These rules are applied as lemmas in PVS when reasoning about TIC models. Two flaws of original reasoning rules have been discovered and avoided.

3.3 Supplementary Rules and Proof Strategies

The TIC reasoning rules validated in Section 3.2 capture primitive properties of sets of intervals. They are inadequate to support domain-specific characteristics. For example, if a *continuous* timed trace *tr crosses* a threshold TH at an interval i in a way $tr(\alpha(i)) < TH \wedge tr(\omega(i)) > TH$, then we can deduce that i can be decomposed into three connected intervals, where the values of tr are larger than TH everywhere in the last subinterval and are equal to TH in the middle subinterval. This property is expressed by the following PVS lemma, where `o` is the *function composition* operator in PVS and function `continuous` expresses the property that a timed trace `tr` is continuous over the time domain.

```
mid_ivl_exi: LEMMA continuous(tr) => subset?(
    AllS((LIFT(tr) o LIFT(ALPHA)) < LIFT(TH) and
         (LIFT(tr) o LIFT(OMEGA)) > LIFT(TH)),
    concat(AllS(TRUE), concat(AllS(LIFT(tr) = LIFT(TH)),
                              AllS(LIFT(tr) > LIFT(TH)))));
```

Note that the above domain-specific property is not captured by any existing TIC

reasoning rule. It is derived from the classic *intermediate value* theorem of continuous functions. Its correctness has been validated in PVS, and we can hence apply `mid_ivl_exi` when analyzing continuous dynamics.

To make the reasoning process in PVS more automated and shield the detailed TIC encoding from users, we have developed several PVS proof strategies. Each strategy combines repetitive proof commands which are frequently used in practice. These strategies mainly cope with quantified PVS formulas, since the PVS prover possesses powerful capabilities (such as automatic deduction and simplification) for reasoning about primitive formulas which are represented in the propositional logic. According to the quantifier type, these strategies are classified into two groups. One eliminates the *universal* quantifier by *skolemization*, and the other removes the *existential* quantifier by proper instantiation. In addition, they usually automatically expand PVS functions which encode the TIC semantics, and detailed encoding of TIC can hence be hidden from users. We present below the strategy `AssignInvlnTime` which offers a flexible way to assign an interval and a time point to a user-specified formula.

```
1: (defstep AssignInvlnTime (fnum &OPTIONAL ivl pt)
2:   (try (else (expand "OOS" fnum) (else (expand "OCS" fnum)
3:           (else (expand "COS" fnum) (else (expand "CCS" fnum)
4:              (else (expand "AllS" fnum) (skip))))))
5:        (then (if ivl (inst fnum ivl) (inst? fnum))
6:              (expand "Everywhere?" fnum)
7:              (if pt (inst fnum pt) (inst? fnum)))
8:        (skip)))
```

Using the above strategy, we can either instantiate explicit values of an interval (denoted by `ivl` at line 1) and a time point (by `pt`), or let the PVS prover automatically fix the values by using the PVS proof command `inst?` (at lines 5 and 7). This strategy handles all interval types by repeatedly applying the basic PVS proof strategy

`else`[2] from line 2 to line 4 to expand proper PVS functions which encode interval brackets. Note that at line 6 `AssignInvlnTime` automatically expands the function `Everywhere?` (defined in Section 3.1.4). In other words, the strategy hides the detailed encoding of TIC, namely, `Everywhere?`, from users.

We have constructed twenty-five supplementary rules and eleven PVS strategies, and their PVS specifications are available in Appendixes A.3 and A.4, respectively. They can assist in simplifying the verification of TIC models in practice.

3.4 Summary

In this chapter, we have chosen the *deep semantic encoding* approach to model TIC into PVS, including the TIC constructs and TIC reasoning rules. We have also developed a collection of supplementary reasoning rules and proof strategies to simplify proofs. These serve as a solid foundation in our verification system; we can rigorously carry out verification of TIC models with a high level of automation as we will show in the next chapter.

Through the above procedure, we have also demonstrated the usability of PVS as a generic theorem prover to cope with an expressive specification language, namely TIC. To be specific, the base logic of the PVS specification language, typed higher order logic, has facilitated the encoding of the TIC semantics, for instance, TIC expressions; and the large mathematical library predefined in PVS has supported the advanced mathematics in TIC such as continuous functions and integration. Moreover, the PVS prover has assisted us to discover two subtle flaws in the original TIC reasoning rules.

[2]The `else` proof strategy [105] executes the first step, and if that does nothing, then the second step is executed.

There were two other works on supporting TIC proofs by the theorem proving approach. Dawson and Goré [40] validated the correctness of TIC reasoning rules using Isabelle/HOL [99]. However, TIC semantics was incompletely encoded in their work. For example, operators used to construct TIC predicates and expressions were not modeled. It is hence difficult to support TIC verification in general. Cerone [23] defined several axioms to formalize TIC semantics. In his encoding of the TIC concatenation operator, two *both-open* intervals are allowed to be linked, although this interpretation is different from the original definition [50]. Moreover, Cerone's work considered only five TIC reasoning rules. In contrast, we have taken into account the complete TIC semantics and all TIC reasoning rules. One subtle flaw of a reasoning rule, which had not been identified before, has been discovered from our rigorous validation process. Furthermore, our system supports mathematical analysis including differential and integral calculus, which is not handled by those previous works.

Chapter 4

Machine-Assisted Proof Support for TIC

Validation of complex real-time computing systems modeled in TIC usually involves mathematical reasoning and takes into account arbitrary infinite intervals of continuous time. When systems are complex, it is difficult to guarantee the correctness of each proof step and keep track of proof details in a paper-and-pencil manner. Machine-assisted proof support thus becomes important and necessary.

Based on the previous chapter which encodes TIC semantics and reasoning rules in PVS, we demonstrate in this chapter how our verification system can ease proofs of TIC models with a high level of automation. We begin by describing the translation from TIC schemas which represent system properties and TIC predicates which specify desired requirements to PVS specifications. Next, a tool supporting the graphical editing and the automatic translation of TIC is presented. Following an illustration of a general proof procedure, we use a typical hybrid application, a temperature control system, to show the advantages of the verification system.

4.1 Translating TIC Models to PVS Specifications

Using TIC, system properties and requirements are modeled at the interval level: TIC schemas capture system properties, and TIC predicates specify requirements. We describe below the way to represent them in PVS respectively.

- TIC schemas are used to structure and compose models, collating pieces of information, encapsulating them and naming them for reuse. Each schema denotes a composite type made up of a set of bindings; each binding relates a declared variable with its restrictive values. This modeling feature enables schemas to be used as types, to support component-based design [77, 71, 42].

  Each schema is represented by a PVS *record* type. Specifically, we construct a set of records in PVS, where schema declarations are denoted by record accessors associated with corresponding types and schema predicates are used to constrain relationships over the record accessors. Moreover, implicit properties indicated by certain kinds of functions in TIC, such as continuity and integrability, are captured by additional constraints in PVS to further restrict the record type.

- A TIC predicate specifies a requirement of a system or some components, and is constructed based on the TIC schemas of the relevant system or components. Each TIC predicate is represented by a PVS *theorem* formula which is constructed based on the PVS specifications that represent corresponding TIC schemas.

The above relationships between TIC models and PVS specifications can be informally illustrated below, where variable `temp` in the PVS type declaration is auxiliary for referring to record accessors used in the generated `Predicate` specification. Note that the translated PVS specifications closely follow the structure of TIC models.

schema_name *declaration* --- *predicate*	`SchemaName: TYPE = {temp: [# Declaration #] | Predicate }`
requirement_name == *predicate*	`RequirementName : THEOREM Predicate`

To automatically translate TIC models to PVS specifications, we develop an automated process which consists of three steps: *scanning*, *parsing* and *translating*. A scanner splits TIC models into a sequence of meaningful tokens such as TIC interval brackets, mathematical operators and so on. A parser constructs a set of abstract syntax trees (ASTs) for TIC models based on those tokens. Each AST represents a TIC model, and the leaves of an AST denote the primitive elements of a TIC model, namely, constants, timed traces or interval operators. A translator traverses an AST in a top-down manner to produce corresponding PVS specifications.

For example, the following TIC predicate named *SimpleReq* depicts a requirement that temperature in a room is always lower than the value 40. Variable *Room* is a TIC schema which models the behavior of temperature which is denoted by variable *tmp*. The symbol "." is the Z selection operator [123] for selecting a schema variable; for example, *sys.tmp* refers to the variable *tmp* declared in a schema named *Room*.

$$SimpleReq == \forall\, sys : Room \bullet \mathbb{I} = [\!\!\lceil sys.tmp < 40 \rceil\!\!]$$

The above TIC predicate is parsed to an AST shown in Figure 4.1 in a structural manner. By traversing the AST, we obtain the following PVS specification, where the PVS projection function ‘ maps the Z selector operator. We remark that function `LIFT` (defined in Chapter 3.1.3) is required to model the timed trace *sys.tmp* and the constant 40 where both are in the interval brackets $[\!\!\lceil \rceil\!\!]$; `LIFT(sys‘tmp)` and `LIFT(40)` are functions whose arguments are time points and intervals.

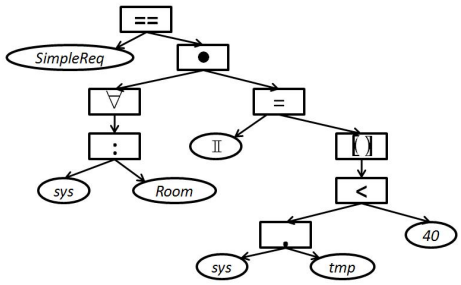


Figure 4.1: The abstract syntax tree of the requirement *SimpleReq*

```
SimpleReq: THEOREM forall (sys: Room): fullset = AllS(LIFT(sys‘tmp) < LIFT(40));
```

We have developed a tool based on HighSpec [43] which provided functionalities such as graphical editing, syntax and type checking, and automatic projection, for an integrated formal modeling technique, OZTA, a combination of Object-Z [121] and Timed Automata [4]. We have extended the HighSpec graphical editor (for instance, encoding the TIC special symbols ⟦ and ∫) to support the editing of TIC models. We have also reused the infrastructure of HighSpec to construct an translator which implements the translation algorithm as mentioned in this section. Using the tool, the previous TIC predicate *SimpleReq* and the TIC models of the case study in the next section are automatically translated to PVS specifications.

4.2 General Proof Procedure

In general, to validate a real-time computing system against an expected requirement is equivalent to proving that the TIC schemas which represent system properties can logically imply the TIC predicate which denotes the requirement. A proof of TIC models is indeed a deduction which starts from hypotheses (namely, system properties) and proceeds in a forward manner towards a proof goal (namely, an expected

requirement). At each deductive step, a TIC reasoning rule, a supplementary rule, or a mathematical law is applied. Usually a proof can be decomposed into several sub-proofs for simpler subsystems.

As highlighted in Chapter 1.1, we aim at developing a verification system to systematically conduct TIC proofs with a high degree of automation. Moreover, a reasoning process in the verification system is intended to follow a manner similar to corresponding manual TIC arguments. Based on our experiments, we present here a general proof procedure, which is effective and efficient in practice.

The procedure starts with a proof sequent (introduced in Chapter 2.3) where its consequent represents an expected requirement in terms of intervals. The main objective is to eliminate quantified formulas in the sequent by assigning, manually or automatically, appropriate values to intervals and time points. The PVS prover can directly manipulate the resulting sequent and automatically discharge many tedious parts of the proof, for instance, reasoning about linear arithmetic and sets.

1. Add constraints which model system properties as new antecedents to the sequent. At the beginning of a proof, the consequent contains only names of PVS records, where those records model a system or components by the predicates over their accessors. These constraints can be inserted to the sequent as new antecedents by applying the PVS proof command `typepred` to relevant record names during a proof process.

2. Use TIC reasoning rules and supplementary rules (formalized and validated in Chapters 3.2 and 3.3). The sequent can be resulted in two directions. (1) *Backward proof*: generating new consequents if some of the current consequents match the conclusion of a rule. (2) *Forward proof*: generating new antecedents if some of the current antecedents satisfy the hypotheses of a rule.

3. Instantiate intervals and time points. This step removes the quantifiers which denote intervals and time points in the sequent. Values of intervals and time points can be manually specified or automatically fixed by the PVS prover.

4. Apply mathematical laws. As TIC models often contain continuous dynamics modeled by the integral and differential calculus, we can choose dedicated mathematical laws from the NASA PVS library to support the analysis.

5. Invoke the PVS proof command `grind` as the last step to automatically discharge the rest of the proof.

In the above procedure, the first two steps manipulate a proof goal at the interval level, and the last step reduces human effort by utilizing the PVS reasoning power on accomplishing proofs at low levels. Furthermore, at Steps 3 and 4, human heuristics are sometimes needed to guide the prover to improve the efficiency. For example, we can directly specify a proper interval value instead of letting the PVS prover try out all possible values which can be many. In the next section, we will demonstrate how this proposed proof procedure can facilitate the analysis of our case study.

4.3 Case Study - A Temperature Control System

A temperature control system [98, 128] is a hybrid application which controls the temperature by turning a heater on or off. This system is requested to fulfill important requirements such as safety requirements.

We firstly describe the system properties and requirements in TIC models as well as in their translated PVS specifications. Next, we present the verification of two requirements, where the first illustrates the advantages of the general proof procedure,

and the second demonstrates the ability to handle induction proofs in the verification system. At the end, we summarize the experimental result.

4.3.1 Specifications of System Properties and Requirements

The control system consists of two subsystems. Subsystem *plant* represents the physical environment where temperature changes *continuously*, following different integral equations with respect to the heater status. Subsystem *controller* models the control logic, namely, turning on or off the heater according to the temperature from *plant*.

In the following TIC schema *Plant* which models the subsystem *plant*, temperature is denoted by variable *tmp_out*, and variable *heater_in* indicates the heater status.

- When the heater is *off* ($heater\_in = 0$) in an interval $[b, e]$, temperature decreases following the equation $tmp\_out(e) = tmp\_out(b) - \int_b^e (0.1 * tmp\_out)$.
- When the heater is *on* ($heater\_in = 1$) in an interval $[b, e]$, temperature increases following the equation $tmp\_out(e) = tmp\_out(b) + \int_b^e (6 - 0.1 * tmp\_out)$.

The corresponding PVS record type is shown below, where PVS variables (for example, `Trace`) and functions (for instance, `LIFT`) for encoding TIC constructs are defined in Chapter 3.1. Note that the temperature is declared to be a continuous function in TIC as indicated by $\twoheadrightarrow$, and this feature is also captured by the PVS function `continuous` from the NASA PVS library.

Plant

$tmp\_out : \mathbb{T} \twoheadrightarrow \mathbb{R};\ heater\_in : \mathbb{T} \to \{0, 1\}$

$[\![heater\_in = 0]\!] \subseteq [\![tmp\_out(\omega) = tmp\_out(\alpha) - (1/10) * \int tmp\_out]\!]$
$[\![heater\_in = 1]\!] \subseteq [\![tmp\_out(\omega) = tmp\_out(\alpha) + 6 * \delta - (1/10) * \int tmp\_out]\!]$

```
Plant: TYPE = { temp: [# tmp_out: Trace, heater_in: BTrace #] |
    subset?(AllS(LIFT(temp`heater_in) = LIFT(0)),
            AllS((LIFT(temp`tmp_out) o LIFT(OMEGA)) =
                 (LIFT(temp`tmp_out) o LIFT(ALPHA)) -
                    LIFT(1/10) * TICIntegral(temp`tmp_out))) AND
    subset?(AllS(LIFT(temp`heater_in) = LIFT(1)),
            AllS((LIFT(temp`tmp_out) o LIFT(OMEGA)) =
                 (LIFT(temp`tmp_out) o LIFT(ALPHA)) + LIFT(6) * LIFT(DELTA)
                  - LIFT(1/10) * TICIntegral(temp`tmp_out))) AND
    continuous(temp`tmp_out)};
```

In subsystem *controller*, the heater is on ($heater\_out = 1$) when the temperature *tmp_in* is not higher than the minimum value 20; and the heater is off ($heater\_out = 0$) when the temperature is not lower than the maximum value 40.

Controller

$tmp\_in : \mathbb{T} \nrightarrow \mathbb{R};\ heater\_out : \mathbb{T} \to \{0, 1\}$

$[\![tmp\_in \leq 20]\!] \subseteq [\![heater\_out = 1]\!] \wedge [\![tmp\_in \geq 40]\!] \subseteq [\![heater\_out = 0]\!]$

```
Controller: TYPE = { temp: [# tmp_in: Trace, heater_out: BTrace #] |
    subset?(AllS(LIFT(temp`tmp_in) <= LIFT(20)),
            AllS(LIFT(temp`heater_out) = LIFT(1))) AND
    subset?(AllS(LIFT(temp`tmp_in) >= LIFT(40)),
            AllS(LIFT(temp`heater_out) = LIFT(0))) AND
    continuous(temp`tmp_in)};
```

The connections between two subsystems *plant* and *controller* and the initial condition of the control system are specified in the following TIC schema *System*; the heater is off and the temperature equals 30 at the starting time point 0.

System

$con : Controller;\ pla : Plant$

$\mathbb{I} = [\![con.tmp\_in = pla.tmp\_out]\!] \wedge \mathbb{I} = [\![pla.heater\_in = con.heater\_out]\!]$

$[\![\alpha = 0]\!] \subseteq [\![pla.tmp\_out(\alpha) = 30 \wedge con.heater\_out(\alpha) = 0]\!]$

```
System: TYPE = { temp: [# con: Controller, pla: Plant #] |
    fullset = AllS(LIFT(temp`con`tmp_in) = LIFT(temp`pla`tmp_out)) AND
    fullset = AllS(LIFT(temp`pla`heater_in) = LIFT(temp`con`heater_out)) AND
    subset?(AllS(LIFT(ALPHA) = LIFT(0)),
            AllS((LIFT(temp`pla`tmp_out) o LIFT(ALPHA)) = LIFT(30) AND
                 (LIFT(temp`con`heater_out) o LIFT(ALPHA)) = LIFT(0)))};
```

Based on the above specifications of system properties, we can specify a safety requirement that the temperature is always within a valid range from the value 20 to the value 40 in all nonempty intervals.

$$Safety == \forall\, s : System \bullet \mathbb{I} = [\![s.pla.tmp\_out \leq 40 \land s.pla.tmp\_out \geq 20]\!]$$

```
Safety: THEOREM forall (s: System): fullset =
    AllS(LIFT(s`pla`tmp_out) <= LIFT(40) and LIFT(s`pla`tmp_out) >= LIFT(20));
```

Using TIC, we can specify important timing requirements. As shown below, the requirement *Length* says that given an interval starting from the time point 0 (namely, $\alpha = 0$), the accumulation of the lengths of the intervals in which the heater is on is always less than three-fourths of the length of the interval. We remark that this requirement is not supported in the original [98, 128] as they lack the capacity to model continuous behavior.

$$Length == \forall\, s : System \bullet [\![\alpha = 0]\!] \subseteq [\![\int s.con.heater\_out \leq \tfrac{3}{4} * \delta]\!]$$

```
Length: THEOREM forall (s: System): subset?( AllS(LIFT(ALPHA) = LIFT(0)),
    AllS(TICIntegral(s`con`heater_out) <= LIFT(3/4) * LIFT(DELTA)));
```

The above way which represents TIC schemas as PVS record types supports the modeling technique of Z, namely, using schemas as types to specify large scale systems. For example, the record accessor `con` in the record `System` denotes the subsystem *controller*. This approach is different from that of Gravell and Pratten [54] who discussed issues on embedding Z into both PVS and HOL [53]. They interpreted Z schemas as Boolean-valued functions of records, and it is thus difficult to cope with the case where schemas are declared as types. On the other hand, Stringer-Calvert et al. [124], who applied PVS to prove Z refinements for a compiler development, excluded the support of schemas, as their work focused on modeling Z partial functions in PVS.

4.3.2 Proofs of Requirements

In the following, we present the proofs of above two requirements. The proposed general proof procedure (in Section 4.2) eases the analysis of the first requirement, and the capability of supporting arithmetic reasoning and induction technique of our verification system decreases the proof complexity of the second requirement.

Proving the Requirement *Safety*

This requirement is concerned with the validity for *all* intervals. Instead of checking the temperature for all intervals, we use the *proof by contradiction* method. Namely, we show that there exists *no* interval in which the temperature is outside the valid range. By applying the reasoning rule `EmpCC_to_All` defined in Chapter 3.2, we further reduce the proof complexity by checking the existence for just one type of intervals, specifically, *both-closed* intervals.

The proof is divided into two sub-proofs which check whether the temperature is greater (or lower) than the maximum (or minimum). Each sub-proof relies on a lemma that depicts the continuous behavior of the temperature respectively. For example, the lemma *Decreasing* specifies here that the temperature at the ending point of a *both-closed* interval is not higher than that at the starting point of the interval provided the temperature is not lower than the value 40 in the interval.

$$Decreasing == \forall\, s : System \bullet \\ [s.pla.tmp\_out \geq 40] \subseteq [s.pla.tmp\_out(\omega) \leq s.pla.tmp\_out(\alpha)]$$

Decreasing can be systematically proved by the proposed general proof procedure (in Section 4.2). Its reasoning process and proof commands used are listed below.

```
 1:  ((skosimp)
 2:   (expandsubset)
 3:   (typepred "s!1")(typepred "s!1‘con")
 4:   (assignsubset -1)
 5:   (("1" (typepred "s!1‘pla")
 6:     (assignsubset -1)
 7:     (("1" (lemma "Integral_ge_0")
 8:       (inst - "ALPHA(x!1)" "OMEGA(x!1)" "s!1‘pla‘tmp_out") (ground)
 9:       (("1" (grind)) ("2" (grind))
10:        ("3" (use "cont_Integrable?")(grind))
11:        ("4" (skosimp) (grind))))
12:      ("2" (rewrite "Invariant_True_R")
13:        (expintervaltotime 1) (grind))))
14:    ("2" (rewrite "Invariant_True_R")(grind))))
```

- At lines 3 and 5, we add new antecedents, which model properties of the whole system `s!1` and its subsystems *controller* `s!1‘con` and `plant s!1‘pla`, to the proof sequent by applying `typepred` to relevant names. For instance, `(typepred "s!1")` inserts the connections between `s!1‘con` and `s!1‘pla`.

- The proof strategies defined in Chapter 3.3 simplify the reasoning process (at lines 2, 4, 6, and 13). For instance, the application `(assignsubset -1)` at line 4 directs the PVS prover to automatically instantiate an interval to the *first* antecedent of the sequent.

- As the temperature changes continuously, we exploit specialized lemmas from the NASA PVS library to cope with advanced analysis (at lines 7 and 10). In particular, the lemma `Integral_ge_0` (at line 7) represents a mathematical law that the integral of an integrable function over a *both-closed* interval is nonnegative if this function has nonnegative values throughout the interval.

- At each branch of the proof, we invoke the PVS proof command `grind` to automatically discharge the sub-proof (at lines 9, 10, 11, 13 and 14).

In the above proof, all instantiations of intervals and time points are automatic. Nevertheless the assignment at line 8 is manual for using the mathematical law of integral calculus. This is acceptable, since human heuristics are helpful here to increase the efficiency by selecting proper argument values. We remark that for this manual assignment, the PVS prover checks the correctness of user-specified values (so, `OMEGA(x!1)` must be greater than or equal to `ALPHA(x!1)`).

Proving the Requirement *Length*

This requirement considers all intervals starting with the time point 0, namely, $[\![\alpha = 0]\!]$. The number of these intervals is *infinite*. We apply the *proof by induction* method to cope with the analysis for any arbitrary interval, based on a common assumption of discrete-valued functions such as a discrete timed trace. Specifically, a discrete timed trace is usually assumed to hold the *finite variability* property [137], which that the discrete timed trace changes *finitely* many times in a bounded interval. In other words, we assume that an interval can be classified into one of the following groups with respect to a discrete timed trace: (1) the timed trace is constant throughout the interval, or (2) the interval can be decomposed into a sequence of connected sub-intervals where the timed trace has different values in adjacent sub-intervals. This property is formalized in PVS as presented in Chapter 5.1.1.

The discrete timed trace in subsystem *controller* is the heater status *heater_out*. According to the finite variability, any interval of $[\![\alpha = 0]\!]$ can be divided into a sequence of connected subintervals, where the heater is off in the *first* subinterval (as the initial heater status is specified in the schema *System*) and the heater is either *off* or *on* in the *last* subinterval. Moreover, for intervals during which the heater is off, the integration of the heater status is 0, namely, $\int heater\_out = 0$. We can thus focus on checking the requirement over a special subset of $[\![\alpha = 0]\!]$: the heater is on in

the last subinterval. These special intervals can be formed by the following function *superCon*, which takes three parameters: k is a natural number acting as a counter, and *base* and *unit* are sets of intervals. The function returns a set of intervals by repeatedly concatenating *unit* to *base* k times .

$$
\begin{array}{|l}
superCon : \mathbb{N} \times \mathbb{P}\,\mathbb{I} \times \mathbb{P}\,\mathbb{I} \rightarrow \mathbb{P}\,\mathbb{I} \\
\hline
\forall\, k : \mathbb{N};\ base, unit : \mathbb{P}\,\mathbb{I} \bullet superCon(k, base, unit) = \\
\qquad \text{if } k = 0 \text{ then } base \text{ else } superCon(k-1, base, unit) \frown unit
\end{array}
$$

Those special intervals above can hence be easily captured by an application of *superCon*, namely, $superCon(k, base, unit)$, where the values of *base* and *unit* are given below. Both *base* and *unit* model the sequential behavior of the heater status, specifically, from the off state ($s.con.heater\_out = 0$) to the on state ($s.con.heater\_out = 1$). In addition, *base* captures the *first* change of the heater status, namely, the heater becoming on from its initial off state (as indicated by $\alpha = 0$).

$$
\begin{array}{l}
base = [\![\alpha = 0 \wedge s.con.heater\_out = 0]\!] \frown [\![s.con.heater\_out = 1]\!]; \\
unit = [\![s.con.heater\_out = 0]\!] \frown [\![s.con.heater\_out = 1]\!]
\end{array}
$$

For the sake of simplicity, we take the following lemma *end_with_On* as an example to show how inductive proofs can be easily conducted in our system. The lemma states that the heater is always *on* at the end of any above special interval.

$$
\begin{array}{l}
end\_with\_On == \forall\, s : System \bullet \forall\, k : \mathbb{N} \bullet \\
\qquad superCon(k, base, unit) \subseteq [\![\text{true}]\!] \frown [\![s.con.heater\_out = 1]\!]
\end{array}
$$

The reasoning process of *end_with_On* and proof commands used are given below.

```
1:  ((skosimp)
2:   (induct "k")
3:   (("1" (expandsubset) (expand "superCon") (expandconcat -1)
4:     (assignconcat 1 "i1!1" "i2!1") (grind))
5:    ("2" (skosimp) (expandsubset) (expand "superCon")
6:     (expandconcat -2 1) (expandconcat -1)
7:     (assignconcat 1 "TwoTOneIvl(i1!1, i1!2)" "i2!2") (grind))))
```

Because *superCon* is defined recursively, proof by induction is applicable. We invoke the PVS proof command `induct` at line 2, namely, `(induct "k")`, to make the PVS prover employ its induction scheme to variable `k` whose type is natural numbers. Two sub-proof goals are thus automatically generated. One is the base case where $k = 0$ (at line 3), and the other corresponds to the inductive case (at line 5).

The proof of the inductive case involves concatenation over three sets of intervals, as indicated by entering the proof strategy `expandconcat` (at line 6) twice which expands the function `concat`. If the proof is carried out by hand, we have to examine sixteen situations one by one with respect to interval types from three sets. In contrast, this tedious work is automatically accomplished in our verification system after specifying appropriate interval values at line 7 (where function `TwoTOneIvl` constructs a new interval from two adjacent intervals).

4.3.3 Experimental Results

We summarize our experimental results in Table 4.1, which lists lemma names associated with the steps of proof commands entered and the execution time (in Seconds) of the PVS prover. The experiments are conducted on the SunOS 5.10 platform with PVS version 3.2. In total, we have proved twelve lemmas, and some of these lemmas have been explained, namely, *Decreasing*, *Safety*, *end_with_On* and *Length*. We briefly explain the rest lemmas below, while their PVS specifications and complete proof scripts are available online [27].

- *Increasing* describes the situation where the temperature increases in the interval, with temperature not greater than the minimum value 20.
- *Safe1* claims that the temperature is always lower then or equal to the maximum

Lemma Name	Steps	Time (*Second*)	Lemma Name	Steps	Time (*Second*)
Decreasing	21	14.45	Off_On_Off	44	20.92
Increasing	24	16.66	end_with_On	14	21.88
Safe1	28	14.15	invariant	79	458.8
Safe2	27	14.68	super_and_BT	23	10.59
Safety	7	12.88	Length_Cover	98	562.32
On_Off_On	46	22.47	Length	26	59.67

Table 4.1: Validation results of the temperature control system

value 40, and *Safe2* claims that the temperature is always higher than or equal to the minimum value 20.

- *On_Off_On* (*Off_On_Off*) checks the length bound of the interval (1) in which the heater is off (on) and (2) which is between two consecutive intervals during which the heater is on (off).

- The lemma *invariant* shows that the special intervals highlighted in the previous subsection (using the function *superCon*) satisfy the requirement *Length*.

- The lemma *super_and_BT* indicates that any interval, which starts with the time point 0 and can be decomposed into k parts of subintervals according to the finite variability, can also be formed by executing *superCon* for k times.

- *Length_Cover* shows that any interval starting with 0 can be constructed in one of the three ways based on *superCon*.

During the proofs, mathematical reasoning, in particular the theories of integral calculus, is frequently involved (for lemmas *Decreasing*, *Increasing*, *On_Off_On* and *Off_On_Off*), since the temperature changes continuously in the control system. Moreover, the induction method plays an important role in tackling the analysis over infinite intervals (for lemmas *end_with_On*, *invariant* and *super_and_BT*). Last but not least, our developed supplementary rules for domain-specific features can

make proofs easier. For instance, the supplementary rule `mid_ivl_exi` defined in Chapter 3.3 has been applied to verify lemmas *Safe1* and *Safe2*.

