

---

# Run Time verification of Hybrid Systems

---

Ph.D Thesis

*Bader Alouffi*

This thesis is submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy

---

Software Technology Research Laboratory  
De Montfort University  
Leicester - United Kingdom

*May 2016*

# Dedication

This thesis is dedicated to my late mother **Dleel alharbi** who was supporting me on a daily basis since the day that i was born, and gave me all of the emotional support that I needed as well as her prayers. I will be ever grateful for all what she had done for me, and i am really sorry that she has not lived to see me graduate. It was her dream. And, only because of her, it became true.

A special dedication to my father **Mohammad alouffi** for all his prayers, sacrifices, and endless support without which I could not have accomplished my thesis.

To my wife **Samah Alquhtani** for her endless love and support, but most of all for her patience and dedication to the family and the home which allowed me to complete this thesis, and finally, a loving dedication to my son **Alwaleed** .

To my Brothers **Saleh, Suliman and Majed** and to my Sisters **Modi, Maha, Khlood and Kholah** for all the help and motivtion.

# Abstract

The growing use of computers in modern control systems has led to the development of complex dynamic systems known as hybrid systems, which integrates both **discrete** and **continuous** systems. Given that hybrid systems are systems that operates in real time allowing for changes in continuous state over time periods, and discrete state changes across zero time, their modelling, analysis and verification becomes very difficult.

The formal verifications of such systems based on specifications that can guarantee their behaviour is very important especially as it pertains to safety critical applications. Accordingly, addressing such verifications issues are important and is the focus of this thesis. In this thesis, in order to actualise the specification and verification of hybrid systems, Interval Temporal Logic(ITL) was adopted as the underlying formalism given its inherent characteristics of providing methods that are flexible for both propositional and first-order reasoning regarding periods found in hardware and software system's descriptions.

Given that an interval specifies the behaviour of a system, specifications of such systems are therefore represented as a set of intervals that can be used to gain an understanding of the possible behaviour of the system in terms of its composition whether in sequential or parallel form. ITL is a powerful tool that can handle both forms of composition given that it offers very strong and extensive proof and specification techniques to decipher essential system properties including safety, liveness

---

and time projections. However, a limitation of ITL is that the intervals within its framework are considered to be a sequence of discrete states. Against this backdrop, the current research provides an extension to ITL with the view to deal with verification and other related issues that centres around hybrid systems.

The novelty within this new proposition is new logic termed SPLINE Interval Temporal Logic (SPITL) in which not only a discrete behaviour can be expressed, but also a continuous behaviour can be represented in the form of a spline i.e. the interval is considered to be a sequence of continuous phases instead of a sequence of discrete states. The syntax and semantics of the newly developed SPITL are provided in this thesis and the new extension of the interval temporal logic using a hybrid system as a case study. The overall framework adopted for the overall structure of SPITL is based on three fundamental steps namely the formal specification of hybrid systems is expressed in SPLINE Interval Temporal Logic, followed by the executable subset of ITL, called Tempura, which is used to develop and test a hybrid system specification that is written in SPITL and finally a runtime verification tool for ITL called AnaTempura which is linked with Matlab in order to use them as an integrated tool for the verification of hybrid systems specification.

Overall, the current work contributes to the growing body of knowledge in hybrid systems based on the following three major milestones namely:

- i** the proposition of a new logic termed SPITL;
- ii** executable subset, Tempura, integrated with SPITL specification for hybrid systems; and
- iii** the development of a tool termed AnaTempura which is integrated with Matlab to ensure accurate runtime verification of results.

# Declaration

I declare that the work described in this thesis is original work undertaken by me for the degree of Doctor of Philosophy at the Software Technology Research Laboratory (STRL), at De Montfort University, United Kingdom.

No part of the material described in this thesis has been submitted for any award of any other degree or qualification in this or any other university or college of advanced education.

This thesis is written by me and produced using L<sup>A</sup>T<sub>E</sub>X.

**Bader Alouffi**

# Acknowledgments

First and foremost, my truthful thankfulness goes to the most merciful ALLAH for all the things he blessed me with throughout my whole life, without those blessings, I would not be here standing in this position at all. After studying for three degrees (including this one), at three universities, in three different countries, I have learned one important thing - I could never have done any of this, particularly the research and writing that went into this thesis, without the love, support and encouragement of a lot of people.

Most importantly, I would like to thank my supervisor, **Dr. Francois Seiwe**, whom without his support, encouragement and guidance this thesis would not have been possible to achieve. I am so happy that I was able to finish my Ph.D under his supervision. The love and care he offered to his students, including me, has affected this work in so many good ways. Also, many thanks and gratitude goes to my previous supervisor **Dr. Antonio Cau**, the one behind this project. For his critical comments, technical suggestions and professional guidance have always improved this thesis since day one.

I want to express my deepest thanks to **The Graduate School Office (GSO)**, for all the help and support.

Last but not least, I would like to thank every member of the **Software Technology Research Laboratory (STRL)** for providing the academic and home-like environment and the support whenever needed.

# Contents

Dedication	I
Abstract	II
Declaration	IV
Acknowledgments	V
Table of Contents	XI
List of Figures	XIII
List of Tables	XIV
Bibliography	XIV
List of Abbreviations	XV
Listings	XV
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Problem Statement . . . . .	5
1.3 Research Objectives . . . . .	5
1.4 Research Question . . . . .	6

1.5	Scope of the Research . . . . .	7
1.6	Research Methodology . . . . .	7
1.7	Success Criteria . . . . .	9
1.8	Thesis Structure . . . . .	9
<b>2</b>	<b>Literature review</b>	<b>12</b>
2.1	Introduction . . . . .	13
2.2	Hybrid System . . . . .	13
2.2.1	Systems definition . . . . .	13
2.2.2	Systems specification . . . . .	15
2.2.3	Formal modelling . . . . .	15
2.2.4	Formalism of Hybrid Systems . . . . .	16
2.2.5	Discrete and Continuous systems formalism . . . . .	16
2.2.6	Hybrid Systems control . . . . .	18
2.2.7	Hybrid Systems specification . . . . .	20
2.2.8	Hybrid Linear automata . . . . .	22
2.3	Formal Methods . . . . .	23
2.3.1	Formal methods specification . . . . .	24
2.3.2	Classification of formal methods . . . . .	25
2.3.3	Temporal Logic (TL) . . . . .	28
2.3.3.1	Time in temporal logic . . . . .	29
2.3.3.2	Temporal Logic classification . . . . .	30
2.3.3.3	Propositional versus First order . . . . .	31
2.3.3.4	Computational versus Linear Time . . . . .	32
2.3.3.5	Time points versus Intervals . . . . .	36
2.3.3.6	Duration Calculus . . . . .	39
2.3.3.7	Discrete or Continuous . . . . .	40



2.4	Runtime verification . . . . .	41
2.4.1	Contemporary Runtime verification Methods . . . . .	41
2.4.2	A conceptual view of Runtime verification . . . . .	43
2.4.2.1	Temporal Logic-based monitoring methods . . . . .	43
2.4.3	Runtime verification versus Model Checking . . . . .	45
2.4.4	Runtime verification versus Testing . . . . .	46
2.4.5	Runtime verification Applications . . . . .	47
2.4.6	Matlab and Simulink . . . . .	47
2.5	Summary . . . . .	48
<b>3</b>	<b>Preliminaries</b>	<b>50</b>
3.1	Introduction . . . . .	51
3.2	Interval Temporal Logic . . . . .	51
3.2.1	Syntax of ITL . . . . .	52
3.2.1.1	Expressions . . . . .	53
3.2.1.2	Formulae . . . . .	54
3.2.2	Semantics . . . . .	55
3.2.3	Derived formulae . . . . .	59
3.2.3.1	Derived constructs . . . . .	61
3.2.3.2	Derived constructs related to expressions . . . . .	61
3.3	An Executable subset of ITL (Tempura) . . . . .	62
3.3.1	The Language: Tempura . . . . .	63
3.3.2	The Tool: AnaTempura . . . . .	64
3.3.3	AnaTempura mechanism . . . . .	66
3.4	Summary . . . . .	68
<b>4</b>	<b>Spline Interval Temporal logic(SPITL)</b>	<b>69</b>
4.1	Introduction . . . . .	70

4.2	Spline background . . . . .	70
4.2.1	Spline types . . . . .	72
4.2.1.1	Linear Spline . . . . .	72
4.2.1.2	Quadratic Spline . . . . .	73
4.2.1.3	cubic Spline . . . . .	74
4.3	Spline Interval Temporal logic(SPITL) . . . . .	77
4.3.1	Discrete changes in SPITL . . . . .	78
4.3.2	Continuous changes in SPITL . . . . .	79
4.3.3	Syntax of SPITL . . . . .	80
4.3.4	Phase definition In (SPITL) . . . . .	81
4.3.5	Timed expressions definition In (SPITL) . . . . .	81
4.3.6	Semantics of SPITL Expressions . . . . .	82
4.3.7	Semantics of SPITL formulae . . . . .	83
4.3.8	Derived formulae . . . . .	84
4.3.8.1	Derived constructs . . . . .	85
4.3.8.2	Expressions derived constructs . . . . .	85
4.3.9	Discrete and Continuous changes Examples . . . . .	86
4.3.9.1	Discrete Examples . . . . .	86
4.3.9.2	Continuous Examples . . . . .	87
4.4	Spline example . . . . .	88
4.5	Summary . . . . .	89
<b>5</b>	<b>Runtime verivcation of hybrid system Framework</b>	<b>90</b>
5.1	Introduction . . . . .	91
5.2	General overview of the framework . . . . .	94
5.3	System specifications (SPITL) . . . . .	94
5.4	Modelling specifications in Tempura . . . . .	96

5.5	Matlab/Simulink (s-function) . . . . .	96
5.6	An Automatic function to Inject assertion points using AnaTempura . . . . .	99
5.7	Chapter summary . . . . .	102
<b>6</b>	<b>Design and Implementation</b>	<b>103</b>
6.1	Overview . . . . .	104
6.2	Simulink and Model based Implementation . . . . .	104
6.3	AnaTempura . . . . .	105
6.4	Steps to compiling the Design . . . . .	105
6.5	AnaTempura and Assertion point . . . . .	106
6.6	Matlab Engine . . . . .	108
6.7	S Function . . . . .	108
6.8	FIFO Pipe . . . . .	109
6.9	Simulink Model . . . . .	110
6.10	Summary . . . . .	112
<b>7</b>	<b>Case study and Evaluation</b>	<b>113</b>
7.1	Overview . . . . .	114
7.2	Mine pump system (the case study) . . . . .	114
7.2.1	Case Study Description . . . . .	114
7.2.2	Specification of mine pump system in SPITL . . . . .	117
7.2.2.1	Functional requirement . . . . .	118
7.2.2.2	Timing requirement . . . . .	119
7.2.3	Writing the requirement in Tempura . . . . .	119
7.2.4	C-Mex Code and S-function . . . . .	125
7.2.5	Simulink model . . . . .	128
7.2.6	Case study results . . . . .	129
7.3	Summary . . . . .	132

<b>8</b>	<b>Conclusion and Future Work</b>	<b>133</b>
8.1	Introduction . . . . .	134
8.2	Summary of Thesis . . . . .	134
8.3	Research Question revisited . . . . .	136
8.4	Criteria for Success and Analysis . . . . .	138
8.4.1	Extended ITL formalism to reason about hybrid systems . . .	138
8.4.2	Extended AnaTempura . . . . .	139
8.5	Future Directions . . . . .	140
<b>9</b>	<b>Appendix A</b>	<b>141</b>
9.1	Mine Pump Controller wrapper . . . . .	141
9.2	Mine Pump Controller . . . . .	144
9.3	Matlab Engine code . . . . .	159
9.4	Fifo Pipe . . . . .	161
<b>10</b>	<b>Appendix B</b>	<b>163</b>
10.1	Assertion points . . . . .	163
10.2	Tempura Code . . . . .	165

# List of Figures

2.1	Hybrid System fundmetal framework . . . . .	18
2.2	Temporal Logic classification [24] . . . . .	31
2.3	LTL path [16] . . . . .	33
2.4	CTL path [27] . . . . .	35
2.5	Points based . . . . .	37
2.6	Interval based . . . . .	37
2.7	Discrete time [22] . . . . .	40
2.8	Continuous time . . . . .	41
3.1	Chop of finite interval . . . . .	58
3.2	Chop of infinite interval . . . . .	58
3.3	Chopstar of finite interval . . . . .	58
3.4	Chopstar of finite interval final infinite . . . . .	59
3.5	Chopstar of infinite interval . . . . .	59
3.6	Tempura example . . . . .	64
3.7	The Analysis Process . . . . .	65
3.8	General System Architecture of AnaTempura[177] . . . . .	68
4.1	control points . . . . .	71
4.2	Linear spline . . . . .	72
4.3	Quadratic spline . . . . .	73
4.4	Cubic spline . . . . .	76

## LIST OF FIGURES

---

4.5	discrete changes . . . . .	78
4.6	continuous changes . . . . .	79
4.7	Discrete changes Example . . . . .	86
4.8	Continuous changes Example . . . . .	87
4.9	Leaking gas burner example . . . . .	88
5.1	General framework . . . . .	95
5.2	The integration of ITL/Tempura within MATLAB/Simulink using C-MEX S-function . . . . .	97
5.3	Illustration of how S-function is integrated with ITL/Tempura frame- work. . . . .	98
5.4	Illustration of assertion points (Adapted from kun thesis) . . . . .	100
5.5	Illustration of runtime verification based on AnaTempura . . . . .	101
7.1	Mine Pump System modified from [120] . . . . .	115
7.2	Matlab engine Code starting connection in AnaTempura . . . . .	120
7.3	Variable update on the Ana tempura external console . . . . .	121
7.4	Flow of S-Function [113] . . . . .	125
7.5	The Simulink Block . . . . .	129
7.6	the Simulink model scope plot . . . . .	130
7.7	Mine pump test cases results in AnaTempura . . . . .	131

# List of Tables

3.1	Syntax of ITL . . . . .	53
3.2	Semantics of ITL . . . . .	57
3.3	Derived formulae . . . . .	60
3.4	Frequently used concrete derived constructs . . . . .	61
3.5	Frequently used derived constructs related to expressions . . . . .	61
3.6	Operations in Tempura . . . . .	63
4.1	Syntax of SPITL . . . . .	80
4.2	Semantics of <i>SPITL</i> . . . . .	82
4.3	Semantics of <i>SPITL</i> formulae . . . . .	83
4.4	Derived formulae . . . . .	84
4.5	Frequently concrete derived constructs . . . . .	85
4.6	Frequently derived constructs related to expressions . . . . .	85

# List of Abbreviations

HS	Hybrid System
TL	Temporal logic
ITL	Interval Temporal Logic
LTL	Linear temporal logic
CTL	Computation Tree Logic
DC	Duration Calculus
SPITL	Spline Interval Temporal logic
FIPA	first-in, first-out pipe in C
CENG	MATLAB Engine API for C
CMEX	executable files for standalone MATLAB for C
S-FUNCTION	A computer language description of a Simulink block written in c
TL	Temporal logic
RVHSF	Run Time Verification of Hybrid system Framework
API	Application Program Interface



# Chapter 1

## Introduction

### *Objectives:*

---

- To present an introduction and research scope.
  - To identify the research problem statement and the motivation.
  - To highlight the research objectives and the success criteria.
  - To provide the adopted research methodology.
  - To provide thesis's outline.
-

## 1.1 Motivation

Essentially, any form of systems that is a mixture of continuous or real time dynamics and discrete events are collectively known as **hybrid systems**. These discrete and continuous dynamics coexist and interact producing changes in response to both discrete and dynamic events as described by a difference equation in time. They are a form of mathematical model for a part of the real world where discrete and continuous parts interact with each other. Such systems can model all kinds of situations, from biological systems [4] to a controller interacting with its environment [6], from electronic circuits [5] to mechanical systems [7]. In the context of the computer science community, a hybrid system is regarded primarily as a discrete (computer) program that interacts with an analogue environment. In computer science parlance, one of the key objectives in hybrid systems is to extend standard program analysis techniques to systems which encompasses some kind of continuous dynamics with much emphasis on the discrete event dynamics. In such systems, the main issue of concern to computer scientist is verification. One of the most important attributes of a hybrid system is to ascertain and specify its behaviour.

Due to the vast number of applications which can be modelled as hybrid systems, having efficient ways of analysing their behaviour enables us to learn useful information about the parts of the real world they model. Such analysis tells us about what happens as time evolves in a system, and we can then decide whether we are comfortable with the behaviour we see. When the behaviour of a dynamic system is analysed, it is important to ensure that two kinds of properties are satisfied:

1. The first property is to ensure that the hybrid system yields accurate information or output whether now or in the future. If the behaviour of such systems is determined to be accurate then one can be contended to leave the system to operate within the overall device that encapsulates it.