4.4 Summary

Machine-assisted proof support for expressive specification languages usually facilitates complex and tedious proofs by offering certain degree of automation and at the same time assuring the correctness of proof processes. The translation from TIC models to PVS specifications is automatic in our verification system. A general proof procedure has also been proposed to effectively conduct TIC proofs with a high degree of automation, in particular, by exploiting the PVS automatic reasoning power for linear arithmetic, sets and propositional logic. Moreover, the application to the temperature control system has demonstrated the capabilities of our verification system, especially in supporting advanced mathematical analysis such as integral calculus, and in analyzing arbitrary infinite intervals.

Chapter 5

Supporting DC in the Verification System

Up to now, we have developed a verification system based on PVS to facilitate TIC proofs. With the support of TIC which is highly expressive, the verification system is generic enough to be applied to support other interval-based specification languages. As introduced in Chapter 2.3, DC [137] is another popular real-time specification language based on the interval temporal logic [94] and integral calculus. We will show in this chapter the applicability of the verification system by extending it to handle basic DC [139] which is the core of other DC extensions [140, 141].

Firstly, DC constructs are elaborately modeled in TIC. Next, DC axioms and reasoning rules are formalized based on the encoding, and they can be in turn validated using our verification system. Last but not least, we apply the resulting system to a typical case study of DC, and present the improper proof step of the original DC proof which is discovered in our experiment.

5.1 Modeling DC Semantics in TIC

In this section, we demonstrate the way to model DC semantics including special DC constructs using TIC, and in turn using PVS as well. We also discuss how to resolve the differences between TIC and DC. The demonstration follows a bottom-up path, from basic DC constructs to complex ones. Corresponding PVS specifications are provided when they are needed.

5.1.1 State Variables

State variables are functions from time to Boolean values, namely, $\{0, 1\}$ where 0 means false and 1 means true. Each state variable is integrable in all intervals. We thus represent a state variable as an integral function whose range consists of values 0 and 1, that is, $\mathbb{T} \xrightarrow{\square} \{0, 1\}$ in TIC, where the symbol $\xrightarrow{\square}$ captures the integrability of a declared function [49]. We further define below a PVS type named `DCState` to denote the type of state variables, where the function `Integrable?` maps $\xrightarrow{\square}$ and the PVS type `BTrace` as defined in Chapter 3.1.2 indicates that a timed trace `bt` is Boolean-valued.

```
DCState: TYPE = {bt: BTrace | forall (a, b: Time): Integrable?(a, b, bt)};
```

It is usually practical to assume that a state variable holds the *finite variability* property. This property is crucial for analyzing arbitrary (infinite) intervals by means of induction (as it has been exploited to check the requirement *Length* in Chapter 4.3.2). To be specific, this property allows a non-empty interval to be decomposed into a *finite* sequence of subintervals, where a state variable is constant in each subinterval. We provide the PVS specifications which model the property, followed by explanations. Our modeling is similar to the work of Skakkebæk [119] who also formalized

the finite variability property in a recursive manner.

```
k: var posnat; dcs1: var DCState; i: var II;
Cnst(dcs1)(i): bool = ALPHA(i) < OMEGA(i) and exists (x: real):
   forall (t: Time): t < OMEGA(i) and t > ALPHA(i) => dcs1(t) = x;
fvl(k)(dcs1)(i): RECURSIVE bool = IF k = 1 THEN Cnst(dcs1)(i)
   ELSE exists (m: Time): m < OMEGA(i) and m > ALPHA(i) and
     Cnst(dcs1)((OO, (ALPHA(i), m))) and fvl(k - 1)(dcs1)((OO, (m, OMEGA(i))))
   ENDIF MEASURE k;
fvr(k)(dcs1)(i): RECURSIVE bool = IF k = 1 THEN Cnst(dcs1)(i)
   ELSE exists (m: Time): m < OMEGA(i) and m > ALPHA(i) and
     fvr(k - 1)(dcs1)((OO, (ALPHA(i), m))) and Cnst(dcs1)((OO, (m, OMEGA(i))))
   ENDIF MEASURE k;
fv(k)(dcs1)(i): bool = fvl(k)(dcs1)(i) and fvr(k)(dcs1)(i);
DCState_is_FV: AXIOM forall i: ALPHA(i) = OMEGA(i) or exists k: fv(k)(dcs1)(i);
```

- `Cnst` is a function to check whether a state variable `dcs1` is equal to a constant `x` at all time points in a non-empty interval. Note that the values of a state variable at interval endpoints are ignored in DC. That is to say, we can investigate only the values within *both-open* intervals in TIC.

- Function `fvl` *recursively* decomposes an interval into a sequence of *both-open* subintervals where two successive subintervals share one endpoint. At a step of the decomposition, an interval `i` as a parameter is divided to two *both-open* intervals, where a state variable `dcs1` is constant in the first interval `(OO, (ALPHA(i),m))` and `fvl` continuously divides the second interval `(OO, (m, OMEGA(i)))`. The decomposition stops when variable `k` is equal to the value 1. The variable is a positive natural number and acts as a counter by using the PVS *measure function* [104], namely, `MEASURE k`, to measure and terminate the decomposition. In other words, `fvl` unfolds an interval from the left-hand end with respect to a state variable. Similarly, we can define function `fvr` to recursively unfold an interval from the right-hand end.

Both `fvl` and `fvr` play an important role to enable the application of the *proof by induction* method. For instance, we can guide the PVS prover to invoke its induction scheme to the variable `k` by entering the proof command `(induct "k")`. We remark that the construction of the constituent subintervals in both `fvl` and `fvr` excludes interval endpoints (as indicated by the *both-open* interval type). This is acceptable because of the irrelevance of interval endpoints when interpreting DC state variables.

- `DCState_is_FV` is a PVS axiom declaration. It describes that for a state variable, any interval can be categorized to two cases: either the interval is a pointer, or there exists a positive natural number `k` such that functions `fvl` and `fvr` hold with respect to the state variable and the interval.

5.1.2 State Expressions

State expressions are formed by applying the propositional logical operators, such as negation ($\neg$), conjunction ($\wedge$) and disjunction ($\vee$), to connect state variables. Semantically, state expressions are interpreted to be functions from time to the Boolean values. We describe here how to convert three primitive logical operations in state expressions to arithmetic operations in terms of time.

Let $S1$ and $S2$ be state expressions, and t be a time point.

- $(\neg\, S1)(t) = 1 - S1(t)$;
- $(S1 \wedge S2)(t) = S1(t) * S2(t)$;
- $(S1 \vee S2)(t) = 1 - (1 - S1(t) * (1 - S2(t)) = S1(t) + S2(t) - S1(t) * S2(t)$, since in propositional logic, we have $(S1 \vee S2)(t) \Leftrightarrow (\neg\, (\neg\, S1 \wedge \neg\, S2))(t)$.

5.1.3 Temporal Variables

Temporal variables in DC are functions from intervals to real numbers. They can be directly represented as the interval operators of TIC. There are two predefined temporal variables in DC. One is ℓ which represents the length of an interval, and the other is $\int$ for accumulating state variables throughout an interval. ℓ is equal to the interval operator δ in terms of functionality. $\int$ is also supported in TIC [49]. The PVS specifications for δ and $\int$ of TIC have been described in Chapters 3.1.2 and 3.1.3, respectively.

5.1.4 Formulas

DC Formulas are evaluated with respect to intervals only. In other words, their semantics is independent of interval endpoints. Hence, we model them to be a subset of predicates of TIC. Note that predicates in TIC usually rely on both time points and intervals. As shown in the following PVS specifications, a DC formula is a special predicate `p` which satisfies function `DCFormula?`. `DCFormula?` returns true provided `p` has the same value for any arbitrary two time points (`t1` and `t2`) which are respectively inside two intervals (`i1` and `i2`) whose endpoints are identical. In addition, there is no constraint on the interval types.

```
DCFormula?(p: TPred): bool =                          % i1, i2: II; t1, t2: Time
   forall i1, i2: ALPHA(i1)= ALPHA(i2) and OMEGA(i1) = OMEGA(i2) =>
      forall t1, t2: t_in_i(t1, i1) and t_in_i(t2, i2) => p(t1, i1) = p(t2, i2);
DCFormula: TYPE = {p: TPred | DCFormula?(p)};
```

A special operator in DC is the chop operator $\frown$ which is used to specify that two formulas hold respectively in two successive intervals, which overlap at one time point. Note that only *both-closed* intervals are considered in DC. The chop operator may look

like the concatenation operator $\curvearrowright$ in TIC, since both can model sequential behavior at the interval level. However, they are different: firstly, $\frown$ links DC formulas which are represented as predicates of TIC, while $\curvearrowright$ concatenates sets of intervals; furthermore, there are no overlap and no gap between two connected intervals in TIC. We thus cannot simply replace $\frown$ by $\curvearrowright$. For example, a point interval i, namely, $\alpha(i) = \omega(i)$, may satisfy the DC formula $\phi \frown \psi$ where ϕ and ψ are DC formulas, although it is impossible for a point interval to be the result of a concatenation operation in TIC since a point interval cannot be divided into two non-overlapped non-empty intervals. We define a function in TIC as shown below to represent $\frown$, where $\mathbb{I} \to \mathbb{B}$ denotes the type of DC formulas and $\mathbb{B}$ (where $\mathbb{B} ::= \text{true} \mid \text{false}$) denotes the Boolean type.

$$\_ \frown \_ : (\mathbb{I} \to \mathbb{B}) \times (\mathbb{I} \to \mathbb{B}) \to (\mathbb{I} \to \mathbb{B})$$

$$\forall \phi, \psi : \mathbb{I} \to \mathbb{B};\ i : \mathbb{I} \bullet (\phi \frown \psi)(i) \Leftrightarrow$$
$$\exists i1, i2 : \mathbb{I} \bullet \alpha(i) = \alpha(i1) \wedge \omega(i) = \omega(i2) \wedge \omega(i1) = \alpha(i2) \wedge \phi(i1) \wedge \psi(i2)$$

The above definition takes two predicates of TIC (ϕ and ψ) as arguments, and returns a predicate of TIC ($\phi \frown \psi$) which holds in an interval i if and only if there are two subintervals $i1$ and $i2$ such that ϕ and ψ are true in respective subintervals. Furthermore, $i1$ and $i2$ share one endpoint, and their other endpoints are equal to the endpoints of i. Note that we are concerned with only the interval endpoints without constraining the type of $i1$ and $i2$. The corresponding PVS specification of the TIC function is given below.

```
dcf1, dcf2: var DCFormula;
DCChop(dcf1, dcf2)(t, i): bool = exists (i1, i2: II):
    ALPHA(i) = ALPHA(i1) and OMEGA(i) = OMEGA(i2) and OMEGA(i1) = ALPHA(i2) and
    Everywhere?(dcf1, i1) and Everywhere?(dcf2, i2);
```

Based on the encoding of $\frown$, we can specify two frequently used DC operators, $\Diamond$ (eventually) and $\Box$ (always), in TIC and in PVS, by following the DC syntax construction style introduced in Chapter 2.3, namely, $\Diamond\phi == (\mathit{true} \frown \phi) \frown \mathit{true}$ and

$\Box\phi == \neg \Diamond(\neg \phi)$. The result of each operation is also a DC formula. Function `TTRUE` in the following PVS specifications always return the true value in PVS, regardless of intervals and time points, namely, `TTRUE(t, i): bool = true`.

```
<>(dcf1): DCFormula = DCChop(DCChop(TTRUE, dcf1), TTRUE);
[](dcf1): DCFormula = not(<>(not(dcf1)));
```

We have presented so far the way to formalize DC constructs using TIC and PVS. The differences between DC and TIC have been discussed and solutions have been proposed such as supporting the chop operator and the finite variability of state variables. This encoding serves as a foundation for performing rigorous validation of DC axioms and reasoning rules as well as proofs of DC models in the following sections.

5.2 Validating DC Axioms and Reasoning Rules

DC reasoning rules are derived from axioms of state durations. It is important to guarantee the correctness of these axioms and reasoning rules when developing machine-assisted proof support for DC. Existing work [64, 110, 119] directly assumes the validity of the axioms. Our approach is different from them in that we can formalize axioms based on the elaborate encoding in the previous section, and we can rigorously validate them as well as the reasoning rules using our verification system.

DC axioms and reasoning rules are required to be valid for all *both-closed* intervals. They are modeled as TIC predicates and their correctness can be checked with respect to intervals. We remark that intervals in TIC are classified to four basic types according to the inclusion/exclusion of interval endpoints. That is to say, we validate DC axioms and reasoning rules for more than just *both-closed* intervals. Nevertheless, involving more interval types has no effect on the validation result, as we preserve the

independency of DC formulas on interval endpoints and in turn interval types in the corresponding TIC predicates.

Mathematical analysis, especially mathematical laws of integral calculus, is important during the validation. Reusing axiom **DCA5** from Chapter 2.3 as an example, the axiom captures a relation between state durations and the chop operator. We formalize the axiom in the following TIC predicate and PVS specifications, where the symbol S represents a state expression and variables x and y represent non-negative real numbers. Note that we model the axiom for all non-empty intervals (indicated by $[\![\ ]\!]$ and `AllS` respectively).

$$DCA5 == \mathbb{I} = [\![(\int S = x) \frown (\int S = y) \Rightarrow \int S = x + y]\!]$$

```
S: var DCState; x, y: var nnreal;
DC_DCA5: LEMMA fullset =
    AllS(DCChop(TICIntegral(S) = LIFT(x), TICIntegral(S) = LIFT(y))
         => TICIntegral(S) = LIFT(x + y));
```

The validation of $DCA5$ involves mathematical analysis. For instance, one proof sequent in the reasoning process is shown below. The first two antecedents indexed by `-1` and `-2` depict that the integral of a state expression `S!1` on two intervals `il!1` and `ir!1` are respectively equal to non-negative real numbers `x!1` and `y!1`. Intervals `il!1` and `ir!1` are automatically generated from the interval `x!2` based on the chop definition, and the constraints over their endpoints are specified by the antecedents indexed by -3, -4 and -5. The proof goal indexed by `1` is to prove the integral of `S!1` on `x!2` is equal to the sum of `x!1` and `y!1`.

```
{-1} Integral(ALPHA(ir!1), OMEGA(ir!1), S!1) = y!1
{-2} Integral(ALPHA(il!1), OMEGA(il!1), S!1) = x!1
[-3] ALPHA(x!2) = ALPHA(il!1)
[-4] OMEGA(x!2) = OMEGA(ir!1)
[-5] OMEGA(il!1) = ALPHA(ir!1)
|-------
{1}  Integral(ALPHA(x!2), OMEGA(x!2), S1!1) = x!1 + y!1
```

To accomplish the above proof goal, we need to invoke the lemma `Integral_split` by the proof command `lemma`. `Integral_split` captures the *additivity of integration on intervals*, and its contents are also displayed below.

```
Rule? (lemma "Integral_split")
this simplifies to:
{-1} FORALL(a, b, c: Time, f: [Time -> real]):
       Integrable?(a, b,f) AND Integrable?(b, c, f)
       => Integrable?(a, c, f) AND
          Integral(a, b, f) + Integral(b, c, f) = Integral(a, c, f)
...
```

Besides the mathematical analysis of continuous dynamics, induction is another common method to handle complex DC formulas. For example, reasoning rule **DC15** from the DC book [137] describes that if the duration of a state expression S is positive on an interval, then the interval can be chopped into *three* subintervals: the duration of S is zero in the first interval, and S is true almost everywhere in the second interval. The reasoning rule is presented below, where $[\![S]\!]$ is short for $\int S = \ell \wedge \ell > 0$ in DC.

DC15 $\int S > 0 \Rightarrow (\int S = 0) \frown [\![S]\!] \frown \text{true}$

The validation of the above rule is sketched as follows, and the formal reasoning process which costs more than 50 proof commands can be found in Appendix B. Firstly, based on the axiom `DCState_is_FV` (defined in Section 5.1.1), we can deduce that an interval in which the duration of S is positive (namely, $\int S > 0$) can be

divided into k subintervals and S is constant throughout each subinterval. Next, we apply the induction scheme available in PVS to k, and this results in two cases.

- The base case is $k = 1$, which indicates that S is constant over the whole interval. Because the range a state expression is Boolean-valued, S is equal to either the value 0 or the value 1. Moreover, the hypothesis $\int S > 0$ restricts that the constant value can only be 1. Therefore, the rule is valid as we can form the first and third intervals as point intervals, where duration on point intervals equals the value 0 and the predicate "true" also holds in point intervals.

- The inductive case assumes that the rule is valid when $k = n$ where n is an arbitrary positive natural number. And we need to prove that the rule is still valid when $k = n + 1$. We apply the finite variability property formalized in Section 5.1.1, particularly, the function `fvr` which unfolds an interval from the *right-hand* end. To be specific, when we expand the application `fvr(n + 1)` for an interval, the interval is decomposed into two successive *both-open* subintervals where S is constant over the second subinterval. It is easy to prove that the rule holds in the first subinterval by the inductive hypothesis. It is not hard to see that the second subinterval has little influence on the validity of the rule, as the predicate "true" also holds over the second subinterval.

Therefore, the reasoning rule is proved by induction.

Currently, we have formalized and checked all DC axioms and the reasoning rules which are used in our example studies as illustrated in the following section.

5.3 Handling DC Proofs

So far, we have encoded the DC constructs modeled in TIC and in PVS to the verification system, formalized DC axioms and reasoning rules based on the encoding, and validated them using the resulting verification system. We can thus handle DC proofs in a manner which closely follows manual DC arguments. In this section, we demonstrate the usability of our approach by a DC case study, a gas burner [137].

A gas burner is a software-embedded system in a safety-critical context. Let *Leak* be a state variable modeling the critical behavior, namely, $Leak : \mathbb{T} \rightarrow \{0, 1\}$, where the value 1 means that gas is leaking and the value 0 means no leaking. There are two design properties for the gas burner system. The first property (also mentioned in Chapter 2.3) is that any leak should last for not longer than 1 time unit. The second property is that the interval length between two consecutive leaks must be at least 30 time units. Both properties are modeled in DC below.

$$Des1 == \Box(\lceil Leak \rceil \Rightarrow \ell \leq 1)$$
$$Des2 == \Box(\lceil Leak \rceil \frown \lceil \neg\, Leak \rceil \frown \lceil Leak \rceil \Rightarrow \ell \geq 30)$$

A real-time requirement is that the proportion of the leaking time in an interval is always less than or equal to one-twentieth of the interval, provided the interval lasts at least 60 time units. This requirement is expressed below in DC based on *Leak*.

$$GbReq == \ell \geq 60 \Rightarrow 20 * \int Leak \leq \ell$$

We remark that requirement *GbReq* here is different from the requirement *Length* of the temperature control system as presented in Chapter 4.3. *Length* is concerned with the intervals which start from the time point 0 ($\alpha = 0$) . However, the intervals which are considered by *GbReq* are restricted by their lengths ($\ell \geq 60$).

The above properties and requirement are DC formulas. They are modeled by the following PVS specifications where function `pq` represents the abbreviation $[\![\ ]\!]$ and other PVS symbols for DC constructs are defined in Section 5.1.

```
Leak: DCState;
Des1: DCFormula = [](pq(Leak) => LIFT(DELTA) < LIFT(1));
Des2: DCFormula = [](DCChop(DCChop(pq(Leak), pq(not(Leak))), pq(Leak))
                      => LIFT(DELTA) >= LIFT(30));
GbReq: DCFormula = LIFT(DELTA) >= LIFT(60)
                   => LIFT(20) * TICIntegral(Leak) <= LIFT(DELTA);
```

To prove the correctness of the design properties, we need to show that the formula $Des1 \wedge Des2 \Rightarrow GbReq$ is *valid*. That is, the formula is true in all intervals. Hence, our proof goal is $\mathbb{I} = [\![Des1 \wedge Des2 \Rightarrow GbReq]\!]$, which is equivalently converted to the following goal $[\![Des1]\!] \cap [\![Des2]\!] \subseteq [\![GbReq]\!]$. That is to say, an interval in which both design properties hold is also the interval in which the requirement is true. This proof goal is expressed below by a PVS theorem named `ProofGoal`.

```
ProofGoal: theorem subset?(intersection(AllS(Des1), AllS(Des2)), AllS(GbReq));
```

Using our verification system, the proof process for the above goal can comply strictly with the original process [137] in terms of the order of applying DC reasoning rules and lemmas. These rules and lemmas are also validated. In the following, we informally describe important steps in the process associated with a simplified proof script.

```
("1"... (lemma "Math_PL1") ...
 (("1" ... (lemma "Lemma3_5") ...
   (("1" ... (lemma "Lemma3_6") ...
     (("1" ... (lemma "DC_DC8") ...
```

1. Lemma `Math_PL1` specifies the property that an interval i whose length is more than or equal to 60 time units can be partitioned into a sequence of $n + 1$ parts of intervals where n is a natural number. In addition, the length of each

subinterval except the last one is equal to 30 time units, while the length of the last subinterval is less than 30 time units.

2. From both design properties we can deduce that for an interval whose length is less than or equal to 30 time units, the duration of leaking is less than or equal to 1 time unit. This property is modeled by lemma `Lemma3_5`.

3. Based on `Lemma3_5`, we can deduce that gas can be leaking for at most n time units throughout the first n subintervals of size 30, which are produced at Step 1. This result is captured by lemma `Lemma3_6`.

4. From `Lemma3_5` and the second design property $Des2$, we conclude that the leaking duration is less than 1 time unit in the last subinterval which follows the first n subintervals. We can thus apply the DC reasoning rule $DC8$ as shown below.

   DC8 $(\int S \leq x) \frown (\int S \leq y) \Rightarrow \int S \leq x + y$

   This rule is modeled by `DC_DC8` in the simplified proof script, and it is used to sum up the leaking duration of the whole interval i at Step 1.

After executing the above steps, the proof is automatically accomplished.

From the rigorous verification using our system, we have identified an improper deductive step in the original presentation [137]. To be specific, a proof obligation before applying $DC8$ is $(\int Leak \leq n) \frown (\int Leak \leq 1) \Rightarrow (\int Leak \leq n + 1)$, and the original proof adopts the axiom $DCA5$ (mentioned in Chapter 2.3 and Section 5.2). However, it is obvious that $DCA5$ cannot resolve the obligation. Instead, the appropriate reasoning rule is $DC8$, which easily discharges the proof obligation.

The gas burner is a typical DC case study to which our extended verification system has been applied in this section. We have presented the corresponding PVS specifi-

cations of the system design and real-time requirement. The DC reasoning rules and lemmas used have been formalized and checked. Thus the reasoning process in the verification system follows a manner similar to the manual DC arguments. We have also shown the discovery of the incorrect proof step in its original presentation.

5.4 Summary

We have generalized the verification system developed in Chapters 3 and 4 to support other interval-based specification languages, in particular, DC. This enhancement has been accomplished by applying TIC to model DC. We have elaborately encoded DC constructs in TIC, and further formalized and validated the DC axioms as well as reasoning rules. The resulting system enables users to carry out DC proofs along the lines of the hand proof without knowing the detailed encoding. Furthermore, the rigorous verification capability of the system elevates the confidence level of DC proofs, for example, the improper proof step identified in the original manual DC arguments of the gas burner.

Some researchers have investigated machine-assistant proof for DC. Skakkebaek and Shankar [120] developed a proof checker with PVS, and Heilmann [64] applied Isabelle [106] to support mechanized proof. Chakravorty and Pandya [24] digitized a subclass of DC, Interval Duration Calculus, into another subclass for discrete systems. However, in the above works, the duration operator as the key construct of DC is not semantically encoded and properties of it are assumed as axioms. As shown in this chapter, we have encoded the duration operator based on the latest NASA PVS library, and we can hence directly validate those properties regarding DC durations in our verification system.

Chapter 6

Modeling Simulink Library Blocks

As presented in Chapters 3 and 4, we have systematically developed a verification system to support the formal verification of TIC models with a high degree of automation. This machine-assisted proof support can be used to expand the application range of TIC in the development of complex systems. In the following three chapters, we construct a framework based on the verification system to complement Simulink [85] which has been widely used in industry for specifying and simulating dynamic systems.

A Simulink diagram formed by connecting blocks with wires represents a set of mathematical relationships which model system behavior over time. Simulink adopts continuous-time semantics [70] to support dynamic systems such as hybrid control systems. Its simulation facility allows system behavior to be visually observed for specific parameter values over specific simulation periods. However, simulations are deficient in checking system behavior for large parameter values or over infinite simulation periods. In addition, open systems whose exact input functions are usually unknown are unanalyzable in Simulink because simulations are inapplicable to these

systems. Moreover, Simulink lacks timing analysis which becomes necessary due to the increasing usage of embedded systems in real-time safety-critical situations [108].

Recently, formal methods have received more attention to improve the development of the embedded real-time systems by their rigorous semantics and formal verification capability [67, 132]. We here apply TIC to complement Simulink: functional and timing aspects of Simulink diagrams are formally captured in TIC; important (timing) requirements are rigorously validated by well-defined TIC reasoning rules and the strong support of mathematical analysis in TIC.

As introduced in Chapter 2.4, Simulink library blocks are templates to produce elementary blocks of Simulink diagrams. They are classified into various categories: continuous, discrete, mathematical functions and so on. Unfortunately, their semantics is *informally*, and even *partially*, defined in the original documentation [84]. It is necessary and important to formally model these blocks using TIC.

In this chapter, we firstly present the basic structure of TIC schemas denoting elementary blocks. The structure captures the time-dependent relationships of elementary blocks. Next, we construct TIC library functions and highlight their features with examples. Lastly, we discuss the construction of these TIC library functions and their validation. These TIC functions can serve as an accurate and thorough documentation to accompany original descriptions [84] of Simulink library blocks.

6.1 TIC Schemas for Simulink Elementary Blocks

An elementary block denotes a time-dependent mathematical relationship between its inputs and outputs. In Simulink, inputs and outputs always have values at any time point. Namely, they are total functions of time. We consider here a common type

of elementary blocks which is *multiple inputs and single output*. Other types such as multiple inputs and outputs can be handled similarly. Furthermore, each elementary block has its sample time as its execution rate in simulation.

An elementary block is modeled by a TIC schema: essential attributes including the block inputs, output, parameters and sample time are captured in the declaration part, and the block behavior is specified in terms of intervals in the predicate part. As sample times can be either continuous or discrete, we classify TIC schemas of elementary blocks to two groups according to their sample times.

When the sample time of an elementary block is equal to the value 0, the block executes continuously. Its output at a time point relies on inputs either at the same time point (for example, a summation) or through an interval (for example, an integral operation). We specify continuous behavior at a high level, namely, at the interval level, rather than the time point level.

Definition 1 (Continuous basic block) *A TIC schema for a continuous elementary block is a 6-tuple* $(Ins, Out, Ps, st, \mathcal{F}, \mathcal{V})$*, where Ins denotes a set of inputs and each input is a timed trace of the type* $\mathbb{T} \to \mathbb{R}$*, Out is the output which is also a timed trace, Ps denotes a set of block parameters of the type* $\mathbb{R}$*, st is the sample time,* $\mathcal{F}$ *denotes the mathematical relationship which depends on the inputs and block parameters, namely,* $\mathcal{F} : Ins \times Ps \to Out$*, and* $\mathcal{V}$ *is a mapping assigning real numbers to block parameters. These attributes satisfy two constraints which capture continuous behavior of the block: one is* $\mathbb{I} = [\![\mathcal{F}(Ins, Ps) = Out]\!]$ *indicating that* $\mathcal{F}$ *holds for all non-empty intervals; the other is* $st = 0$ *to limit the sample time value.*

When the sample time of an elementary block is positive, the block executes discretely. To be specific, the block changes its output at *sample time hits* which are integer multiples of the sample time, and keeps its output constant between any two

consecutive sample time hits. This discrete behavior is captured by modeling the behavior for each *sample time interval* which is *left-closed and right-open* and whose endpoints are a pair of consecutive sample time hits.

Definition 2 (Discrete basic block) *A TIC schema for a discrete elementary block is also a 6-tuple* $(Ins, Out, Ps, st, \mathcal{F}, \mathcal{V})$*, where the types of attributes are the same as those in Definition 1. However, the constraints that these attributes satisfy are different from Definition 1 as the behavior is discrete here. Specifically,* $st > 0$ *restricts the value of the sample time, and the other constraint specifies the discrete behavior for all sample time intervals which are denoted by* $[\![\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st)\!]$ *where* $\mathbb{N}$ *represents the set of all natural numbers.*

For many discrete elementary blocks in practice, the constraints specifying their behavior can be expressed in the following form: $[\![\exists\, k : \mathbb{N} \bullet \alpha = k{*}st \wedge \omega = (k{+}1){*}st)\!] \subseteq [\![\mathcal{F}(Ins(\alpha), Ps) = Out)\!]$. Namely, the output values during a sample time interval are dependent on the input values at the starting point of the sample time interval. An example is the *Zero-Order Hold* library block as mentioned in Section 6.2.

Above two definitions capture two basic types of TIC schemas in terms of their schema structure. These definitions serve as a guideline to construct TIC library functions in the next section, where the mathematical relationship $\mathcal{F}$ will be explicitly specified with respect to a particular Simulink library block. Note that the range of the above timed traces is real numbers to represent the data type *double* in Simulink. This is acceptable since different data types in Simulink only affect simulation efficiency. Nevertheless, our approach can be extended to support multi-dimensional values. For example, vectors of values can be represented as *sequences* of values in TIC.

6.2 TIC Library Functions for Simulink Library Blocks

In Simulink, an elementary block is generated by assigning particular values to the parameters of a library block. This parameterization technique is also adopted by our TIC library functions which model Simulink library blocks. To be specific, these library functions return TIC schemas which represent elementary blocks.

As we focus on the mathematical relationships denoted by elementary blocks, irrelevant block parameters are ignored, such as parameters used for block appearance. We divide the remainder of block parameters to three groups, namely, *operands*, *sample times*, and *operators*, according to their effect on the mathematical relationships. We describe below how to model the general structure of TIC library functions, which takes into account the first two groups of block parameters. In the end, the way to handle the last group is presented.

- A continuous library block always produces continuous elementary blocks whose sample times are 0, and we thus consider only the operand parameters.

  Definition 3 (Continuous library block) *A TIC library function for a continuous library block takes a set of arguments which denote the operand parameters of the library block and returns a TIC schema which conforms to Definition 1 with respect to its attributes and the constraints over the attributes.*

  For example, the *Integrator* library block is a continuous library block; and its output value at the ending point of an interval is equal to the sum of its output value at the starting point of the interval and the integration of its input over the interval. In addition, the output value at the time point 0 is stored via the

InitialCondition block parameter, which is represented by variable *IniVal* in the following TIC library function.

$$
\begin{array}{|l}
\hline
Integrator : \mathbb{R} \rightarrow \mathbb{P}[In_1 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \mathbb{T} \twoheadrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, init : \mathbb{R} \bullet Integrator(init) = [In_1 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \mathbb{T} \twoheadrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\quad st = 0 \land IniVal = init \land Out(0) = IniVal \land \\
\quad \mathbb{I} = [\![Out(\omega) = Out(\alpha) + \textstyle\int_\alpha^\omega In_1]\!]]
\end{array}
$$

In a schema returned by the function *Integrator*, namely, $Integrator(init)$, the predicate $IniVal = init$ which associates the argument *init* with *IniVal* corresponds to the mapping $\mathcal{V}$ in Definition 1. Moreover, the predicate $\mathbb{I} = [\![Out(\omega) = Out(\alpha) + \int_\alpha^\omega In_1]\!]$ which explicitly specifies the mathematical relationship $\mathcal{F}$ in terms of intervals conforms to the first constraint in Definition 1. Note that we indicate the continuity feature of the block output *Out* by $\twoheadrightarrow$.

- A discrete library block always creates discrete elementary blocks whose sample times are positive, and we consider its sample time and operand parameters.

  Definition 4 (Discrete library block) *A TIC library function for a discrete library block takes a set of arguments denoting the sample time and operand parameters, and returns a TIC schema which conforms to Definition 2 with respect to its attributes and the constraints over the attributes.*

  For instance, the *Zero-Order Hold* library block is a discrete library block. Output values of this library block through a sample time interval are equal to its input value which is sampled at the beginning point of the sample time interval. The sample time is determined from the *SampleTime* block parameter, and it is denoted by variable *st* in the following TIC library function.

$$
\begin{array}{|l}
\hline
ZOH : \mathbb{T} \rightarrow \mathbb{P}[In_1, Out : \mathbb{T} \rightarrow \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet ZOH(t) = [In_1, Out : \mathbb{T} \rightarrow \mathbb{R};\ st : \mathbb{T} \mid st > 0 \land st = t \land \\
\quad [\![\exists\, k : \mathbb{N} \bullet \alpha = k * st \land \omega = (k+1) * st]\!] = [\![Out = In_1(\alpha)]\!]]
\end{array}
$$

In a schema returned by the above function *ZOH*, predicates $st > 0$ and $st = t$ constrain the sample time value. As mentioned in Definition 2, we specify discrete behavior for all sample time interval. Namely, the sample-and-hold behavior of an elementary block created by the *Zero-Order Hold* library block is depicted by the last predicate.

- Other library blocks can generate either continuous or discrete elementary blocks. A TIC library function for such a library block captures both types of behavior by returning different TIC schemas according to the sample time assigned to the library block. Namely, the structure of a returned schema conforms to Definition 1 when the sample time is 0, and conforms to Definition 2 otherwise.

  Taking the *Relational Operator* library block as an example, this library block outputs the value 1 when its first input is larger than or equal to its second input, and the value 0 otherwise. If its sample time is specified to be *continuous*, namely, $st = 0$, its output at any time point relies on its inputs at the same time point. Otherwise, the comparison is executed *discretely*: the output values during a sample time interval are dependent on the inputs at the starting point of the sample time interval. The above relation between the types of behavior and the sample time is represented by two conjunctive implications in the following TIC library function *Relation_geq*.

$$
\begin{array}{|l}
Relation\_geq : \mathbb{T} \rightarrow \mathbb{P}[In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \mathbb{T} \rightarrow \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet \\
\quad (t = 0 \Rightarrow Relation\_geq(t) = [In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \mathbb{T} \rightarrow \{0,1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge (\!| In_1 \geq In_2 |\!) = (\!| Out = 1 |\!) \wedge (\!| In_1 < In_2 |\!) = (\!| Out = 0 |\!)]) \\
\wedge \quad (t > 0 \Rightarrow Relation\_geq(t) = [In_1, In_2 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \mathbb{T} \rightarrow \{0,1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge (\!| \exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st |\!) = \\
\qquad\quad (\!| (In_1(\alpha) \geq In_2(\alpha) \Rightarrow Out = 1) \wedge (In_1(\alpha) < In_2(\alpha) \Rightarrow Out = 0) |\!)])
\end{array}
$$

So far we have described the general structure of TIC library functions which involves the sample times and operand parameters of library blocks. Ideally, we shall

establish a one-to-one mapping from block types to TIC library functions, such as the function *Integrator* for the *Integrator* library block. However, this kind of mapping is inapplicable to operator parameters, because these parameters can cause a library block to produce elementary blocks with different functionalities. We thus construct multiple TIC library functions for one library block which has operator parameters; and each library function captures a particular functionality.

Reusing the *Relational Operator* library block as an instance, its *Relational Operator* block parameter is an operator parameter. The default value of *Relational Operator* is " >= " indicating that generated elementary blocks check whether their first input is not less than their second input. However, when the value is specified as " == ", the comparison is to check if the first input equals the second input. We define below a TIC library function named *Relation_eq* to represent the equivalence comparison for the *Relational Operator* library block.

$$
\begin{array}{|l}
Relation\_eq : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet \\
\quad (t = 0 \Rightarrow Relation\_eq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\quad\ st = 0 \wedge (\!| In_1 = In_2 |\!) = (\!| Out = 1 |\!) \wedge (\!| In_1 \neq In_2 |\!) = (\!| Out = 0 |\!)]) \\
\wedge\ (t > 0 \Rightarrow Relation\_geq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\quad\ t = st \wedge st > 0 \wedge (\!| \exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st |\!) = \\
\qquad (\!| (In_1(\alpha) = In_2(\alpha) \Rightarrow Out = 1) \wedge (In_1(\alpha) \neq In_2(\alpha) \Rightarrow Out = 0) |\!)]) \\
\end{array}
$$

6.3 Discussions and Discoveries

Up to now we have presented the structure of TIC library functions which capture various behaviors of the Simulink library block at the interval level. In this section, we discuss the way of constructing and validating these library functions with some discoveries which have been confirmed by senior application engineers of MathWorks.

Case	Relation	Relay	Case	Relation	Relay
1	$OnV > OffV \wedge InV = OffV$	off	5	$OnV > OffV \wedge InV > OnV$	on
2	$OnV > OffV \wedge OnV > InV > OffV$	off	6	$OnV = OffV \wedge InV = OffV$	on
3	$OnV > OffV \wedge InV = OnV$	on	7	$OnV = OffV \wedge InV < OffV$	off
4	$OnV > OffV \wedge InV < OffV$	off	8	$OnV = OffV \wedge InV > OffV$	on

Table 6.1: The initial relay state of the *Relay* library block in different cases

We aim to capture the time-dependent mathematical relationships denoted by library blocks as their intrinsic semantics. Unfortunately, the original Simulink documentation [84] specifies library blocks in a narrative and sometimes partial manner. This can lead to ambiguous interpretation of library blocks and further obstruct the proper usage of Simulink.

For example, the *Relay* library block switches its output according to its relay status, either *on* or *off*. Its original description states that: when the relay is on, the block remains on until the input drops below the value of the *Switch off point* block parameter; when the relay is off, it remains off until its input exceeds the value of the *Switch on point* block parameter. In addition, the *Switch on point* value must be greater than or equal to the *Switch off point* value. However, the description misses the specification of the initial behavior, namely, the relay status at the time point 0. It is thus necessary to clearly capture this initial behavior to avoid confusion of the initial relay state.

To formally model a library block based on its informal and particularly incomplete description, we investigate the behavior by means of simulations under all possible circumstances where the library block may be used in practice. Specifically, we assign different values to the operand parameters of the library block and feed the inputs with various types, continuous or discrete, to simulate all possible situations in which the block may be applied. Reusing the *Relay* library block as an example, we consider the relationship of two parameters and the relationship between the initial input value

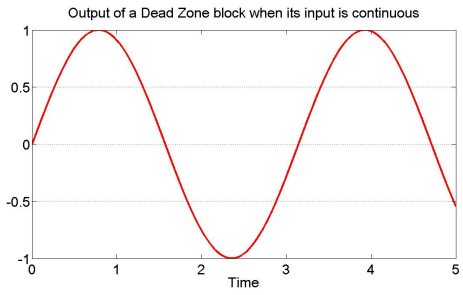


Figure 6.1: An incorrect simulation result of the *Dead Zone* library block

and these two parameters. As shown in Table 6.1, there are eight cases to observe the initial relay state, where OnV denotes the *Switch on point* value, $OffV$ denotes the *Switch off point* value, and InV denotes the input value at the time point 0.

We also exploit simulations to validate TIC library functions. Namely, we check if a TIC library function of a library block conforms to the simulation results of the library block. When an inconsistency occurs, we carefully analyze the original description of the block again and consult with our partners at Simulink, to identify the problem and refine the library function if needed. Not only can this way elevate our confidence of the TIC library functions, it also helps us discover below a bug in a library block.

The output of the *Dead Zone* Library block depends on the relation between its input and a region constrained by a lower limit and an upper limit: (1) the output is zero if the input is within the region, (2) the output is the input minus the upper limit if the input is greater than or equal to the upper limit, (3) the output is the input minus the lower limit if the input is less than or equal to the lower limit.

Unfortunately, the above original definition is inconsistent with the case where the input is continuous and the upper limit equals the lower limit. Figure 6.1 depicts a particular simulation result of an elementary block of the *Dead Zone* library block:

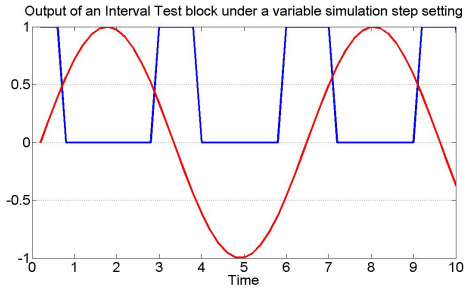


Figure 6.2: A wrong simulation result of the *Interval Test* library block.

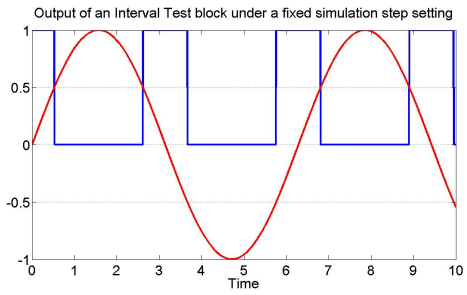


Figure 6.3: A correct simulation result of the *Interval Test* library block.

its output is identical to its input, where both limits are equal to the value 0.5 and the input is a sine wave. However, the input and output should be different according to the definition, as the input is greater than or less than 0.5 most of the time.

We remark that checking the TIC library functions by simulations must be extremely careful, especially when simulation results are different. Because simulations in Simulink can be influenced by simulation settings such as the step size of a simulation, a library block may thus behave correctly in one setting but wrongly in another.

A particular instance is the *Interval Test* library block. The library block outputs the value 1 if its input is within the values of parameters *Lower limit* and *Upper limit*,

and the value 0 otherwise. Figures 6.2 and 6.3 show the respective simulation results of an instantiated library block when *Lower limit* equals the value -0.5 and *Upper limit* equals the value 0.5. The input of the instantiated library block is a continuous wave, and the output is a square line. From Figure 6.2, it is easy to see that the block behaves wrongly when the simulation step size is automatically determined by Simulink as a default setting: the moments when input values are outside the range $[-0.5 \ldots 0.5]$ are imprecisely captured. However, the block behaves correctly in Figure 6.3 when we manually fix the simulation step size to $1e-3$.

Therefore, if incorrect behavior of a library block is caused by simulation settings, we treat that the TIC library function of the library block is still valid. In addition, we record the settings which lead to abnormal behavior. The above way also facilitates the analysis of the case when simulation results are unexpected, and hence increases confidence in system design denoted in Simulink.

6.4 Summary

In this section, we have presented the general structure of TIC library functions which formally model the time-dependent mathematical relationships as the denotational semantics of Simulink library blocks. We have further discussed the way to elaborately construct and validate these library functions, with the illustration of discoveries of incomplete semantics of the *Relay* library block and a bug of the *Dead Zone* library block. These functions can serve as an accurate and thorough documentation to accompany [84] for Simulink library blocks, and form the foundation used for the automatic transformation from Simulink diagrams to TIC models.

Interpreting Simulink diagrams in other formal notations or programming languages usually focuses on discrete behavior [1, 10, 21, 22, 89, 128, 129]. Our first attempt

was to apply Timed Communicating Object-Z (TCOZ) [78, 79, 80, 109] to represent Simulink diagrams. TCOZ blends Object-Z [45] and Timed CSP [116] for designing real-time and concurrent systems with digital components. However, like those previously mentioned formal notations and programming languages, TCOZ faces the same drawback of lacking support of the continuous-time semantics which is adopted by Simulink. On the other hand, TIC is based on the continuous-time domain and supports elementary calculus, such as integral calculus, which is commonly used in control engineering, an important application domain of Simulink. To the best of our knowledge, we are the first to model Simulink diagrams in terms of continuous time. The TIC expressive power enables us to handle a wide range of Simulink library blocks.

Currently, we have modeled 44 Simulink library blocks of 9 categories including *continuous*, *logic operations*, *discrete*, *math operation*, and so on. These block names are given in Appendix C.1, and the corresponding TIC library functions are available online [30]. The main reason for modeling these library blocks is their frequent usage in practice. For example, all 22 library blocks of the *Commonly Used* category defined in [84] are supported, and their specific TIC library functions are given in Appendix C.3. Moreover, these library blocks have been used in 17 Simulink demos which cover the areas of aerospace and automobile systems. Note that library blocks of the *Ports and Subsystems* category are improper to be captured, because their functionalities are usually unpredictable until they are instantiated in specific Simulink diagrams. We will present how to handle these library blocks in Chapter 7.

Chapter 7

Transforming Simulink Diagrams into TIC Schemas

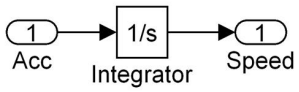
The TIC library functions constructed in the previous chapter model Simulink library blocks which are templates of generating elementary blocks as the basic units of Simulink diagrams. Based on those functions, we develop a strategy here to transform Simulink diagrams into TIC schemas. The transformation starts from elementary blocks and follows a bottom-up order in which a Simulink diagram is constructed. During the transformation, functional and timing aspects of an elementary block are captured, and the hierarchical structure of a diagram and wires between components in the diagram are retained. Moreover, the transformation is automatic as the strategy has been implemented in Java.

In the following, we firstly present the way to transform elementary blocks, wires and diagrams. Next, we describe a simple but effective algorithm for deriving unspecified sample times of elementary blocks in complex diagrams which may contain loop structure. Lastly, we highlight how to cope with the library blocks of the *Ports and*

Subsystems category (note that they are not captured in the previous chapter), especially conditional subsystems such as *Triggered* subsystems and *Enabled* subsystems.

7.1 Transforming Elementary Blocks

A Simulink elementary block is produced by a library block by using the parameterization technique. We reuse the Simulink diagram shown in Figure 2.1 from Chapter 2.4 as a running example in this chapter. The diagram with its simplified textual contents is displayed in Figure 7.1. Note that the elementary block *Integrator* is produced by the *Integrator* library block with value 4 for the block parameter *InitialCondition*.




```
1  System{ Name             "calculation"
2    Block{ BlockType       Inport
3           Name            "Acc"       }
4    Block{ BlockType       Integrator
5           Name            "Integrator"
6           InitialCondition "4"        }
7    Line{ SrcBlock         "Integrator"
8          DstBlock         "Speed"  }
9    Line{ SrcBlock         "Acc"
10         DstBlock         "Integrator" }}
```

Figure 7.1: A system *calculation* in Simulink graphical and textual contents

An elementary block is represented by a TIC schema which is produced by applying relevant parameter values to the TIC library function which models the corresponding library block. During a transformation, two aspects are taken into account.

One is the criteria for selecting an appropriate TIC library function for an elementary block. The primary criterion is the *BlockType* block parameter which indicates the functionality of a block. Nevertheless, this parameter is inadequate to distinguish special library blocks which contain operator parameters and can generate elementary blocks with different functionalities. Thus, operator parameters are also considered to

be additional criteria. Recalling the *Relational Operator* library block in Chapter 6.2, it has an operator parameter *Relational Operator* which determines the functionality of its produced elementary blocks. Hence, the criteria of choosing a TIC library function consist of two block parameters, namely, *BlockType* and *Relational Operator*.

The other aspect is about sample times. A sample time of an elementary block is determined in one of the following ways: by the *SampleTime* block parameter, by the library block type (for example, elementary blocks of a continuous library block always have continuous sample times), or by blocks which connect to the block inputs. In addition, the last way relies on the assumption that all sample times of the blocks are specified. We formalize below the third way based on the rules from [85]. We will also present a simple but effective algorithm in Section 7.4 to deal with the case that the above three ways are inapplicable.

Let *Blk_In* denote the blocks which connect to the inputs of an elementary block, and *InST* be a function of the type $InST : Blk\_In \rightarrow \mathbb{T}$ which associates the blocks with their sample times.

- We firstly check whether sample times of all blocks in *Blk_In* are identical. If so, we assign the identical value to be the sample time of the elementary block. Otherwise, we return the value -1 as modeled by the following function *AllEq* to indicate that the sample time is unspecified.

$$
\begin{array}{|l}
AllEq : \mathbb{P}\, Blk\_In \rightarrow (\mathbb{T} \cup \{-1\}) \\
\hline
\forall\, ins : \mathbb{P}\, Blk\_In \bullet \exists\, res : \mathbb{T} \bullet AllEq(ins) = \\
\quad \mathsf{If}\, \forall\, in : ins \bullet InST(in) = res\ \mathsf{Then}\ res\ \mathsf{Else}\ -1 \\
\end{array}
$$

- Next, we check whether there is a sample time of a block in *Blk_In* which is the greatest common integer divisor (GCD) of the sample times of other blocks in *Blk_In*. If so, we assign the GCD to be the sample time of the elementary

block. Otherwise, we return the value -1 as modeled by the following function *ExiFast* to indicate that the sample time is unspecified.

$$
\begin{array}{|l}
ExiFast : \mathbb{P}\, Blk\_In \rightarrow (\mathbb{T} \cup \{-1\}) \\
\hline
\forall\, ins : \mathbb{P}\, Blk\_In \bullet \exists\, res : \mathbb{T} \bullet ExiFast(ins) = \\
\quad \mathsf{If}\ \exists\, in1 : ins \bullet \forall\, in2 : ins \mid in1 \neq in2 \bullet \\
\qquad \exists\, k : \mathbb{N} \mid k > 1 \bullet InST(in2) = InST(in1) * k \wedge InST(in1) = res \\
\quad \mathsf{Then}\ res\ \mathsf{Else}\ -1
\end{array}
$$

- Lastly, when *AllEq* and *ExiFast* return the value -1, we derive the sample time according to the solver[1] used in that diagram which contains the elementary block. As modeled by the following function *STP*, if a *variable-step* solver is used, the sample time is continuous, namely, equaling the value 0. Otherwise, the sample time is the result of function *CalGCD* which returns the GCD of sample times of *Blk_In* if such a GCD exists or the value 0 otherwise.

$$
\begin{array}{|l}
STP : \mathbb{P}\, Blk\_In \times Solver \rightarrow \mathbb{T} \\
\hline
\forall\, ins : \mathbb{P}\, Blk\_In;\ s : Solver \bullet STP(ins, s) = \\
\quad \mathsf{If}\ AllEq(ins) < 0\ \mathsf{Then} AllEq(ins) \\
\qquad \mathsf{Else}\ (\mathsf{If}\ ExiFast(ins) < 0\ \mathsf{Then}\ ExiFast(ins) \\
\qquad\quad \mathsf{Else}\ (\mathsf{If}\ s = Variable\_Step\ \mathsf{Then}\ 0\ \mathsf{Else}\ CalGCD(ins)))
\end{array}
$$

The above two aspects are important to capture the functional and timing properties of an elementary block. We here use the elementary block *Integrator* in Figure 7.1 as an example: (1) the selection criterion is the *BlockType* block parameter whose value is *Integrator* and hence the TIC library function *Integrator* in Chapter 6.2 is chosen, (2) the sample time is 0 since the type of the *Integrator* library block is continuous. This elementary block is modeled by the following schema *calculation_Integrator* which is

[1]There are two types of solvers for simulations in Simulink: variable-step solvers vary the simulation step size, while fixed-step solvers keep the simulation step size constant. $Solver == \{Variable\_Step, Fixed\_Step\}$.

the result by applying the value 4 as the initial value to the library function. The expanded form of the schema is also given.

$$calculation\_Integrator \mathrel{\widehat=} Integrator(4)$$

calculation_Integrator

$$In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}$$

$$st = 0 \land IniVal = 4 \land Out(0) = IniVal \land \mathcal{I} = [\![Out(\omega) = Out(\alpha) + \int_\alpha^\omega In_1]\!]$$

To preserve the hierarchical structure of a Simulink diagram, a transformed TIC schema is named in a conventional manner which composes the name of the path in the diagram. For example, the schema name *calculation_Integrator* is formed by appending the block name *Integrator* to the system name *calculation* with the symbol "_". This naming manner is also adopted by other approaches [10, 22, 129].

7.2 Transforming Wires

In Simulink, wires represent input and output relations between connected blocks, and the communication is infinitely fast. Namely, a destination block can receive the value which is produced by a source block at the same time point. As Simulink adopts the continuous-time domain, wires have values at all time points. In other words, a source (or destination) block can write (or read) value to (or from) a wire according to its own execution rate, namely, its sample time. This feature hence allows Simulink to support multi-rate discrete systems as well as hybrid systems, and both can contain blocks with different sample times.

Each wire is converted into an equation of two timed traces at the interval level. The equivalent timed traces indicate the interfaces of connected blocks respectively.

Specifically, let *src* denote the output of a source block ($src : \mathbb{T} \rightarrow \mathbb{R}$) and *dst* denote the input of a destination block ($dst : \mathbb{T} \rightarrow \mathbb{R}$), the equivalence of these timed traces holds for all non-empty intervals, namely, $\mathbb{I} = [\![src = dst]\!]$. We remark that this way is also applicable to normal (sub)systems. However, a different way to handle conditionally executed subsystems will be demonstrated in Section 7.5.

7.3 Transforming Diagrams

A Simulink diagram is made up by linking blocks with wires. It is thus necessary to retain the components and connections of a diagram in transformed TIC models. Our method is similar to the one of Arthan et al [10]: the transformation of a diagram is performed after the components of the diagram are transformed into TIC schemas. A diagram is modeled by a TIC schema in the following way.

- Each component is declared as a schema variable. If a component is a block, the type of the variable is the TIC schema which models the block. Otherwise, the component is an interface, such as an input port, and the variable is declared to be a total function from time to real numbers (more details are in Section 7.5).
- Each wire is represented by a TIC predicate in the way as described in the previous section, where the variables used in predicate are the schema variables from the schema declaration.

A Simulink block can be either an elementary block or a diagram itself representing a subsystem. The way to handle elementary blocks is illustrated in Section 7.1 and Section 7.5 shows how to deal with subsystems.

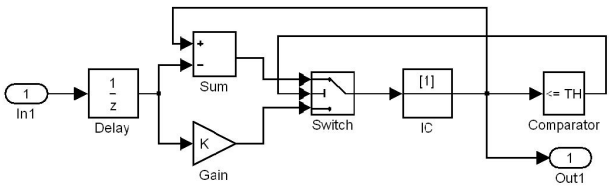


Figure 7.2: A diagram with specified sample times for blocks *Delay* and *IC*.

7.4 Computing Unspecified Sample Times

In Section 7.1, we described three ways of determining sample times of elementary blocks. However, it is often a case that these ways are inapplicable in practice, as users usually assign particular values to sample times of some blocks and leave other blocks with unspecified sample times.

Taking Figure 7.2 as a running example here, the diagram consists of six elementary blocks, one input port (*In1*) and one output port (*Out1*), and only the sample times of blocks *Delay* and *IC* are specified with values 1 and 2 respectively. We remark that in the beginning the sample time of elementary block *Switch* is unknown by using the method in Section 7.1 since its block type is not continuous and the sample times of blocks *Sum*, *Comparator* and *Gain* which connect to its inputs are unspecified.

It is thus necessary to develop an algorithm to handle the unspecified sample times which cannot be directly computed in the previous way. Our algorithm can automatically compute *derivable* sample times which are a subset of unspecified sample times. The unspecified sample time of an elementary block is derivable provided one of the following conditions holds.

1. One of the blocks connecting to the elementary block has continuous sample time;

2. All sample times of the blocks connecting to the elementary block are specified;

3. All sample times of the blocks connecting to the elementary block are either specified or derivable.

When the elementary block fulfills condition 1, its sample time is equal to the value 0 [85]. When condition 2 is satisfied, the sample time can be computed by applying the function *STP* defined in Section 7.1. When condition 3 holds, the sample time can be derived after finishing the computation of the derivable sample times of the blocks connecting to the elementary block.

According to the above conditions, we can deduce that the sample times of *Sum*, *Gain* and *Comparator* in Figure 7.2 are derivable by condition 2 and the sample time of *Switch* is also derivable by condition 3.

Based on the computability property of derivable sample times, we have developed a simple and effective algorithm to handle unspecified sample times. As shown below, the algorithm starts with a non-empty list *BLKS* of elementary blocks whose sample times are unspecified, and *repeatedly* modifies the list *until* its termination condition at line 15 holds. Specifically, it terminates when either all sample times are specified or none of the unspecified sample times is derivable. The first case is examined by method *Empty* which checks if *BLKS* is empty, and the second case is examined by method *CheckEq* which checks if there exists a change of *BLKS* each time the *for* loop from line 3 to line 14 executes.

Algorithm 1: Deal with all unspecified sample times

BLKS : a non-empty list of elementary blocks with unspecified sample times

```
 1:  repeat
 2:      iniBLKS ← BLKS
 3:      for all i = 1 to BLKS.length do
 4:          b ← BLKS[i]
 5:          if ExistsContinuous(b.GetInBLKSST()) then
 6:              b.st ← 0
 7:              BLKS ← Delete(BLKS, b)
 8:          else if AllSTKnown(b.GetInBLKSST()) then
 9:              b.st ← CallSTP(BLKS, b)
10:              BLKS ← Delete(BLKS, b)
11:          else skip
12:          end if
13:          i ← i + 1
14:      end for
15:  until Empty(BLKS) or CheckEq(BLKS, iniBLKS)
```

An element b in *BLKS* is analyzed with respect to three cases (line 5 to line 12).

- Lines 5 to 7 correspond to condition 1. Method *ExistsContinuous* checks whether there exists a block whose sample time is continuous and the block connects to b (namely, the block is in the blocks returned by method *GetInBLKSST*). If *ExistsContinuous* returns true, the sample time of b is equal to the value 0, and then b is deleted from *BLKS* by method *Delete*.

- Lines 8 to 10 correspond to condition 2. Method *AllSTKnown* checks if all sample times of the blocks connecting to b are specified. If *AllSTKnown* returns true, we apply method *CallSTP* which implements the function *STP* defined in Section 7.1 to calculate the sample time of b, and then delete b from *BLKS*.

- Line 11 indicates that there is inadequate information to calculate the sample time of b. We simply do nothing to finish the analysis of b.

We illustrate how the algorithm can systematically compute the unspecified sample times in Figure 7.2. Initially, *BLKS* consists of four blocks, *Sum*, *Gain*, *Switch* and *Comparator*. After the first time the *for* loop is executed, *Gain*, *Sum* and *Comparator* are deleted from *BLKS* with specified sample times which are 1, 1 and 2 respectively. After the second execution of the *for* loop, *Switch* is deleted with its sample time which is 1. The algorithm hence terminates since *BLKS* becomes empty.

Note that the Simulink diagram contains two loops: One consists of *Sum*, *Switch* and *IC*; and the other comprises *Switch*, *IC* and *Comparator*. Loop structure is often used in Simulink for modeling differential equations or feedback control systems. Nevertheless, [129] is unable to compute block sample times in loops. In contrast, our algorithm can calculate derivable sample times in complex Simulink diagrams containing loop structure.

7.5 Dealing with the *Ports and Subsystems* Category

The transformation presented in Section 7.1 handles the elementary blocks whose library blocks are modeled by the TIC library functions defined in Chapter 6.2. However, it is difficult and impracticable to define TIC library functions to model the library blocks of the *Ports and Subsystem* category, because the behavior of their instances as produced elementary blocks in particular diagrams is usually unpredictable. We hence directly model their instances in TIC during the transformation. We demonstrate below how to deal with these library blocks based on their usage, specifically, either denoting interfaces or creating subsystems. Appendix C.2 lists the library block names of this category supported so far.

- Library blocks *Inport*, *Outport*, *Enable*, and *Trigger* are designed to create (sub)system interfaces. For example, instances of the *Outport* library block

represent outputs of (sub)systems. An instance of one of these four library blocks is transformed to a total function of time in a TIC schema which models the (sub)system which contains the instance. Furthermore, an instance of the *Inport* or *Outport* library block in a *plain* or *enabled* subsystem (which will be explained more in the following context) is declared to be continuous if the output of the block which connects to the instance is continuous, as the instance inherits the sample time from the block.