2. The second property is to be able to ascertain when the system provides information that are not accurate or complete from which the system can then be remodelled to accomplish what we desire and considered extremely important.

These types of properties have long been considered in the field of hybrid systems theory, with stability and captivate being key concepts that are of paramount importance. Stability is the idea that, if a system trajectory starts close to a point in space, then it will always remain close to that point in space. This is a property of the first type, where going far away from the point in space is a ‘bad thing’ or inaccurate output. On the other hand, captivate encompasses the idea that a system trajectory will keep getting closer and closer to a desired point in space, which is a property of the second type, where getting close to the desired point is a ‘good thing’ or an accurate output.

Stability and captivate capture the intuition that we have about good behaviour of a hybrid system, but are not easy to establish automatically. However, it is desirable to use automatic methods of analysis on hybrid systems to get a lot of information about a system within time, cost, resources and technical performance objectives.

Performing automatic analysis on a system allows us to think about it in a way which makes the best use of our intelligence, intuition, and time. The main challenge applying these constructs to hybrid systems is that a programmer must consider both the discrete and the continuous time behaviour to understand when and where a program execution must be suspended, by drawing inference from program properties that must be checked during runtime. The combinatorial state space explosion of hybrid systems complicates this task. Yet, a more fundamental limitation is the lack of a mechanism for controlling thread schedules that would enable a programmer to enforce his or her choices.

Verification hybrid systems is considerably hard given the unpredictability of

their execution which has the tendency of generating a new sets of program bugs. Moreover, standard tools for carrying out verification activities are ineffective despite a wide array of tools that have been constructed to assist programmers in verification hybrid systems. Under normal circumstances, such verification tools are expected to provide the programmer with maximum verification power with with little efforts of programming exerted, however in practice there always seem to be a trade off between the efforts of the programmer in using such tools and specificity which entails minimisation of false or inaccurate results.

In fact, popular bugs identified within the general properties of hybrid systems such as violations of atomicity and data races can sometimes prove difficult to trace and debug given the low specificity of such tools. Although such tools are proficient at verification some forms of bugs, they do not leverage on the knowledge of a programmer regarding their code, as such, properties that are implicitly specified may not correlate with real bugs. This is important given that not all forms of bugs are attributed to violations of atomicity or data races because it is possible that a program can be race free whilst still generating inaccurate results or make data corrupt depending on the application. Similarly, not all data races are bugs as research suggests that roughly 2-10% of reported data races incidence are harmful[1].

Implicit specifications has a competitive edge in that it is not required by a programmer to identify the exact properties to be checked for but also possess the disadvantage that the programmer may lack the requisite knowledge of the exact properties that are being checked. In fact, within the verification protocol community, there is a general lack of consensus regarding what data-race freedom and atomicity entails given that both terms have multiple definitions that are inconsistent.

In the light of the above, an important question that comes to mind is "what are the implications of this for the current study?" The answer to this important

question is presented in the problem statement as highlighted in Section 1.2.

## 1.2 Problem Statement

In the world of computing especially as it pertains to hybrid systems, the verifications of the level of correctness of computer programs that interacts with continuous environments is one of the key issues that computer scientists have tried to resolve. Detecting bugs in complex computer software systems is a challenging task and given the rate at which the software industry is growing, there is a massive interest towards the development of automated tools that can assist in the verification process. Accordingly, a number of formal verification tools which provides high specificity at the expense of a considerably high effort has been developed. Although such tools are a very powerful but they are equally very complex given that they require a great deal of skills to put them into correct use, thereby limiting their usefulness to ordinary programmers, small programs or abstractions of large programs. There is therefore the need to develop a robust yet simple framework that can be used for verification and simulation of the behaviour of a system in terms of its properties such as liveness and safety.

## 1.3 Research Objectives

The central aim of this research is to develop a framework that can be used to verify and simulate a computer system's behaviour in terms of safety and liveness properties, using executable subset of Interval Temporal Logic (ITL) and its extension for the development of a hybrid system termed Spline Interval Temporal Logic (SPITL). This entails the use of Tempura with subsequent integration with AnaTempura and Matlab in order to verify such a hybrid system model done within

Simulink. The intended outcome is to improve the interpreter Tempura by merging multiple assertion points thereby making them to receive the points.

## 1.4 Research Question

**How can Interval Temporal logic be extended in order to specify hybrid systems, which integrates both discrete and continuous systems. In order to provide properties that capture the dynamic behaviour of hybrid systems and how can these properties be formally verified at runtime and how can this verification can be inserted in to the hybrid system model in matlab simulink?**

We propose to address the overall research question, a set of research questions that tackle each of the underlying issues.

- RQ 1. What is the appropriate formalism technique that is required for the specification and verification of hybrid systems?
- RQ 2. What properties of a hybrid systems can be expressed in SPITL?
- RQ 3. Does the formalism have adequate tool support in order to simulate and verify hybrid systems?
- RQ 4. How can we describe the behaviour of hybrid systems using Interval Temporal Logic?
- RQ 5. How can we characterise the whole time interval instate of characterising fixed points on the interval?
- RQ 6. Can we have new operators in ITL that can deal with states durations?

RQ 7. Can the proposed extension of ITL be used to reason about hybrid systems?

RQ 8. How do we verify at runtime the behaviour of hybrid system under investigation using our framework?

## 1.5 Scope of the Research

In order to propose solutions to the problems and research questions outlined in the preceding section, the aim of this thesis is to present a formal approach for the specification of hybrid systems using an extended formalism from the well-known logic ITL, called Spline Interval Temporal logic. Subsequently, an integrated framework for the specification and runtime verification of hybrid systems is provided. Development of such a framework requires the following components:

- Defining the system behaviours.
- Specifying the properties using SPITL.
- Modelling the system behaviour in Tempura.
- Communicating the Run time verification tool AnaTempura with the external hybrid system simulation platform, Matlab Simulink.
- Verifying the systems in order to prove if the system satisfies its properties.

## 1.6 Research Methodology

The research methodology adopted in this research is based on constructive research approach whereby a contribution to knowledge is based on the development of a new solution to an identified problem. Accordingly formal framework is developed for known problem which pertains to the inability of the run time verifier, AnaTempura,

to verify hybrid systems using assertion points approach. This was achieved by improving the interpreter Tempura by refining the assertion point feature enabling it to merge multiple assertion points in real time. The overall methodology involves four distinct steps as follows:

- **Step 1: Overview and background**

This step highlights the basic concepts of the runtime verification used in hybrid systems. It shows the deferences and features of the hybrid systems in general. Then, it shows the hybrid systems properties of interest and associated issues. Additionally, it lays the foundation to the understanding of all approaches upon which the current research problem is based which is derived as a gap required to be filled from previous studies. Accordingly, a detailed studying and understanding of past work within the same can assist in the recognition of their weakness and boundaries from which the basis of the current work is established.

- **Step 2: Architecture**

This step introduces the proposed framework. It defines the general conceptual framework and the main components as it relates to the overall research problem. It explains how individual entities of the research interact with each other with the view to achieve the expected aim of the research. Additionally, it shows the logical background of the framework and the techniques to be adopted.

- **Step 3: Implementation**

This step presents the implementation of the overall framework. The implementation includes hybrid systems properties of interest and the expected behaviour of the system during the runtime. It entails formal specification of hybrid systems expressed using Interval Temporal Logic (ITL) and its ex-



tension SPITL, and a formal model and verification of hybrid systems using AnaTempura and Matlab.

- **Step 4: Evaluation**

This step pertains to how the capability of the proposed framework can be ascertained and how its associated components can be validated and verified within the hybrid systems

## 1.7 Success Criteria

The success of this research will be measured based on how the aforementioned research objectives are accomplished.

Thus, the accomplishment criteria of the objectives as following:

- Extending the Interval temporal logic in order to specify the hybrid system which can express both discrete and continuous behaviour of a hybrid system model.
- Improving the interpreter Tempura by refining the assertion point feature and make it able to merge multiple assertion points in runtime.
- Improving the interpreter Tempura by refining the assertion point feature and make it able to merge multiple assertion points in runtime.
- Linking AnaTempura and Matlab to establish communication between them whilst ensuring that sending and receiving inputs are guaranteed.

## 1.8 Thesis Structure

This thesis report is organised into 8 chapters as follows:

- In this chapter a brief overview and outlines of the motivations, research objectives and methodology, success criteria and overall structure of the thesis are presented.
- Chapter 2 presents a comprehensive and underlying description of the most relevant aspects of temporal logic and hybrid systems. The chapter starts with a brief overview of temporal logic, types and tools. Then, hybrid system background are presented in the following subsections. Finally, hybrid system related works are discussed.
- Chapter 3 provides an overview of temporal logic in general and ITL in particular, showing its syntax and semantics and presents a justification of our selection of ITL. Tempura the executable subset of ITL are discussed using examples. In addition, a review of the Anatempura tool is presented, followed by a discussion of its features and architecture and its use in our framework.
- Chapter 4 shows the proposed extension to Interval Temporal called, Spline Interval Temporal Logic (SPITL), in which not only a discrete behaviour can be expressed, but also a continuous behaviour can be represented by a form of a spline. The syntax and semantics of the SPITL are presented in this chapter.
- In chapter 5, detailed description of the proposed framework and architecture as well as its components are presented. Each component and their interaction with each other are described in detail.
- Chapter 6 presents the implementation of the proposed framework and formalises different properties of hybrid systems. In this chapter, testing is illustrated using AnaTempura and Matlab. Similarly, correctness verification for the abstract model is illustrated.

- Chapter 7 provides the evaluation criteria and the results of implementing the framework components. The results will be used to measure the effectiveness of the proposed framework. This chapter compares our extended framework with other existing frameworks that tackle similar problem.
- Chapter 8 presents the main conclusions that stems from this research and identify key areas and new directions for further work.

# Chapter 2

## Literature review

### *Objectives:*

---

- To describe the concept of Hybrid systems.
  - To explain the formal specification.
  - To introduce Temporal logic and its types.
  - To review the runtime verification methods.
  - To investigate the related work
-

## 2.1 Introduction

In this chapter, a detailed review of extant literature related to the current work including the background to temporal logic and its applications are presented. It discusses the specification and its uses and the most relevant terms related to specification, such as formal specification and formal specification approaches. Additionally, temporal logic history, and how we can classify temporal logic, as well as the temporal logic applications, is discussed. The chapter concludes with a review of runtime verification methods.

## 2.2 Hybrid System

A hybrid System is a system consisting of a collection of a continuous-valued and discrete variables. Hybrid systems arise in embedded control when digital controllers, computers, and subsystems modelled as finite state machines are coupled with controllers and plants modelled by partial or ordinary differential equations or difference equations. Hybrid systems model exist in many important applications such as air traffic management systems [6,7], highway systems [7,8], and manufacturing [10]. In the sub-section that follows, the classification of hybrid systems detailing important background issues are presented.

### 2.2.1 Systems definition

*Definition:* In the IEEE Standard Dictionary of Electrical and Electronic Terms a system was defined as “a combination of components that act together to perform a function not possible with any of the individual parts” . It is a set of components or interdependent components that interact with each other to become a single entity or an integrated whole. It then follows that one component depend on the other for

functionalities within the overall system [1,2]. The general properties of systems are studied in a number of fields including computer science, control theory, dynamical systems, mathematical programming, discrete systems, hybrid system, system engineering and simulation languages. In such fields, abstract properties of a system are investigated with the view to gain an understanding of the principles and concepts that are independent of domain, temporal scale or types [1,2] fundamentally, every system has a structure which contains its components or associated parts that are interlinked with each other either directly or indirectly; behaviour which entails processes that transmogrify inputs into outputs; interconnectivity whereby every parts or processes within the system are interlinked based on structural and behavioural relationships. Accordingly, the structure or behavioural pattern of a system can be disaggregated into subsystems or sub-processes [1,2]. Some systems share common characteristics, including: [2]

- A system has structure, it contains parts (or components) that are directly or indirectly related to each other.
- A system has behaviour, it contains processes that transform inputs into outputs.
- A system has interconnecting: the parts and processes are connected by structural and/or behavioural relationships.
- A system's structure and behaviour may be decomposed via subsystems and sub-processes to elementary parts and process steps.

The term system may also refer to a set of rules that governs structure and/or behavior. Alternatively, and usually in the context of complex systems, the term institution is used to describe the set of rules that govern structure and/or behavior. [2]

### 2.2.2 Systems specification

In systems theory, there is a fine distinction between system structure and system [2] The external behaviour of a system pertains to the connection between its input and output. The

input/output of the system's behaviour is made up of input time segments matched with out-

put time segments derived from a real model or system . In terms of the underlying structures, a system is made up of its state and transition mechanism (illustrating how inputs

are transformed from a current state into other states). A deep knowledge of the system structure enhances the ability to specify, analyse, simulate and verify its behaviour[2,3].

### 2.2.3 Formal modelling

**Definition:** A model is generally defined as a description of a system or subsystem using formal terms that covers a given set of knowledge or information [4]. Therefore, among the characteristics of a system is that it should be formal and should contain information in a way that is consistent and unambiguous. Additionally, a model must possess some level of abstraction that can be represented only by selected information. This is important given that the choice of right model is a complex activity when it comes to abstracting information as it is mainly driven by the perspective of the modellers and his/her overall objectives to be realized [4].

The systems or subsystems formal specification within models is an imperative for most of the activities, such as:

- Checking the system properties.

- Simulation so as to be able to check the behaviour of the system.
- Mathematical proof techniques for verification of the behaviour of a system.

### 2.2.4 Formalism of Hybrid Systems

Essentially, any form of systems that is a mixture of continuous or real time dynamics and discrete events are collectively known as **hybrid systems**. However, Before discussing hybrid systems it is important to have an understanding of some terms relating to it such as continuous and discrete dynamics (will be discussed in next section below). A dynamical system has its roots from the mathematical science which is time-dependent.

In other words, the system has various states and processes depending on what time it is operating. Some examples include water flow in a pipe, the pendulum of the clock. On the other hand discrete dynamics depend on fixed points, and can be integrated with the continuous dynamic.

The hybrid system as illustrated in figure 2.1 contains both the continuous and discrete dynamics, which can give it a great deal of flexibility, and hence can be used for applications especially nowadays, with the wealth of dynamic applications. Finally, the term hybrid automaton, consists of discrete state transitions and continuous evolution, which will be discussed later.

### 2.2.5 Discrete and Continuous systems formalism

The general differential equation systems, that have continuous states and continuous time, were derived as the class of continuous Time Systems (CS). Also, automata, for example, and other systems that operated on a discrete time base were derived as the class of (DS). The next major advancement in systems formalisms was the combination of discrete systems and continuous systems formalisms into



one consistent hybrid systems [HS] [1, 2]. In the sections that follows a background description of these formalisms, discrete systems, continuous systems and Hybrid systems are presented. Given that hybrid system is the main focus of the current research, a greater deal of attention is given to their specification, simulation, and verification. A more fundamental choice is that between Discrete or Continuous of a flow of time. It implies that it would be composed of a sequence of instances where each non-final point is followed by another immediate point. We can therefore say that a property is correct in the following moment and also correct all time or at some future time. This can be formulated in first-order logic:

$$\forall x, y(x < y \rightarrow \exists z(x < z \wedge z \leq y \wedge \forall w(x < w \wedge w \leq y \rightarrow z \leq w)))$$

Temporal logics mostly used for program reasoning consider time as discrete where the present instant matches to the program's present state and by the finite model property. Hence the temporal structure which matches with a series of states of a program execution is the non negative integers. continuous refers to a linear ordering in which we can find another different point between any two distinct points. This can be mathematically represented as

$$\forall x, y(x < y \rightarrow \exists z(x < z < y))$$

The idea of the flow of time can be modeled using rational or the real numbers, which can represent the flow of continuous time [22, 23]. Philosophers have been studying tense logics interpreted over a continuous time structure. Cau in [21], proposed the application of dense time temporal logics to reasoning about concurrent programs. Dense time temporal logics can also be used in real time programs where strict, quantitative performance requirements are placed on programs [30].

### 2.2.6 Hybrid Systems control

a system that has processes of distinct traits that will lead to the desired result, more specially, interacting continuous and discrete dynamics as shown in Figure 2.1. The main characteristic of of Hybrid systems is that they generate mixed signals that include a combination of discrete-valued and continuous signals. In essence, some of these signals are values derived from a continuous set of real numbers, for example, while other values are derived from a discrete finite set of symbols denoted by a,b,c.

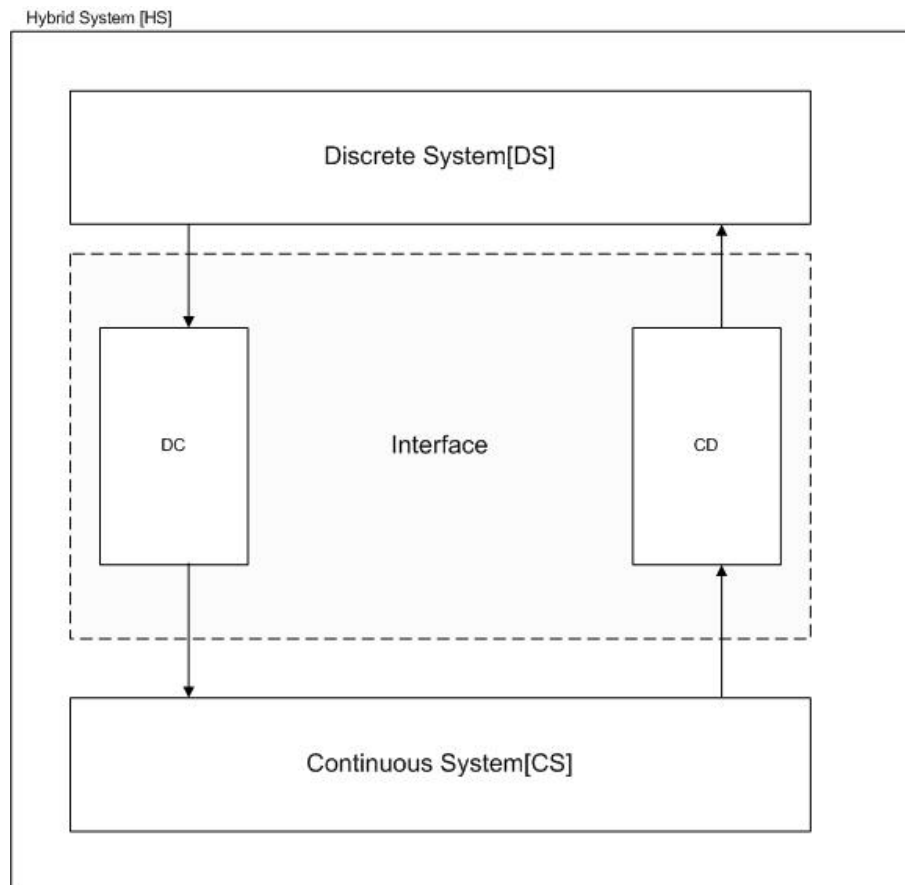


Figure 2.1: Hybrid System fundmetal framework

In manufacturing, for instance, different components may be produced or processed in specific machines. However, only the sequence or arrival of a component would trigger the process. In essence, manufacturing process comprises the events

that are driven by the dynamics of processes involved and the moving parts of the machines. The time-driven dynamics were often studied in isolation from the event-driven dynamics. While time-driven dynamics made use of Petri net models or automata or even PLC, logic expressions, event-driven dynamics were studied through the based on knowledge derived from differential equations.

To meet the high performance specification, and to understand the system's behaviour, we need to model all dynamics as well as their interlinkages and interactions. Optimisation of the entire process of manufacturing must be tackled in a meaningful and consistent way. In instances whereby event and dynamics driven by time are not coupled in a tight fashion or the system performance requirements are difficult to actualise, simpler models that are constructed separately for separate phenomena can be adequate.

However, the best response can be derived from hybrid systems when they are applied in instances where the interaction between the discrete and continuous parts are noticeable and when they are expected to deliver on specifications whose performance are expected to be high.

Hybrid models may be deployed to greater advantages especially in areas like control of automotive engine where control algorithms with specific and tailored properties are implemented based on embedded controllers, capable of reducing gas consumption, and emissions consumption without compromising the performance of the car.

To fulfil the highly challenging design requirements in control systems designs for issues such as idle speed control or cut off frequency of an engine [11], hybrid models that are capable of representing behaviours that are based on events and time can come really handy and must be deployed accordingly. Also some processes or systems, that requires accuracy, and demand high performance such as, the chemical related processes, manufacturing of robotic systems, air traffic control systems

as well as transportation systems, , would greatly benefit from the hybrid system models [12, 13, 14, 15, 16].

Hybrid systems emerge from the interlinkage between algorithms for discrete planning and those for continuous processes, thereby providing the basic methodology and framework for the synthesis and analysis of intelligent and autonomous systems [17]. Essentially, hybrid systems are important in designing supervisory controllers for continuous systems, and also key in designing intelligent control systems with a high degree of independence.

The hierarchical organization of complex systems is another important way hybrid systems emerge in which a hierarchical structure assists in managing complexity given that higher levels in the hierarchical structure require models that are less detailed (e.g. discrete abstraction) regarding the functioning of the lower levels thereby prompting the interlinkage between continuous and discrete components [18].

### **2.2.7 Hybrid Systems specification**

The approaches of hybrid systems are different depending on the level of complexity of the discrete or continuous dynamics as to whether it stress the importance of the synthesis and analysis of results or consider results only or simulations only. Some hybrid systems are an extension of theoretical idea of a system that are developed based on ordinary differential equations which include discrete time and variables that are applicable to systems with switching mechanism.

There are additional approaches , in which “intelligence” derived from continuous control systems based on linear and non-linear differential equations are combined with supervisory control of DESs that are based on finite automata and Petri nets to derive, with disparaging success, in terms of synthesis and analysis of results.

The availability of efficient simulation and analysis software tools for the design of hybrid systems, is important because of the complex nature of hybrid systems – a

fact that is well-recognized by the research community of hybrid systems developers and a lot of software program have been developed to tackle their complexity

Few of the available software used for such purposes will be discussed here but it is important to state here that the pool of software applications changes with time as progress and advancement is attained in hybrid system research. As such, simulation and modelling tools are expected to be developed with robust algorithms so that they can address so that problems that arise due to the interfacing of discrete and continuous dynamics can be addressed swiftly [10, 11, and 12].

Matlab, Simulink, and Stateflow software [13] are software tools that can be adopted for visual modelling and simulation of hybrid systems that are based on discrete-time, continuous-time and dynamics that are driven by events and can make the use of hybrid systems in terms of testing, implementing, and debugging much more easier.

Ptolemy II [14] is a set of software tool that supports concurrent and heterogeneous modelling and hybrid system design. It supports many computational models such as finite state machines, continuous-time systems, discrete event systems as well as the suitable interfaces that facilitates the modelling and simulation of hybrid system through the efficient coordination of the interaction of these interfaces.

Modelica [16]-an object oriented language which was developed for paradigms and physics modelling of hybrid systems. Simulation software and tools including MathModelica and Dymola which are compatible with Modelica can also be applied to conduct simulation of physical systems that shows the characteristics of hybrid systems[17,18].

HCC [19] is also an object oriented language that supports the modelling of the dynamics of hybrid systems. Another programming language known as Shift was also developed for the description of the dynamic networks of hybrid automata. Its development was motivated due to the need for the high level specification require-

ments and analysis of applications in the automotive industry.

Shift has been adopted in various domains of application. OmSim is a software tool for modelling and simulation activities based on Omola, which is equally an object oriented language used for the representation of discrete-event and continuous-time dynamical systems [20,21].

Charon is a coding language for modelling hierarchical structures in hybrid system. Similarly, HyTech has been used in hybrid systems for verification purposes [22,23]. Kronos [24] and UPPAAL [25] are real-time systems modelled by timed automata for the sole aim of verifying hybrid systems. software tools have also been developed and applied for various purposes in the chemical industry [3].

### **2.2.8 Hybrid Linear automata**

Hybrid automata were introduced originally in the early 1990s [27] and they provide a modelling formalism which can be used as a basis for algorithmic analysis and specification of hybrid systems. They are adopted to construct dynamical systems that comprises of analogue and discrete components which come into play when an interaction is established between and the physical world in real time.

Many of the proposed systems got a good attraction because of their simplicity, but as application's requirements increase, and as the complexity increases, these systems are not satisfactory for the up to date applications. A great a example of Hybrid Linear automata, the Hybrid Automata (HA) [40] is a FSM where each state is characterized by a set of continuous variables and equations to express the system when in that state. Movements from one transition to another is triggered either by external actions or when a certain condition is satisfied. Each transition is labelled with a guarded command to be executed when the transition takes place, all of the above frameworks were developed having in mind systems where the discrete component is dominant and the continuous one is relatively simple. Therefore, their

application to chemical processing systems becomes problematic, when faced with a substantially more complex continuous element and tight interactions between the discrete and continuous parts. Pantelides proposes a general framework for discrete and continuous processing systems operating in the continuous time Domain (1995).

## 2.3 Formal Methods

Formal methods entails methods that have very strong mathematical basis in their constructs. They are distinct from structured methods whose constructs are defined properly but lack high level mathematical basis for the description of the functionalities of hybrid systems [17]. Formal methods allow for the precise specification of system functionalities while structured methods allows for the accuracy of a system structures specification. Formal method consists of some vital components including a semantic model, a notation which act as the language for specification, a verification system and refinement calculus, guidelines for development and supporting tools[42]:

- The semantic model is a logical structure or sound mathematical framework where all terms, formulae, and rules used have meanings that are concise and precise The semantic model should have a reflection of the fundamental computational model of the application under consideration.
- The language of specification is a set of notations that are employed for the description of the behavioural pattern of the system under consideration and it must possess adequate semantics within the semantic model.
- verification system and refinement calculi are complete rules that ensures that properties and specifications are verified and refined.
- Development guidelines are steps that depicts how the methods are employed.

- Supporting tools include extensions like proof assistant, a syntax, an animator, and type checker, and a prototyper.

### 2.3.1 Formal methods specification

In the past, specification may have been written in natural language or informal language. Because of that, producing formal specification was not part of common software engineering practice [22]. Software developers were not usually familiar with using formal specification languages, and training in using these languages was both time consuming and expensive [23]. However, today the specification are written in formal specification languages such as temporal logic, so we are translating a non mathematical description, such as English and diagrams, into formal specification language [24].

What is more, the formal specification, which uses mathematical notation, is used precisely to describes the functionality, structure and interfaces of software systems. This process does not include the programming languages details needed to produce an implementation [23].

The reason behind this is that the system developer works at a higher level of abstraction than the programmer, so, they have the chance to define system functionality concisely without worrying about other aspects of implementation that they have nothing to do with, such as the functional behaviour of the system, algorithms, efficiency and memory management[25].

This abstraction decreases the specification error rate and removes the confusion that such details bring to the specification reader, and allows him to recognize the defined functionality. This permits the verification of implementation [26].

A formal specification provides a dependable point of reference for researchers who want to study the customers needs, those who execute the programs in order to ensure that the needs are met, those who evaluate the outcome of the execution,



and those who write instruction manuals for the system.

Formal specification of a system can be concluded in the early stages of program development, since it is not dependent on the program code. This formal specification has to be modified as the design progresses and the designers better understand the customers needs. But it is a powerful tool creating a mutual understanding among all parties involved in the system.

According to Gehani [27], formal specification are used for several reasons which are:

- Uncertainties, oversights and inconsistencies can be detected in the formulation of informal problems informal in the entire process leading to formalisation.
- The correctness of the model based on formal framework can be ascertained by mathematical means.
- Analysis can be conducted on a system that is specified in a formal way to possess or not to possess wanted properties.
- A formal specified system can be integrated within a larger system with additional level of certainty.
- The formal model (partly) forms the basis of automated development methods and tools like simulations.
- For systems designs that are specified based on formal protocols , comparison between components can be achieved can be easily compared with each other.

### **2.3.2 Classification of formal methods**

To write detailed formal specification for any software systems, five basic approaches have been used, these are:

- **Algebraic approach:** This approach emerged in the mid 70s as a technique to deal with data structures in an implementation-independent manner. In this approach, implicit definition of operations are given by linking the behaviour of different processes without defining state. An example of this approach is OBJ language[41] and PLUSS. In this sense, equational logic [42], a branch of first order logic, constitutes that part which deals exclusively with sentences in the form of identities chosen as the specification formalism and universal algebra and category theory provided the underlying semantical techniques [43].
- **Model-based approach:** In this approach, we build the system model using familiar mathematical constructs such as sets and sequences. The system operations have been defined as modification of the system state[44]. Unlike algebraic specification, the state of the system is not hidden and the state changes are straightforward to define, but again there is no explicit representation of concurrency; this is the approach most widely used by Z notation[45] and Vienna development Method (VDM)[46].
- **Process Algebraic approach:** This approach is an explicit model of concurrent processes which represents behavioural pattern through constraints on the communication between the processes that are allowed to be observed (e.g  $\pi$ -Calculus[47] and calculus of communication systems (CCS) [50]).
- **Logic-based approach:** Here, properties of systems, such as specification of program behaviour at a low-level and specification of behaviour in terms of system timing. An example of this approach is Temporal and Interval Temporal logic (e.g. the method that we are considered on our work)[51]. will be discussed on more details later on this chapter and chapter 3.
- **Net-based approach:** In this approach, an implicit synchronized model of

the system based on (causal) data flow via a network, such as the representation of situations under which data can flow between two nodes of a network. An example of this approach is Petri nets, and predicate transition networks [52].

Methods and their associated tools which supports the verification and analysis of Hybrid system are available. For example, **HyTech** by Dr. T.A. Henzinger [5] “ is a symbolic model checker for linear hybrid automata ”, which is the one that is mostly related to the our study [4], and has a subclass of hybrid automata that can be automatically analysed through the computation of polyhedral state sets. A distinguishing characteristics of HyTech is its inherent capability to carry out parametric analysis, i.e. to identify the values of the parameters for the design of a linear hybrid automaton system to ascertain whether it satisfies a requirement based on temporal-logic. HyTech is regarded as the most successful in terms of application to systems that entails a complex interplay between continuous and discrete dynamics.

[60] and [61] developed a framework based on a logic for real-time systems (Real Time Logic (RTL)) and a language for system specification known as Modechart. RTL was first developed by [62], and was derived from the work of Harel [63]. A methodological framework for the verifications of systems properties identified in modechart was detailed in [65]. A limitation of both Modechart and HyTech is that they are only appropriate during the development phase of hybrid systems requiring formal verification. They lack the ability to process code-level analysis from source using certain properties as benchmark. Furthermore, both formalisms are not compositional, rendering them potentially useless for evolution of large scale systems. A new study on the monitoring of real-time constraints using RTL[70] and interval model checking is based on Linear Time Logic (LTL). Although the analysis and the underlying logic presented in the study are suitable for the expression of

real-time properties, they both lack compositional attributes and also lack the ability to handle analysis of source code. However, the current work seeks to address this limitations as it will be shown later.

Temporal Rover [72] is a tool used for specifying and verifying and/or validating systems protocols and systems that are reactive. With the tool, verification of real-time events and temporal properties that are relative can be automated. The formal specification is developed by integrating Temporal Logic [13] and a programming language such as C, C++ and Java. Temporal-logic assertions are enclosed as part of codes that are executable in combination with formal specifications and can be simulated using the Temporal Rover simulator but suffers a limitation in that its verification ability is based on simulations and is not compositional methods. Furthermore the pre and post conditions analysis it provides are not sufficient for tackling complex parallelism.

### **2.3.3 Temporal Logic (TL)**

Temporal logic has become one of the most important formalisms for specifying, verifying and reasoning about systems that interact with their environment [13]. The formal language with its proof theory, decision algorithms and associated method of practical application, has found many uses in dealing with programs [14].

Temporal logic is considered to be a very suitable formal method for specifying and verifying concurrent and reactive systems [17]. By ‘ temporal logic ’ we mean “ a family of logics and logical techniques which can be applied to a wide array of problems, both abstract and concrete ” [18].

Temporal logic formulas can describe sequences of state changes and properties of behaviours, and, hence, can span a wide range of problems in various fields with a richer notation [19].

As temporal languages are increasingly employed to cover a variety of uses, as

mentioned above, there is growing interest to include the use of past operators to the temporal logic languages [27].

In the next sections, we will give an overview of temporal logic starting from the models of time.

### **2.3.3.1 Time in temporal logic**

Time has been studied in disciplines such as physics, philosophy and computer science. It has been one of the most paradoxical concepts of philosophy throughout history [14,17]. The concept of time has been studied in order to introduce a satisfactory definition of time since there is no common understanding of time that has been given till now. The main reason is that each definition has covered some aspects of time whilst excluding others. The time concept has been studied in various disciplines in order to introduce a common language for time.

In many science applications such as physics, mathematics and first order predicate calculus, which is used to reason about expressions containing the time variable, time has been represented as another variable. Therefore, there is apparently no need for a special temporal logic [12,14].

In philosophy, temporal logic has been an important subject, as some of the ancient philosophers used some form of temporal logic to analyse the structure of time. Plato[23] defined it as the ‘ moving image of eternity ’ while Aristotle described it as ‘ the number of motion with respect to earlier and later ’. Philosophers found it useful to introduce special temporal operators for the analysis of temporal connectives in languages. The verbs ‘ incipit ’ (it begins) and ‘ desinit ’ (it ends) are found in Aristotle’s Physics books [21,22]. These new operators were soon seen as potentially valuable in analysing the structure of time [25].