Reusing the system *calculation* in Figure 7.1 as an example, its input port *Acc* is an instance of the *Inport* library block, and its output port *Speed* is an instance of the *Outport* library block. These ports are represented by two functions declared in the following schema *calculation* which models the system. These functions are in turn used in the predicates of *calculation*. Moreover, *Speed* is continuous as indicated by $\nrightarrow$ because the elementary block *Integrator* outputs continuously. Note that the schema *calculation_Integrator* which models *Integrator* was presented in Section 7.1.

calculation

$$Acc : \mathbb{T} \rightarrow \mathbb{R};\ Integrator : calculation\_Integrator;\ Speed : \mathbb{T} \nrightarrow \mathbb{R}$$

$$\mathbb{I} = (\!|Acc = Integrator.In_1|\!) \land \mathbb{I} = (\!|Integrator.Out = Speed|\!)$$

- The *Subsystem* library block is applied to form plain subsystems which virtually reduce the number of blocks displayed in Simulink diagrams and form the hierarchical structure of Simulink diagrams. The plain subsystems are thus treated in the same way of transforming diagrams as described in Section 7.3.

  Library blocks *Enabled Subsystem* and *Triggered Subsystem* are used to build enabled subsystems and triggered subsystems respectively which are *conditionally executed* subsystems. Here we illustrate our solution for dealing with these subsystems with examples in the following two subsections.

7.5.1 Triggered Subsystems

A *triggered* subsystem executes each time a trigger event occurs. A trigger event is determined by a control input which is an instance of the *Trigger* library block. There are three types of trigger events, *rising*, *falling*, and *either*, according to the direction the control input crosses the value 0. For instance, a *rising* trigger event occurs, when the control input rises from a negative or zero value to a positive value.

When no events occur, triggered subsystems always hold their outputs at the last value between trigger events [85]. In addition, Simulink constrains the sample times of components in a triggered subsystem in the following way: all blocks and interface ports such as an input have the same sample times of its control input. To model a triggered subsystem, we focus on modeling the way which assigns values to system inputs under different circumstances of the system control input, whether a trigger event occurs or not. This is because the subsystem outputs and the behavior of its components are dependent on the subsystem inputs.

Triggered subsystem can be classified into two groups in terms of their control inputs, which can be either continuous or discrete. We present here how to handle triggered subsystems whose control inputs are continuous. Appendix D.1 shows the way to support the other group where control inputs are discrete.

When the control input of a triggered subsystem is continuous, a trigger event occurs only at a point-interval where the starting point equals the ending point. The subsystem behavior is modeled for three cases: the presence of trigger events, the absence of trigger events in non-initial intervals whose starting points are positive, and the absence of trigger events in initial intervals which start with the time point 0. We remark that the last case is not specified in the Simulink documentation [84, 85].

For example, a triggered subsystem named *trigsys* as shown in Figure 7.3 outputs the

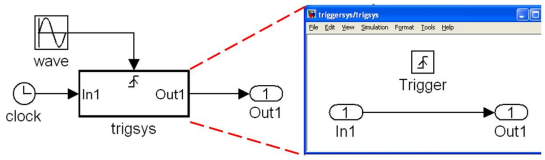


Figure 7.3: A triggered subsystem with a continuous control input

moments when its continuous control input *Trigger* rises from a negative or zero value to a positive value. In the following schema *sys_trigsys* which denotes the triggered subsystem, the control input is declared to be a total function *Trigger* whose range consists of two values, 1 and 0, where the value 1 indicates the presence of a trigger event and the value 0 indicates the absence. Because we aim at modeling the subsystem behavior with respect to trigger events, we hence simply define control inputs as functions without specifying how to detect trigger events. In addition, *sys_trigsys* captures the timing feature that trigger events occur only at point-intervals by the predicate $\lfloor Trigger = 1 \rfloor \subseteq \lfloor \alpha = \omega \rfloor$ where $\lfloor\ \rfloor$ indicates *both-closed* intervals.

sys_trigsys

$$Trigger : \mathbb{T} \to \{0, 1\};\ In1, Out1 : \mathbb{T} \to \mathbb{R}$$

$$\lfloor Trigger = 1 \rfloor \subseteq \lfloor \alpha = \omega \rfloor \land \mathbb{I} = (\!| In1 = Out1 |\!)$$

sys

$$clock : sys\_clock;\ trigsys : sys\_trigsys;\ \ldots$$

$$\ldots$$

$$\lfloor trigsys.Trigger = 1 \rfloor \subseteq \lfloor clock.Out = trigsys.In1 \rfloor \qquad \text{[Predicate1]}$$

$$(\!| trigsys.Trigger = 0 \land \alpha > 0 |\!) \subseteq (\!| trigsys.In1(\alpha) = trigsys.In1 |\!) \qquad \text{[Predicate2]}$$

$$(\!| trigsys.Trigger = 0 \land \alpha = 0 |\!) \subseteq (\!| trigsys.In1 = 0 |\!) \qquad \text{[Predicate3]}$$

In the above schema *sys* which includes the subsystem *trigsys* by the declaration *trigsys* : *sys_trigsys*, the conditionally executed behavior of *trigsys* is depicted by

three predicates which constrain subsystem input *trigsys.In*1. Predicate1 states that *trigsys.In*1 equals the output of elementary block *clock* when a trigger event happens. Predicate2 restricts the values of *trigsys.In*1 in a non-initial interval (indicated by $\alpha > 0$), where no trigger event occurs, to be the *trigsys.In*1 value at the beginning of the non-initial interval. Predicate3 captures the default value of *trigsys.In*1 when no trigger event happens in an initial interval (by $\alpha = 0$), namely, the value 0. Note that this default value is missed in the Simulink documentation.

We informally explain below the reason to represent the last value between trigger events by using α in Predicate2. When a trigger event occurs at a time point t, we have $clock.Out(t) = trigsys.In1(t)$ according to Predicate1. Based on Predicate2, we can deduce that the set of intervals as denoted by $[\![trigsys.Trigger = 0 \wedge \alpha > 0]\!]$ contains a *left-open* interval i such that (1) its starting point is t, namely, $\alpha(i) = t$, and (2) the values of *trigsys.In*1 in i are equal to $trigsys.In1(\alpha(i))$, namely, $trigsys.In1(t)$. Furthermore, we can imply that in any interval which starts at t and ends before a time point t' at which the next trigger event happens, the values of *trigsys.In*1 are the same as $trigsys.In1(t)$. For instance, for a point-interval $[x \ldots x]$ where $t < x < t'$, we can find a *left-open and right-closed* interval $(t \ldots x]$ such that $trigsys.In1(t) = trigsys.In1(x)$ because of the TIC expression $[\![trigsys.In1(\alpha) = trigsys.In1]\!]$.

7.5.2 Enabled Subsystems

An *enabled* subsystem executes when the value of its control input which is an instance of the *Enabled* library block is positive. Namely, an enabled subsystem starts its execution from the moment when its control input value crosses zero from a negative value and continues its execution in the interval in which the control input values remains positive. When an enabled subsystem is disabled, it can output either its last values or its initial output values. We demonstrate here how to model the enabled

subsystems which output their last values when they are disabled. Nevertheless, the enabled subsystems which output their initial output values can be handled similarly with auxiliary variables for storing the initial output values.

We concentrate on specifying the behavior which associates the subsystem inputs with their control inputs. Unlike triggered subsystems which restrict all components of a triggered subsystem to have the same sample time, enabled subsystems in Simulink can include components with different sample times [85]. This loose restriction results in the difficulty of representing the last values of enabled subsystems, especially when the control inputs are discrete. We present below our solution to model the enabled subsystems whose control inputs are discrete. Appendix D.2 shows how to support the enabled subsystems whose control inputs are continuous.

When the control input of an enabled subsystem is discrete, we can represent the behavior of its control input by specifying its value at every sample time hits. Nevertheless, the logic that Simulink uses to update enabled subsystems with different sample times of their components is complicated to be identified, as discussed by Tripakis et al. [129]. Currently, we restrict ourselves to cope with a subset of enabled subsystems: all sample times of components within an enabled subsystem are inherited by the inputs of the subsystem, and all subsystem inputs have the same sample times. These restrictions are weaker than those of Tripakis et al., as we allow the sample time of the control input to be different from other sample times.

Taking enabled subsystem *open* as displayed in Figure 7.4 as an example, it multiplies its continuous input *volumeIn* by the constant -0.1 when it is enabled; otherwise, it outputs the last value. Its control input *Enable* is linked by elementary block *inverse* whose sample time equals one time unit. In the following TIC schemas, *tank_open_K* represents elementary block K and *tank_open* represents *open*. In this example, the input *volumeIn* of *open* is connected by a continuous block, so it is defined by a

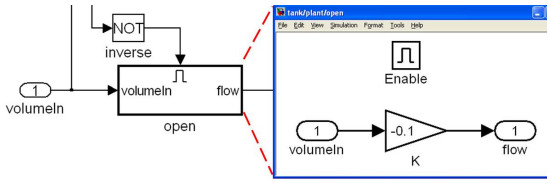


Figure 7.4: An enable subsystem *open* in a system *tank*.

continuous function in *tank_open*. Therefore, the sample time of K is equal to the value 0, and the output *flow* of *open* is hence continuous based on the discussion at the beginning of Section 7.5.

$$tank\_open\_K \mathrel{\widehat=} Gain(0, -0.1)$$

$$\begin{array}{|l} tank\_open \\ \hline Enable : \mathbb{T} \to \mathbb{R};\ volumeIn, flow : \mathbb{T} \pfun \mathbb{R};\ K : tank\_open\_K \\ \hline \mathbb{I} = [\![volumeIn = K.In_1]\!] \land \mathbb{I} = [\![K.Out = flow]\!] \end{array}$$

The components and wires in *open* are captured in *tank_open*. The conditionally executed behavior of *open* is specified in the following schema *tank* which denotes the system *tank*. Specifically, we model the relation between the input *open.volumeIn* and the control input *open.Enable* in every sample time interval. Note that a sample time interval is *left-closed and right-open* (indicated by $[\,)$ in Definition 2). Moreover, discrete systems in Simulink execute only at sample time hits; particularly, elementary block *inverse* outputs values to *open.Enable* every 1 time unit. As *open* is either enabled or disabled at the endpoints of a sample time interval, so there are four cases regarding the *open* status at the endpoints. The sample time intervals during which *open* is disabled are further distinguished to two groups based on their starting points, because when an enabled subsystem is disabled in the initial sample time interval, its last value is 0 by default. We remark that this default value is obtained from our experience, although it is unspecified in the Simulink documentation [85].

tank

$volumeIn : \mathbb{T} \pfun \mathbb{R}$; $open : tank\_open$; ...

...

$[open.Enable \leq 0 \land open.Enable(\omega) > 0 \land \alpha = 0 \land \omega = 1) \subseteq [open.volumeIn = 0)$ [Predicate1]

$[open.Enable \leq 0 \land open.Enable(\omega) \leq 0 \land \alpha = 0 \land \omega = 1) \subseteq [open.volumeIn = 0 \land open.volumeIn(\omega) = 0)$ [Predicate2]

$[open.Enable \leq 0 \land open.Enable(\omega) > 0 \land \exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1) \subseteq [open.volumeIn(\alpha) = open.volumeIn)$ [Predicate3]

$[open.Enable \leq 0 \land open.Enable(\omega) \leq 0 \land \exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1) \subseteq [open.volumeIn(\alpha) = open.volumeIn \land open.volumeIn(\alpha) = open.volumeIn(\omega))$ [Predicate4]

$[open.Enable > 0 \land open.Enable(\omega) > 0 \land \exists\, k : \mathbb{N} \bullet \alpha = k \land \omega = k + 1) \subseteq [volumeIn = open.volumeIn)$ [Predicate5]

$[open.Enable > 0 \land open.Enable(\omega) \leq 0 \land \exists\, k : \mathbb{N} \bullet \alpha = k \land \omega = k + 1) \subseteq [volumeIn = open.volumeIn \land volumeIn(\omega) = open.volumeIn(\omega))$ [Predicate6]

The first four predicates depict the behavior when *open* is disabled in a sample time interval (indicated by $open.Enable \leq 0$). Predicate1 and Predicate2 are concerned with the initial sample time interval which starts with the time point 0: the values of *open.volumeIn* are equal to 0 in the interval. In addition, if *open* is still disabled at the ending point, namely, $open.Enable(\omega) \leq 0$, then the value of *open.volumeIn* is 0 at the ending point (in Predicate2). Predicate3 and Predicate4 deal with the non-initial sample time intervals (denoted by $[\exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1)$), where the last value is the *open.volumeIn* value at the starting point ($open.volumeIn(\alpha)$). Furthermore, if *open* is still disabled at the ending point, we assign the *open.volumeIn* value at the ending point to be the last value (in Predicate4). The way of representing last values here is similar to the one handling triggered subsystems in Section 7.5.1.

Predicate5 and Predicate6 model the behavior when *open* is enabled in a sample time interval (indicated by $open.Enable > 0$). To be specific, input *open.volumeIn* of *open* is equal to input *volumeIn* of *tank* at the same time point in the sample time interval. Moreover, if *open* becomes disabled at the ending point (denoted by $open.Enable(\omega) \leq 0$), the *volumeIn* value at the ending point can be considered as

the last value for *open.volumeIn* (namely, $volumeIn(\omega) = open.volumeIn(\omega)$ in Predicate6), based on mathematical theories of continuous functions (*volumeIn*). In the case where *volumeIn* is discrete, we can assign the *volumeIn* value at the *most recent* sample time hit before the ending point to be the last value.

7.6 Summary

Up to now, we have developed a translator which implements the strategy to automatically transform Simulink diagrams to TIC models. The transformation can preserve the functional and timing aspects of Simulink elementary blocks and the hierarchical structure of diagrams. We have also presented an algorithm to compute unspecified sample times in (complex) Simulink diagrams. We have further discussed how to handle the library blocks of the *Ports and Subsystems* category, in particular enabled subsystems and triggered subsystems. The conditional execution behavior is captured by precisely modeling the relationship between subsystem inputs and control inputs under different circumstances, such as an enabled subsystem with a discrete control input and continuous system inputs. We currently support 7 library blocks of this category, and hence our approach covers totally 51 library blocks of 10 categories (other 44 library blocks are modeled in Chapter 6). Based on transformed TIC models of Simulink diagrams, we can specify requirements over diagrams, and rigorously check their validity with a high grade of automation, as we illustrate in the following chapter.

Chapter 8

Validation beyond Simulink

The significant and novel point of our formal framework is to exploit TIC modeling and reasoning features to support the validation beyond Simulink. Previous chapters (6 and 7) have illustrated how to precisely and concisely capture the functional and timing aspects of Simulink diagrams in TIC. Based on the TIC models of diagrams, we can specify important timing requirements over the whole system or some components. Moreover, additional properties of open systems, such as the bounds of an environment variable, can be expressed as well. We can further rigorously validate diagrams against these requirements, with a high degree of automation by applying the verification system developed in chapters 3 and 4.

Recently, there have been a number of works on transforming Simulink into other formal notations or programming languages. Adams and Clayton [1], Arthan et al. [10] transformed Simulink diagrams to the Z notation [136] by capturing the functional behavior of one cycle. Cavalcanti et al. [22] extended that work by applying Circus [135] to model the functionality and concurrency of diagrams. Their approaches focused on the verification of Simulink diagrams, specifically, verifying whether Simulink dia-

grams are correctly implemented in the programming language Ada. This is different from our approach: we focus on the validation of Simulink diagrams, namely, checking if Simulink diagrams fulfill various requirements. Moreover, they consider only single-rate discrete systems, and timing information is missing. In contrast, we can handle multi-rate discrete and hybrid systems, and the timing information is retained as well.

Meenakshi et al. [89] used the *NuSMV* model checker [35] to analyze single-rate discrete Simulink diagrams. Tripakis et al. [21, 129] applied the synchronous programming language *Lustre* [57] to support multi-rate discrete Simulink diagrams. Tiwari et al. [128] converted differential equations denoted in Simulink to difference equations for constructing discrete transition systems. However, the discretization of infinite transition systems can decrease accuracy when checking properties of continuous dynamics [97]. Our approach is different in that we can directly represent and analyze continuous Simulink diagrams.

There are other approaches [56, 118] which took into account Simulink/Stateflow[1] Models (SSMs). Sims et al. [118] verified SSMs with an invariant checker *Salsa* [15], and the transformation from SSMs to the input language of the checker was performed by hand. Gupta et al. [56] developed a hybrid verification system *CheckMate* [117] which contained some customized Simulink blocks to increase the modeling capacity. *CheckMate* was designed to check functional behavior, and it lacked support of timing analysis. Jersak et al. [72] transformed Simulink diagrams into SPI [142] models for timing analysis, while the transformation abstracted functional behavior. On the other hand, we support the validation for both functional and timing behavior.

Latest work of the MathWorks is the Simulink Design Verifier [86] which can auto-

[1]Stateflow [87] combines flow diagrams and statecharts [61] to specify control logic and can be integrated with Simulink.

matically generate test input sequences for a Simulink diagram or prove properties about this diagram. Currently the tool is restricted to the analysis of discrete-time systems and primitive properties such as linear integer arithmetic. The objective of this tool is to apply model checker to generate test cases. This is different from our approach, as we aim to validate Simulink diagrams for all possible cases. In addition, we also support the analysis of continuous-time systems.

This chapter starts by extending the translation described in Chapter 4.1 to deal with axiomatic definitions [136] which are the type of TIC library functions. Next, we define a set of supplementary rules dedicated to Simulink modeling characteristics to make validation more automated. Thirdly, we illustrate the benefits of our framework by an example study of a brake control system which involves continuous and discrete behavior. The chapter ends by comparing our approach with other existing works.

8.1 Translating TIC Library Functions

In Chapter 6.2, we have defined TIC library functions for modeling Simulink library blocks. These library functions return the TIC schemas which capture the time-dependent relationships denoted by Simulink blocks. These library functions are axiomatic definitions. The translation presented in Chapter 4.1 considers two types of TIC models, namely, the TIC schemas which represent system properties and the TIC predicates which denote requirements.

We describe here the way to automatically translate TIC axiomatic definitions to PVS specifications. An axiomatic definition usually declares a global variable and specifies constraints of the declared variable. Moreover, constraints are assumed to hold whenever the variable is used. A general form can be depicted as below, where *Predicate* represents the constraints of a variable *Decl_Name* whose type is *Expression*. The

definition of a function *square* is given as a concrete example as well.

$$\begin{array}{|l} Decl\_Name : Expression \\ \hline Predicate \end{array} \qquad \begin{array}{|l} square : \mathbb{N} \rightarrow \mathbb{N} \\ \hline \forall\, n : \mathbb{N} \bullet square(n) = n * n \end{array}$$

PVS *constant declarations* [104] introduce new constants, specifying their types and optionally providing a value. The term *constant* in the PVS higher-order logic refers to not only the usual (0-ary) constants, but also functions and relations. In our approach, an axiomatic definition is transformed to a PVS function which is a constant declaration. To be specific, for an axiomatic definition, the name of its variable *Decl_Name* is used as the identifier of a PVS function; the type of the variable *Expression* is converted into PVS specifications; and the constraint *Predicate* is translated to a PVS *axiom* whose identifier is *Decl_Name* as well. These mappings are sketched below. We also present the corresponding PVS function for the function *square*.

```
1  Decl_Name : Expression;
2  Decl_Name : AXIOM Predicate;
```

```
1  square : [nat -> nat];
2  square : AXIOM FORALL (n: nat):
3                   square(n) = n * n;
```

We remark that the *name overloading* technique is applied to construct PVS specifications of axiomatic definitions. In the above PVS specifications of the axiomatic function *square*, line 1 declares that a constant `square` is a total function whose domain and range are natural numbers, and lines 2 and 3 model the function property as an axiom whose identifier is also `square`.

A TIC library function is an axiomatic definition, and is thus transformed to a PVS function which returns a set of PVS *records*. Specifically, the parameters of the PVS function correspond to the arguments of the TIC library, and the set of records represents a schema which is returned by the TIC library function. Taking the TIC library function *Integrator* in Chapter 6.2 as an example, it is transformed to the following PVS function `Integrator` where the keywords such as `Trace`, `Time`, `AllS`, `ALPHA`, `OMEGA`, and `LIFT` encode TIC semantics in PVS as defined in Chapter 3.1.

$$
\begin{array}{|l}
\hline
Integrator : \mathbb{R} \rightarrow \mathbb{P}[In_1 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, init : \mathbb{R} \bullet Integrator(init) = [In_1 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = 0 \land IniVal = init \land Out(0) = IniVal \land \mathbb{I} = [\![Out(\omega) = Out(\alpha) + \textstyle\int_{\alpha}^{\omega} In_1]\!]] \\
\hline
\end{array}
$$

```
1  Integrator: [real -> setof[[# In1: Trace, Out: Trace, IniVal: real, st: Time #]]];
2  Integrator: AXIOM  FORALL (init: real): Integrator(init) =
3      { temp: [# In1: Trace, Out: Trace, IniVal: real, st: Time #] |
4        temp‘st = 0 AND temp‘IniVal = init AND temp‘Out(0) = temp‘IniVal AND
5        fullset = AllS((LIFT(temp‘Out) o LIFT(OMEGA)) = (LIFT(temp‘Out) o LIFT(ALPHA)) +
6                              TICIntegral(LIFT(ALPHA), LIFT(OMEGA), temp‘In1)) AND
7        continuous(temp‘Out)};
```

- Line 1 declares the type of the function. Namely, `Integrator` is from real numbers to a set of records (by `setof[[# ... #]]`). Each record in the set comprises four accessors, and each accessor is associated with a corresponding type. For example, the type of the accessor `Out` is a timed trace (`Trace`) as a function from time to real numbers.

- The `AXIOM` specification from line 2 to line 7 models the properties of the function. An auxiliary variable *temp* is defined at line 3 for easily referencing accessors by using the PVS projection function (‘). This is similar to the way mentioned in Chapter 4.1. For instance, `temp‘st` refers to the accessor `st`. Line 4 captures the timing feature such as its continuous sample time, and relationships between accessors and parameters. The predicate at lines 5 and 6 represents the integration operation in PVS. Note that continuous behavior is supported based on the NASA PVS library [19], such as the function `TICIntegral` for an integral operation and the function `continuous` indicating the continuity feature of the output `Out`.

The above PVS function `Integrator` follows closely the TIC library function *Integrator* in terms of the structure. Schema predicates in *Integrator* are converted to the constraints in `Integrator` for *all* record accessors. Moreover, `Integrator` facilitates the proofs in PVS, in particular, when applying the property of an elementary block of the *Integrator* library block. Namely, we only need to enter the proof command `lemma` with the function name once to retrieve properties of four accessors during a proof.

8.2 Facilitating TIC Validation of Simulink Diagrams

We define a collection of supplementary rules dedicated to Simulink modeling features to increase the efficiency of TIC validation. We categories these rules into two groups: one group deals with connections in Simulink diagrams, and the other handles discrete systems modeled in Simulink.

Wires in Simulink diagrams are represented by equations in TIC (as shown in Chapter 7.2). Each equation contains two timed traces which denote connected block ports. When reasoning about TIC models of Simulink diagrams, we often need to substitute one timed trace for another provided they indicate a connection. In other words, values of these two timed traces are equal in any interval. However, this kind of substitution is tedious in PVS since we need to completely expand detailed encoding of TIC semantics to the low level of time points enable the PVS prover to automatically discharge proof goals. To simplify the substitution process, we define a set of rewriting rules to replace one timed trace by another at the interval level under various circumstances. For example, the following rule `BB_eq_sub` passes a constant value `v` between two equivalent timed traces `tr1` and `tr2`, namely, from `tr1` to `tr2` and vice versa. This rule saves five proof steps such as expanding the function `AllS` which encodes the interval brackets $[\![\cdot]\!]$ in PVS.

```
BB_eq_sub: LEMMA FORALL (tr1, tr2: Trace, v: real):
     fullset = AllS(LIFT(tr1) = LIFT(tr2))
  => AllS(LIFT(tr1) = LIFT(v)) = AllS(LIFT(tr2) = LIFT(v));
```

Discrete systems with periodic execution are a common application domain of Simulink. The time domain of these discrete systems in Simulink is decomposed into a sequence of sample time intervals as defined in Chapter 6.1, and systems are updated at sample time hits. Based on these features, we define several domain-specific rules to ease the analysis of this domain. For instance, to verify a safety requirement `pl` of a discrete system, we can apply the following rule `CO_to_All` if `pl` is independent on interval operators (namely, α, ω, and δ). Note that a safety requirement must be valid in every non-empty interval. The rule simplifies the checking of `pl` by considering only the sample time intervals, which are expressed in PVS at lines 2 and 3, rather than

all non-empty intervals. In addition, the function `No_Term?` at line 1 indicates that `pl` is not affected by any interval operator.

```
1  CO_to_All: LEMMA st > 0 AND No_Term?(pl) =>
2       subset?(COS(exNat( lambda(k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(st) AND
3                                  LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(st))),
4               COS(pl))
5    => fullset = AllS(pl);
```

We have constructed 25 supplementary rules (in Appendix A.3) which have been validated in PVS. These rules facilitate the validation of Simulink diagrams by elevating the grade of automation. In the following section, we will show the feasibility and benefit of our approach from our experimental studies.

8.3 Implementation and Experimental Study

To enhance the usability of our approach, we apply Java technology to implement the framework. The work flow is shown in Figure 8.1. Two translators automatically transform system designs. Specifically, *Sim2TIC* transforms Simulink diagrams into TIC models, and *TIC2PVS* translates the TIC models of requirements and Simulink library blocks and diagrams to PVS specifications. Moreover, supplementary rules (from the previous section) are imported to simplify reasoning processes.

Using this framework, we can rigorously validate various systems modeled in Simulink, such as multi-rate discrete or hybrid systems. Open systems are also supported, as we can formally specify environment properties in TIC based on the transformed TIC models denoting Simulink diagrams. With the powerful verification capability, the framework can handle the analysis of continuous dynamics and can reason about important timing requirements such as bounded response requirements.

Here we apply our framework to an adapted brake control system which is designed to

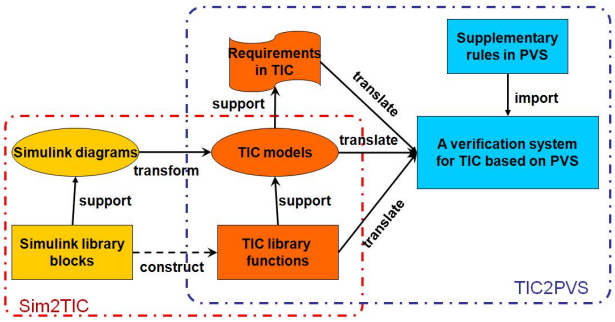


Figure 8.1: The framework structure to model and validate Simulink diagrams

prevent a vehicle from over speeding by automatically turning on its brakes in time. The control system is *open* and possesses both continuous and discrete behavior. We firstly present the Simulink diagram of the system as well as its transformed TIC schemas. Next, we sketch the validation of two important requirements of the control system. One requirement is concerned with computational accuracy, and the other requirement checks the response time within which brakes should respond.

8.3.1 Specifications of System Design and Requirements

As shown in Figure 8.2, the Simulink diagram modeling the brake control system consists of three subsystems: *plant*, *sensor* and *brake*.

- Subsystem *plant* represents the continuous behavior of vehicle speed. Speed is calculated by elementary block *Integration* which continuously accumulates the acceleration rate from elementary block *Switch*. For different status of brakes, which is indicated by input port *command* from subsystem *brake*, *Switch* outputs a respective acceleration rate which is from either input port *on* or input port

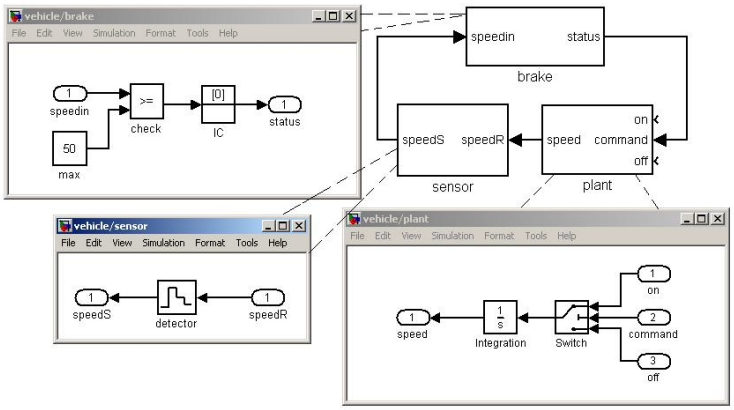


Figure 8.2: The brake control system in Simulink

off. Specifically, when the value of *command* is 0, *Switch* outputs its third input; when the value of *command* is 1, *Switch* outputs its first input.

- Subsystem *sensor* converts continuous speed into discrete speed. The conversion is performed by elementary block *detector* which is an instance of the *Zero-Order Hold* library block. The block samples vehicle speed from subsystem *plant* every 1 second via output port *speedR* and outputs the sampled speed to subsystem *brake* via output port *speedS*.

- Subsystem *brake* controls brakes according to the speed from subsystem *sensor*. Elementary block *check* outputs the value 1 to enable brakes if its input value is not lower than 50 meter/second which is the specified via elementary block *max*, and outputs the value 0 to disable brakes otherwise. Initially, brakes are disabled, and this is depicted by elementary block *IC* which outputs the value 0 at the time point 0 and outputs the value of *check* after the time point 0.

We apply the strategy described in Chapter 7 to automatically transform the Simulink diagram to TIC schemas. Firstly, there are six elementary blocks in Figure 8.2, and

they are created by different Simulink library blocks. Three of these library blocks are presented in Chapter 6.2: the *Integrator* library block for block *Integration*, the *Zero-Order Hold* library block for block *detector*, and the *Relational Operator* library block for block *check*. We demonstrate below the other three library blocks.