Classical logic deals with timeless propositions, so logic formulas can characterize only static states and properties. Temporal propositions typically contain some

reference to time conditions, so temporal logic formulas can be used to describe sequences of state changes and properties of behaviours. Therefore, temporal logic can cover a wide range of problems in different fields and areas with richer notations [15].

The various temporal logics can be adopted in conjunction with qualitative temporal properties:

- **Safety:** nothing bad happens to the system.
- **Liveness:** something good eventually happens to the system.
- **Fairness:** something good happens fairly.

Depending on the view of time (whether time is linear or branching, or whether time is discrete or continuous) and the types of temporal semantics (interval semantics, point semantics, linear semantics, branching semantics and partial order semantics), we can classify temporal logic. In the next section, we will discuss the classification of temporal logic systems in details.

For an appropriate definition of any temporal logic, the following are necessary:

- **Syntax:** the language for describing the time or temporal systems;
- **Semantics:** the model of time to derive the meaning of a logic formula.

The main question we need to ask is what is the system structure of time that should be used? (model of time)[12].

### 2.3.3.2 Temporal Logic classification

Most temporal logic can be classified along a number of axes. We will list the most popular axes that can be used to classify temporal logic systems which are:

- Propositional versus first order.

- Linear versus branching.
- Points (instances) versus intervals.
- Discrete versus continuous.
- Past versus future tense.

as are shown in the next Figure 2.2.

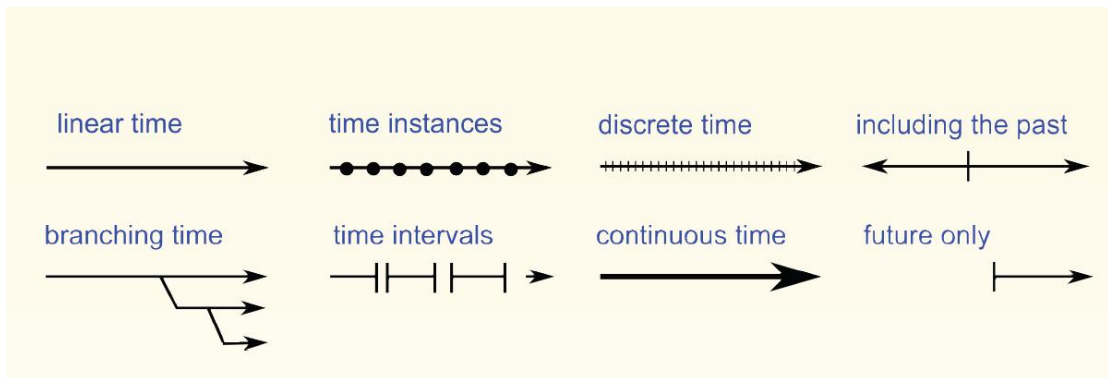


Figure 2.2: Temporal Logic classification [24]

Next, the most common criteria to distinguish between temporal logic systems is described [12,13,14].

### 2.3.3.3 Propositional versus First order

Propositional temporal logic is similar to the classical propositional logic. In propositional temporal logic, problems are expressed in generic language such as the set of propositional letters, the classical propositional connectives  $\neg$ ,  $\vee$  and  $\wedge$  and a set of temporal operators [55,56]. When creating a program from formal specification it is crucial to use propositional temporal logics since they have the finite model property. The created model is similar to a finite state machine; but, the model accepts infinite strings.

First order temporal logic (FOTL) is similar to predicate logic. Different kinds of FOTL have been suggested[14]; however the generic language consists of predicate symbols, variables, constants, boolean connectives and temporal operators[18]. A difference can also arise as a result of enabling or disabling restrictions on the interaction of quantifiers and temporal operators.

Lack of restrictions or freedom in some cases might lead to logics that cannot be decided. For instance, enabling modal operators within the freedom of quantifiers can cause a serious problem. On the other hand, one can have a restricted FOTL composed of propositional temporal logic together with a first order language for defining the atomic propositions by disabling such quantification over temporal operators[56].

#### **2.3.3.4 Computational versus Linear Time**

There are two main contrasting views that have tried to explain the structure of time. One view is that the course of time is linear because time flows in only one direction and the other view is that time has a branching tree like nature.

According to the theory of linear time, at any instant there is only one possible future moment [26]. According to the branching theory of time, at each moment of time, time can split into alternate courses portraying different possible futures, which mean that at any moment, time has many futures but only one linear past[14]. So, if linear temporal logic has the linear structure of time we call it linear time logic (LTL); however we call it branching (computational) time logic if it has the branching time structure [14,22,23,24].

Depending on the two views stated above, we can classify a system of temporal logic as either a linear time logic or a system of branching time logic. The nature of time assumed in the semantics is normally reflected in the temporal modalities of a temporal logic system [23].



When it comes to a linear time logic, the flow of events can be explained along a single time line in temporal modalities. On the other hand, in branching time logic systems, modalities enable quantification over possible futures. We can get different logics by changing the structure of the language of the logic in both linear and branching time temporal logic systems [14,15].

Linear Temporal Logic (LTL): This is a widely accepted type of formalism which is useful for the specifying and verifying systems that are concurrent and reactive [16]. Within LTL, time is modelled as a sequence of states which is sometimes known as computation path. In general, the future is not determined, so several paths are considered, representing different possible futures, from which any path can happen to be the main path that is achieved as shown in Figure 2.3



Figure 2.3: LTL path [16]

*-Formula of LTL:* The formula in LTL is defined inductively as follows:

- $\top$  and  $\perp$  are formulas.
- All atomic propositions  $p \in FP$  are linear temporal logic formulas.
- If  $F$  is a formula, then  $\neg F$  is a formula.
- If  $F_1, \dots, F_n$  are formulas, where  $n \geq 2$ , then  $(F_1 \wedge \dots \wedge F_n)$  and  $(F_1 \vee \dots \vee F_n)$  are formulas.
- If  $F$  and  $G$  are formulas, then  $(F \rightarrow G)$  and  $(F \leftrightarrow G)$  are formulas.
- If  $F$  is a formula, then  $\circ F$ ,  $\diamond F$ , and  $\square F$  are formulas.

- If  $F$  and  $G$  are formulas, then  $F \mathcal{U} G$  and  $F \mathcal{R} G$  are formulas.

The symbols  $\circ, \diamond, \square, \mathcal{U}, \mathcal{R}$  are called temporal operators.

Now we explain their meaning informally. The formulas of LTL are true or false on computation paths, that is sequences of states  $s_0, s_1, \dots$ . The formula  $\square F$  means that  $F$  is true at all states along the path. The formula  $\diamond F$  means that  $F$  is true at some state on the path. The formula  $\circ F$  means that  $F$  is true at the next state after the initial one, that is, at  $s_1$ .

The formulas  $F \mathcal{U} G$  and  $F \mathcal{R} G$  will be formally defined below because they are a bit more complex [14,15,16]. Any two formula  $F$  and  $G$  called equivalent ( $F \equiv G$ ) if for every path  $\sigma$  we have  $\sigma \models F$  if and only if  $\sigma \models G$ .

Examples of linear time temporal logic formula:

- Liveness: Every request is followed by a grant.  $\square(request \rightarrow \circ Grant)$
- Safety:  $p$  never happens.  $\square \neg p$
- Fairness:  $p$  happens infinitely often.  $(\square \circ p) \rightarrow f$
- Another natural example, we may want to express that a professor and a student cannot be borrowers from the library at the same time:  $\square \neg(borrower\_student \wedge borrower\_prof)$
- $\square (S \rightarrow \diamond T)$

The informal meaning of this formula is: Whenever  $S$  holds, in the future  $T$  is bound to hold [18].

**Computational Temporal Logic (CTL):** Computational Temporal Logic (CTL): Computational Temporal Logic, is a branching time logic, which means that its structure model of time is tree like and has many branches (paths), any one of which might be the actual computation path. In this model of time we should specify the path before any computation as shown in Figure 2.4 [25,26,27].

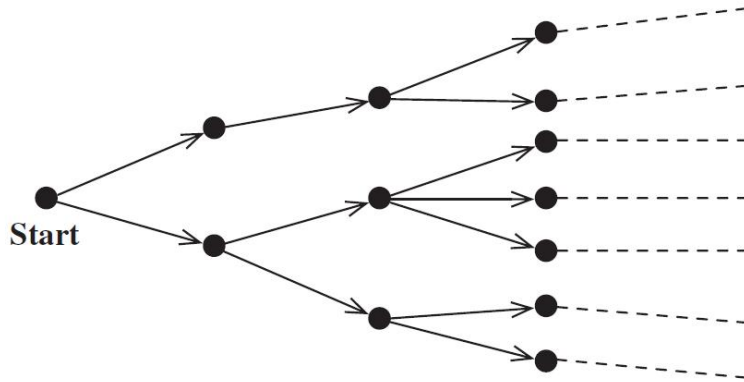


Figure 2.4: CTL path [27]

-*Formula of CTL:* The formula in CTL is defined in pairs inductively as follows:

Firstly Path part:

- $\mathcal{A}$ : means  $\forall$  paths(inevitably)
- $\mathcal{E}$ : means  $\exists$  some paths(possibly)

The formula in CTL has the tree like (branches), if the branch is computed then inside the branch it has the same syntax of LTL formula, and it is defined inductively as follows

- $\top$  and  $\perp$  are formulas.
- All atomic propositions  $p \in FP$  are linear temporal logic formulas.
- If  $F$  is a formula, then  $\neg F$  is a formula.
- If  $F_1, \dots, F_n$  are formulas, where  $n \geq 2$ , then  $(F_1 \wedge \dots \wedge F_n)$  and  $(F_1 \vee \dots \vee F_n)$  are formulas.
- If  $F$  and  $G$  are formulas, then  $(F \rightarrow G)$  and  $(F \leftrightarrow G)$  are formulas.
- If  $F$  is a formula, then  $\circ F$ ,  $\diamond F$ , and  $\square F$  are formulas.

- If  $F$  and  $G$  are formulas, then  $F \mathcal{U} G$  and  $F \mathcal{R} G$  are formulas.

The symbols  $\circ, \diamond, \square, \mathcal{U}, \mathcal{R}$  are called temporal operators. Now we explain their meaning informally. The formulas of CTL are true or false on computation paths, that is sequences of states  $s_0, s_1, \dots$ . The formula  $\square F$  means that  $F$  is true at all states along the path. The formula  $\diamond F$  means that  $F$  is true at some state on the path. The formula  $\circ F$  means that  $F$  is true at the next state after the initial one, that is, at  $s_1$ . The formulas  $F \mathcal{U} G$  and  $F \mathcal{R} G$  explained in LTL section [28,29,30].

Some examples of branching time temporal logic formula: Safety: bad thing never happens:  $\mathcal{A}\square(\neg bad\_thing)$  Fairness:  $p$  happens infinitely often.  $\mathcal{E}(\square \circ p) \rightarrow f$   $\mathcal{E}\diamond(P \wedge \neg q)$  Which means: There exists a state where  $p$  holds but  $q$  does not hold.  $\mathcal{A}\square(p \rightarrow \mathcal{A}\diamond q)$  Which means: Whenever  $p$  holds, eventually  $q$  holds.  $\mathcal{A}\square(\mathcal{E}\diamond q)$  Which informally means: That at all the paths  $q$  holds after some time.

### 2.3.3.5 Time points versus Intervals

The choice between time instants and time intervals has been a centre of focus in philosophy when using temporal logic. Temporal logics normally represent time either as point based or intervals. Until the last decade, logic scholars were greatly interested in point based temporal logics.

Prior and Pnueli considered time as a discrete sequence of points in their model of temporal logic and used it in system specification and verification [14]. Modelling the refinement of a system specification is a widely recognized problem when using a point-based temporal logic [17]. However, the interval based approach is more efficient than the point based approach since it can provide efficient representation of temporal facts. For example, the interval notion is necessary to show continuous processes and to make temporal statements in AI applications; because of this, temporal statements are based on intervals [14].

In a point based temporal logic model, the formula evaluated as true or false of points in time is as shown in figure 2.5.

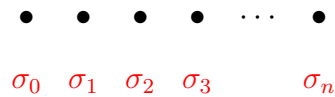


Figure 2.5: Points based

However, in interval based temporal logic the formula is evaluated over intervals of time as shown in Figure 2.6.

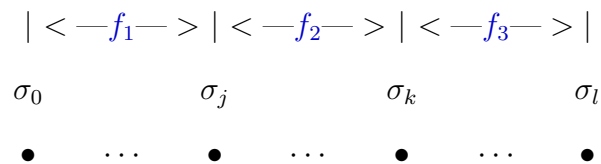


Figure 2.6: Interval based

The claim is that use of intervals greatly simplifies the formulation of certain correctness properties [24]. There are many scientists who proposed use of the interval in many areas; however when it comes to philosophical logic, Simons and Galton suggested the need for intervals with regard to conceptual structures in natural language [29,30,31].

Formal tools for reasoning in artificial intelligence have sprung up from Interval based temporal logics. Major contributions in this area were carried out by Allen [38]. Allen proposed thirteen relations between intervals, called Allen's relations. He provided an axiomatization and representation result of interval structures, and interval-based theory of actions and events.

Interval based logics have been used in other areas of computer science. One of the first applications of interval temporal logic (ITL) in computing for design of hardware components was developed by Moszkowski [10] which we will use on our study. ITL “ is a linear temporal logic over (in)finite time ”. It has been widely adopted to address various problems ranging across specification and verification of hardware devices[23,24,25,34] and temporal logic programming[13,27] to multimedia documents specification [36] and interaction between human and computer [37]. Expressiveness and natural notation were the basis of ITL . Accordingly operators including loops, conditional statements and assignments that are considered imperative and high level operators are easily defined in ITL, making it execution seamless [38]. This intriguing features of ITL renders it an improved alternative to tackling problems that stems from conventional point-based temporal logics.

According to Pnueli and Vardi[14,12], programming languages for specification purposes requires the full power of consistent expressions which is the term used to describe a codified method of searching, defined by Stephen Kleene in 1956 [13]. It is a well-established fact that chop and chopstar offers the power of expressive ability to ITL [10].

Additionally, there is a growing industrial interest in ITL; for instance, veracity have adopted ITL concepts in their temporal language [10] and a temporal logic called Sugar has been introduced by IBM containing ITL (like) operators and these works targets are making the logic more usable for industrial design engineers[11].

The nature of interval temporal logic can be viewed from two distinct perspectives, according to philosophy. Intervals can be viewed as points, which are the only primitive objects, or they are primitive objects in the logic. The majority of interval based logics construct intervals out of points, for example [10,11]. The following is

an example of instant (points) time temporal logic formula:

$$(\sigma, 6) \models \diamond(p)$$

. This formula can be defined informally as: there exists a point where p holds. The following is an illustration of interval time temporal logic formula:

$$A ; B$$

The above formula can be defined as: The interval decomposed (chopped) into a prefix interval and suffix interval, such that A holds over the prefix interval and B over the suffix interval, or A holds for that interval if it is infinite. The issue below is linked to the underlying structure of time.

### 2.3.3.6 Duration Calculus

Duration Calculus (DC) is an extension of the Interval Temporal Logic (ITL) based on the work of Halpern, Manna, and Moszkowski [23, 46], which was originally introduced by Zhou et al. [72] in 1991. DC was applied successfully in case studies of software embedded systems, including gas burner [12] and a railway crossing [13]. It has also been employed for the definition real time semantics of other languages.

The difference between ITL (i.e. underlying logic adopted in this thesis) and DC is that whereas DC is based on intervals of real numbers, ITL is based on a discrete-time domain. The reason for DC operating on a continuous-time domain is that many of the applications it can handle are based on the area of hybrid systems where a discrete computer component interacts with a continuous environment based on sensors and actuators. Accordingly, in this thesis, it was established that the ITL extension known as SPITL in this research can serve as a potential alternative to DC especially as it pertains to specificity of hybrid systems offering competitive edge

based on its executable subset known as Tempura and the tool and the assertion tool termed AnaTempura.

### 2.3.3.7 Discrete or Continuous

A more fundamental choice is that between Discrete or Continuous of a flow of time. It implies that it would be composed of a sequence of instances where each non-final point is followed by another immediate point. We can therefore say that a property is correct in the following moment and also correct all time or at some future time. This can be formulated in first order logic as:

Temporal logics mostly used for program reasoning consider time as discrete where the present instant matches to the present state and by the finite model property. Hence the temporal structure which matches with a series of states of a program execution is the non negative integers as it is shown in Figure 2.7.

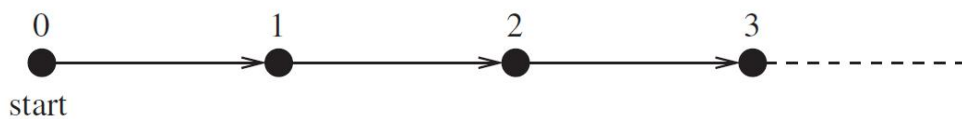


Figure 2.7: Discrete time [22]

Here, each of the black circles represents a classical propositional state, and the arrows represent the accessibility relation, in our case the ‘step’ to the next moment in time. Note that we also have one state identified as the ‘start of time’. Dense refers to a linear ordering in which we can find another different point between any two distinct points. This can be mathematically represented as:

The idea of the flow of time can be modelled using rational or real numbers, which can represent the flow of dense time [33] as is shown in Figure 2.8.





Figure 2.8: Continuous time

Philosophers have been studying tense logics interpreted over a dense time structure. Cau [8] proposed the application of dense time temporal logics to reasoning about concurrent programs. Dense time temporal logics can also be used in real time programs where strict, quantitative performance requirements are placed on programs [9].

## 2.4 Runtime verification

In this section, we classify and evaluate existing runtime verification methods used for safety critical hybrid systems i.e. This section has two goals. The first is to give the reader important background information on existing verification methods for safety critical hybrid systems.

The second is to help the reader to understand the rationale behind the research direction of this thesis by examining previous and contemporary theories of runtime verification. This examination is necessary because we intend to suggest in subsequent chapters that many different runtime verification scheme have a distinct underlying formal model and theory.

### 2.4.1 Contemporary Runtime verification Methods

Runtime verification presents useful approaches that is capable of checking that software is correct by measuring it against certain performance indicators that are

already defined. It can be used to conduct a safety measure in online applications or as a tool for detecting computer bugs. An advantage is that the approaches have a strong formal basis. However, given the problems faced during program execution monitoring, verification tools for runtime events will require further refinements and adjustments for them to be applicable in real-time and embedded systems[52].

Although monitoring activities to mainly through the use hardware probes can be beneficial regarding non-intrusion systems where there are no alternative options[53]. Nowadays, the extent of improving the complexity of hardware platforms whilst leveraging on monitors based on hardware can lead to problems pertaining to the anticipated visibility of the program that is targeted for execution. Conversely, the desire for observation of the targets that are non-intrusive may become difficult to handle by relying entirely on extra layer of software[54].

Nevertheless, a number of past works in the hybrid system verification community depends mainly on the inclusion of extra software for the target application at hand. At the minimum, these methods require a deep knowledge of instrumentation which puts additional layer of complexity on the behaviour of system software within an embedded system. Given the modular nature of some tools for conducting runtime verification, there exist definite potential for the optimisation of extra hardware that can allow for the required processing of observations that requires verification[55,56,57].

Peterson and Savaria [60] in their work, presented the integration of verification protocols based on assertion with an on-chip monitor and submitted that if a scheme that is minimally invasive is used for the investigation of an embedded system, it can be organised by integrating on-chip and hybrid monitoring with the resultant effect of generating the necessary observations required for the verification approaches of other runtime functions. However there are still room for plenty areas of investigation. For instance, the integration of on-chip hardware platform for the

observation of how behavioural pattern of hybrid systems are executed and the use of currently available chip interfaces to allow the observations of events such as monitoring via remote systems as runtime verification protocols are research activities that are currently ongoing.

### **2.4.2 A conceptual view of Runtime verification**

All Runtime verification schemes make use of a facility that monitors the behaviour of a controller that is based on a predefined set of rules that constitute behaviour that are deemed acceptable[64]. For instance if the behaviour at hand is safety related, then certain conditions to ascertain the safety of the system must have been defined using well established algorithm that ensures the safety of the entire system[68].

The use of feedback controllers allows for the easy monitoring of such events by supplying the necessary information that is used as a form of identification parameter to ensure that the defined safety tests and threshold has been passed. With the right type of information executed within the controller, safety limits can be set and used as a basis or threshold for ascertaining the safety of a system[70].

There are many different architectures and technical approaches used for runtime verification methods for real time systems such as hybrid system. Some of them are defined in the subsections that follows[73,74,75].

#### **2.4.2.1 Temporal Logic-based monitoring methods**

This method has gained momentum in the past 5 years with advent of efficient runtime dynamic checking algorithms that can check the safety properties of programs at run time[24]. These dynamic checking algorithms are variants of the powerful model checking algorithms found in most model checking tools. Temporal logic based methods were developed mainly by the computer science and the autonomous

space systems community [30,31,32,33].

There are two basic concepts behind the use temporal logic based methods. First, the expression of a monitorable safety property is expressed in some form of temporal logic like linear time logic (LTL), past-time linear time logic (PTLTL) [36,32,33]. These logics are used to describe a safety verification case, which defines what events or conditions need to be monitored to ensure the safeness of a current execution with respect to a safety specification.

The system safety specification is derived from the system requirements. From the system safety specification, executable versions of the temporal logic expressions are generated. These executable versions of the temporal formulas are loaded onto the safety monitor[38].

The second important aspect is the evaluation of the execution sequence by the safety monitor. The processor must know what relevant information to send the checker. A monitor script expresses what events and conditions are to be automatically extracted from the running program, and then forwarded to the safety monitor for evaluation. The event recognizer (which runs on the processor) is responsible for detecting when events and conditions are to be extracted, such as state changes, or time event changes[39].

The event recognizer forwards an execution trace consisting of extracted events and conditions from the running program to the safety monitor. The monitor then executes a run-time checking algorithm that tests the execution trace against the executable safety specification (i.e.executable versions of temporal logic formulas).

Several excellent examples of temporal logic based runtime verification have been reported in the literature. Most notably is the Temporal Rover system built by Time Rover[34], The temporal Rover allows the user to specify LTL and MTL formulae as comments in a program that is written in C, C++, Java, or VHDL. These formulae express the safety conditions of the executing system. The formulae

are then transformed into executable code at compile time and linked with the application program. During execution of the application program, the generated code from the LTL or MTL formulae verify the behaviour of the program against the formal temporal specification (e.g. LTL and MTL formulae).

Other examples of temporal logic based runtime verification tools are the Java Path finder developed by NASA Ames [31], and the MAC (Monitor and Checking) toolset developed by Lee and Kim at University of Pennsylvania, computer science department [37]. Finally, the Error Confirmation Wrapping System (ECWS) developed by LASS national laboratories of France is noteworthy because the researchers developed an efficient wrapping language and tool-set to expedite the creation and loading of temporal logic formulas for the run-time checker [37,38]. In addition, experimental fault-injection results on the effectiveness of this approach were reported in [38] which showed the approach is robust in detecting both SW and HW induced errors.

### **2.4.3 Runtime verification versus Model Checking**

A model checking is a model with a combination of a specific model and a group of scenarios or computation which ensures that specific models meet their requirements based on a predefined criteria or the level of correctness of the property under consideration. On the other hand, runtime verification has its roots derived from the model checking, but a number of differences still exist. First, in model checking all the scenarios are considered on a true or false basis depending on the correctness property. However, the runtime verification entails a further check to ascertain if the current execution is an element of a group of correctness properties. Second, runtime verification deals with executions that are finite in nature but model checking deals executions with infinite boundaries. Third, model checking deals with a predefined model to check the computations, but runtime verification may consider a finite

executions on an incremental basis.

There exist an additional feature that differentiates runtime verification from model checking based on the fact that runtime verification deals with black box systems, given that it has no foreknowledge of what the scenario is likely to be.

On the other hand, model checking is based on the construction of models that are predefined and tailored for a specific system. Aside from the traditional model checking technique, there exist an advanced model checking that describes a precise model of the underlying system, which is called bounded-model checking techniques [7].

Model checking technique suffer many problems among which is problem pertaining to **state explosion** whereby the system has to deal with each possible state, which is usually huge. As for runtime verification based on single runtime there are no such problems like state explosion.

#### **2.4.4 Runtime verification versus Testing**

One property shared by testing and runtime verification is one that pertains to finite set of executions which both of them possess, although testing is less sophisticated in terms of its underlying technique. In testing, inputs are fed into the system in a sequential manner followed by an observation of its required output whether it meets the desired criteria or not. Oracle [80] test is another form of testing that shares similarities with runtime verifications whereby a test design is embedded with the system under consideration at during runtime.

Despite this resemblance in mode of operation, differences still exist. For instance, testing does not require high level specification and it uses a sequence of input for testing purposes unlike runtime verification where such tests are rarely used.

### 2.4.5 Runtime verification Applications

- Runtime verification can be adopted to play a complementary role in proving theorem and checking of models given the level of difficulty in understanding other methods apart from runtime verification.
- In instances where only few information about the system under consideration is known, runtime verification offers better services compared to both theorem proving and model checking.
- In situations whereby an application highly depend on a given environment and there is little or no information regarding this environment, runtime verifications has been established to outperform other methods.
- For scenarios or instances where security is a major concern in the system thereby requiring additional layer of checking, runtime verifications offers a competitive edge compared to model checking.

In general, the runtime method excels in the applications that are dynamic and in which predictions of results is hard. These dynamic systems are increasing every day, hence the urgency regarding the desire to improve on the overall mechanism of runtime verification to ascertain required output within time. Runtime verifications readily finds application in self-organizing, self-healing and adaptive systems.

### 2.4.6 Matlab and Simulink

The use of Matlab [40], a tool for mathematical programming, is actually increasing in a large number of fields. Together with its dynamic simulation toolbox Simulink [41], originally developed for control and automation applications, it has become a powerful tool that is suitable for a large number of applications. In the field of

runtime verification and hybrid systems, the number of users of Matlab/Simulink has also been increasing rapidly in the last years. The tool is suitable for many applications in this field as for example the study of energy consumption, control strategies, hydraulic and air flow studies, IAQ, comfort, sizing problems. More and more studies are being published using Matlab/Simulink environment for development of specific tools and for simulations of buildings and technical building services. In this thesis, a synthesis on the use of Matlab/Simulink for the verifying and simulating hybrid systems. This work uses the tool AnaTempura in order to be linked to Matlab/ Simulink to get accurate runtime verification results for hybrid systems.

## 2.5 Summary

In this work we propose the use of Runtime verification concepts for checking temporal properties in hybrid systems. As the name implies, Runtime verification is a phenomenon which entails verification of design at runtime through additional hardware and software monitoring as well as some form of recovery mechanism. However, most Runtime verification approaches suffers from the well-known problems that pertains to state space explosion. In practice, additional techniques including design abstraction and/or compositional reasoning are employed to address problems pertaining to state space explosion [4]. Overall, the current work adopts Tempura within a Matlab/Simulink simulation framework to guarantee much more robust and accurate verification methods for hybrid systems.

It is common to utilize additional techniques such as design abstraction and/or compositional reasoning [4] to cope with state space explosion. On our study we success to resolve the state space explosion by using Tempura which will be discussed in more details later. As well as coming this technique to work with a Matlab/Simulink as a simulation methods for more accurate verification methods for hybrid systems.



The Tool AnaTempura is designed to support the step-by-step methodology of handling verification of hybrid systems. This tool helps engineers in handling verification of hybrid systems in a comprehensive way. AnaTempura helps the user by performing its functions in an intelligent way. AnaTempura automatically monitors hybrid systems execution and analyses the system's run-time behaviours.

AnaTempura successfully linked with Matlab techniques. Therefore, the tool has become more effective and powerful as well as more friendly user interface. Both AnTempura and MATLAB are helpful in the analysis of the behaviours of the system and reveal the evolutionary development process of the system. AnaTempura considers possible error cases comprehensively. It is tolerant to many user errors. The tool checks for the errors, corrects the errors whenever possible, and gives relevant prompt information

This chapter has examined the background details and information drawn from the literature which form the basis of the current work. It discussed Temporal Logic (TL) by presenting an overview of its modus operandi after which it's a description of the behavioural pattern of hybrid systems are presented.

Additionally, the chapter touched on specific system issues by classifying them into two distinct aspects namely : general hybrid systems and then those whose run time are based on verification methods. The implications of these summaries for the research questions and problem statement highlighted for the current work are discussed in the chapter that follows.

# Chapter 3

## Preliminaries

### *Objectives:*

---

- Present Overview of *ITL*.
  - Describe the syntax and semantics of *ITL*.
  - Present the language Tempura and its tool AnaTempura.
-

## 3.1 Introduction

A hybrid system will pass through several steps in order for its behaviour to be verified. As such, the steps that represent the behaviour of such systems need to be classified and expressed. Interval Temporal Logic (ITL) can be used to establish all forms of behavioural properties that is desired within a system. For instance, it can be used to understand the behaviour of computer virus in a model. Therefore, ITL has been chosen to be the formal language that will be used in the present research to focus on the verification of a hybrid model. The existence of Tempura which is the executable subset of ITL makes it a very suitable language to be used in the present research. In addition, Tempura offers framework that is executable for developing and experimenting with suitable ITL specification. Therefore, Tempura will be used in this research to ascertain if a good or bad behaviour occurs in a system using ITL description and system traces as the underlying tool.

This chapter provides a contextual understanding about temporal logics and then provide elaboration of temporal logics with the help of some examples. Also, comparison of temporal logic in terms of the ITL language is also presented. Furthermore, the following are explained in detail i.e. ITL syntax and informal semantics, also its executable form Tempura and its syntax and then further its semi-automatic tool Anatempura. This chapter will also present the reasoning behind the choice of ITL for this research.. In the last a critical review of similar and existing research on the use of formal verification is presented.

## 3.2 Interval Temporal Logic

As noted in Chapter 1, Tempura, an executable subset of ITL (i.e. a programming language derived from ITL [13]) is adopted for the current work. In addition, ITL

is very useful for the description system traces. ITL is an essential temporal logic for applicable for both propositional and reasoning based on first order logic on intervals of time. It is very useful in the formal depiction discrete systems that are linear in nature for many reasons. ITL is distinct from other temporal logics given its capability to tackle sequential as well as parallel composition. Also, ITL offers a very strong and extensible specification structure for reasoning purposes regarding properties involving liveness, safety and projected time. Additionally, Tempura and AnaTempura presents framework that are easily executable with animation techniques that can be used for experimentation and specifications based on ITL [13, 14, 18, 21, 27].

### 3.2.1 Syntax of ITL

The key characteristic of ITL is an *interval*. An interval  $\sigma$  is considered to be a (in)finite sequence of states  $\sigma_0, \sigma_1 \dots$ , where a state  $\sigma_i$  is a mapping from the set of variables  $\text{Var}$  to the set of values  $\text{Val}$ . The length  $|\sigma|$  of an interval  $\sigma_0 \dots \sigma_n$  is equal to  $n$  (one less than the number of states in the interval (this has always been a convention in ITL), i.e., a one state interval has length 0).

The syntax of ITL is defined in Table 3.1 where

$z$  is an integer value,

$a$  is a static integer variable (does not change within an interval),

$A$  is a state integer variable (can change within an interval),

$v$  a static or state integer variable,

$g$  is a integer function symbol,

$q$  is a static Boolean variable (does not change within an interval),

$Q$  is a state Boolean variable (can change within an interval),

$p$  is a predicate symbol.

Expressions $e ::=$	$z$   $a$   $A$   $g(e_1, \dots, e_n)$   $\circ A$   $\text{fin } A$
Formulae $f ::=$	$\text{true}$   $q$   $Q$   $p(e_1, \dots, e_n)$   $\neg f$   $f_1 \wedge f_2$   $\forall v. f$   $\text{skip}$   $f_1 ; f_2$   $f^*$

Table 3.1: Syntax of ITL

### 3.2.1.1 Expressions

The syntax is explained with some examples below:

Expressions are built inductively as follows:

- Constants ( $z$ ):

We denote Constants by letters of the form  $z$  for examples:  $z_0, z_1$  to denote values like 0,4,9 and so on.

- Individual variables:

- By convention, capital letters are used to denote state variables which are variables whose values can change within an interval for example  $A, B, C, \dots$

- Small letters to denote static variables which are variables whose values does not change within an interval for example  $a, b, c, \dots$

- Letters of the form  $v$  are used to denote a variable which can either be a static or a state variable.

- Functions :

- $g(e_0, e_1, e_2, \dots, e_k)$  where  $k \geq 0$  and  $e_0, e_1, e_2, \dots, e_k$  are expressions.

- $+$  and  $mod$  are among common functions used.

-Constants (such as 0,1 etc.) are treated as zero place functions.

- Next:  $\circ e$ , where  $e$  is an expression.

- Fin:  $\text{fin } e$ , where  $e$  is an expression.

Examples include:  $A + B$ ,  $a - b$ ,  $A + a$ ,  $v \text{ mod } C$  and so on.

Some examples of syntactically legal expressions are given below:

$$I + (\circ J + 2)$$

This expression adds the value of I in the current state, the value of J in the next state and the constant 2.

$$I + (\circ J) - (\circ I)$$

This expression adds the value of I in the current state to the value of J in the next state and subtracts the value of I in the next state from the result [?, ?].