1. The *Constant* library block produces the elementary block *Max*. This library block always outputs the value stored in its *Constant Value* block parameter.

$$
\begin{array}{|l}
Constant : \mathbb{R} \rightarrow \mathbb{P}[Out : \mathbb{T} \rightarrow \mathbb{R};\ IniVal : \mathbb{R}] \\
\hline
\forall\, cv : \mathbb{R} \bullet Constant(cv) = \\
\quad [Out : \mathbb{T} \rightarrow \mathbb{R};\ IniVal : \mathbb{R} \mid cv = IniVal \wedge \mathbb{I} = [\![Out = IniVal]\!]]
\end{array}
$$

2. The *Switch* library block produces the elementary block *Switch*. This library block switches its output between first and third input based on the value of its second input. The selection condition is determined by the *Threshold* block parameter which assigns the switch threshold and the *Criteria for passing first input* block parameter which specifies how to check the second input; whether the second input is greater than or equal to the threshold value, purely greater than the threshold value, or nonzero. Thus, the *Criteria for passing first input* parameter is treated as an operator parameter (as introduced in Chapter 6.2), as its value can result in different functionalities of generated elementary blocks. In this Simulink diagram, the *Switch* library block passes its first input when its second input is purely greater than its threshold value.

$$
\begin{array}{|l}
Switch\_G : \mathbb{T} \times \mathbb{R} \rightarrow \mathbb{P}[In_1, In_2, In_3, Out : \mathbb{T} \rightarrow \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T};\ th : \mathbb{R} \bullet (t = 0 \Rightarrow Switch\_G(t, th) = [In_1, In_2, In_3, Out : \mathbb{T} \rightarrow \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T} \mid \\
\quad st = 0 \wedge th = TH \wedge [\![In_2 > TH]\!] = [\![Out = In_1]\!] \wedge [\![In_2 \leq TH]\!] = [\![Out = In_3]\!]]) \\
\wedge\ \ (t > 0 \Rightarrow Switch\_G(t, th) = [In_1, In_2, In_3, Out : \mathbb{T} \rightarrow \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge th = TH \wedge \{\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\} \subseteq \\
\qquad \{(In_2(\alpha) > TH \Rightarrow Out = In_1(\alpha)) \wedge (In_2(\alpha) \leq TH \Rightarrow Out = In_3(\alpha))\}])
\end{array}
$$

3. The *IC* library block produces the elementary block *IC*. This library block outputs the value specified via the *Initial value* block parameter when the simulation starts. Thereafter, the block outputs its input value.

$$
\begin{array}{|l}
InitCond : \mathbb{T} \times \mathbb{R} \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T};\ init : \mathbb{R} \bullet (t = 0 \Rightarrow InitCond(t, init) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = 0 \land init = IniVal \land (\!|0 = \alpha|\!) \subseteq (\!|Out(\alpha) = IniVal|\!) \land (\!|0 < \alpha|\!) \subseteq (\!|Out = In_1|\!)]) \\
\land\ \ (t > 0 \Rightarrow InitCond(t, init) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad\qquad t = st \land st > 0 \land init = IniVal \land (\!|\alpha = 0 \land \omega = st|\!) = (\!|Out = IniVal|\!) \land \\
\qquad\qquad (\!|\exists\, k : \mathbb{N}_1 \bullet \alpha = k * st \land \omega = (k+1) * st|\!) \subseteq (\!|Out = In_1(\alpha)|\!)])
\end{array}
$$

Next, the transformation captures the timing aspect of the diagram, namely, sample times. In the diagram, the sample time of *detector* is specified with the value 1, and the sample times of *on* and *off* are assigned to be continuous. Note that subsystem *plant* is an open system modeling physical environment which usually evolves continuously. By the method presented in Chapter 7.1, the sample time of *Integration* is 0 because of its block type, and *Switch* has a continuous sample time since one sample time of its inputs is continuous such as the input port *on*. Using the algorithm stated in Chapter 7.4, we can derive other unspecified sample times: the sample time of *check* is computed to be 1 due to the elementary block *detector*, and the sample time of *IC* is in turn calculated to be the value 1. Note that the elementary block *max* has no effect on the computation of the sample time of *check*.

We illustrate the transformed TIC schemas below, which also preserve the functionalities of the elementary blocks and the connections in the diagram.

- Schema *vehicle_sensor_detector* represents the elementary block *detector*, and it is an application the *ZOH* library function. Schema *vehicle_sensor* denotes the subsystem *sensor*. We remark that the input port *speedR* is declared to be continuous (indicated by $\twoheadrightarrow$) because the output of the subsystem *plant* is continuous. The reason is explained at the beginning of Chapter 7.5.

$vehicle\_sensor\_detector \mathrel{\widehat=} ZOH(1)$

vehicle_sensor

$speedR : \mathbb{T} \pfun \mathbb{R};\ speedS : \mathbb{T} \to \mathbb{R};\ detector : vehicle\_sensor\_detector$

$\mathbb{I} = [\![speedR = detector.In_1]\!] \land \mathbb{I} = [\![detector.Out = speedS]\!]$

- The following three schemas model three components of the subsystem *brake*, namely, blocks *max*, *IC*, and *check*.

$vehicle\_brake\_max \mathrel{\widehat=} Constant(50)\ \ vehicle\_brake\_IC \mathrel{\widehat=} InitCond(1, 0)$
$vehicle\_brake\_check \mathrel{\widehat=} Relation\_geq(1)$

Schema *vehicle_brake* represents the subsystem *brake*.

vehicle_brake

$speedin, status : \mathbb{T} \to \mathbb{R};\ IC : vehicle\_brake\_IC$
$max : vehicle\_brake\_max;\ check : vehicle\_brake\_check$

$\mathbb{I} = [\![speedin = check.In_1]\!] \land \mathbb{I} = [\![max.Out = check.In_2]\!]$
$\mathbb{I} = [\![check.Out = IC.In_1]\!] \land \mathbb{I} = [\![IC.Out = status]\!]$

- Initially the speed of a vehicle is 0, which is captured by the elementary block *Integration*, as shown by $Integrator(0)$. Furthermore, the output *speed* of the subsystem *plant* is defined to be continuous due to *Integration*.

$vehicle\_plant\_Switch \mathrel{\widehat=} Switch\_G(0, 0)$
$vehicle\_plant\_Integration \mathrel{\widehat=} Integrator(0)$

vehicle_plant

$on, off, speed : \mathbb{T} \pfun \mathbb{R};\ Switch : vehicle\_plant\_Switch$
$Integration : vehicle\_plant\_Integration;\ command : \mathbb{T} \to \mathbb{R}$

$\mathbb{I} = [\![on = Switch.In_1]\!] \land \mathbb{I} = [\![Switch.Out = Integration.In_1]\!]$
$\mathbb{I} = [\![off = Switch.In_3]\!] \land \mathbb{I} = [\![Integration.Out = speed]\!]$
$\mathbb{I} = [\![command = Switch.In_2]\!]$

- The whole control system comprises the above three subsystems.

$$
\begin{array}{|l}
\hline
vehicle \\
brake : vehicle\_brake;\ plant : vehicle\_plant;\ sensor : vehicle\_sensor \\
\hline
\mathbb{I} = [\![sensor.speedS = brake.speedin]\!] \land \mathbb{I} = [\![plant.speed = sensor.speedR]\!] \\
\mathbb{I} = [\![brake.status = plant.command]\!] \\
\hline
\end{array}
$$

The brake control system is designed to satisfy several important requirements. For example, *sensor* should measure speed with an acceptable accuracy, and brakes should be enabled in time to avoid speeding. These requirements can be easily and precisely represented as TIC predicates based on the transformed TIC schemas, over the whole system or some components. We here specify two requirements which are used to show how validation can be rigorously conducted in the next section.

Requirement *Approximation* checks the computational accuracy between vehicle speed *speed* of *plant* and the sensed speed *speedS* of *sensor*. Specifically, the sensor should measure the speed within an accuracy of 10 meters/second in any non-empty interval.

$$Approximation == \forall\, v : vehicle \bullet \mathbb{I} = [\![|v.sensor.speedS - v.plant.speed| \leq 10]\!]$$

Requirement *Response* concerns the bounded response of brakes. Brakes should respond *in time* when the vehicle speed is too high. Namely, brakes must be enabled within 1 second ($\delta < 1$) and remain enabled till the end of a *both-closed* interval, during which the vehicle speed is not lower than 50 meter/second ($speed \geq 50$) and the interval lasts more than 1 second. Note that the concatenation operator $\frown$ as introduced in Chapter 2.1 connects two sets of intervals end-to-end.

$$
\begin{array}{l}
Response == \forall\, v : vehicle \bullet \\
\quad [v.plant.speed \geq 50 \land \delta > 1] \subseteq [\delta < 1) \frown [v.brake.status = 1]
\end{array}
$$

8.3.2 Validating System Design against Requirements

Validating systems denoted by Simulink diagrams against requirements is non-trivial, because these systems usually contain continuous dynamics and requirements often concern behavior over arbitrary (infinite) intervals. After transforming diagrams to TIC schemas, we can apply well-defined TIC reasoning rules and mathematical laws (of arithmetic and calculus) to rigorously prove if the TIC schemas logically imply the TIC predicates which represent the requirements.

To reduce the complexity of manual proofs in TIC, we exploit the verification system, as developed in Chapters 3 and 4 and extended in Section 8.1, to accomplish machine-assisted proofs. The main objective of validation is to assign proper values to the quantified predicates of intervals and time points, where an assignment is often automatic. Resulting (propositional) predicates usually can be automatically discharged by the verification system.

We sketch below the reasoning of the brake control system against the requirements described early to show the capability of our framework in effectively carrying out formal validation. Before the reasoning, we need to add environmental properties.

Adding Environmental Properties

In practice, it is difficult to identify the exact function which models the acceleration behavior of a vehicle, as the behavior would be affected by a number of (environmental) factors such as wind and the road condition. Open systems are unanalyzable in Simulink since simulation is inapplicable. On the other hand, loose information about environmental properties, for instance, the range of acceleration value in the brake control system, is usually available. This loose information can be represented

as TIC predicates which are formed based on the TIC schemas of Simulink diagrams, and it becomes possible to formally analyze open systems with the loose information.

Regarding the brake control system, the bounds of acceleration in different brake status are known as below. When brakes are disabled, the acceleration as indicated by the input port *off* is between 0 and 10 meter/second^2. When brakes are enabled, the acceleration as indicated by the input port *on* is between -10 and 0 meter/second^2. The above properties can be denoted by the following TIC predicate, which is further translated into a PVS axiom.

$$InputAssump == \forall\, v : vehicle \bullet \\ \mathbb{I} = [\![0 \leq v.plant.off \leq 10]\!] \wedge \mathbb{I} = [\![-10 \leq v.plant.on \leq 0]\!]$$

```
InputAssump: AXIOM FORALL (v: vehicle):
   fullset = AllS(LIFT(v‘plant‘off) >= LIFT(0) and LIFT(v‘plant‘off) <= LIFT(10))
and fullset = AllS(LIFT(v‘plant‘on) >= LIFT(-10) and LIFT(v‘plant‘on) <= LIFT(0))
```

Checking the Requirement *Approximation*

This requirement considers the difference between *speed* in the subsystem *plant* and *speedS* in the subsystem *sensor* at all non-empty intervals. Due to the block *detector*, *speedS* is updated at sample time hits and remains constant between sample time hits. Hence, the checking requires the analysis of continuous dynamics, namely, *speed* which is the integration of acceleration.

We apply the supplementary rule `CO_to_All` defined in Section 8.2 to shrink the intervals that are needed to be examined, from all intervals to all sample time intervals. In addition, by the functionality of *detector* and the connections in the system *vehicle*, the proof goal can be converted to compare the *speed* value at the starting point of a sample time interval with the *speed* values at other time points of that interval.

$$\forall\, i : [\![\exists\, k : \mathbb{N} \bullet \alpha = k \wedge \omega = k + 1)\!] \bullet \forall\, t : i \bullet \\ |v.plant.speed(t) - v.plant.speed(\alpha(i))| \leq 10$$

Using the *skolemization* technique of the PVS prover, we further reduce the complexity by checking the difference of the *speed* values at the starting point and an arbitrary fixed time point. We also exploit the lemma `Integral_bound` from the NASA PVS library [19], which relates *integration bounds* of an integrated function and *bounds* of the integrated function, to deduce that the difference is between -10 and 10. We remark that the environmental properties of the acceleration as added previously are necessary for analyzing the continuous behavior of the vehicle speed.

Checking the Requirement *Response*

This requirement concerns a timing relation between the brake status and the vehicle speed. The subsystem *sensor* which is between *plant* and *brake* executes discretely every 1 second. However, an endpoint of an interval i, where $i \in [\![v.plant.speed \geq 50 \wedge \delta > 1]\!]$, may not be a sample time hit. We apply the *proof by exhaustion* method to systematically resolve the above non-trivial difficulty.

Firstly, we develop a theorem to associate any arbitrary time point with a sample time (ST). The theorem as shown below categories *all* non-empty intervals into four groups according to their endpoints, whether they are sample time hits or not.

$$endpoint\_ST == \forall\, i : \mathbb{I} \bullet \exists\, m, p : \mathbb{N};\ n, q : \{0\} \cup \mathbb{R}^{+} \mid n < ST \wedge q < ST \bullet \alpha(i) = m * ST + n \wedge \omega(i) = p * ST + q$$

Next, we check *Response* for i which is one of four groups. Here we analyze the basic case where the endpoints of i are sample time hits, namely, $n = 0 \wedge q = 0$. In other words, i consists of multiple sample time intervals. The analysis here also facilitates the analysis of three other cases, as other types of intervals can be formed by appending an interval which lasts less than 1 second to the front or the back of multiple sample time intervals.

We further reduce the complexity of checking an implication relationship between two predicates in the basic case. To be specific, given two predicates which are independent of interval operators, rather than checking if they satisfy an implication relationship over the whole interval i, we only need to reason about the validity of the relationship over an arbitrary sample time interval which is inside i, by the skolemization technique. The theorem is provided below and has been validated.

```
Multi_Sample_Intervals: LEMMA No_Term?(tp1) AND No_Term?(tp2) AND x < y =>
    ((FORALL (k: {n: nat | x <= n AND n < y}):
      subset?( COS( tp1 AND LIFT(ALPHA) = LIFT(k) * LIFT(ST) AND
                    LIFT(OMEGA) = LIFT(k) * LIFT(ST)), COS(tp2)))
 => subset?( COS( tp1 AND LIFT(ALPHA) = LIFT(x) * LIFT(ST) AND
                  LIFT(OMEGA) = LIFT(y) * LIFT(ST)), COS(tp2)));
```

We remark that the above theorem is generic and can be applied for other systems which possess periodic behavior. For example, it can check the *exportable interval properties* defined in Interval Temporal Logic [95].

By applying the theorem `Multi_Sample_Intervals`, it is not difficult to prove that the requirement *Response* is valid for the basic case.

Experimental Results

We summarize our experimental results in Table 8.1, which lists the lemma names associated with the steps of proof commands entered from users and the execution time (in Seconds) of the PVS prover. In total, we have proved 16 lemmas, and some of them have been explained, such as *Approximation* and *Response*. The lemma *Response_L*1 corresponds to the analysis of the basic case mentioned previously. Moreover, the lemma *Safety* guarantees that a vehicle will not be speeding. Detailed specifications and complete proof scripts of these lemmas are available [30].

In this section, we applied our framework to formally model and rigorously validate

Lemma Name	Steps	Time	Lemma Name	Steps	Time
ACC_Range	11	77.73	Response_L2	68	26.26
Plant_Speed	43	73.98	Response_L3	49	26.40
Approximation	26	13.78	Response_L4	53	28.70
V_Initial	8	10.46	Response	18	15.74
Brake_Prop	27	45.75	ACC_On	16	66.66
V_at_sample	22	23.26	overlimit1	42	112.20
V_within_sample	27	21.56	overlimit2	55	185.88
Response_L1	46	24.83	Safety	48	103.45

Table 8.1: Experimental results of the validation of the brake control system

the Simulink diagram of the brake control system. The automatic transformation from Simulink to TIC preserves the functional and timing aspects of the brake control system which involves continuous and discrete behavior as well as discrete logic. The powerful verification system for TIC enables us to analyze continuous dynamics such as the integral equation of computing the vehicle speed, and to use common proof methods such as proof by exhaustion. Moreover, we conducted validation beyond Simulink. Particularly, we checked the control system as an open system by specifying the range of acceleration in TIC, and we also verified a timing requirement.

8.4 Summary

This chapter completes our framework which is designed to exploit TIC to complement Simulink and in turn to increase confidence in the system design modeled in Simulink. We presented the algorithm to translate axiomatic definitions to PVS functions. We have enhanced the verification system developed in Chapters 3 and 4 to implement the algorithm so as to support the automatic translation. We also defined a collection of supplementary rules dedicated to Simulink modeling features.

By the expressive power of TIC, this framework can capture not only complex Simulink diagrams, but also specify precisely various environmental properties of open systems and important requirements over the whole system or particular components. The machine-assisted proof support facilitates formal verification with a high level of automation (for instance, arithmetic reasoning is automatic). Moreover, we can rigorously validate system design against timing requirements beyond Simulink.

Chapter 9

Conclusion

This chapter serves two purposes. Firstly, a conclusion of the whole thesis is given, summarizing the main contributions and secondly, a discussion on future work directions is also presented.

9.1 Main Contributions of the Thesis

Real-time computing systems are increasingly embedded in safety-critical situations, and high level confidence of their design thus becomes necessary and important. Interval-based specification languages, such as TIC and DC, have been widely used to model and reason about real-time systems by their expressive power and formal verification capability.

When real-time computing systems are complex, manual proving of these systems against various requirements becomes an obstacle since it is difficult to guarantee the correctness of each proof step and to keep track of all proof details. Moreover, proofs usually involve the analysis of both continuous and discrete behavior.

This thesis presents the research work on providing machine-assisted proof support for interval-based specification languages and further expanding the usage of these languages in embedded real-time computing system development. The five main contributions of this thesis can be summarized as follows.

- In Chapter 3, we encoded the TIC semantics in PVS, formalized and checked all TIC reasoning rules based on the encoding, and defined a collection of supplementary rules and proof strategies to simplify reasoning processes of TIC models in PVS and hide the detailed encoding.

  The specification language of PVS is based on typed higher-order logic, which facilitates the encoding of expressive TIC constructs. The PVS interactive prover enables us to check the correctness of the TIC reasoning rules. We discovered subtle flaws of original reasoning rules from our rigorous validation procedure.

- Based on the above work, we developed a verification system for TIC in Chapter 4. The verification system can automatically translate TIC models to PVS specifications. We also proposed a general proof procedure to systematically conduct verification in a manner similar to manual TIC arguments, but with a high level of automation by exploiting the automatic reasoning power of PVS, in particular for sets, linear arithmetic, and propositional logic.

  Using the verification system, we can formally validate various systems (for example, hybrid systems) against important requirements (such as safety and bounded liveness). The application to the temperature control system illustrated the advantages of our verification system: advanced mathematical analysis, including integral calculus, is supported, and common proof methods can be applied to handle complex proofs (for instance, proof by induction to deal with arbitrary infinite intervals).

- We investigated the applicability of the verification system to handle other interval-based specification languages. In particular, we extended the system to support DC proofs in Chapter 5. We modeled DC constructs in TIC, formalized and validated all DC axioms and DC reasoning rules. The extended verification system allows users to rigorously carry out DC proofs in the way close to those by hand without knowing the detailed encoding.

- An important goal of formal specification languages is to improve the usage of industrial tools, which often have informal semantics and lack verification power. We explored the application of TIC to complement Simulink which is applied widely in industry for modeling and simulating dynamic systems.

  Using Simulink, systems are denoted by wired block diagrams which represent a set of time-dependent mathematical functions. Elementary blocks are created from Simulink library blocks by the parameterization technique. Unfortunately, these library blocks are described informally and sometimes partially in their original documentation. It is hence necessary to precisely model the functionalities of Simulink library blocks.

  We constructed a library of TIC functions in Chapter 6 to capture the essential characteristic of Simulink library blocks, that is, the time-dependent mathematical relationship between input(s) and output(s) denoted by a library block. To the best of our knowledge, our work is the first attempt to model Simulink library blocks in terms of continuous time. Our TIC library functions cover a wide range of library blocks, to be specific, 51 library blocks from 10 categories including *continuous*, *discrete*, *signal routing* and so on. Moreover, from our rigorous procedure of constructing and validating these library functions, we discovered incomplete semantics and a bug of those Simulink library blocks.

- Based on the above work of modeling Simulink library blocks, we designed a

formal framework in Chapters 7 and 8 to model and validate Simulink diagrams. This framework also makes use of the verification system for TIC developed early to ease proof complexity.

Chapter 7 presented a strategy to automatically transform Simulink diagrams to TIC models. The transformation preserves the functional and timing aspects, and it can derive unspecified sample times of complex Simulink diagrams. Conditionally executed subsystems, specifically, *Enabled* subsystems and *Triggered* subsystems, are also supported.

We further defined supplementary reasoning rules dedicated to Simulink modeling features in Chapter 8. With the framework, we can specify important (timing) requirements of a Simulink diagram or some of its components, and specify environment properties in TIC when the system denoted by the diagram is *open*. By the formal verification power of TIC, we can rigorously conduct validation beyond Simulink, such as analyzing *open* systems and reasoning about bounded timing response requirements.

In summary, the research work in this thesis attempts to develop a verification system based on the generic theorem prover PVS to support the highly expressive interval-based specification language TIC with a high grade of automation. It also generalizes the verification system to cope with another interval-based specification language DC, and facilitates the usage of TIC in complementing Simulink.

9.2 Future Work Directions

Based on this thesis, there are a number of directions for future work that may be beneficial to the verification system of interval-based specification languages and its

applications. In this section, some of these possible research directions are briefly discussed.

9.2.1 Higher Automation for Verifying TIC Models

We developed a verification system in Chapters 3 and 4.1 to show the feasibility and advantages of exploiting the powerful theorem prover PVS to provide machine-assisted proof support for TIC, in particular, the support of the analysis of both continuous and discrete behavior over arbitrary (infinite) intervals of continuous time. Proofs of TIC models are not fully automatic in general, as this is the price to be paid for the TIC highly expressive power. However, we can elevate the automation degree of TIC proofs in the following aspects.

Define more intelligent proof strategies We defined 11 proof strategies in Chapter 3.3 to make TIC proofs more automated. These strategies were designed according to the encoding of TIC semantics in PVS. Their main purpose was to save users from explicitly mastering the detailed encoding. Nevertheless, these strategies offer limited help for handling TIC verification.

We are inspired by the work [7, 8, 91] which developed a tool TAME (Timed Automata Modeling Environment) based upon PVS, allowing users to specify and prove properties of three classes of automata without great effort. TAME utilized the capability of proof strategies in PVS and provided a collection of high level strategies dedicated to assisting mechanized proofs of particular properties, specifically, proofs of invariants and of weak refinement by means of induction.

Currently, we are in the process of developing more intelligent strategies to implement heuristics found from our experiments. One of the challenges for automating proofs is to instantiate appropriate interval values and time point values so as to eliminate

the quantifiers which denote intervals and time points respectively in a proof sequent. For example, when assigning a time point within an interval during a proof is needed, the proof usually can be successfully discharged in one of the following ways of the assignment: one of the endpoints or the middle point of the interval. Another challenge is to choose proper TIC reasoning rules, supplementary rules, and mathematical laws. In the future, we aim to develop strategies to enable the verification system to handle TIC proofs with a higher degree of automation, such as backtracking, choosing the next proof command, treating generated sub-goals in distinct ways and so on.

Model checking TIC Model checking is an important and practical technique for developing correct systems by automatically discovering bugs, although the technique is limited with respect to infinite state systems, for example, continuous time used in TIC. Several approaches [52, 60, 90] discussed the possibility of model checking another interval-based specification language, that is, DC. Their algorithms were designed for a particular subset of DC based on the decidability of DC [138], though the subset they work on restricts the expressive power of DC, for instance, [60] concerning discrete time rather than continuous time, and [52] handling systems with periodical behavior.

We are interested in exploring the feasibility of model checking TIC by identifying a suitable subset of TIC which is decidable and is still expressive enough to model systems in general. We are studying the literature [138] to investigate the effect of TIC basic constructs, especially the integrator operator and the concatenation operator, on the decidability of TIC. Once the subset is identified, we plan to implement our model checking algorithm by applying popular model checkers of real-time systems, such as Kronos [18] and Uppaal [13]. We will also study the work [2] which showed the regularity of a restricted subclass of hybrid automata [66], especially its support of expressive guards in the differential equation format.

Recently, SRI has developed several tools to support infinite state systems at a certain level. The latest PVS [103] includes a model checker and integrates the predicate abstraction technique which can semi-automatically map an infinite system to a finite system. The SAL toolkit [41] contains an infinite bounded model checker which allows state variables to be real or unbounded integers. Infinite state systems can be represented in SAL as formulations of continuous or real-time behavior. One of our goals is to observe the applicability of these tools to directly handle TIC proofs.

9.2.2 Further Development for Supporting Simulink Diagrams

We constructed a formal framework to model and validate Simulink diagrams in Chapters 6, 7 and 8. With TIC expressiveness, we defined a set of TIC semantic functions to represent 51 Simulink library blocks (out of 130 total library blocks [84]) of 10 categories. From our rigorous procedure of constructing and validating these library functions, we discovered incomplete semantics and one bug of those library blocks in their original descriptions [84]. This shows that, with a complete coverage of all Simulink library blocks, our work has a potential to enhance the overall reliability of Simulink. Having complete coverage is our future research work, which is not trivial as we have to understand the timed-dependent semantics of every library block and some of those are domain-specific. Moreover, it requires continuous effort to deal with the constantly update of Simulink library blocks.

Stateflow [87] is another important product of the MathWorks. It allows hierarchical state-machine diagrams to be combined with flowchart diagrams in a flexible way. It becomes desirable to include the formal verification for Stateflow diagrams, as Simulink and Stateflow are complementary and they are increasingly used as an integrated tool-chain in many applications. However, Stateflow raises many more semantic problems than Simulink because of its informal definition. Its documentation [87]

(which is 896 pages long) describes the semantics by showing some specific simulation results. Existent work [59] and [58] attempted to propose operational semantics and denotational semantics, respectively, with respect to a subset of Stateflow. There were several tools, such as SF2SMV [11], SF2SAL [59], and SF2Lustre [115], which aimed to support formal analysis of Stateflow. These tools focused on checking state reachability of a particular subset of Stateflow.

We plan to apply a recently developed model checker, PAT [76, 125, 126, 127], to analyze Stateflow diagrams. The analysis will include the checking of reachability, deadlock and refinement, and the verification of LTL formulae. Furthermore, PAT's strength is in verifying liveness properties under fairness assumptions, such as weak fairness and strong local/global fairness. We remark that fairness assumptions are often necessary in system verification, whereas existing languages and tools, for instance, Stateflow, have limited capabilities to support fairness modeling as well as verification.

Other development plans that are worthy of pursuit are to (1) integrate the analysis results of Simulink and of Stateflow[1], and (2) develop our proposed verification tools as plug-ins to Simulink.

9.2.3 Expanding the Verification System

So far the verification system developed in this thesis can facilitate the validation of complex systems, where both system designs and requirements are specified in TIC and the validation exploits the TIC reasoning rules defined in [50]. In [82], refinement

[1]Recently, there were some efforts on composing different analysis techniques. For example, [107] proposed an interfacing technique to combine purely functional analysis schemes with state-based modeling and analysis methods.

laws of TIC were defined to structure specifications and develop designs and programs: to allow a system to be decomposed into components that may be implemented in hardware or software, or a combination of both; and to allow refinement to various combinations of components including piping, parallelism, etc. We can encode these laws to the verification system and in turn to support refinement checking during system development, for example, rigorously deriving specifications for a system from the requirements of the system [63, 73].

Chapter 5 demonstrated the extension of the verification system by supporting the basic DC, which adopts real numbers to model time and uses Boolean-valued functions of time to model states of real-time systems. We aim to expand the verification system to handle other variants of the basic DC: Mean Value Calculus [140] replaces integrals of Boolean-valued functions by their mean values so as to describe properties over point intervals, and Extended Duration Calculus [141] investigates how DC can be combined with real analysis. Furthermore, dealing with DC refinement [111] is also one of our aims.

Looking at the applicability of the verification system to other formal specification languages which integrate interval-based specification languages is also a future research direction. A possible candidate is Real-time Object-Z [122], an integration of the object-oriented, state-based specification language Object-Z [121] with TIC.

Bibliography

[1] M. M. Adams and P. B. Clayton. ClawZ: Cost-effective formal verification for control systems. In *ICFEM'05: Proceedings of the 7th International Conference on Formal Engineering Methods*, pages 465–479. Springer, 2005.

[2] M. Agrawal, F. Stephan, P. S. Thiagarajan, and S. Yang. Behavioural approximations for restricted linear differential Hybrid Automata. In *HSCC'06: Proceedings of the 9th International Workshop on Hybrid Systems: Computation and Control*, pages 4–18. Springer, 2006.

[3] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *ICALP'90: Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 322–335. Springer, 1990.

[4] R. Alur and D. L. Dill. The theory of timed automata. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 45–73, London, UK, 1992. Springer-Verlag.

[5] R. Alur and T. A. Henzinger. A really temporal logic. In *FOCS'89: Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 164–169. IEEE Computer Society, 1989.

[6] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106. Springer-Verlag, 1991.

[7] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000.

[8] M. Archer, C. L. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.

[9] M. Archer, B. D. Vito, and C. Muñoz. Developing user strategies in PVS: A tutorial. In *STRATA'03: Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics*, pages 16–42, 2003.

[10] R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. Clawz: Control laws in Z. In *ICFEM'00: Proceedings of the 3rd International Conference on Formal Engineering Methods*, pages 169–176. IEEE Computer Society, 2000.

[11] C. Banphawatthanarak, B. H. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *Proceedings of the 10th International Symposium on Computer Aided Control System Design*, pages 581–586. IEEE, 1999.

[12] P. A. Barnard. Graphical techniques for aircraft dynamic model development. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*. American Institute of Aeronautics and Astronautics, August 2004.

[13] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control*, pages 232–243. Springer, 1995.

[14] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1):37–68, 1999.

[15] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *TACAS'00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 378–394. Springer-Verlag, 2000.

[16] J. P. Bowen and M. J. C. Gordon. Z and HOL. In *Z User Workshop*, pages 141–167. Springer, 1994.

[17] J. P. Bowen and M. J. C. Gordon. A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5-6), 1995.

[18] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *CAV'98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 546–550. Springer, 1998.

[19] R. W. Butler. Formalization of the integral calculus in the PVS theorem prover. Technical report, NASA Langley Research Center, Virginia, October 2004.

[20] V. Carreño and C. Muñoz. Safety verification of the small aircraft transportation system concept of operations. In *Proceedings of the 5th Aviation Technology Integration and Operations Conference*. American Institution of Aeronautics and Astronautics, 2005.

[21] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. In *EMSOFT'03: Proceedings of the 3rd International Conference on Embedded Software*, pages 84–99. Springer, 2003.

[22] A. Cavalcanti, P. Clayton, and C. O'Halloran. Control law diagrams in Circus. In *FM'05: Proceedings of the 13th International Symposium of Formal Methods Europe*, pages 253–268. Springer, 2005.

[23] A. Cerone. Axiomatisation of an interval calculus for theorem proving. *Electronic Notes in Theoretical Computer Science*, 42, 2001.

[24] G. Chakravorty and P. K. Pandya. Digitizing interval duration logic. In *CAV'03: Proceedings of the 15th International Conference on Computer Aided Verification*, pages 167–179. Springer, 2003.

[25] C.-L. Chang and R. C.-T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., 1997.

[26] C. Chen. A continous-time approach to modeling and validating Simulink models. In the Doctoral Symposium of the 14th International Symposium on Formal Methods, 2006.

[27] C. Chen. A Verification System for interval-based specification languages, 2008. `http://www.comp.nus.edu.sg/~chenchun/verifysys`.

[28] C. Chen and J. S. Dong. Applying Timed Interval Calculus to Simulink diagrams. In *ICFEM'06: Proceedings of the 8th International Conference on Formal Engineering Methods*, pages 74–93. Springer, 2006.

[29] C. Chen, J. S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing*. Accepted.

[30] C. Chen, J. S. Dong, and J. Sun. A Formal Framework for Modeling and Verifying Simulink Diagrams, 2007. `http://www.comp.nus.edu.sg/~chenchun/SimInTIC`.

[31] C. Chen, J. S. Dong, and J. Sun. Machine-assisted proof support for validation beyond Simulink. In *ICFEM'07: Proceedings of the 9th International Conference on Formal Engineering Methods*, pages 96–115. Springer, 2007.

[32] C. Chen, J. S. Dong, and J. Sun. A verification system for timed interval calculus. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 271–280. ACM, 2008.

[33] C. Chen, J. S. Dong, J. Sun, and A. Martin. A verification system for interval-based specification languages. *ACM Transactions on Software Engineering and Methodology*. Accepted.

[34] B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *FOSE'07: 2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.

[35] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An open source tool for symbolic model checking. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 359–364. Springer-Verlag, 2002.

[36] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[37] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 2000.

[38] E. M. Clarke and J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996. Other working group members: R. Alur, R. Cleaveland, D. Dill, A. Emerson, S. Garland, S. German, J. Guttag, A. Hall, T. Henzinger, G. Holzmann, C. Jones, R. Kurshan, N. Leveson, K. McMillan, J. Moore, D. Peled, A. Pnueli, J. Rushby, N. Shankar, J. Sifakis, P. Sistla, B. Steffen, P. Wolper, J. Woodcock and P. Zave.

[39] J. Davies and S. Schneider. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.

[40] J. E. Dawson and R. Goré. Machine-checking the Timed Interval Calculus. In *AI'02: Proceedings of the 15th Australian Joint Conference on Artificial Intelligence*, pages 95–106. Springer-Verlag, 2002.

[41] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *CAV'04: Proceedings of the 16th International Conference on Computer Aided Verification*, pages 496–500. Springer, 2004.

[42] A. Dimov and S. Ilieva. System level modeling of component based software systems. In *CompSysTech'04: Proceedings of the 5th international conference on Computer systems and technologies*, pages 1–6. ACM, 2004.

[43] J. S. Dong, P. Hao, X. Zhang, and S. Qin. HighSpec: a tool for building and checking OZTA models. In *ICSE'07: Proceedings of the 28th International Conference on Software Engineering*, pages 775–778. ACM, 2006.

[44] R. C. Dorf and R. H. Bishop. *Modern Control Systems*. Prentice Hall, 9th edition, 2000.

[45] R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Macmillan, 2000.

[46] B. Dutertre. Elements of mathematical analysis in PVS. In *TPHOLs'96: Proceedings of the 9th Internaional Conference on Theorem Proving in Higher Order Logics*, pages 141–156. Springer, 1996.

[47] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, 1997.

[48] C. J. Fidge. Modelling discrete behaviour in a continuous-time formalism. In *IFM'99: Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 170–188. Springer, 1999.

[49] C. J. Fidge, I. J. Hayes, and B. P. Mahony. Defining differentiation and integration in Z. In *ICFEM'98: Proceedings of the 2nd International Conference on Formal Engineering Methods*, pages 64–73. IEEE Computer Society, 1998.

[50] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In *MPC'98: Proceedings of the 4th International Conference on Mathematics of Program Construction*, pages 188–206. Springer, 1998.

[51] Ford. Structure analysis using Matlab/Simulink/Stateflow - modeling style guidelines. Technical report, Ford Motor Company, 1999.

[52] M. Fränzle. Model-checking dense-time Duration Calculus. *Formal Aspects of Computing*, 16(2):121–139, 2004.

[53] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem-proving environment for higher-order logic*. Cambridge University Press, New York, NY, USA, 1993.

[54] A. M. Gravell and C. H. Pratten. Embedding a formal notation: Experiences of automating the embedding of Z in the higher order logics of PVS and HOL. In *TPHOLs'98: Proceedings of 11th International Conference on Theorem Proving in Higher Order Logics, supplementary proceedings*, pages 73–84. Springer, 1998.

[55] D. Griffioen and M. Huisman. A comparison of PVS and Isabelle/HOL. In *TPHOLs'98: Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, pages 123–142. Springer, 1998.

[56] S. Gupta, B. H. Krogh, and R. A. Rutenbar. Towards formal verification of analog designs. In *ICCAD'04: Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, pages 210–217. IEEE Computer Science, 2004.

[57] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Septebber 1991.

[58] G. Hamon. A denotational semantics for Stateflow. In *EMSOFT'05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 164–172. ACM, 2005.

[59] G. Hamon and J. M. Rushby. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer*, 9(5-6):447–456, 2007.

[60] M. R. Hansen. Model-checking discrete Duration Calculus. *Formal Aspects of Computing*, 6(6A):826–845, 1994.

[61] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[62] J. Harrison. Theorem proving for verification (invited tutorial). In *CAV'08: Proceedings of the 20th International Conference on Computer Aided Verification*, pages 11–18. Springer, 2008.

[63] I. J. Hayes, M. A. Jackson, and C. B. Jones. Determining the specification of a control system from that of its environment. In *FM'03: Proceedings of the 12th International Symposium of Formal Methods Europe*, pages 154–169. Springer, 2003.

[64] S. T. Heilmann. *Proof Support for Duration Calculus*. PhD thesis, Department of Information Technology, Technical University of Denmark, January 1999.

[65] C. L. Heitmeyer, J. Kirby, B. G. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, 1998.

[66] T. A. Henzinger. The theory of hybrid automata. In *LICS'96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society, 1996.

[67] T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM'06: Proceedings of the 14th International Symposium on Formal Methods*, pages 1–15. Springer, 2006.

[68] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria. Software engineering and formal methods. *Communications of the ACM*, 51(9):54–59, 2008.

[69] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(9):890–904, 1986.

[70] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, March 2005.

[71] J.-J. Jeng and B. H. C. Cheng. Specification matching for software reuse: a foundation. In *SSR'95: Proceedings of the 1995 Symposium on Software reusability*, pages 97–105. ACM, 1995.

[72] M. Jersak, Y. Cai, D. Ziegenbein, and R. Ernst. A transformational approach to constraint relaxation of a time-driven simulation model. In *ISSS'00: Proceedings of the 13th International Symposium on System Synthesis*, pages 137–142. IEEE Computer Society, 2000.

[73] C. B. Jones, I. J. Hayes, and M. A. Jackson. Deriving specifications for systems that are connected to the physical world. In *Formal Methods and Hybrid Real-Time Systems*, pages 364–390. Springer, 2007.

[74] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, June 2000.

[75] P. A. Laplante. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. IEEE Press, 2nd edition, 1997.

[76] Y. Liu, J. Sun, and J. S. Dong. An analyzer for extended compositional process algebras. In *ICSE Companion'08: Companion of the 30th International Conference on Software Engineering*, pages 919–920. ACM, 2008.

[77] R. L. London and K. R. Milsted. Specifying reusable components using Z: realistic sets and dictionaries. In *IWSSD'89: Proceedings of the 5th International Workshop on Software Specification and Design*, pages 120–127. ACM, 1989.

[78] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *ICSE'98: Proceedings of the 20th International Conference on Software Engineering*, pages 95–104. IEEE Computer Society, 1998.

[79] B. P. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

[80] B. P. Mahony and J. S. Dong. Deep semantic links of TCSP and Object-Z: TCOZ approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.

[81] B. P. Mahony and I. J. Hayes. A case study in timed refinement: A central heater. In *Proceedings of the BCS/FACS 4th Refinement Workshop*, pages 138–149. Springer, 1991.

[82] B. P. Mahony and I. J. Hayes. A case-study in timed refinement: A mine pump. *IEEE Transactions on Software Engineering*, 18(9):817–826, 1992.

[83] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[84] The MathWorks. *Simulink® 7 - Reference*, March 2008.

[85] The MathWorks. *Simulink® 7 - Using Simulink*, March 2008.

[86] The MathWorks. *Simulink Design Verifier*, 2008.

[87] The MathWorks. *Stateflow® and Stateflow® coder™ 7 - User's Guide*, March 2009.

[88] R. Mattolini and P. Nesi. An interval logic for real-time system specification. *IEEE Transactions on Software Engineering*, 27(3):208–227, 2001.

[89] B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for translating Simulink models into input language of a model checker. In *ICFEM'06: Proceedings of the 8th International Conference on Formal Engineering Methods*, pages 606–620. Springer, 2006.

[90] R. Meyer, J. Faber, J. Hoenicke, and A. Rybalchenko. Model checking Duration Calculus: a practical approach. *Formal Aspects of Computing*, 20(4-5):481–505, 2008.

[91] S. Mitra and M. Archer. PVS strategies for proving abstraction properties of automata. *Electronic Notes in Theoretical Computer Science*, 125(2):45–65, 2005.

[92] G. Moretti. Design complexity requires system-level design. *Electronics Design, Strategy, and News*, pages 28–32, March 2005.

[93] L. E. Moser, P. M. Melliar-Smith, Y. S. Ramakrishna, G. Kutty, and L. K. Dillon. The real-time graphical interval logic toolset. In *CAV'96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 446–449, 1996.

[94] B. C. Moszkowski. *Executing temporal logic programs.* Cambridge University Press, New York, NY, USA, 1986.

[95] B. C. Moszkowski. A complete axiomatization of Interval Temporal Logic with infinite time. In *LICS'00: Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, pages 241–252. IEEE Computer Society, 2000.

[96] C. Muñoz, V. Carreño, and G. Dowek. Formal analysis of the operational concept for the Small Aircraft Transportation System. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 306–325. Springer, 2006.

[97] C. Muñoz, V. Carreño, G. Dowek, and R. W. Butler. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer*, 4(3):371–380, 2003.

[98] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 549–572. Springer-Verlag, 1992.

[99] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[100] J. S. Ostroff. *Temporal logic for real time systems*. John Wiley & Sons, Inc., New York, NY, USA, 1989.

[101] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[102] S. Owre and N. Shankar. Writing PVS proof strategies. In *STRATA'03: Proceedings of Design and Application of Strategies/Tactics in Higher Order Logics*, pages 1–15, 2003.

[103] S. Owre and N. Shankar. A brief overview of PVS. In *TPHOLs'08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 22–27. Springer, 2008.

[104] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. SRI International, November 2001.

[105] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. SRI International, November 2001.

[106] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[107] L. T. X. Phan, S. Chakraborty, P. S. Thiagarajan, and L. Thiele. Composing functional and state-based performance models for analyzing heterogeneous real-time systems. In *RTSS'07: Proceedings of the 28th IEEE Real-Time Systems Symposium*. IEEE Computer Society.

[108] A. Pnueli. Embedded systems: Challenges in specification and verification. In *EMSOFT'02: Proceedings of the 2nd International Conference on Embedded Software*, pages 1–14. Springer, 2002.

[109] S. Qin, J. S. Dong, and W.-N. Chin. A semantic foundation for tcoz in unifying theories of programming. In *FM'03: Proceedings of the 12th International Symposium of Formal Methods Europe*, pages 321–340. Springer, 2003.

[110] T. M. Rasmussen. *Interval logic - Proof Theory and Theorem Proving*. PhD thesis, Technical University of Denmark, 2002.

[111] A. P. Ravn. *Design of Embedded Real-Time Computing Systems*. PhD thesis, Technical University of Denmark, 1995.

[112] J. M. Rushby. Formal methods and their role in the certification of critical systems. Technical report, SRI International, mar 1995. Also available as part of the FAA Digital Systems Validation Handbook (the guide for aircraft certification).

[113] J. M. Rushby. Theorem proving for verification. In *MOVEP'00: Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, pages 39–57. Springer, 2000.

[114] J. M. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, 1998.

[115] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In *EMSOFT'04: Proceedings of the 4th International Conference on Embedded Software*, pages 259–268. ACM, 2004.

[116] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[117] B. I. Silva and B. H. Krogh. Formal verification of hybrid systems using Checkmate: a case study. In *Proceedings of the American Control Conference*, pages 1679–1683, 2000.

[118] S. Sims, R. Cleaveland, K. Butts, and S. Ranville. Automated validation of software models. In *ASE'01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 91–96. IEEE Computer Society, 2001.

[119] J. U. Skakkebaek. *A Verification Assistant for a Real-Time Logic*. PhD thesis, Department of Computer Science, Technical University of Denmark, 1994.

[120] J. U. Skakkebæk and N. Shankar. Towards a Duration Calculus proof assistant in PVS. In *FTRTFT'94: Proceedings of the 3rd International Symposium Or-*

ganized on Formal Techniques in Real-Time and Fault-Tolerant Systems, pages 660–679. Springer, 1994.

[121] G. Smith. *The Object-Z specification language*. Kluwer Academic Publishers, 2000.

[122] G. Smith and I. J. Hayes. An introduction to real-time Object-Z. *Formal Aspects of Computing*, 13(2):128–141, 2002.

[123] J. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[124] D. W. J. Stringer-Calvert, S. Stepney, and I. Wand. Using PVS to prove a Z refinement: A case study. In *FME'97: Proceedings of the 4th International Symposium of Formal Methods Europe*, pages 573–588. Springer, 1997.

[125] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV'09: Proceedings of the 21th International Conference on Computer Aided Verification*, pages 709–714. Springer, 2009.

[126] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and verifying event-based fairness enhanced systems. In *ICFEM'08: Proceedings of the 10th International Conference on Formal Engineering Methods*, pages 5–24. Springer, 2008.

[127] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking of parameterized systems. In *FM'09: the 16th International Symposium of Formal Methods*, 2009. Accepted.

[128] A. Tiwari, N. Shankar, and J. M. Rushby. Invisible formal methods for embedded control systems. *Proceedings of the IEEE*, 91(1):29–39, 2003.

[129] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 4(4):779–818, 2005.

[130] N. Völker. Two semantic embeddings of Z schemas in Isabelle/HOL. Technical report, Department of Computer Science, University of Essex, 2001.

[131] A. Wabenhorst. Induction in the Timed Interval Calculus. *Theoretical Computer Science*, 300(1-3):181–207, 2003.

[132] F. Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, August 2004.

[133] F. Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*. Springer, 2006.

[134] L. Wildman. Requirements reformulation using formal specification: A case study. In *CRPIT'02: Proceedings of the Conference on Application and Theory of Petri Nets*, pages 75–83. Australian Computer Society, Inc., 2002.

[135] J. Woodcock. Using Circus for safety-critical applications. *Electronic Notes in Theoretical Computer Science*, 95:3–22, 2004.

[136] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[137] C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer-Verlag, 2004.

[138] C. Zhou, M. R. Hansen, and P. Sestoft. Decidability and undecidability results for Duration Calculus. In *STACS'93: Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 58–68. Springer, 1993.

[139] C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[140] C. Zhou and X. Li. A Mean Value Calculus of durations. In *A classical mind: essays in honour of C. A. R. Hoare*, pages 431–451. Prentice-Hall International (UK) Ltd., 1994.

[141] C. Zhou, A. P. Ravn, and M. R. Hansen. An extended duration calculus for hybrid real-time systems. In *Hybrid Systems*, pages 36–59. Springer, 1993.

[142] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI - a system model for heterogeneously specified embedded systems. *IEEE Transactions on Very Large Scale Integration Systems*, 10(4):379–389, 2002.

Appendix A

Encoding of TIC in PVS

A.1 Basic Definitions

The encoding of TIC semantics is defined in a series of PVS theories.

The domains of time and interval are modeled in the `TIC_time_interval` theory.

```
TIC_time_interval : THEORY
  BEGIN
   Time: TYPE = nnreal;

   InterVal_Type: TYPE = {OO, CO, OC, CC}; % O stands for open (exclude), and C stands for close (include)

   % define the general type of valid intervals
   GenInterVal: TYPE = [invt: InterVal_Type, {stp: Time, etp: Time | stp <= etp}];

   gi: var GenInterVal;

   % II denotes the type of all possible intervals
   II: TYPE = {gi | (proj_1(gi) = OO AND proj_1(proj_2(gi)) < proj_2(proj_2(gi)))
                 or (proj_1(gi) = CO AND proj_1(proj_2(gi)) < proj_2(proj_2(gi)))
                 or (proj_1(gi) = OC AND proj_1(proj_2(gi)) < proj_2(proj_2(gi)))
                 or (proj_1(gi) = CC AND proj_1(proj_2(gi)) <= proj_2(proj_2(gi)))};

   i: var II;

   COInterVal: TYPE = {i | proj_1(i) = CO };
   OOInterVal: TYPE = {i | proj_1(i) = OO };
   OCInterVal: TYPE = {i | proj_1(i) = OC };
```

```
   CCInterVal: TYPE = {i | proj_1(i) = CC };

   % a function returns the type of a given interval
   Typeof(i): InterVal_Type = Proj_1(i);

  END TIC_time_interval
```

Three predefined interval operators, α, ω, and δ, are defined in the `TIC_interval_operators` theory. Note that the *importing* clause here is used to access entities defined in the `TIC_time_interval` theory.

```
TIC_interval_operators  : THEORY
  BEGIN

   IMPORTING TIC_time_interval;

   i: var II;
   t: var Time;

   Term: TYPE = [II -> Time]; % interval operators are defined as the type Term.

   ALPHA(i): Time = Proj_1(Proj_2(i));      % return the starting point
   OMEGA(i): Time = Proj_2(Proj_2(i));      % return the ending point
   DELTA(i): Time = OMEGA(i) - ALPHA(i);    % return the length of the interval

  End TIC_interval_operators
```

This theory models timed trace and expressions. By the name overloading technique of PVS, the function `LIFT` unifies different functionalities of primitive TIC elements to the same type of functions whose arguments are time and intervals. In the definition of integration, we adopt the function Integral from the NASA PVS library.

```
TIC_expression : THEORY
  BEGIN

   IMPORTING TIC_time_interval, TIC_interval_operators,
             analysis@derivatives[Time], % the analysis package is from the NASA PVS library
             analysis@integral[Time], analysis@continuous_functions[Time],;

   i: var II;
   t: var Time;
   tm: var Term;

   Trace: TYPE  = [Time -> real];
```

```
  % the range of a binary trace contains of only values 0 and 1.
  BTrace: TYPE = [Time -> {x: real | x = 0 OR x = 1}];

  tr: var Trace;

  TExp: TYPE = [Time, II -> real];

  % a non zero TIC expression is a special TIC expression that returns non zero
  % values, and it is used to handle the division mathematical operation.
  NZero_TExp: TYPE = { exp: TExp | forall (t, i): exp(t, i) /= 0};

  % a non negative TIC expression is a special TIC expression that returns non negative
  % values. It is used to support function composition since time value is not negative.
  NNeg_TExp: TYPE = { exp: TExp | forall(t, i): exp(t, i) >= 0};

  expl, expr: var TExp;
  nzeroexp: var NZero_TExp;
  nnegexp: var NNeg_TExp;

  % The function LIFT handles three types of TIC basic elements.
  LIFT(tr)(t, i): real = tr(t);     % a timed trace
  LIFT(x: real)(t, i): real = x;    % a constant
  LIFT(tm)(t, i): real = tm(i);     % an interval operator

  % basic mathematical operations.
  +(expl, expr)(t, i): real = expl(t, i) + expr(t, i);
  -(expl, expr)(t, i): real = expl(t, i) - expr(t, i);
  -(expl)(t, i): real = - expl(t, i);
  *(expl, expr)(t, i): real = expl(t, i) * expr(t, i);
  /(expl, nzeroexp)(t, i): real = expl(t, i) / nzeroexp(t, i);
  o (expl, nnegexp)(t, i): real = expl(nnegexp(t, i), i);

  % The following lemma requires users to prove the continuity of a timed trace
  % which is integrated.
  Integral_tr: var Trace;
  Integral_tr: LEMMA continuous(Integral_tr);
  b, e: var NNeg_TExp;
  TICIntegral( b, e, Integral_tr)(t, i): real = Integral(b(t, i), e(t, i), Integral_tr);

 END TIC_expression
```

In the following `TIC_predicate` theory, predicates are defined as PVS functions from time points and intervals to Boolean.

```
TIC_predicate : THEORY
  BEGIN

   importing TIC_expression
```

```
TPred: TYPE = [Time, II -> bool];

i: var II;
t: var Time;

TTRUE(t, i): bool = true;
FFALSE(t, i): bool = false;

expl, expr: var TExp;

% Basic (in)equations over expressions.
<(expl, expr)(t, i): bool = expl(t, i) < expr(t, i);
<=(expl, expr)(t, i): bool = expl(t, i) <= expr(t, i);
>(expl, expr)(t, i): bool = expl(t, i) > expr(t, i);
>=(expl, expr)(t, i): bool = expl(t, i) >= expr(t, i);
=(expl, expr)(t, i): bool = expl(t, i) = expr(t, i);
/=(expl, expr)(t, i): bool = expl(t, i) /= expr(t, i);

tpl, tpr: var TPred

% Typical logic operators
not(tpl)(t, i): bool = NOT tpl(t, i);
or(tpl, tpr)(t, i): bool = tpl(t, i) OR tpr(t, i);
and(tpl, tpr)(t, i): bool = tpl(t, i) AND tpr(t, i);
=>(tpl, tpr)(t, i): bool = NOT tpl(t, i) OR tpr(t, i);
<=> (tpl, tpr)(t, i): bool = (tpl => tpr)(t, i) AND (tpr => tpl)(t, i);

% Quantified operators (i.e., existential and universal operators) are used to model
% sample time intervals. A quantified predicate is a function from natural number to
% timed predicates
QuaPred: TYPE = [nat -> TPred];
qp: var QuaPred;
exNat(qp)(t, i): bool = EXISTS (k: nat): (qp(k)(t, i));
allNat(qp)(t, i): bool = FORALL (k: nat): (qp(k)(t, i));

QuaPred1: TYPE = [posnat -> TPred];
qp1: var QuaPred1;
exNat1(qp1)(t, i): bool = EXISTS (k: posnat): (qp1(k)(t, i));
allNat1(qp1)(t, i): bool = FORALL (k: posnat): (qp1(k)(t, i));

END TIC_predicate
```

The key constructor in TIC is the interval brackets. A pari of interval brackets associated with a predicate denotes a *set* of intervals and in each of which the predicate holds at every time point. Four basic types of interval brackets and one general interval type are encoded in the `TIC_IExpression` theory. The theory also specifies conventional set operators over TIC expressions and the TIC concatenation operator.

```
TIC_IExpression  : THEORY
  BEGIN

    importing TIC_predicate

    i: var II;
    oo: var OOInterVal;
    co: var COInterVal;
    oc: var OCInterVal;
    cc: var CCInterVal;
    t: var Time;
    tp: var TPred;

    % checks if a time point t is within an interval i based on the interval type
    t_in_i(t, i): bool =  (Typeof(i) = OO AND t > ALPHA(i) AND t < OMEGA(i))
                       or (Typeof(i) = OC AND t > ALPHA(i) AND t <= OMEGA(i))
                       or (Typeof(i) = CO AND t >= ALPHA(i) AND t < OMEGA(i))
                       or (Typeof(i) = CC AND t >= ALPHA(i) AND t <= OMEGA(i));

    % check if the a predicate holds at all time points within an interval
    Everywhere?(tp, i): bool =   forall t: t_in_i(t, i) => tp(t, i);

    PII: TYPE = setof[II];
    POC: TYPE = setof[OCInterVal];
    POO: TYPE = setof[OOInterVal];
    PCO: TYPE = setof[COInterVal];
    PCC: TYPE = setof[CCInterVal];

    ocsl, ocsr: var POC;
    oosl, oosr: var POO;
    cosl, cosr: var PCO;
    ccsl, ccsr: var PCC;
    iisl, iisr: var PII;

    OOS(tp): POO = {oo | Everywhere?(tp, oo)};     % (- tp -)
    OCS(tp): POC = {oc | Everywhere?(tp, oc)};     % (- tp -]
    COS(tp): PCO = {co | Everywhere?(tp, co)};     % [- tp -)
    CCS(tp): PCC = {cc | Everywhere?(tp, cc)};     % [- tp -]
    AllS(tp): PII =  {i | Everywhere?(tp, i)};     % [(- tp -)]

    % return true when a predicate holds for all intervals.
    AllTrue(tp): bool = FORALL i: Everywhere?(tp, i);

    % constrain two connected intervals in terms of interval types
    ConcatType((l, r, re: II)): bool =
      (Typeof(re) = OO AND ((Typeof(l) = OC AND Typeof(r) = OO) OR (Typeof(l) = OO AND Typeof(r) = CO)))
   OR (Typeof(re) = CO AND ((Typeof(l) = CC AND Typeof(r) = OO) OR (Typeof(l) = CO AND Typeof(r) = CO)))
   OR (Typeof(re) = OC AND ((Typeof(l) = OO AND Typeof(r) = CC) OR (Typeof(l) = OC AND Typeof(r) = OC)))
   OR (Typeof(re) = CC AND ((Typeof(l) = CO AND Typeof(r) = CC) OR (Typeof(l) = CC AND Typeof(r) = OC)));

    % the concatenation operator "^" in [(- tp1 -)] ^ [(- tp2 -)]
```

```
    concat(iisl, iisr): PII = {i | exists (i1, i2: II): member(i1, iisl) AND member(i2, iisr) AND
       OMEGA(i1) = ALPHA(i2) AND ALPHA(i1) = ALPHA(i) AND OMEGA(i2) = OMEGA(i) AND ConcatType(i1, i2, i)};

    % check whether two intervals are adjacent.
    adjacent_intervals?(i, j: II): bool = OMEGA(i) = ALPHA(j) AND
      (Typeof(i) = OO AND Typeof(j) = CO OR Typeof(i) = OO AND Typeof(j) = CC
    or Typeof(i) = OC AND Typeof(j) = OO OR Typeof(i) = OC AND Typeof(j) = OC
    or Typeof(i) = CO AND Typeof(j) = CO OR Typeof(i) = CO AND Typeof(j) = CC
    or Typeof(i) = CC AND Typeof(j) = OO OR Typeof(i) = CC AND Typeof(j) = OC);

    % compose two connected intervals to form an interval
    TwoTOneInterval((l: II), (r: {tmp: II | adjacent_intervals?(l, tmp)})): II =
        IF Typeof(l) = OO AND Typeof(r) = CO OR Typeof(l) = OC AND Typeof(r) = OO
           THEN (OO, (ALPHA(l), OMEGA(r)))
           ELSIF Typeof(l) = OO AND Typeof(r) = CC OR Typeof(l) = OC AND Typeof(r) = OC
               THEN (OC, (ALPHA(l), OMEGA(r)))
               ELSIF Typeof(l) = CO AND Typeof(r) = CO OR Typeof(l) = CC AND Typeof(r) = OO
                   THEN (CO, (ALPHA(l), OMEGA(r))) ELSE (CC, (ALPHA(l), OMEGA(r)))
        ENDIF;

    % conventional set operations between two sets of intervals.
    TIC_U(oosl, oosr): POO = union(oosl, oosr);
    TIC_U(ocsl, ocsr): POC = union(ocsl, ocsr);
    TIC_U(cosl, cosr): PCO = union(cosl, cosr);
    TIC_U(ccsl, ccsr): PCC = union(ccsl, ccsr);
    TIC_U(iisl, iisr): PII = union(iisl, iisr);
    TIC_I(oosl, oosr): POO = intersection(oosl, oosr);
    TIC_I(ocsl, ocsr): POC = intersection(ocsl, ocsr);
    TIC_I(cosl, cosr): PCO = intersection(cosl, cosr);
    TIC_I(ccsl, ccsr): PCC = intersection(ccsl, ccsr);
    TIC_I(iisl, iisr): PII = intersection(iisl, iisr);

  End TIC_IExpression
```