### 3.2.1.2 Formulae

Formulas are built inductively as follows:

- Predicates  $p(e_0, e_1, e_2, \dots, e_k)$  where  $k \geq 0$  and  $e_0, e_1, e_2, \dots, e_k$  are expressions. Predicates include  $\leq$  and other basic relations.
- Equality:  $e_1 = e_2$ ; where  $e_1$  and  $e_2$  are expressions.
- Logical connectives:  $\neg f$  and  $f_1 \wedge f_2$ , where  $f, f_1$  and  $f_2$  are formulas.
- Universal Quantifier :  $\forall v.f$  where  $f$  is formulae.
- Skip: **skip** is true on an interval  $\sigma$  iff  $\sigma$  has length 1 (unit interval).
- Chop:  $f_1 ; f_2$ , where  $f_1$  and  $f_2$  are a formulas.
- Chopstar:  $f^*$ , where  $f$  is a formulae.

Some examples of syntactically legal formulas are given below:

$$\neg(J = 2) \wedge (K = 4)$$

This formulae states that the value of J is 2 in the current state and the value of K is 4 in the current state.

$$\neg(I = 2) \wedge (\circ J = I + 2)$$

This formulae states that the formulae is true if  $I$  equal to 2 in the current and the value of  $J$  in the next state would be  $I+2$ .

Note that the operator  $\circ$  can be used both for expressions (e.g.,  $\circ I$ ) and for formulas, e.g.,  $\circ(I = 5)$  [12, 13].

### 3.2.2 Semantics

The informal semantics of the most interesting constructs are as follows:

- $\circ A$ : if interval is non-empty then the value of  $A$  in the next state of that interval else an arbitrary value.
- $\text{fin } A$ : if interval is finite then the value of  $A$  in the last state of that interval else an arbitrary value.
- $\neg f$ :  $f$  does not holds for that interval.
- $f_1 \wedge f_2$ :  $f_1$  holds for that interval and  $f_2$  holds for that interval.
- $\text{skip}$  unit interval (length 1).
- $f_1 ; f_2$  holds if the interval can be decomposed (“chopped”) into a prefix and suffix interval, such that  $f_1$  holds over the prefix and  $f_2$  over the suffix, or if the interval is infinite and  $f_1$  holds for that interval.
- $f^*$  holds if the interval is decomposable into a finite number of intervals such that for each of them  $f$  holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which  $f$  holds.

To define the formal semantics, we introduce the following notations:

- $\Sigma$  denotes the set of sequences of states.

- $\Sigma^\omega$  denotes the set of infinite sequences of states.
- $\Sigma^+$  denotes the set of non-empty finite sequences of states.
- $\sigma_{i \rightarrow j}$  for  $0 \leq i \leq j \leq |\sigma|$  denotes a subinterval  $\sigma_i \sigma_{i+1} \cdots \sigma_j$ .

Let  $\mathcal{E}_\sigma[\dots]$  be the “meaning” (semantic) function from  $(\Sigma^+ \cup \Sigma^\omega) \times \text{Expressions}$  to  $Val$  and let  $\mathcal{M}_\sigma[\dots]$  be the “meaning” function from  $(\Sigma^+ \cup \Sigma^\omega) \times \text{Formulae}$  to  $Bool$  (set of Boolean values,  $\{\text{tt}, \text{ff}\}$ ) and let  $\sigma = \sigma_0 \sigma_1 \dots$  be an interval from  $(\Sigma^+ \cup \Sigma^\omega)$ . We write  $\sigma \sim_v \sigma'$  if the intervals  $\sigma$  and  $\sigma'$  are identical with the possible exception of their mappings for the variable  $v$ .

The formal semantics of ITL, except the chop and chopstar operators, is listed in Table 3.2.



$\mathcal{E}_\sigma[z]$	=	$z$
$\mathcal{E}_\sigma[a]$	=	$\sigma_0(a)$ and for all $0 < i \leq  \sigma , \sigma_i(a) = \sigma_0(a)$
$\mathcal{E}_\sigma[A]$	=	$\sigma_0(A)$
$\mathcal{E}_\sigma[g(e_1, \dots, e_n)]$	=	$g(\mathcal{E}_\sigma[e_1], \dots, \mathcal{E}_\sigma[e_n])$
$\mathcal{E}_\sigma[\bigcirc A]$	=	$\begin{cases} \sigma_1(A) & \text{if }  \sigma  > 0 \\ \text{choose-any-from}(Val) & \text{otherwise} \end{cases}$
$\mathcal{E}_\sigma[\text{fin } A]$	=	$\begin{cases} \sigma_{ \sigma }(A) & \text{if } \sigma \text{ is finite} \\ \text{choose-any-from}(Val) & \text{otherwise} \end{cases}$
$\mathcal{M}_\sigma[\text{true}]$	=	$\text{tt}$
$\mathcal{M}_\sigma[q]$	=	$\sigma_0(q)$ and for all $0 < i \leq  \sigma , \sigma_i(q) = \sigma_0(q)$
$\mathcal{M}_\sigma[Q]$	=	$\sigma_0(Q)$
$\mathcal{M}_\sigma[p(e_1, \dots, e_n)] = \text{tt}$	iff	$p(\mathcal{E}_\sigma[e_1], \dots, \mathcal{E}_\sigma[e_n])$
$\mathcal{M}_\sigma[\neg f] = \text{tt}$	iff	not $(\mathcal{M}_\sigma[f] = \text{tt})$
$\mathcal{M}_\sigma[f_1 \wedge f_2] = \text{tt}$	iff	$(\mathcal{M}_\sigma[f_1] = \text{tt})$ and $(\mathcal{M}_\sigma[f_2] = \text{tt})$
$\mathcal{M}_\sigma[\text{skip}] = \text{tt}$	iff	$ \sigma  = 1$
$\mathcal{M}_\sigma[\forall v \cdot f] = \text{tt}$	iff	(for all $\sigma'$ s.t. $\sigma \sim_v \sigma', \mathcal{M}_{\sigma'}[f] = \text{tt}$ )

Table 3.2: Semantics of ITL

The semantics of chop (;) is as follows:

$$\mathcal{M}_\sigma[f_1 ; f_2] = \text{tt} \text{ iff}$$

exists  $k$ , such that  $0 \leq k \leq |\sigma|$ , and if  $\mathcal{M}_{\sigma_0 \rightarrow \sigma_k}[f_1] = \text{tt}$  and  $\mathcal{M}_{\sigma_k \rightarrow \sigma_{|\sigma|}}[f_2] = \text{tt}$

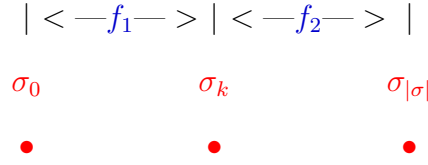


Figure 3.1: Chop of finite interval

or the interval is infinite and  $\mathcal{M}_\sigma[f_1] = \text{tt}$ )

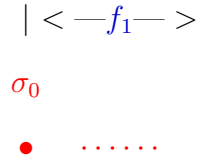


Figure 3.2: Chop of infinite interval

The semantics of chopstar ( $f^*$ ) is as follows:

$\mathcal{M}_\sigma[f^*] = \text{tt}$  iff

if  $\sigma$  is finite then (exists  $l_0, \dots, l_n$ , such that  $l_0 = 0$  and  $l_n = |\sigma|$ )

and for all  $0 \leq i < n$ ,  $l_i \leq l_{i+1}$  and  $\mathcal{M}_{\sigma_{l_i \rightarrow l_{i+1}}}[f] = \text{tt}$ )

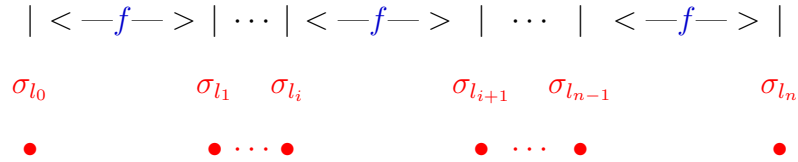


Figure 3.3: Chopstar of finite interval

Else (exists  $l_0, \dots, l_n$ , such that  $l_0 = 0$  and  $\mathcal{M}_{\sigma_{l_n \rightarrow \sigma_{| \sigma|}}} \llbracket f \rrbracket = \text{tt}$ )

and for all  $0 \leq i < n$ ,  $l_i \leq l_{i+1}$  and  $\mathcal{M}_{\sigma_{l_i \rightarrow \sigma_{l_{i+1}}}} \llbracket f \rrbracket = \text{tt}$ )

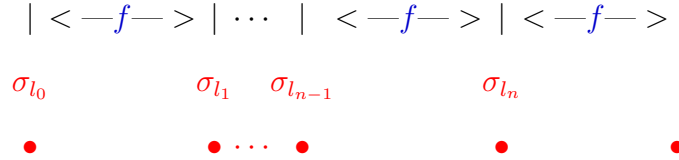


Figure 3.4: Chopstar of finite interval final infinite

or

(exist an infinite number of  $l_i$  such that  $l_0 = 0$  and for all  $0 \leq i$ ,  $l_i \leq l_{i+1}$  and  $\mathcal{M}_{\sigma_{l_i \rightarrow \sigma_{l_{i+1}}}} \llbracket f \rrbracket = \text{tt}$ )

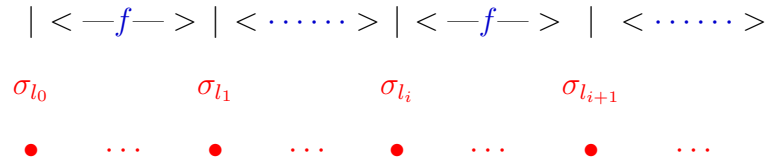


Figure 3.5: Chopstar of infinite interval

### 3.2.3 Derived formulae

Now, we are using the basic operators such as ; and skip and true to derive and define a new formulae, in order to help us in formulating and constructing a logical argument or proof.

The common derived formulae listed in Table 4.4 as follow:

$\text{false}$	$\hat{=}$	$\neg \text{true}$	false value
$\bigcirc f$	$\hat{=}$	$\text{skip} ; f$	next
$\bigcirc^w f$	$\hat{=}$	$\neg \bigcirc \neg f$	weak next
$\text{more}$	$\hat{=}$	$\bigcirc \text{true}$	interval with $\geq 2$ states
$\text{empty}$	$\hat{=}$	$\neg \text{more}$	one state interval
$\text{inf}$	$\hat{=}$	$\text{true} ; \text{false}$	infinite interval
$\text{finite}$	$\hat{=}$	$\neg \text{inf}$	finite interval
$\diamond f$	$\hat{=}$	$\text{finite} ; f$	sometimes in the
$\square f$	$\hat{=}$	$\neg \diamond \neg f$	always in the
$\diamond_{\text{I}} f$	$\hat{=}$	$f ; \text{true}$	some initial subinterval
$\square_{\text{I}} f$	$\hat{=}$	$\neg(\diamond_{\text{I}} \neg f)$	all initial subintervals
$\diamond_{\text{S}} f$	$\hat{=}$	$\text{finite} ; f ; \text{true}$	some subinterval
$\square_{\text{S}} f$	$\hat{=}$	$\neg(\diamond_{\text{S}} \neg f)$	all subintervals

Table 3.3: Derived formulae

### 3.2.3.1 Derived constructs

In this part, the concrete derived constructs are introduced in Table 4.5 as follow:

$\text{if } f_0 \text{ then } f_1 \text{ else } f_2$	$\hat{=}$	$(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
$\text{if } f_0 \text{ then } f_1$	$\hat{=}$	$\text{if } f_0 \text{ then } f_1 \text{ else true}$	if then
$\text{fin } f$	$\hat{=}$	$\square(\text{empty} \supset f)$	final state
$\text{halt } f$	$\hat{=}$	$\square(\text{empty} \equiv f)$	terminate interval when
$\text{keep } f$	$\hat{=}$	$\boxtimes(\text{skip} \supset f)$	all unit subintervals
$\text{while } f_0 \text{ do } f_1$	$\hat{=}$	$(f_0 \wedge f_1)^* \wedge \text{fin } \neg f_0$	while loop
$\text{repeat } f_0 \text{ until } f_1$	$\hat{=}$	$f_0 ; (\text{while } \neg f_1 \text{ do } f_0)$	repeat loop

Table 3.4: Frequently used concrete derived constructs

### 3.2.3.2 Derived constructs related to expressions

In this part, the derived constructs related to expressions are introduced in Table 4.6 as follow:

$A := exp$	$\hat{=}$	$\circ A = exp$	assignment
$A \approx exp$	$\hat{=}$	$\square(A = exp)$	equal in interval
$A \leftarrow exp$	$\hat{=}$	$\text{finite} \wedge (\text{fin } A) = exp$	temporal assignment
$A \text{ gets } exp$	$\hat{=}$	$\text{keep}(A \leftarrow exp)$	gets
$\text{stable } A$	$\hat{=}$	$A \text{ gets } A$	stability
$\text{len}(exp)$	$\hat{=}$	$\exists I \bullet (I = 0) \wedge (I \text{ gets } I + 1) \wedge (I \leftarrow exp)$	interval length

Table 3.5: Frequently used derived constructs related to expressions

### 3.3 An Executable subset of ITL (Tempura)

We are interested in the compositional specification and verification of hybrid systems. The formalism that we require for this purpose has to be dual in the sense that it allows reasoning about behaviours of systems (compositional specification aspect) as well a framework to execute and simulate them (verification aspect). An important motivating factor for choosing ITL as our underlying formal framework is the existence of subset known as Tempura which is executable and supported by an interpreter. Originally proposed by Ben Moszkowski [10, 13], Tempura is a strict, executable subset of ITL.

A Tempura program is deterministic i.e. no arbitrary choice (either of computation length or variable assignment) can be made during execution. For e.g. neither the formula skip nor the formula  $(I = 0 \vee I = )$  is executable, as both are non-deterministic. The syntax of Tempura is restricted to exclude formula such a  $\neg$  and  $\vee V$ . Data types in Tempura are integers, booleans and lists, out of which more complex ones can be built. Tempura operations is listed in table 3.6.

Tempura is derived from functional programming, imperative programming and logic programming methods and provides an avenue for the rapid development and analysis of specifications that are consistent with ITL framework. The adoption of ITL and Tempura provides the twin advantages of the traditional methods based on proofing blended with appropriate speed and ease of computer-based analysis through execution and simulation. We enumerate some benefits of using Tempura to validate compositional specification.

- Modular and reusable tempura test suites can be built.
- Several specification can be compared over a range of test data.
- In contrast to model checking, execution can be used to check theorems that

are not decidable

- Tempura can be expanded upon to contain very important programming constructs, whilst retaining its distinct temporal feel.
- Interval Temporal Logic serves as the single unifying logical and computational formalisation at all stages of analysis.

operator	Usage
Local variables	exists $v : f$
Next	next $f$
sequential composition	$f_1 ; f_2$
Parallel composition	$f_1$ and $f_2$
conditional	if $b$ then $f_1$ else $f_2$
Iteration	$f$ chopstar
Equality	$exp_1 = exp_2$
assignment	$X := e$
Always	always $f$
Sometimes	sometimes $f$
length of an interval	$Len(v)$

Table 3.6: Operations in Tempura

### 3.3.1 The Language: Tempura

Tempura is a subset of ITL that is executable and the syntax of Tempura reflects the relationship. Tempura has state and static variables defined over primitive types such as integers and booleans and over derived entities like lists. Lists in Tempura range over the primitive types and over lists themselves. Lists are analogous to Arrays or Vectors in imperative programming languages. Tempura provides standard operations over expressions such as  $+$ ,  $-$ ,  $*$ ,  $div$ ,  $mod$ ,  $=$ ,  $or$ , and.

Many interesting operators can be further defined over the syntactical constructs [34]. Tempura communicates with external entities using the parametrised input and output functions. Inputs could be read from the keyboard ordinary or from

an external program. Outputs are produced on the terminal, written to a file or streamed to an external program. The following is a simple example to demonstrate the use of lists in Tempura.

```
exists A,I:{A=[2,4,3,8,5,6] and
I =0 and I gets I+1 and halt(I=5) and
stable(struct(A)) and
forall i<5:(swap_list(A,I,I+1) ) and
always output(I) and always output(I+1) and always output(A)}.

/* exchange two elements of a list */
define swap_list(L,i, j) = {
forall k < |L|: {
if k=i then (L[k] gets L[j])
else if k=j then L[k] gets L[i]
else L[k] gets L[k]
}
}.
}
```

Figure 3.6: Tempura example

### 3.3.2 The Tool: AnaTempura

An integral part of the executable framework for ITL is a semi-automatic tool, called AnaTempura[13]. It presents an integrated platform for the verification of runtime of systems based on ITL and its subset Tempura, which is executable. AnaTempura has the following support characteristics:

- Support the specification.
- Support for verification and validation such as runtime testing and simulation combined with formal specification.