A.2 TIC Reasoning Rules

We have formalized and validated all TIC reasoning rules [50]. The following `TIC_rules` theory contains the PVS specifications of those reasoning rules.

```
TIC_rules: THEORY
  BEGIN

   IMPORTING TIC_IExpression, analysis@continuous_functions_props[Time];

   tr1, tr2: VAR Trace;
   i: VAR II;
```

```
t: VAR Time;
tp1, tp2, tp3, tp4: VAR TPred;

% return true if a predicate is independent of interval operators.
No_Term?(tp1): bool = FORALL (i1, i2: II): FORALL t :
   t_in_i(t, i1) AND t_in_i(t, i2) => tp1(t, i1) = tp1(t, i2);

% return true if a predicate is independent of time points within an interval
Only_Interval?(tp1): bool = FORALL i: FORALL (t1, t2: Time):
   t_in_i(t1, i) AND t_in_i(t2, i) => tp1(t1, i) = tp1(t2, i);

%%%% Monotonic rules %%%%
MONO_RULE: LEMMA AllTrue(tp1 => tp2) IMPLIES subset?(AllS(tp1), AllS(tp2));
MONO_RULE_CO: LEMMA AllTrue(tp1 => tp2) IMPLIES subset?(COS(tp1), COS(tp2));
MONO_RULE_OC: LEMMA AllTrue(tp1 => tp2) IMPLIES subset?(OCS(tp1), OCS(tp2));
MONO_RULE_OO: LEMMA AllTrue(tp1 => tp2) IMPLIES subset?(OOS(tp1), OOS(tp2));
MONO_RULE_CC: LEMMA AllTrue(tp1 => tp2) IMPLIES subset?(CCS(tp1), CCS(tp2));

%%%% AND rules %%%%
AND_RULE_OO: LEMMA TIC_I(OOS(tp1), OOS(tp2)) = OOS(tp1 and tp2);
AND_RULE_CO: LEMMA TIC_I(COS(tp1), COS(tp2)) = COS(tp1 and tp2);
AND_RULE_OC: LEMMA TIC_I(OCS(tp1), OCS(tp2)) = OCS(tp1 and tp2);
AND_RULE_CC: LEMMA TIC_I(CCS(tp1), CCS(tp2)) = CCS(tp1 and tp2);
AND_RULE: LEMMA TIC_I(AllS(tp1), AllS(tp2)) = AllS(tp1 and tp2);

%%%% Or rules %%%%
OR_RULE: LEMMA subset?(TIC_U(AllS(tp1), AllS(tp2)), AllS(tp1 or tp2));
OR_RULE_OO: LEMMA subset?(TIC_U(OOS(tp1), OOS(tp2)), OOS(tp1 or tp2));
OR_RULE_CO: LEMMA subset?(TIC_U(COS(tp1), COS(tp2)), COS(tp1 or tp2));
OR_RULE_OC: LEMMA subset?(TIC_U(OCS(tp1), OCS(tp2)), OCS(tp1 or tp2));
OR_RULE_CC: LEMMA subset?(TIC_U(CCS(tp1), CCS(tp2)), CCS(tp1 or tp2));

%%%% properties of fullset and empty set %%%%
True_n_Uni : LEMMA AllS(TTRUE) = fullset;
False_n_Emp : LEMMA AllS(FFALSE) = emptyset;

% [(- tp1 -)] = fullset => [(- not tp1 -)] = emptyset
All_to_Emp: LEMMA AllS(tp1) = fullset => AllS(not tp1) = emptyset;

% [(- tp1 -)] = fullset => ALPHA, OMEGA, DELTA don't occur free in tp1
All_to_Emp_noterm: LEMMA AllS(tp1) = fullset => No_Term?(tp1);

% if ALPHA(i) = OMEGA(i), then forall t: i, we have t = ALPHA(i)
Emp_to_All_CC: LEMMA FORALL i: ALPHA(i) = OMEGA(i) => (FORALL t: t_in_i(t, i) => t = ALPHA(i));

% if ALPHA, OMEGA, DELTA don't occur freely in tp1,
% then [(- not tp1 -)] = emptyset => [(- tp1 -)] = fullset
Emp_to_All: LEMMA No_Term?(tp1) => (AllS(not tp1) = emptyset => AllS(tp1) = fullset);

Emp_toAll_only_interval: LEMMA Only_Interval?(tp1) => (AllS(not tp1) = emptyset => AllS(tp1) = fullset);
```

```
% if tp is term-free, then [- tp -] = emptyset => [(- not tp -)] = fullset
EmpCC_to_All: LEMMA No_Term?(tp1) => (CCS(tp1) = emptyset => AllS(not tp1) = fullset);

%%%% Not rule %%%%
% DIFF_RULE shows that [(- not tp -)] subset (II \ [(- tp -)])
DIFF_RULE : LEMMA subset?(AllS(not tp1), difference(fullset, AllS(tp1)));

co1, co2, co3, co4: var COInterVal;
cos1, cos2, cos3, cos4: var PCO;
ocs1, ocs2: var POC;
ccs1, ccs2: var PCC;
i1, i2, i3, i4: var PII;

%%%% Concatenation monotonicity %%%%
CONC_MONO_CO_CO: LEMMA subset?(cos1, cos3) AND subset?(cos2, cos4)
                    => subset?(concat(cos1, cos2), concat(cos3, cos4));
CONC_MONO_CC_OC: LEMMA subset?(ccs1, ccs2) AND subset?(ocs1, ocs2)
                    => subset?(concat(ccs1, ocs1), concat(ccs2, ocs2));
CONC_MONO_RULE: LEMMA subset?(i1, i3) AND subset?(i2, i4) => subset?(concat(i1, i2), concat(i3, i4));

%%%% Concatenation associativity %%%%
CONC_ASSO_CO_CO : LEMMA concat(concat(cos1, cos2), cos3) = concat(cos1, concat(cos2, cos3));
CONC_ASSO_RULE: LEMMA concat(concat(i1, i2), i3) =  concat(i1, concat(i2, i3));

%%%% Concatenation union %%%%
% (S uni T) ; U = (S ; U) uni (T ; U)
CONC_UNION_CO_CO_L: LEMMA concat(TIC_U(cos1, cos2), cos3) = TIC_U(concat(cos1, cos3), concat(cos2, cos3));
CONC_UNION_L: LEMMA concat(TIC_U(i1, i2), i3) =  TIC_U(concat(i1, i3), concat(i2, i3));

% S; (T uni U) = (S ; T) uni (S ; U)
CONC_UNION_CO_CO_R: LEMMA concat(cos1, TIC_U(cos2, cos3)) = TIC_U(concat(cos1, cos2), concat(cos1, cos3));
CONC_UNION_R: LEMMA concat(i1, TIC_U(i2, i3)) = TIC_U(concat(i1, i2), concat(i1, i3));

%%%% Concatenation intersection %%%%
% (S int T) ; U subseteq (S ; U) int (T ; U)
CONC_INTER_CO_CO_L: LEMMA subset?(concat(TIC_I(cos1, cos2), cos3),
                                  TIC_I(concat(cos1, cos3), concat(cos2, cos3)));
CONC_INTER_L: LEMMA subset?(concat(TIC_I(i1, i2), i3), TIC_I(concat(i1, i3), concat(i2, i3)));

% S ; (T int U) subseteq (S ; T) int (S ; U)
CONC_INTER_CO_CO_R: LEMMA subset?(concat(cos1, TIC_I(cos2, cos3)),
                                  TIC_I(concat(cos1, cos2), concat(cos1, cos3)));
CONC_INTER_R: LEMMA subset?(concat(i1, TIC_I(i2, i3)), TIC_I(concat(i1, i2), concat(i1, i3)));

%%%% Concatenation with fixed-length intersection %%%%
CONC_FIX_r: var posreal;

%   ((S int [- delta = r -)) ; T) int ((U int [- delta = r -)) ; V)
% = (S int U int [- delta = r -)) ; (T int V)
CONC_FIX_CO_CO_L: LEMMA
   TIC_I(concat(TIC_I(cos1, COS(LIFT(DELTA) = LIFT(CONC_FIX_r))), cos2),
```

```
          concat(TIC_I(cos3, COS(LIFT(DELTA) = LIFT(CONC_FIX_r))), cos4))
   =  concat(TIC_I(TIC_I(cos1, cos3), COS(LIFT(DELTA) = LIFT(CONC_FIX_r))), TIC_I(cos2, cos4));
   CONC_FIX_L: LEMMA
      TIC_I(concat(TIC_I(i1, AllS(LIFT(DELTA) = LIFT(CONC_FIX_r))), i2),
          concat(TIC_I(i3, AllS(LIFT(DELTA) = LIFT(CONC_FIX_r))), i4))
   =  concat(TIC_I(TIC_I(i1, i3), AllS(LIFT(DELTA) = LIFT(CONC_FIX_r))), TIC_I(i2, i4));

   %   (S ; (U int [- delta = r -))) int (T ; (V int [- delta = r -)))
   % = (S int T) ; (U int V int [- delta = r -))
   CONC_FIX_CO_CO_R: LEMMA
      TIC_I(concat(cos1, TIC_I(cos2, COS(LIFT(DELTA) = LIFT(CONC_FIX_r)))),
          concat(cos3, TIC_I(cos4, COS(LIFT(DELTA) = LIFT(CONC_FIX_r)))))
   =  concat(TIC_I(cos1, cos3), TIC_I(TIC_I(cos2, cos4), COS(LIFT(DELTA) = LIFT(CONC_FIX_r))));
   CONC_FIX_R: LEMMA
      TIC_I(concat(i1, TIC_I(i2, AllS(LIFT(DELTA) = LIFT(CONC_FIX_r)))),
          concat(i3, TIC_I(i4, AllS(LIFT(DELTA) = LIFT(CONC_FIX_r)))))
   =  concat(TIC_I(i1, i3), TIC_I(TIC_I(i2, i4), AllS(LIFT(DELTA) = LIFT(CONC_FIX_r))));

   %%%% Concatenation Concatenate property %%%%
   CONC_CONC_CO: LEMMA No_Term?(tp1) => COS(tp1) = concat(COS(tp1), COS(tp1));
   CONC_CONC: LEMMA No_Term?(tp1) => AllS(tp1 AND LIFT(DELTA) > LIFT(0)) = concat(AllS(tp1), AllS(tp1));

   %%%% Concatenate durations %%%%
   CONC_DURA_CO_r, CONC_DURA_CO_s: var Time;
   CONC_DURA_CO: LEMMA CONC_DURA_CO_r > 0 AND CONC_DURA_CO_s > 0 AND No_Term?(tp1) =>
     COS(tp1 AND LIFT(DELTA) = LIFT(CONC_DURA_CO_r) + LIFT(CONC_DURA_CO_s)) =
     concat(COS(tp1 AND LIFT(DELTA) = LIFT(CONC_DURA_CO_r)), COS(tp1 AND LIFT(DELTA) = LIFT(CONC_DURA_CO_s)));
   CONC_DURA: LEMMA  CONC_DURA_CO_r > 0 AND CONC_DURA_CO_s > 0 AND No_Term?(tp1) =>
     AllS(tp1 AND LIFT(DELTA) = LIFT(CONC_DURA_CO_r) + LIFT(CONC_DURA_CO_s)) =
     concat(AllS(tp1 AND LIFT(DELTA) = LIFT(CONC_DURA_CO_r)), AllS(tp1 AND LIFT(DELTA) = LIFT(CONC_DURA_CO_s)));

END TIC_rules
```

A.3 Supplementary Rules

The `TIC_supple_rules` theory below includes all TIC supplementary reasoning rules which capture domain specific properties.

```
TIC_supple_rules: THEORY
  BEGIN

   IMPORTING TIC_rules, analysis@continuous_functions_props[Time];

   tr1, tr2: VAR Trace;
   i: VAR II;
```

```
t: VAR Time;
tp1, tp2, tp3, tp4: VAR TPred;
ST: var posreal;

% The rule DISCRETE_n_ALL shows that a property holds always in all intervals provided
% it is true in every sample time interval. Namely,
% If ALPHA, OMEGA, and DELTA do not occur free in predicate tp1,
% then [- exists k: nat @ ALPHA = k * ST /\ OEMGA = (k+1) * ST -) subseteq [- tp1 -)
%   => II = [(- tp1 -)]
DISCRETE_n_ALL_IMP: LEMMA No_Term?(tp1) =>
  (   subset?(COS(exNat(LAMBDA (k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(ST) AND
                 LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(ST))),
              COS(tp1))
   => AllTrue(tp1));

% When interval operators do not affect a predicate tp1, we have
%    (- exi k: N @ alpha = k * st and omega = (k + 1) * st -] subset (- tp1 -]
% => [(- alpha > 0 -)] subset [(- tp1 -)]
DISCRETE_n_ALL_IMP_OC: LEMMA No_Term?(tp1) =>
  (   subset?(OCS(exNat(LAMBDA (k: nat): LIFT(ALPHA) = LIFT(k) * LIFT(ST) AND
                 LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(ST))),
              OCS(tp1))
   => subset?(AllS(LIFT(ALPHA) > LIFT(0)), AllS(tp1)));

% Lemma CONC_LEN_LE shows that
%[(- tp1 /\ delta <= r -)] ; [(- tp2 /\ delta <= s -)] subseteq [(- delta <= r + s -)]
CONC_LEN_LE_s, CONC_LEN_LE_r: var Time;
CONC_LEN_LE: LEMMA subset?(
   concat(AllS(LIFT(DELTA) <= LIFT(CONC_LEN_LE_r)), AllS(LIFT(DELTA) <= LIFT(CONC_LEN_LE_s))),
   AllS(LIFT(DELTA) <= LIFT(CONC_LEN_LE_r) + LIFT(CONC_LEN_LE_s)));

Keep_L_RULE: LEMMA subset?(AllS(tp1 and tp2), AllS(tp1));
Keep_R_RULE: LEMMA subset?(AllS(tp1 and tp2), AllS(tp2));

Intro_L_RULE: LEMMA subset?(AllS(tp1 and tp2), AllS(tp3))
                 => subset?(AllS(tp1 and tp2), AllS(tp1 and tp3));
Intro_R_RULE: LEMMA subset?(AllS(tp1 and tp2), AllS(tp3))
                 => subset?(AllS(tp1 and tp2), AllS(tp2 and tp3));

True_Invariant_L: LEMMA AllTrue(tp1) <=> AllS(tp1) = fullset;
True_Invariant_R: LEMMA AllTrue(tp1) <=> fullset = AllS(tp1);
Invariant_True_R: LEMMA fullset = AllS(tp1) <=> AllTrue(tp1);
Invariant_True_L: LEMMA AllS(tp1) = fullset <=> AllTrue(tp1);

%     ([(- tp1 -)] subset [(- tp2 -)]) /\ ([(- tp1 -)] subset [(- tp3 -)])
% =>  [(- tp1 -)] subset [(- tp2 /\ tp3 -)]
BB_in_common: LEMMA subset?(AllS(tp1), AllS(tp2)) and subset?(AllS(tp1), AllS(tp3))
                 => subset?(AllS(tp1), AllS(tp2 and tp3));

%    ([(- tp1 -)] subset [(- tp2 -)]) /\ ([(- tp2 -)] subset [(- tp3 -)])
% => [(- tp1 -)] subset [(- tp3 -)]
```

```
  BB_in_tran: LEMMA subset?(AllS(tp1), AllS(tp2)) and subset?(AllS(tp2), AllS(tp3))
                => subset?(AllS(tp1), AllS(tp3));

  Insert_BB_R: LEMMA AllTrue(tp1) IMPLIES AllS(tp2) = AllS(tp2 and tp1);
  Insert_BB_L: LEMMA AllTrue(tp1) IMPLIES AllS(tp2) = AllS(tp1 and tp2);

  BB_to_CO_subset: LEMMA subset?(AllS(tp1), AllS(tp2)) => subset?(COS(tp1), COS(tp2));
  BB_to_CC_subset: LEMMA subset?(AllS(tp1), AllS(tp2)) => subset?(CCS(tp1), CCS(tp2));
  BB_to_OC_subset: LEMMA subset?(AllS(tp1), AllS(tp2)) => subset?(OCS(tp1), OCS(tp2));

  BB_to_CO_eq: LEMMA AllS(tp1) = AllS(tp2) => COS(tp1) = COS(tp2);
  BB_to_CC_eq: LEMMA AllS(tp1) = AllS(tp2) => CCS(tp1) = CCS(tp2);

  BB_eq_sub_e1, BB_eq_sub_e2, BB_eq_sub_e3: var TExp;
  BB_eq_sub: LEMMA AllTrue(BB_eq_sub_e1 = BB_eq_sub_e2) =>
      AllS(BB_eq_sub_e1 = BB_eq_sub_e3) = AllS(BB_eq_sub_e2 = BB_eq_sub_e3);
  BB_ge_sub: LEMMA AllTrue(BB_eq_sub_e1 = BB_eq_sub_e2) =>
      AllS(BB_eq_sub_e1 >= BB_eq_sub_e3) = AllS(BB_eq_sub_e2 >= BB_eq_sub_e3);
  BB_le_sub: LEMMA AllTrue(BB_eq_sub_e1 = BB_eq_sub_e2) =>
      AllS(BB_eq_sub_e1 <= BB_eq_sub_e3) = AllS(BB_eq_sub_e2 <= BB_eq_sub_e3);

  % mathematical axiom used in the following proof:
  TH: var real;
  mid_point_existence:  AXIOM continuous(tr1) => FORALL (t1, t2: Time):
       t1 < t2 AND tr1(t1) < TH AND tr1(t2) > TH
   =>  EXISTS (t3: Time): t3 < t2 AND t3 > t1 AND tr1(t3) = TH AND
                          FORALL (t4: Time): t4 > t3 AND t4 < t2 => tr1(t4) > TH;

  mid_ivl_exi: LEMMA continuous(tr1) =>
     subset?( AllS((LIFT(tr1) o LIFT(ALPHA)) < LIFT(TH) and (LIFT(tr1) o LIFT(OMEGA)) > LIFT(TH)),
              concat( AllS(TTRUE), concat( AllS(LIFT(tr1) = LIFT(TH)), AllS(LIFT(tr1) > LIFT(TH)))));

  % forall i: II @ exists m, p: nat; n, q : nonnegative reals | n < ST /\ q < ST @
  %        ALPHA(i) = m * ST + n /\ OMEGA(i) = p * ST + q;
  Endpoints_general_form: LEMMA FORALL (i: II): EXISTS (m, p: nat), (n, q: {x: nnreal | x < ST}):
       ALPHA(i) = m * ST + n AND OMEGA(i) = p * ST + q;

  x, y: var nat;
  Mult_SAMPLE_INTERVALS: LEMMA No_Term?(tp1) AND No_Term?(tp2) AND x <= y
    => ((FORALL (k: {n: nat | x <= n AND n <= y}):
            subset?(COS(LIFT(ALPHA) = LIFT(k) * LIFT(ST) AND
                        LIFT(OMEGA) = (LIFT(k) + LIFT(1)) * LIFT(ST) AND tp1),
                    COS(tp2)))
        => subset?(COS(LIFT(ALPHA) = LIFT(x) * LIFT(ST) AND
                       LIFT(OMEGA) = (LIFT(y) + LIFT(1)) * LIFT(ST) AND tp1),
                   COS(tp2)));

END TIC_supple_rules
```

A.4 Proof Strategies

To make TIC proofs more automated in PVS, we have defined a collection of proof strategies to combine frequently used proof commends and also keep the encoding details of TIC away from users.

```
(defstep ExpandSubset ()
   (then (expand "subset?" +)
         (skosimp)
         (ground))
   "(ExpandSubset): expand and simplify the subset relation in the consequents"
   "Expand and simplify the subset relation in the consequents")

(defstep ExpIntervaltoTime (fnum)
   (try (try (expand "OOS" fnum) (skosimp)
             (try (expand "OCS" fnum) (skosimp)
                  (try (expand "COS" fnum) (skosimp)
                       (try (expand "CCS" fnum) (skosimp)
                            (try (expand "AllS" fnum)(skosimp)
                                 (skip))))))
         (then (expand "Everywhere?" fnum)(skosimp))
         (skip))
    "(ExpandIntervaltoTime <fnum>): Trying to expand the interval brackets in consequent
     <fnum> to predicate which involve time point and interval explicitly"
    "Trying to expand the interval brackets in consequent ~a")

(defstep ExpandTIC (fnum)
  (then (expand "LIFT" fnum) (expand "OR" fnum) (expand "=" fnum)
        (expand ">" fnum) (expand ">=" fnum) (expand "<" fnum)
        (expand "<=" fnum) (expand "-" fnum) (expand "o" fnum)
        (expand "+" fnum) (expand "*" fnum) (expand "exNat" fnum)
        (bddsimp fnum) )
  "(ExpandTIC <fnum>): Assigning time point and interval to all logical and mathematical
   operators within the formula <fnum>"
  "Assigning time point and interval to formula ~a")

(defstep ExpandAllTrue (fnum)
  (then (expand "AllTrue" fnum) (skosimp)
        (expand "Everywhere?") (skosimp)
        (ExpandTIC fnum))
  "(ExpandAllTrue <fnum>): expand function AllTrue at formula <fnum> into explicit representation"
  "Expanding function AllTrue at formula ~a")

(defstep ExpandConcat (fnum &OPTIONAL pos)
  (then (if pos (expand "concat" fnum pos)
                (expand "concat" fnum))
        (skosimp)
        (ground))
```

```
  "(ExpandConcat <fnum>): PVS automatically instantiated two concatenated invervals at antecedent <fnum>"
  "PVS automatically instantiated two concatenated intervals ~a")

(defstep AssignInvlnTime (fnum &OPTIONAL interval point)
    (try (else (expand "OOS" fnum)
               (else (expand "OCS" fnum)
                     (else (expand "COS" fnum)
                           (else (expand "CCS" fnum)
                                 (else (expand "AllS" fnum)
                                       (skip))))))
         (then (if interval (inst fnum interval) (inst? fnum))
               (expand "Everywhere?" fnum)
               (if point (inst fnum point) (inst? fnum)))
         (skip))
    "Assign time point and interval to TIC function at formula <fnum>: if user does not provide value of
     the interval and time point respectively, the values are assigned automatically by the PVS prover."
    "Assign time point and interval to TIC function at formula ~a")

 (defstep AssignAllTrue (fnum &OPTIONAL interval timepoint)
   (then (expand "AllTrue" fnum)
         (if interval (inst fnum interval)
                      (inst? fnum))
         (if timepoint (then (expand "Everywhere?" fnum)
                             (inst fnum timepoint))
                       (skip)))
   "(AssignAllTrue <fnum>): Assign user-defined intervals to function AllTrue
    at antecedent <fnum> and waiting for the user-defined time point"
   "Assign user-defined intervals to function AllTrue at antecedent ~a")

 (defstep AssignSubset (fnum &OPTIONAL val)
   (then (expand "subset?" fnum)
         (if val (inst fnum val)
                 (inst? fnum))
         (ground))
   "(AssignSubset <fnum>): assign user-specified value to antecedent <fnum>"
   "Assign user-specified value to antecedent ~a")

 (defstep AssignConcat (fnum lvl rvl &OPTIONAL pos)
   (then (if pos (expand "concat" fnum pos)
                 (expand "concat" fnum))
         (inst fnum lvl rvl)
         (ground))
   "(AssignConcat <fnum>): assign user-defined intervals to the corresponding connected
     intervals at consequent <fnum>"
   "User assigns interval values to the connected intervals at consequent ~a")
```

Appendix B

Proof of the DC Rule $DC15$

This chapter corresponds to Chapter 5.2 for showing all proof steps when reasoning about the DC rule *DC15* using our extended verification system which support DC proofs. The rule as given below states that if the duration of a state S is positive in an interval, the the interval can be decomposed into three subintervals: the duration of S is zero in the first subinterval, and S is true almost everywhere in the second subinterval.

DC15 $\int S > 0 \Rightarrow (\int S = 0) \frown [\![S]\!] \frown \text{true}$

As explained in Chapter 5, we model DC semantics in TIC, and the chop operator is defined using TIC (refer to Chapter 5.1.4). The following PVS specification is generated from the corresponding TIC model.

```
DC_DC15: LEMMA fullset = AllS( TICIntegral(dcstate1) > LIFT(0)
     => DCChop(DCChop(TICIntegral(dcstate1) = LIFT(0), pq(dcstate1)), TTRUE));
```

The following proof scripts are used to prove the above lemmas and theorems. These scripts are obtained by the Emacs command `M-x show-proofs-pvs-file`. Note that the numbers indicating the subgoal indexes in the scripts are not entered by users, but are used by PVS to keep track of the proof.

```
DC_DC15: proved

  ("" (skosimp) (expand "fullset") (apply-extensionality) (expintervaltotime 1)
      (expand "=>") (ground) (lemma "DC_DC15_pre") (lemma "DCState_is_FV")
      (inst?) (expand "fullset") (decompose-equality) (inst?) (expintervaltotime -1)
      (inst?) (ground) (expand "exNat") (skosimp) (inst?)(ground))))
```

The lemma `DC_DC15_pre` used in the above proof is given below.

```
 DC_DC15_pre: LEMMA forall (k: nat): fvl(k)(dcstate1)(i) =>
   (TICIntegral(dcstate1) > LIFT(0))(t, i)
=>  DCChop(DCChop(TICIntegral(dcstate1) = LIFT(0), pq(dcstate1)), TTRUE)(t, i);
```

The proof of the above lemma invokes an induction to the variable `k` according to the recursive function `fvl` defined in Chapter 5.1.1. Note that the relation between the `DC_DC15_pre` lemma and the `DC_DC15` is captured by the axiom `DCState_is_FV` (also defined in Chapter 5.1.1) which represents the finite variability property for discrete-valued timed traces.

```
DC_DC15_pre: proved

 ("" (induct "k")
    (("1" (skosimp)(expand "fvl" -1)(lemma "Integral_a_to_a")(inst - "ALPHA(i!1)" "dcstate1!1")(grind))
     ("2" (skosimp)(skosimp)(expand "fvl" -2)(ground)
          (("1" (expand "DCChop" 1 1)(inst 1 "i!1" "(CC, (OMEGA(i!1), OMEGA(i!1)))")(ground)
                (("1" (expand "OMEGA" 1) (propax))
                 ("2" (expand "OMEGA" 1) (expand "ALPHA" 1) (propax))
                 ("3" (expand "Everywhere?") (skosimp) (expand "DCChop" 1)
                      (inst 1 "(CC, (ALPHA(i!1), ALPHA(i!1)))" "i!1") (ground)
                      (("1" (expand "ALPHA") (propax)) ("2" (expand "OMEGA") (propax))
                         ("3" (expand "Everywhere?") (skosimp) (lemma "Integral_a_to_a")
                              (inst - "ALPHA(i!1)" "dcstate1!1") (grind))
                         ("4" (expand "Everywhere?")  (skosimp) (inst?) (ground))))
                  ("4" (grind))))
            ("2" (expand "DCChop" -1)(skosimp)(expand "DCChop" 1 1)(inst 1 "il!1" "ir!1")(ground)
                 (("1" (expand "Everywhere?") (skosimp) (expand "DCChop" 1)
                       (inst 1 "(CC, (ALPHA(il!1), ALPHA(il!1)))" "il!1") (ground)
                       (("1" (expand "ALPHA") (propax)) ("2" (expand "OMEGA") (propax))
                        ("3" (expand "Everywhere?") (skosimp) (lemma "Integral_a_to_a")
                             (inst  - "ALPHA(il!1)" "dcstate1!1") (grind) )))
                  ("2" (grind) )))
            ("3" (expand "DCChop" -1) (skosimp) (expand "Everywhere?")
                 (inst -5 "(ALPHA(ir!1) + OMEGA(ir!1))/2") (ground)
                 (("1" (inst -6 "dcstate1!1" "ir!1" "(ALPHA(ir!1) + OMEGA(ir!1))/2") (ground)
```

```
                    (("1" (expand "DCChop" -1 1) (skosimp) (expand "Everywhere?")
                          (inst - "(ALPHA(il!2) + OMEGA(il!2))/2") (ground)
                          (("1" (expand "DCChop" -1) (skosimp) (expand "DCChop" 1 1)
                                (inst 1 "ChoppedInls(il!1, il!2)" "ir!2") (ground)
                                (("1"  (grind)) ("2" (grind))
                                 ("3" (expand "Everywhere?") (skosimp) (expand "DCChop")
                                      (inst 1 "ChoppedInls(il!1, il!3)" "ir!3") (ground)
                                      (("1" (grind)) ("2" (grind)) ("3" (grind))
                                       ("4" (expand "Everywhere?") (skosimp) (lemma "DC_DCA5")
                                            (inst - "dcstate1!1" "0" "0") (expand "subset?")
                                            (inst - "ChoppedInls(il!1, il!3)") (ground)
                                            (("1" (expintervaltotime -1)(inst - "t!3")(ground))
                                             ("2" (expintervaltotime 1) (expand "DCChop")
                                                  (inst 1 "il!1" "il!3") (ground)
                                                  (("1" (grind)) ("2" (grind))
                                                   ("3" (expand "Everywhere?")(skosimp)
                                                        (lemma "DC_DC12")(inst?)(expand "fullset")
                                                        (decompose-equality)(inst - "il!1")
                                                        (expintervaltotime -1)(inst - "t!5")(ground)
                                                        (expand "=>")(inst -18 "t!5")(ground))))))))
                                       ("4" (grind) ))
                                 ("2" (typepred "il!2") (grind))))
                           ("2" (lemma "Integral_split")
                                (inst - "ALPHA(il!1)" "ALPHA(ir!1)" "OMEGA(ir!1)" "dcstate1!1") (ground)
                                (("1" (expand "TICIntegral" (-8 1)) (expandtic (-8 1)) (lemma "DC_DC12")
                                      (inst?) (expand "fullset") (decompose-equality) (inst - "il!1")
                                      (ground) (expand "AllS") (expand "Everywhere?")
                                      (inst - "(ALPHA(il!1) + OMEGA(il!1))/2") (ground)
                                      (("1" (inst - "(ALPHA(il!1) + OMEGA(il!1))/2")  (ground)
                                            (("1" (expand "=>" -2) (expand "TICIntegral")
                                                  (expandtic -2) (grind))
                                             ("2" (typepred "il!1") (grind))))
                                       ("2" (typepred "il!1") (grind))))
                                 ("2" (typepred "dcstate1!1")(inst - "ALPHA(il!1)" "ALPHA(ir!1)"))
                                 ("3" (typepred "dcstate1!1")(inst - "ALPHA(ir!1)" "OMEGA(ir!1)"))))))
                    ("2" (typepred "ir!1") (grind)))))))
("3" (skosimp) (grind) )
("4" (skosimp) (grind) )
("5" (skosimp) (grind) )))
```

Appendix C

Supported Simulink Library Blocks

This chapter includes (1) the names of Simulink library blocks which are supported in our proposed formal framework (as presented in Chapters 6 and 7, and (2) specific TIC library functions which model the Simulink library blocks of the *Commonly Used* category [84].

Currently, we handled 51 library blocks of 10 categories in terms of their functionalities. Among them, 44 library blocks of 9 categories are represented by a set of TIC library functions (in Section C.1). The rest 7 library blocks of the category *Ports and Subsystems* are handled during transformation (in Section C.2). The last section illustrates how TIC can precisely capture all library blocks of the *Commonly Used* category. For the sake of page limit, we ignore descriptions of all 51 library blocks and the rest of TIC library functions. Readers can refer to our online report [30] for more details such as the semantics of these library blocks with their corresponding TIC library functions.