AnaTempura is built upon the C-Tempura interpreter, originally developed by Roger Hale [83, 82] and is currently under the custody of Antonio Cau and Ben Moszkowski. The first Tempura interpreter was programmed by Ben Moszkowski [142] in Prolog and became operational at its fullest scale sometime in December, 1983. Later, he reprogrammed the interpreter in Lisp (mid March, 1984). The C-Tempura interpreter was developed at Cambridge University in early 1985 by Roger Hale [83]. A brief description of the run-time analysis process in AnaTempura is depicted in Figure 3.7. AnaTempura operates on the basis of open architecture that avails the integration of new tool components. It monitors and analyses reactive and time critical systems. It has also been used to analyse the effect of change in the evolution of Legacy system [231, 232]. AnaTempura supports the idea of runtime validation of systems. A possible runtime behaviour of a system can be checked against a property for satisfiability. Given that an ITL property is equivalent to an array of sequences of states or intervals, runtime validation is assessing whether the sequence generated by the system is on the set of sequences corresponding to the property we are investigating [30].

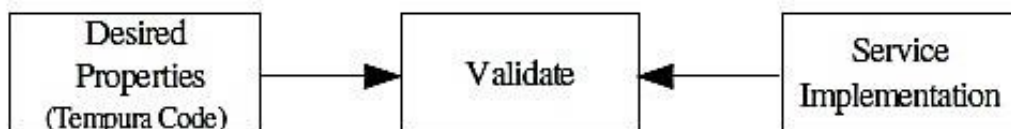


Figure 3.7: The Analysis Process

A formula in Tempura is executable through AnaTempura if:

- It is deterministic.
- The corresponding interval's length is established.

- The values of the variables are identified throughout the corresponding interval.

### 3.3.3 AnaTempura mechanism

The general framework for analysing system in Anatempura can be described as follows:

- **Formulate all properties that are desirable regarding the system of interest in tempura**

The starting point is formulating all desired properties of interest (also referred to as assertions) for the system in Tempura. Establishing properties of systems can be a arduous task. An initial guideline for choosing properties could be formulating safety, liveness, timing and security properties for the system. Properties formulated in Tempura are stored in files and loaded at runtime.

- **Identify places in the code that are suitable and insert assertion points**

The runtime validation procedure adopts assertion points to check whether a system satisfies the desired properties/assertions. The assertion points [30] are injected into the source code of the system and will produce a sequence of data or system states, like values and timestamps of variables and value change respectively, while the system is running. Identifying the location of assertion points can be challenging and depends largely on the kind of assertions that are modelled. Assertions can be pre/post condition properties in which case the assertion points are inserted at the entry and exit points of the program. Assertions can also be invariants in which case assertion points are all possible states defined for the program[10,30].

- **By the use of Tempura, check that the behaviour satisfies the ex-**

**pected properties**

AnaTempura produces an analysis of the system characteristics on a state by state basis as the computational procedures advances. At different states of execution, for which assertion points are stated, values for variables of interest are sent from the system to AnaTempura. The Tempura properties are cross checked with respect to the values received. If the properties are not met AnaTempura will flag the errors by showing the expected output and what the system under current consideration actually supplies. Therefore the method is not merely a keep tracking approach i.e. providing the running results of some properties of the system. It does not just capture the execution results but also compare them with formal properties, AnaTempura performs the validation[13].

Figure 3.8 below shows an overview of the AnaTempura Architecture. Inputs to the system are the source code augmented with assertions points or an ITL specification and the properties of interest. The output is a result stating the satisfiability of that property for the system. For a more visually appealing result, the process of validation can be animated. The tool can analyse programs written in C, Verilog and Java. AnaTempura can be downloaded from [36] and several examples of the tool in action can be found at [142, 34].

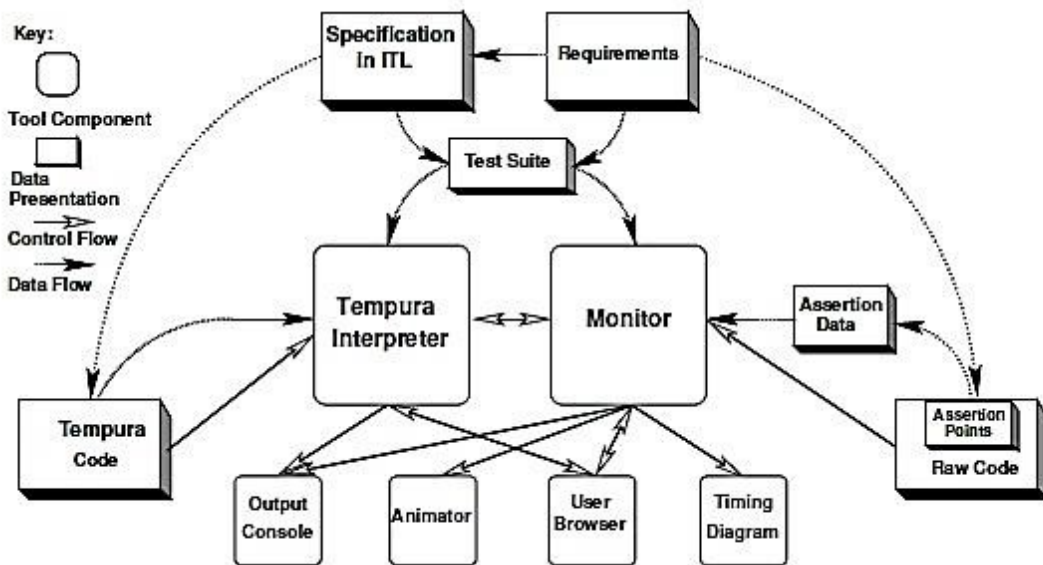


Figure 3.8: General System Architecture of AnaTempura[177]

### 3.4 Summary

Specification of hybrid systems requires representation of properties that are temporal in nature. Interval Temporal Logic (ITL) provides a sound formalism for reasoning about behaviour of systems over periods of time. Its executable subset Tempura and runtime validation engine AnaTempura provide the machinery for prototyping, simulating and debugging temporal behaviour of systems.

# Chapter 4

## Spline Interval Temporal logic(SPITL)

### *Objectives:*

---

- Define a Spline Interval Temporal formalism.
    - modelling in SPITL
    - syntax of SPITL
    - semantics of SPITL
  
  - Show examples that can be specified in SPITL
-

## 4.1 Introduction

This chapter presents a formalism for describing hybrid system in form of splines, based on Interval Temporal Logic (ITL)., called Spline Interval Temporal (SPITL) formalism is introduced with the view to specify hybrid systems and highlight its specifications. This extends ITL with new features in order to mix between continuous and discrete specification of behaviours. The syntax and semantics of SPITL as well as the derived constructs are introduced. This chapter is therefore structured as follows. Section 2 discuss the definition of Spline and its types will be illustrated. In section 4 we first provide the SPITL formalism to reason about hybrid systems. Then we present the syntax and semantics of SPITL with some examples for the discrete and continuous SPITL. Finally we conclude this chapter in section 6.

## 4.2 Spline background

A spline is a piece-wise polynomial function that is smoothly connected at the control points as shown in Figure 4.1. The control points are known as knots. Splines were first employed in numerical analysis for interpolation. Spline interpolation may be preferred to polynomial approximation due to their ability to avoid similar results whilst avoiding oscillation between data points in instances where polynomials of high degrees are used in the approximation. Other useful properties of splines have also been reported one of which include the stability of evaluation and capacity to approximate curves with complex structures. For a comprehensive study of splines, readers are referred to See de Boor [80].

Splines are mathematical model that possess the ability to associate a continuous representation of a curve with a discrete set of points in a given space [84]. Spline fitting is an extremely popular form of piecewise approximation which adopts many

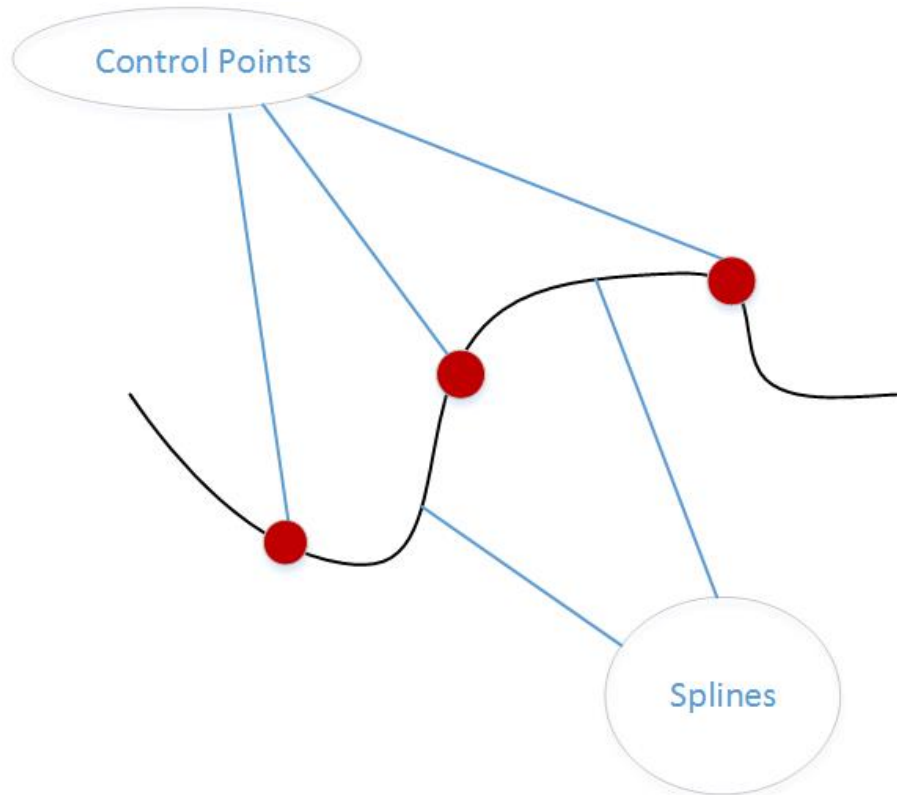


Figure 4.1: control points

forms of polynomials on an interval in which they are fitted to the function at specified points, referred to as control points as highlighted above. The polynomial used can change, but the derivatives of the polynomials are required to meet related interpolator conditions[83].

Boundary conditions are also imposed on the end points of the intervals. Undoubtedly, a critical deciding question is whether the spline is required to approximate or interpolate the control points. In the sections that follows, a brief description of types of spline is provided. Also presented is a discussion on how to adjust the control points of spline curve in order to present the continuous model of our hybrid system.

## 4.2.1 Spline types

### 4.2.1.1 Linear Spline

Given  $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1}), (x_n, y_n)$ , fit linear splines to the data as shown in Figure 4.2. This simply involves forming the consecutive data through straight lines. So if the above data is given in an ascending order, the linear splines are given by  $y_i = f(x_i)$ .

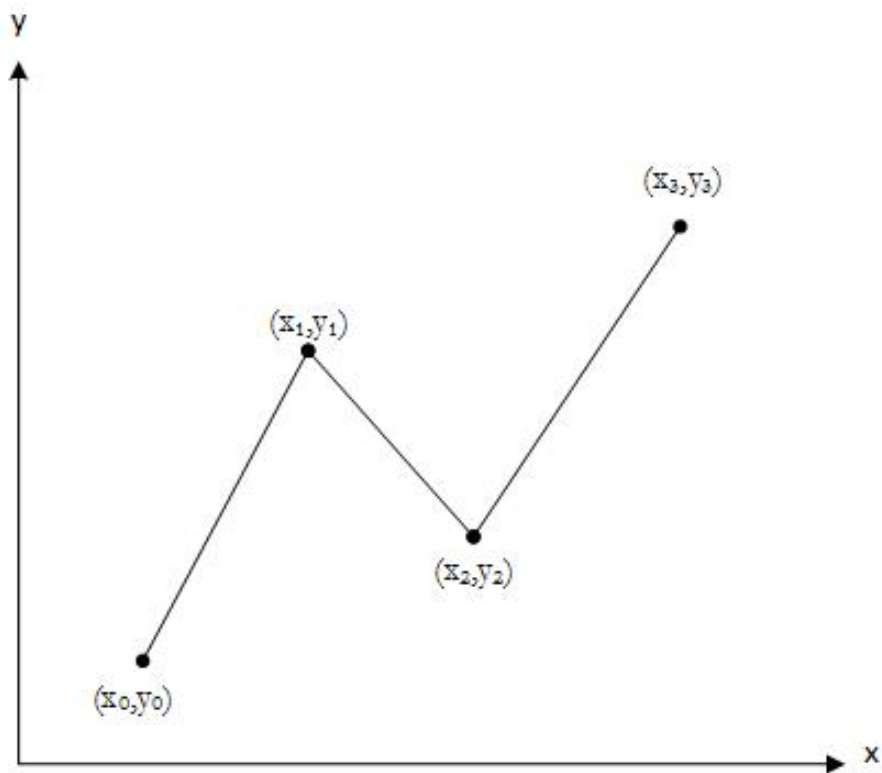


Figure 4.2: Linear spline

$$f(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0), \text{ where } x_0 \leq x \leq x_1$$

$$= f(x_1) + \frac{f(x_2) - f(x_1)}{x_2 - x_1}(x - x_1), \text{ where } x_1 \leq x \leq x_2$$

·  
·  
·



$$= f(x_{n-1}) + \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}(x - x_{n-1}), \text{ where } x_{n-1} \leq x \leq x_n$$

be aware that:

$$\frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \text{ is a slope between } x_i \text{ and } x_{i-1}.$$

### 4.2.1.2 Quadratic Spline

Quadratic, uniquely defined by three points.

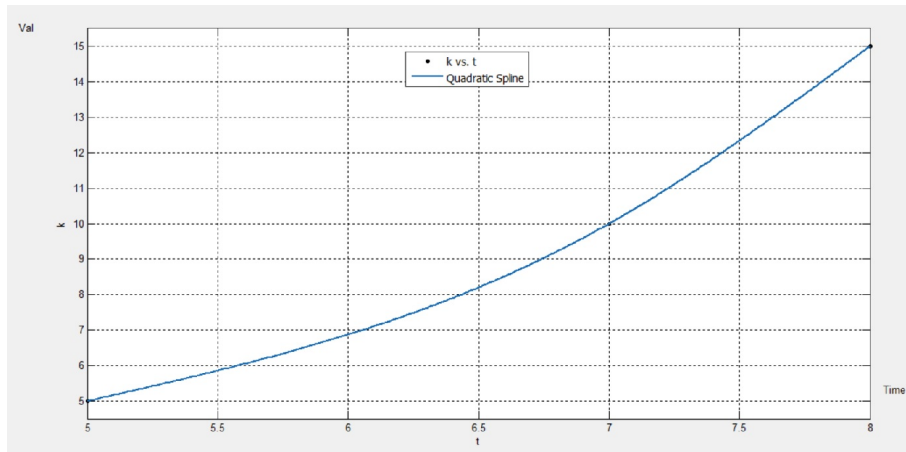


Figure 4.3: Quadratic spline

A quadratic spline has a quadratic function for each interval  $f_i(t) = a_it^2 + b_it + c_i$  where  $t \in [t_i, t_{i+1}]$  and  $i = 1, 2, \dots, n - 1$

so, there are  $3n$  unknowns  $a, b, c$ , we need to setup  $3n$  equations and then solve them.

Therefore, to find the  $3n$  equation, we will do as follows.

- each quadratic spline goes through two consecutive control points

$$f(t_0) = a_1t_0^2 + b_1t_0 + c_1$$

$$f(t_1) = a_1t_1^2 + b_1t_1 + c_1$$

.

.

.

$$f(t_{i-1}) = a_i t_{i-1}^2 + b_i t_{i-1} + c_i$$

$$f(t_i) = a_i t_i^2 + b_i t_i + c_i$$

·  
·  
·

$$f(t_{n-1}) = a_n t_{n-1}^2 + b_n t_{n-1} + c_n$$

$$f(t_n) = a_n t_n^2 + b_n t_n + c_n$$

So, As the quadratic spline goes through two consecutive control points, this gives us 2n equations.

- due to the fact that the first derivatives of two quadratic splines are continuous at the interior control point. For instance, the derivative of first spline  $a_1 t_0^2 + b_1 t_0 + c_1$  is  $2a_1 t + b_1$

and the derivative of the second spline is  $a_2 t_0^2 + b_2 t_0 + c_2$  is  $2a_2 t + b_2$

Therefore,

the tow are equal at  $t = t_1$  and that will give us :

$$2a_1 t_1 + b_1 = 2a_2 t_1 + b_2 \Rightarrow 2a_1 t_1 + b_1 - 2a_2 t_1 - b_2 = 0$$

And so on ..

- We will assume that the first spline is linear, i.e,  $a_1 = 0$ .

This gives us 3n equations and 3n unknowns. Thus, we can solve quadratic splines using these equations. See figure 4.3.