C.1 Library Blocks Modeled by TIC Library Functions

- Continuous Library: Integrator, Derivative.

- Discrete Library: Memory, Discrete-Time Integrator, Unit Delay, Zero-order Delay.
- Logic and Bits Operations Library: Combinational Logic, Comparator to Constant, Compare to Zero, Interval Test, Logical Operator, Relational Operator.
- Math Operations Library: Abs, Add, Bias, Divide, Dot Product, Fcn, Gain, Math Function, MinMax, Product, Sign, Subtract, Sum, Unary Minus.
- Discontinuous Library: Dead Zone, Hit Crossing, Relay, Saturation.
- Signal Routing Library: Bus Creator, Bus Selector, Demux, Mux, Switch.
- Source Library: Constant, Clock, Digital Clock, Ground.
- Signal Attributes Library: Data Type Conversion, IC.
- Sinks: Display, Scope, Terminator.

C.2 Library Blocks Handled in Transformation

- Ports and Subsystems: Enable, Enabled Subsystem, Inport, Outport, Subsystem, Trigger, Triggered Subsystem.

C.3 Commonly Used Simulink Library Blocks in TIC

This section shows how to model library blocks of the Simulink *Commonly Used* category in TIC. Most of TIC specifications are functions and the way of constructing and validating these TIC library functions is explained in Chapter 6. Some Simulink library blocks, particularly blocks*Bus Creator*, *Bus Selector*, *Demux*, *Ground*, *Mux*, *Scope* and *Terminator*, are represented by TIC schemas since they do not involve any operand parameters and sample times. In addition, some library blocks are handled

during the transformation phase as discussed in Chapter 7.5. Namely, library blocks *Inport*, *Outport* and *Subsystem* are not listed here.

Bus Creator library block

BusCreator_2

$$signal1, signal2 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \text{seq}(\mathbb{T} \rightarrow \mathbb{R})$$

$$\#Out = 2 \land \mathbb{I} = \{\!| Out(1) = signal1 \land Out(2) = signal2 |\!\}$$

BusCreator_3

$$signal1, signal2, signal3 : \mathbb{T} \rightarrow \mathbb{R};\ Out : \text{seq}(\mathbb{T} \rightarrow \mathbb{R})$$

$$\#Out = 3 \land \mathbb{I} = \{\!| Out(1) = signal1 \land Out(2) = signal2 \land Out(3) = signal3 |\!\}$$

Bus Selector library block

BusSelector_2

$$In_1 : \text{seq}(\mathbb{T} \rightarrow \mathbb{R});\ signal1, signal2 : \mathbb{T} \rightarrow \mathbb{R}$$

$$\#In_1 = 2 \land \mathbb{I} = \{\!| In_1(1) = signal1 \land In_1(2) = signal2 |\!\}$$

BusSelector_3

$$In_1 : \text{seq}(\mathbb{T} \rightarrow \mathbb{R});\ signal1, signal2, signal3 : \mathbb{T} \rightarrow \mathbb{R}$$

$$\#In_1 = 3 \land \mathbb{I} = \{\!| In_1(1) = signal1 \land In_1(2) = signal2 \land In_1(3) = signal3 |\!\}$$

Constant library block

$$Constant : \mathbb{R} \rightarrow \mathbb{P}[Out : \mathbb{T} \rightarrow \mathbb{R};\ IniVal : \mathbb{R}]$$

$$\forall\, cv : \mathbb{R} \bullet Constant(cv) = [Out : \mathbb{T} \rightarrow \mathbb{R};\ IniVal : \mathbb{R} \mid cv = IniVal \land \mathbb{I} = \{\!| Out = IniVal |\!\}]$$

Demux library block

Demux_2

$$In_1 : \text{seq}(\mathbb{T} \rightarrow \mathbb{R});\ Out_1, Out_2 : \mathbb{T} \rightarrow \mathbb{R}$$

$$\#In_1 = 2 \land \mathbb{I} = \{\!| Out_1 = In_1(1) \land Out_2 = In_1(2) |\!\}$$

Demux_3

$$In_1 : \text{seq}(\mathbb{T} \to \mathbb{R});\ Out_1, Out_2, Out_3 : \mathbb{T} \to \mathbb{R}$$

$$\#In_1 = 3 \land \mathbb{I} = [\![Out_1 = In_1(1) \land Out_2 = In_1(2) \land Out_3 = In_1(3)]\!]$$

Discrete-Time Integrator library block

$$DiscreteIntegrator\_F : \mathbb{R} \times \mathbb{T} \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}]$$

$$\begin{aligned}
&\forall\, init : \mathbb{R};\ t : \mathbb{T} \bullet DiscreteIntegrator(t, init) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
&\qquad st > 0 \land t = st \land IniVal = init \land Out(0) = IniVal \land \\
&\qquad (\!|\exists\, k : \mathbb{N} \bullet \alpha = k * st \land \omega = (k+1) * st|\!) \subseteq (\!|Out = Out(\alpha) \land Out(\omega) = Out(\alpha) + st * In_1(\alpha)|\!)]
\end{aligned}$$

$$DiscreteIntegrator\_B : \mathbb{R} \times \mathbb{T} \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}]$$

$$\begin{aligned}
&\forall\, init : \mathbb{R};\ t : \mathbb{T} \bullet DiscreteIntegrator(t, init) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
&\qquad st > 0 \land t = st \land IniVal = init \land Out(0) = IniVal \land \\
&\qquad (\!|\exists\, k : \mathbb{N} \bullet \alpha = k * st \land \omega = (k+1) * st|\!) \subseteq (\!|Out = Out(\alpha) \land Out(\omega) = Out(\alpha) + st * In_1(\omega)|\!)]
\end{aligned}$$

Gain library block

$$Gain : (\mathbb{T} \times \mathbb{R}) \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ GValue : \mathbb{R};\ st : \mathbb{T}]$$

$$\begin{aligned}
&\forall\, t : \mathbb{T};\ gv : \mathbb{R} \bullet (t = 0 \Rightarrow Gain(t, gv) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ GValue : \mathbb{R};\ st : \mathbb{T} \mid \\
&\qquad st = 0 \land gv = GValue \land \mathbb{I} = [\![Out = In_1 * GValue]\!]) \\
&\land\ (t > 0 \Rightarrow Gain(t, gv) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ GValue : \mathbb{R};\ st : \mathbb{T} \mid \\
&\qquad t = st \land st > 0 \land gv = GValue \land (\!|\exists\, k : \mathbb{N} \bullet \alpha = k * st \land \omega = (k+1) * st|\!) = (\!|Out = In_1(\alpha) * GValue|\!)])
\end{aligned}$$

Ground library block

$$Ground \mathrel{\widehat{=}} [Out : \mathbb{T} \to \mathbb{R} \mid \mathbb{I} = [\![Out = 0]\!]]$$

Integrator library block

$$Integrator : \mathbb{R} \to \mathbb{P}[In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}]$$

$$\begin{aligned}
&\forall\, init : \mathbb{R} \bullet Integrator(init) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \nrightarrow \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
&\qquad st = 0 \land IniVal = init \land Out(0) = IniVal \land \mathbb{I} = [\![Out(\omega) = Out(\alpha) + \textstyle\int_\alpha^\omega In_1]\!]]
\end{aligned}$$

Logical Operator library block

$$Logic\_NOT : \mathbb{T} \to \mathbb{P}[In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T}]$$

$$\begin{aligned}
&\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Logic_NOT(t) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
&\qquad st = 0 \land ([\![In_1 = 0]\!] = [\![Out = 1]\!] \land [\![In_1 \neq 0]\!] = [\![Out = 0]\!]) \\
&\land\ (t > 0 \Rightarrow Logic_NOT(t) = [In_1 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid t = st \land st > 0 \land \\
&\qquad (\!|\exists\, k : \mathbb{N} \bullet \alpha = k * st \land \omega = (k+1) * st|\!) = (\!|(In_1(\alpha) = 0 \Rightarrow Out = 1) \land (In_1(\alpha) \neq 0 \Rightarrow Out = 0)|\!)])
\end{aligned}$$

$$
\begin{array}{|l}
Logic\_AND\_2 : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Logic\_AND\_2(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge (\!|In_1 \neq 0 \wedge In_2 \neq 0|\!) = (\!|Out = 1|\!) \wedge (\!|In_1 = 0 \vee In_2 = 0|\!) = (\!|Out = 0|\!)]) \\
\ \wedge\ (t > 0 \Rightarrow Logic\_AND\_2(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!| \exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st |\!] = \\
\qquad\qquad [\!| (In_1(\alpha) \neq 0 \wedge In_2(\alpha) \neq 0 \Rightarrow Out = 1) \wedge (In_1(\alpha) = 0 \vee In_2(\alpha) = 0 \Rightarrow Out = 0) |\!]])
\end{array}
$$

$$
\begin{array}{|l}
Logic\_OR\_2 : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Logic\_OR\_2(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge (\!|In_1 \neq 0 \vee In_2 \neq 0|\!) = (\!|Out = 1|\!) \wedge (\!|In_1 = 0 \wedge In_2 = 0|\!) = (\!|Out = 0|\!)]) \\
\ \wedge\ (t > 0 \Rightarrow Logic\_OR\_2(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!| \exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st |\!] = \\
\qquad\qquad [\!| (In_1(\alpha) \neq 0 \vee In_2(\alpha) \neq 0 \Rightarrow Out = 1) \wedge (In_1(\alpha) = 0 \wedge In_2(\alpha) = 0 \Rightarrow Out = 0) |\!]])
\end{array}
$$

$$
\begin{array}{|l}
Logic\_NAND\_2 : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Logic\_NAND\_2(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge (\!|In_1 = 0 \vee In_2 = 0|\!) = (\!|Out = 1|\!) \wedge (\!|In_1 \neq 0 \wedge In_2 \neq 0|\!) = (\!|Out = 0|\!)]) \\
\ \wedge\ (t > 0 \Rightarrow Logic\_NAND\_2(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!| \exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st |\!] = \\
\qquad\qquad [\!| (In_1(\alpha) = 0 \vee In_2(\alpha) = 0 \Rightarrow Out = 1) \wedge (In_1(\alpha) \neq 0 \wedge In_2(\alpha) \neq 0 \Rightarrow Out = 0) |\!]])
\end{array}
$$

$$
\begin{array}{|l}
Logic\_NOR\_2 : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Logic\_NOR\_2(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge (\!|In_1 = 0 \wedge In_2 = 0|\!) = (\!|Out = 1|\!) \wedge (\!|In_1 \neq 0 \vee In_2 \neq 0|\!) = (\!|Out = 0|\!)]) \\
\ \wedge\ (t > 0 \Rightarrow Logic\_AND\_2(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!| \exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st |\!] = \\
\qquad\qquad [\!| (In_1(\alpha) = 0 \wedge In_2(\alpha) = 0 \Rightarrow Out = 1) \wedge (In_1(\alpha) \neq 0 \vee In_2(\alpha) \neq 0 \Rightarrow Out = 0) |\!]])
\end{array}
$$

Mux library block

$$Mux\_2 \mathrel{\widehat{=}} [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathrm{seq}(\mathbb{T} \to \mathbb{R}) \mid \#Out = 2 \wedge \mathbb{I} = (\!|Out = \langle In_1, In_2 \rangle|\!)]$$

$$Mux\_3 \mathrel{\widehat{=}} [In_1, In_2, In_3 : \mathbb{T} \to \mathbb{R};\ Out : \mathrm{seq}(\mathbb{T} \to \mathbb{R}) \mid \#Out = 3 \wedge \mathbb{I} = (\!|Out = \langle In_1, In_2, In_3 \rangle|\!)]$$

Product library block

$$
\begin{array}{|l}
Product\_MM : \mathbb{T} \to \mathbb{P}[In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Product\_MM(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = (\!|Out = In_1 * In_2|\!)]) \\
\ \wedge\ (t > 0 \Rightarrow Product\_MM(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!| \exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st |\!] = [\!| Out = In_1(\alpha) * In_2(\alpha) |\!]])
\end{array}
$$

$$
\begin{array}{|l}
Product\_MD : \mathbb{T} \to \mathbb{P}[In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Product\_MD(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = [\![Out = In_1 / In_2]\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Product\_MD(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\![\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st]\!] = [\![Out = In_1(\alpha) / In_2(\alpha)]\!]])
\end{array}
$$

$$
\begin{array}{|l}
Product\_DD : \mathbb{T} \to \mathbb{P}[In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Product\_DD(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = [\![Out = 1/(In_1 * In_2)]\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Product\_DD(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\![\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st]\!] = [\![Out = 1/(In_1(\alpha) * In_2(\alpha))]\!]])
\end{array}
$$

$$
\begin{array}{|l}
Product\_MMM : \mathbb{T} \to \mathbb{P}[In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Product\_MMM(t) = [In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge \mathbb{I} = [\![Out = In_1 * In_2 * In_3]\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Product\_MMM(t) = [In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\![\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st]\!] = [\![Out = In_1(\alpha) * In_2(\alpha) * In_3(\alpha)]\!]])
\end{array}
$$

Relational Operator library block

$$
\begin{array}{|l}
Relation\_eq : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Relation\_eq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge [\![In_1 = In_2]\!] = [\![Out = 1]\!] \wedge [\![In_1 \neq In_2]\!] = [\![Out = 0]\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Relation\_eq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\![\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st]\!] = \\
\qquad\quad [\![(In_1(\alpha) = In_2(\alpha) \Rightarrow Out = 1) \wedge (In_1(\alpha) \neq In_2(\alpha) \Rightarrow Out = 0)]\!]])
\end{array}
$$

$$
\begin{array}{|l}
Relation\_neq : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Relation\_neq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge [\![In_1 \neq In_2]\!] = [\![Out = 1]\!] \wedge [\![In_1 = In_2]\!] = [\![Out = 0]\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Relation\_neq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\![\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st]\!] = \\
\qquad\quad [\![(In_1(\alpha) = In_2(\alpha) \Rightarrow Out = 1) \wedge (In_1(\alpha) \neq In_2(\alpha) \Rightarrow Out = 0)]\!]])
\end{array}
$$

$$
\begin{array}{|l}
Relation\_l : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Relation\_l(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge [\![In_1 < In_2]\!] = [\![Out = 1]\!] \wedge [\![In_1 \geq In_2]\!] = [\![Out = 0]\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Relation\_l(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0, 1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\![\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st]\!] = \\
\qquad\quad [\![(In_1(\alpha) < In_2(\alpha) \Rightarrow Out = 1) \wedge (In_1(\alpha) \geq In_2(\alpha) \Rightarrow Out = 0)]\!]])
\end{array}
$$

$$
\begin{array}{l}
Relation\_leq : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Relation\_leq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge (\!|In_1 \leq In_2|\!) = (\!|Out = 1|\!) \wedge (\!|In_1 > In_2|\!) = (\!|Out = 0|\!)]) \\
\ \wedge\ \ (t > 0 \Rightarrow Relation\_leq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge (\!|\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st|\!) = \\
\qquad\qquad (\!|(In_1(\alpha) \leq In_2(\alpha) \Rightarrow Out = 1) \wedge (In_1(\alpha) > In_2(\alpha) \Rightarrow Out = 0)|\!)])
\end{array}
$$

$$
\begin{array}{l}
Relation\_geq : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Relation\_geq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge (\!|In_1 \geq In_2|\!) = (\!|Out = 1|\!) \wedge (\!|In_1 < In_2|\!) = (\!|Out = 0|\!)]) \\
\ \wedge\ \ (t > 0 \Rightarrow Relation\_geq(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge (\!|\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st|\!) = \\
\qquad\qquad (\!|(In_1(\alpha) \geq In_2(\alpha) \Rightarrow Out = 1) \wedge (In_1(\alpha) < In_2(\alpha) \Rightarrow Out = 0)|\!)])
\end{array}
$$

$$
\begin{array}{l}
Relation\_g : \mathbb{T} \to \mathbb{P}[In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Relation\_g(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge (\!|In_1 > In_2|\!) = (\!|Out = 1|\!) \wedge (\!|In_1 \leq In_2|\!) = (\!|Out = 0|\!)]) \\
\ \wedge\ \ (t > 0 \Rightarrow Relation\_g(t) = [In_1, In_2 : \mathbb{T} \to \mathbb{R};\ Out : \mathbb{T} \to \{0,1\};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge (\!|\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st|\!) = \\
\qquad\qquad (\!|(In_1(\alpha) > In_2(\alpha) \Rightarrow Out = 1) \wedge (In_1(\alpha) \leq In_2(\alpha) \Rightarrow Out = 0)|\!)])
\end{array}
$$

Saturation library block

$$
\begin{array}{l}
Saturation : (\mathbb{T} \times \mathbb{R} \times \mathbb{R}) \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ Llimit, Ulimit : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T};\ ll, hl : \mathbb{R} \bullet (t = 0 \Rightarrow Saturation(t, ll, hl) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ Llimit, Ulimit : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge Llimit \leq Ulimit \wedge ll = Llimit \wedge hl = Ulimit \wedge (\!|In_1 \leq Llimit|\!) = (\!|Out = Llimit|\!) \wedge \\
\qquad (\!|Llimit < In_1 \wedge In_1 < Ulimit|\!) = (\!|Out = In_1|\!) \wedge (\!|In_1 \geq Ulimit|\!) = (\!|Out = Ulimit|\!)]) \\
\ \wedge\ \ (t > 0 \Rightarrow Saturation(t, ll, hl) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ Llimit, Ulimit : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge Llimit \leq Ulimit \wedge ll = Llimit \wedge hl = Ulimit \wedge \\
\qquad\qquad (\!|\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st|\!) = (\!|(Llimit < In_1(\alpha) < Ulimit \Rightarrow Out = In_1(\alpha)) \wedge \\
\qquad\qquad\qquad (In_1(\alpha) \leq Llimit \Rightarrow Out = Llimit) \wedge (In_1(\alpha) \geq Ulimit \Rightarrow Out = Ulimit)|\!)])
\end{array}
$$

Scope library block

$$Scope\_Scalar \mathrel{\widehat{=}} [In_1 : \mathbb{T} \to \mathbb{R}]$$

$$Scope\_V2 \mathrel{\widehat{=}} [In_1 : \mathrm{seq}(\mathbb{T} \to \mathbb{R}) \mid \#In_1 = 2]$$

Sum library block

$$
\begin{array}{|l}
Sum\_PP : \mathbb{T} \to \mathbb{P}[In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Sum\_PP(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = [\!\{Out = In_1 + In_2\}\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Sum\_PP(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!\{\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\}\!] = [\!\{Out = In_1(\alpha) + In_2(\alpha)\}\!]])
\end{array}
$$

$$
\begin{array}{|l}
Sum\_PM : \mathbb{T} \to \mathbb{P}[In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Sum\_PM(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = [\!\{Out = In_1 - In_2\}\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Sum\_PM(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!\{\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\}\!] = [\!\{Out = In_1(\alpha) - In_2(\alpha)\}\!]])
\end{array}
$$

$$
\begin{array}{|l}
Sum\_MM : \mathbb{T} \to \mathbb{P}[In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Sum\_MM(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = [\!\{Out = 0 - In_1 - In_2\}\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Sum\_MM(t) = [In_1, In_2, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!\{\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\}\!] = [\!\{Out = 0 - In_1(\alpha) - In_2(\alpha)\}\!]])
\end{array}
$$

$$
\begin{array}{|l}
Sum\_PPP : \mathbb{T} \to \mathbb{P}[In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T} \bullet (t = 0 \Rightarrow Sum\_PPP(t) = [In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid st = 0 \wedge \mathbb{I} = [\!\{Out = In_1 + In_2 + In_3\}\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Sum\_PPP(t) = [In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad t = st \wedge st > 0 \wedge [\!\{\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\}\!] = [\!\{Out = In_1(\alpha) + In_2(\alpha) + In_3(\alpha)\}\!]])
\end{array}
$$

Switch library block

$$
\begin{array}{|l}
Switch\_G : (\mathbb{T} \times \mathbb{R}) \to \mathbb{P}[In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T};\ th : \mathbb{R} \bullet (t = 0 \Rightarrow Switch\_G(t, th) = [In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = 0 \wedge th = TH \wedge [\!\{In_2 > TH\}\!] = [\!\{Out = In_1\}\!] \wedge [\!\{In_2 \leq TH\}\!] = [\!\{Out = In_3\}\!]]) \\
\quad \wedge \quad (t > 0 \Rightarrow Switch\_G(t, th) = [In_1, In_2, In_3, Out : \mathbb{T} \to \mathbb{R};\ TH : \mathbb{R};\ st : \mathbb{T} \mid t = st \wedge st > 0 \wedge th = TH \wedge \\
\qquad [\!\{\exists\, k : \mathbb{N} \bullet \alpha = k * st \wedge \omega = (k+1) * st\}\!] = [\!\{(In_2(\alpha) > TH \Rightarrow Out = In_1(\alpha)) \wedge (In_2(\alpha) \leq TH \Rightarrow Out = In_3(\alpha))\}\!]])
\end{array}
$$

Terminator library block

$$Terminator\_Scalar \mathrel{\widehat{=}} [In_1 : \mathbb{T} \to \mathbb{R}]$$

$$Terminator\_V2 \mathrel{\widehat{=}} [In_1 : \mathrm{seq}(\mathbb{T} \to \mathbb{R}) \mid \#In_1 = 2]$$

Unit Delay library block

$$
\begin{array}{|l}
UnitDelay : \mathbb{T} \times \mathbb{R} \to \mathbb{P}[In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T}] \\
\hline
\forall\, t : \mathbb{T};\ init : \mathbb{R} \bullet UnitDelay(t, init) = [In_1, Out : \mathbb{T} \to \mathbb{R};\ IniVal : \mathbb{R};\ st : \mathbb{T} \mid \\
\qquad st = t \wedge st > 0 \wedge IniVal = init \wedge [\!\{\alpha = 0 \wedge \omega = st\}\!] \subseteq [\!\{Out = IniVal\}\!] \wedge \\
\qquad [\!\{\exists\, k : \mathbb{N}_1 \bullet \alpha = k * st \wedge \omega = (k+1) * st\}\!] \subseteq [\!\{Out = In_1(\alpha - st)\}\!]]
\end{array}
$$

Appendix D

Handling Conditional Subsystems

This appendix complements chapters 7.5.1 and 7.5.2 to respectively deal with *triggered* subsystems with discrete control inputs and *enabled* subsystems with continuous control inputs.

D.1 Triggered Subsystems with Discrete Control Inputs

When the control input of a triggered subsystem is discrete, trigger events occurs only at sample time hits. In addition, there is no trigger event at time point 0 as the input is constant in the initial sample time interval. Note that discrete behavior in Simulink is piecewise-constantly continuous. We specify the behavior by constraining the values of subsystem inputs in terms of sample time intervals, which are *left-closed and right-open* and formed by a pair of consecutive sample time hits. We remark that triggered subsystems output the last value between any two events.

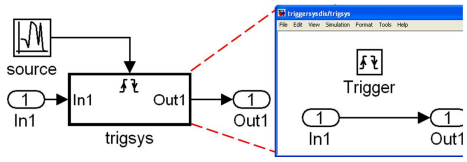


Figure D.1: A triggered subsystem controlled by a discrete input

As subsystem outputs are determined by subsystem inputs, it is thus necessary and important to mode the way of assigning subsystem inputs. Specifically the value of a subsystem input can come from the block which is outside the subsystem and connects to the input or be the last value which is obtained at the time point when the last event happens. Moreover, different trigger event types can lead to different kinds of situations. Particularly, according to the occurrences of trigger events at both endpoints of any sample time interval, if the type is *either*, there are six kinds of situations relevant to the assignment of input values; else the type is either *rising* or *falling*, and there are five kinds of situations since it is impossible that two events occur at a pair of sample time hits.

We take a simple system shown in Figure D.1 as an example. The control input of the triggered subsystem *trigsys* is connected by a source which outputs discretely, every 1 time unit. The type of trigger events is *either*. Namely, a trigger event occurs when the control input rises from a negative or zero value to a positive value or the control input falls from a positive or a zero value to a negative value.

The schema *sys_trigsys* shown below denotes the subsystem *trigsys*: the first predicate constrains that there is no trigger event at the time point 0; the second predicate captures that the time points where trigger events can occur are multiples of the sample time which is 1 in this example.

sys_trigsys

$$Trigger : \mathbb{T} \to \{0, 1\};\ In1, Out1 : \mathbb{T} \to \mathbb{R}$$

$$[\![Trigger = 0]\!] \subseteq [\![\alpha = 0 \land \omega = 0]\!]$$
$$[\![Trigger = 1]\!] \subseteq [\![\exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \alpha = \omega]\!] \land \mathbb{I} = (\!| In1 = Out1 |\!)$$

The following part of the schema *sys* represents the conditional execution of *trigsys* by six predicates. These predicates model the way to assign the subsystem input $trigsys.In1$ based on whether an event occurs at any ending points of every sample time interval, namely, checking $trigsys.Trigger(\alpha)$ and $trigsys.Trigger(\omega)$. Predicate1 and Predicate2 are concerned with the initial sample time interval: the default value of $trigsys.In1$ is 0 during the interval; and if no event happens at the ending point, the value 0 is assigned to $trigsys.In1$ at the ending point (expressed by Predicate1). The last four predicates deal with non-initial sample time intervals (by $[\![\exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1]\!]$). Predicate4 states that when events occur at both ending points, the value of the input port at the starting point ($In1(\alpha)$) is the last value to $trigsys.In1$ in the interval; moreover, if no event occurs at the ending point, one more constraint is added to assign the last value to $trigsys.In1$ at the ending point (by Predicate3). When no event occurs at the starting point but one event at the ending point, the last value during the interval is the input value at the starting point (by Predicate6); furthermore, if no event occurs at the ending point, we need to also assign the last value to $trigsys.In1$ at the ending point (by Predicate5).

sys

$In1 : \mathbb{T} \to \mathbb{R};\ trigsys : sys\_trigsys;\ \ldots$

$\ldots$ [Predicate1]

$(\!|trigsys.Trigger(\omega) = 0 \land \alpha = 0 \land \omega = 1|\!) \subseteq (\!|trigsys.In1 = 0 \land trigsys.In1(\omega) = 0|\!)$

$(\!|trigsys.Trigger(\omega) = 1 \land \alpha = 0 \land \omega = 1|\!) \subseteq (\!|trigsys.In1 = 0|\!)$ [Predicate2]

$(\!|trigsys.Trigger(\alpha) = 1 \land trigsys.Trigger(\omega) = 0 \land \exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1|\!)$
$\quad \subseteq (\!|In1(\alpha) = trigsys.In1 \land In1(\alpha) = trigsys.In1(\omega)|\!)$ [Predicate3]

$(\!|trigsys.Trigger(\alpha) = 1 \land trigsys.Trigger(\omega) = 1 \land \exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1|\!)$
$\quad \subseteq (\!|In1(\alpha) = trigsys.In1|\!)$ [Predicate4]

$(\!|trigsys.Trigger(\alpha) = 0 \land trigsys.Trigger(\omega) = 0 \land \exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1|\!)$
$\quad \subseteq (\!|trigsys.In1(\alpha) = trigsys.In1 \land trigsys.In1(\alpha) = trigsys.In1(\omega)|\!)$ [Predicate5]

$(\!|trigsys.Trigger(\alpha) = 0 \land trigsys.Trigger(\omega) = 1 \land \exists\, k : \mathbb{N}_1 \bullet \alpha = k \land \omega = k + 1|\!)$
$\quad \subseteq (\!|trigsys.In1(\alpha) = trigsys.In1|\!)$ [Predicate6]

D.2 Enabled Subsystems with Continuous Control Inputs

When the control input of an enabled subsystem is continuous, the subsystem executes whenever the value of the input is positive. Here we handle the case that enabled subsystems outputs the most recent values when it is disabled. To model the conditional execution, we use a similar method which has been applied for triggered subsystems to specify how to assign subsystem inputs appropriate values in two circumstances, namely, enabled and disabled. We further restrict that the intervals during which the control input values are positive are left and right-closed.

For example, Figure D.2 shows an enabled subsystem *enablesys* which is controlled by a continuous wave. This continuity feature is captured in the schema *sys_enablesys* which denotes the subsystem *enablesys*, specifically, by the symbol $\twoheadrightarrow$ in the declaration of the control input *Enable*.

$$sys\_enablesys \mathrel{\widehat=} [Enable : \mathbb{T} \twoheadrightarrow \mathbb{R};\ In1, Out1 : \mathbb{T} \to \mathbb{R} \mid \mathbb{I} = (\!|In1 = Out1|\!)]$$

Part of the following schema *sys* specifies the conditional behavior in three TIC predicates. Predicate1 indicates that whenever the subsystem is enabled, the value of the

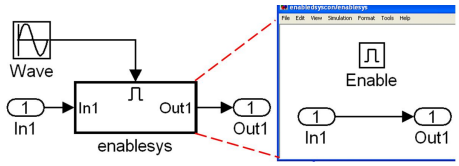


Figure D.2: An enabled subsystem controlled by a continuous input

subsystem input $enablesys.In1$ is assigned by the input port $In1$ which connects the subsystem input. Predicate2 and Predicate3 are concerned with the situation where the subsystem is enabled ($enablesys.Enable > 0$). Specifically, if the interval during which the subsystem is disabled starts with time point 0, the value of $enablesys.In1$ is 0 by default (expressed by Predicate2); else the interval starts with positive time point, and we assign the value of $enablesys.In1$ at the starting point as the last value within the interval (by Predicate3). Note that the reason for choosing the last value is similar to the one for handling triggered subsystems (as discussed in Section 7.5.1).

sys

$In1 : \mathbb{T} \nrightarrow \mathbb{R};\ enablesys : sys\_enablesys;\ \ldots$

$\ldots$

$[\![enablesys.Enable > 0]\!] \subseteq [\![In1 = enablesys.In1]\!]$ [Predicate1]

$[\![enablesys.Enable \leq 0 \land \alpha = 0]\!] \subseteq [\![enablesys.In1 = 0]\!]$ [Predicate2]

$[\![enablesys.Enable \leq 0 \land \alpha > 0]\!] \subseteq$
$\quad [\![enablesys.In1(\alpha) = enablesys.In1]\!]$ [Predicate3]