#### 4.2.1.3 cubic Spline

In these splines, a cubic spline is defened by four points. A cubic spline has a cubic function for each interval.  $f_i(t) = a_i t^3 + b_i t^2 + c_i t + d_1$  where  $t \in [t_i, t_{i+1}]$  and  $i = 1, 2, \dots, n - 1$

so, there are 4n unknowns a,b,c,d we need to setup 4n equations and then solve

them. Therefore, to find the 4n equation, we will do as follows.

- each cubic spline goes through two consecutive control points

$$f(t_0) = a_1t^3_0 + b_1tt^2_0 + c_1t_0 + d_1 \quad (1)$$

$$f(t_1) = a_1t^3_1 + b_1tt^2_1 + c_1t_1 + d_1 \quad (2)$$

.

.

.

$$f(t_{i-1}) = a_it^3_{i-1} + b_itt^2_{i-1} + c_it_{i-1} + d_i \quad (3)$$

$$f(t_i) = a_it^3_i + b_itt^2_i + c_it_i + d_i \quad (4)$$

.

.

.

$$f(t_{n-1}) = a_nt^3_{n-1} + b_nnt^2_{n-1} + c_nt_{n-1} + d_n \quad (5)$$

$$f(t_n) = a_nt^3_n + b_nnt^2_n + c_nt_n + d_n \quad (6)$$

So, As the cubic spline goes through two consecutive control points, this gives us 2n equations.

- due to the fact that the first derivatives of two cubic splines are continuous at the interior control point(i.e, are equal to each other). For instance, the first

$$\text{derivative of first spline } a_1t^3_0 + b_1tt^2_0 + c_1t_0 + d_1 \text{ is } 3a_1t^2 + 2b_1t + c_1 \quad (7)$$

$$\text{and the first derivative of the second spline is } a_1t^3_1 + b_1tt^2_1 + c_1t_1 + d_1 \text{ is } 3a_2t^2 + 2b_2t + c_2 \quad (8)$$

Therefore,

the tow are equal at  $t = t_1$  and that will give us :

$$3a_1t_1^2 + 2b_1t_1 + c_1 = 3a_2t_1^2 + 2b_2t_1 + c_2 \Rightarrow 3a_1t_1^2 + 2b_1t_1 + c_1 + 3a_2t_1^2 + 2b_2t_1 - c_2 = 0$$

(9).

And so on ..

- Also, in cubic splines, the second derivatives of two cubic splines are continuous at the interior control point(i.e, are equal to each other). For instance, the second derivative of first spline  $a_1t^3_0 + b_1tt^2_0 + c_1t_0 + d_1is6a_1t + 2b_1$  (10) and the second derivative of the second spline is  $a_1t^3_1 + b_1tt^2_1 + c_1t_1 + d_1is6a_2t + 2b$  (11)

Therefore,

the tow are equal at  $t = t_1$  and that will give us :

$$6a_1t + 2b_1 + 6a_2t + 2b \Rightarrow 6a_1t + 2b_1 - 6a_2t - 2b = 0$$
 (12).

And so on ..

- We will assume that the first spline is linear, i.e,  $a_1=0$ . Thus, we can solve cubic splines using these equations. Figure 4.4 illustrates the cubic spline .

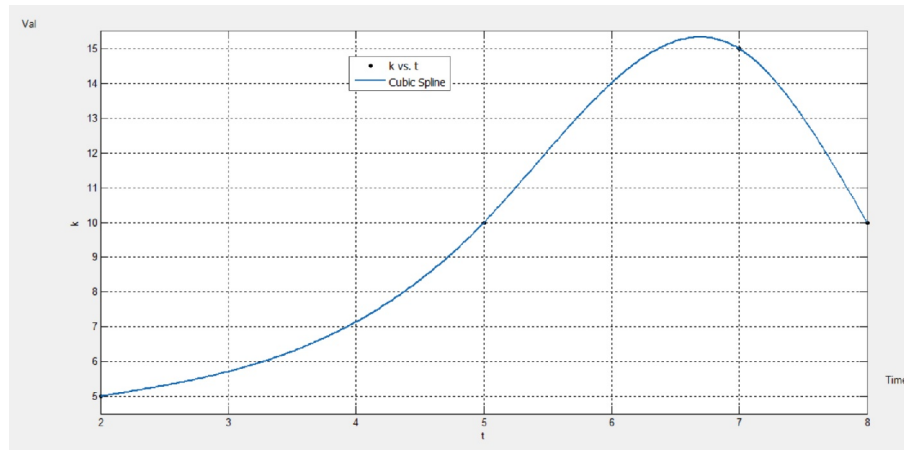


Figure 4.4: Cubic spline

Accordingly, it is possible to model a sequence of phases which represents successive intervals of time given that adjoining phases do not share a time point. Against this

backdrop, the current work only considers spline interpolation using linear splines (splines of degree 1), quadratic splines (splines of degree 2), and cubic splines (splines of degree 3) as discussed above. Generalization to splines of general order is relatively straightforward.

### 4.3 Spline Interval Temporal logic(SPITL)

Spline Interval Temporal logic(SPITL) is a flexible notation that extends ITL to model continuous changes over time in form of splines. SPITL, is presented in order to mix between continuous and discrete specification of hybrid systems behaviours. SPITL has temporal operators to model the behavioural aspects, i.e. it has operators that can put phases in sequence to describe behaviour of hybrid systems. Furthermore, the main differences between ITL and SPITL is that in ITL, interval is considered to be a sequence of states, and in SPITL the interval is considered to be a sequence of phases. A phase replaces a sequence of discrete states with a continuous behaviour represented by a spline. Note that the spline should satisfy the condition that they are continuous from the right in every time point and phases length is greater than zero,i.e. to ensure that there will be a finite number of changes within a finite time interval. Phases in SPITL have duration within a phase. Variables in SPITL change in a continuous fashion where in each phase variables change according to a spline[6]. SPITL can also model discrete change in value.i.e. discrete value remain stable and there are no gaps between phases whiles time is still continuous.

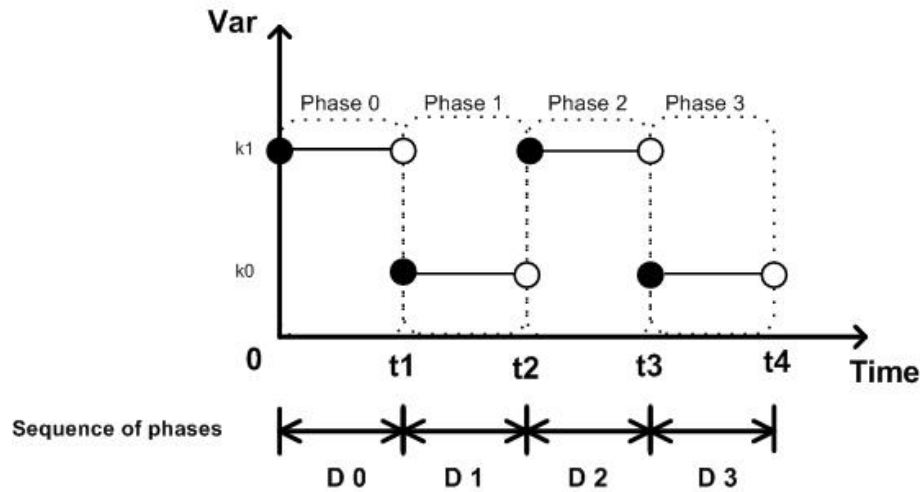


Figure 4.5: discrete changes

### 4.3.1 Discrete changes in SPITL

As mentioned on the previous section that SPITL can model discrete changes as well as the continuous changes. Furthermore, for the discrete behaviour, we have that 'states' within a phase which is remain constant, i.e. non-continuous change in value, see figure 4.5 for discrete changes. The discrete changes model the discrete events of the hybrid system operation as sequence discrete states in time interval. Each event occurs at a particular instant in time interval and impact on the hybrid system behaviour. Between each state, no changes in the system states are assumed to occur; thus the system can directly chop in time from one state to the another.

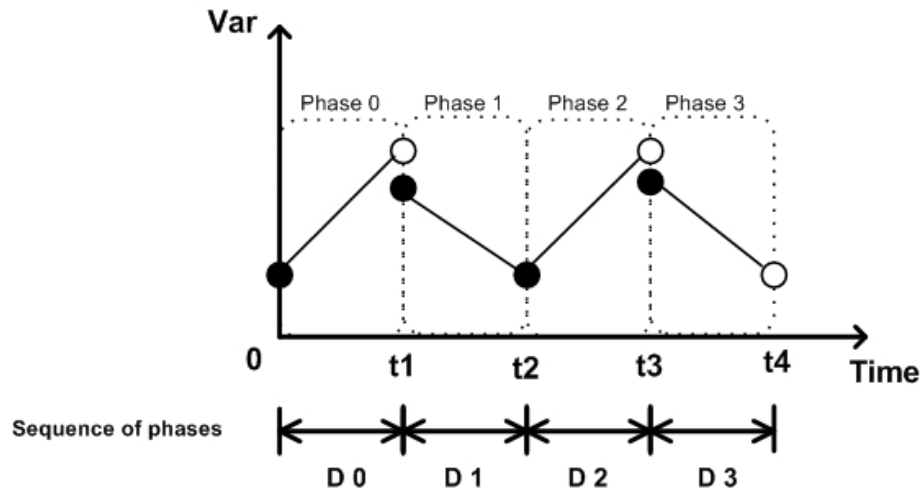


Figure 4.6: continuous changes

### 4.3.2 Continuous changes in SPITL

The reason why this approach is extending ITL using the form of spline is because we need to describe continuous changes as well as the discrete changes (which ITL already do) in order to specify and model hybrid systems. Hence, this approach will concentrate on continuous time type behaviours. Furthermore, the interval in SPITL is a sequence of phases as mentioned before. Phases in SPITL have duration within a phase as illustrated in figure 4.6. Therefore, phases enable us to give semantics to variable with respect to time. For example  $[0 : 10, 1 : 20 >$ , a phase such that values change in a continuous fashion from 10 to 20 according to spline. Next sections will discuss the syntax and semantics of SPITL.

Table 4.1: Syntax of SPITL

<i>Expressions</i>	$e ::= k \mid A \mid g(e_1, \dots, e_n) \mid \circ A \mid [r_0 : k_0, \dots, r_n : k_n >$
<i>Formulae</i>	$f ::= \ell : \text{empty} \mid \text{empty} \mid p(e_1, \dots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid f_1 ; f_2 \mid f^*$

### 4.3.3 Syntax of SPITL

The syntax of *SPITL* is defined in Table 4.1 where:

In the syntax of expressions:

- $k$  denotes a constant value.
- $A$  denotes the value of variable  $A$  in the current phase
- $g(e_0, \dots, e_m)$  denotes a function on expression where  $g$  is a function symbol.
- $\circ A$  denotes the value of variable  $A$  in the next phase, if there is no next phase then it is an arbitrary value from  $\text{Val}$ .

In the syntax of formulae:

- $\ell : \text{empty}$  denotes a single phase with duration  $\ell$ .
- $\text{empty}$  denotes exactly one phase with finite duration.
- $p(e_1, \dots, e_n)$  denotes a predicate on expressions  $p$  where  $p$  is a predicate symbol.
- $\neg f$  denotes a boolean negation of a formulae.
- $f_1 \wedge f_2$  denotes the boolean conjunction of two formulae.



- $f_1 ; f_2$  holds if the interval can be decomposed (“chopped”) into a prefix and suffix interval, such that  $f_1$  holds over the prefix and  $f_2$  over the suffix, I.e, the last phase of the interval over which  $f_1$  holds is the same as the first phase of the interval over which  $f_2$  holds.
- $f^*$  holds if the interval is decomposable into a finite number of intervals such that for each of them  $f$  holds.

to introduce the formal semantics, we must before introduce the following subsection:

### 4.3.4 Phase definition In (SPITL)

Let duration  $\ell$  be an element of  $\mathfrak{R}^{>0}$ . A  $\ell$ -phase  $\delta^\ell$  is a continuous mapping from the set of variables  $Var$  and  $[0, \ell)$  to the set of values  $Val$ .

$$\boxed{\delta^\ell : Var \times [0, \ell) \mapsto Val}$$

Let  $\delta^\ell, \delta_0^\ell, \delta_1^\ell, \delta_2^\ell, \dots$  denote  $\ell$ -phases.

$\Delta_\ell$  denote the set of all possible  $\ell$ -phases.

The set of all phases is denote by  $\Delta$  and is defined as  $\Delta \hat{=} \cup_\ell \Delta_\ell$ .

Let  $\delta, \delta_0, \delta_1, \delta_2, \dots$  denote phases.

Since a phase is a state with time component, Next section will defines Timed expressions that used to model changes over time within a phase.

### 4.3.5 Timed expressions definition In (SPITL)

The syntax of Timed expressions is as follows:

$$e ::= K|A|g(e_0, \dots, e_m)|\circ A|[r_0 : k_0, \dots, r_n : k_n >$$

where:

- As mentioned before on the syntax of SPITL,  $K, A, g(e_0, \dots, e_m), \circ A$  are respectfully constants, variables, functions, next variables.

- $[r_0 : k_0, \dots, r_n : k_n >$  denotes a change in the form of a spline defined by  $n + 1$  control points  $(r_i * \ell, k_i)$  ( $0 \leq i \leq n, n \geq 0$ ) where  $\ell \in \mathfrak{R} > 0$  denotes the length of a phase,  
 $(0 \leq r_0 < r_1 \dots < r_n \leq 1)$  and  $k_i$  ( $0 \leq i \leq n, n \geq 0$ ) are constants.

### 4.3.6 Semantics of SPITL Expressions

Let  $\llbracket \dots \rrbracket$  be the “meaning” (semantic) function from Timed Expressions  $\times \Delta^* to$  ( $[0, \ell_0) \xrightarrow{c} Val$ ) and let  $\sigma$  be a sequences of phases  $\delta_0, \delta_1, \dots, \delta_n$  and let  $\ell_0$  be the duration of the first phase  $\delta_0$  and  $\ell_1$  be the duration of the second phase  $\delta_1$  then denotational semantics is as follows:

$\llbracket k \rrbracket_\sigma$	$\hat{=}$	$(\lambda t \in [0, \ell_0).k)$
$\llbracket A \rrbracket_\sigma$	$\hat{=}$	$(\lambda t \in [0, \ell_0).\delta_0(A)(t))$
$\llbracket g(e_0, \dots, e_m) \rrbracket_\sigma$	$\hat{=}$	$(\lambda t \in [0, \ell_0).g(\llbracket e_0 \rrbracket_\sigma(t), \dots, \llbracket e_m \rrbracket_\sigma(t))$
$\llbracket \bigcirc A \rrbracket_\sigma$	$\hat{=}$	$(\lambda t \in [0, \ell_0).$
		$\begin{cases} \delta_1(A)(\ell_1 * t / \ell_0) & \text{if }  \sigma  > 0 \\ \text{choose}(Val) & \text{if }  \sigma  = 0 \end{cases}$
	)	
$\llbracket [r_0 : k_0, \dots, r_n : k_n > \rrbracket_\sigma$	$\hat{=}$	$spline < (r_0 * \ell_0, k_0), \dots, (r_n * \ell_0, k_n) >$

Table 4.2: Semantics of *SPITL*

Notice that, the time interval of the first and second phases need to be synchronised in the semantic definition of  $\bigcirc A$ , i.e.,  $\bigcirc A$  takes values from the second phase with  $t_1$  such that  $0 \leq t_1 < \ell_1$  but the actual time is from the first phase with time  $0 \leq t_0 < \ell_0$

This function  $sync(t) \hat{=} \ell_1 * t / \ell_0$  ensures that if  $0 \leq t < \ell_0$  then  $0 \leq sync(t) < \ell_1$

### 4.3.7 Semantics of SPITL formulae

Let  $\llbracket \dots \rrbracket$  be the “meaning” (semantic) function from SPITL formulae to  $p(\Delta^*)$ . Let  $te_i (1 \leq i \leq n)$  be timed expressions, then denotational Semantics of SPITL formulae is as follows:

$\llbracket \text{empty} \rrbracket$	≐	$\Delta$
$\llbracket \ell : \text{empty} \rrbracket$	≐	$\Delta_\ell$
$\llbracket p(e_1, \dots, e_n) \rrbracket$	≐	$\{\sigma \in \Delta^* \mid \hat{p}(\llbracket e_0 \rrbracket_\sigma, \dots, \llbracket e_n \rrbracket_\sigma)\}$
$\llbracket \neg f \rrbracket = \text{tt}$	iff	not $\llbracket f \rrbracket = \text{tt}$
$\llbracket f_1 \wedge f_2 \rrbracket = \text{tt}$	iff	$\llbracket f_1 \rrbracket = \text{tt}$ and $\llbracket f_2 \rrbracket = \text{tt}$
$\llbracket f_1 ; f_2 \rrbracket = \text{tt}$	iff	exists $k$ , such that $0 \leq k \leq  \sigma $ , and if $\llbracket f_1 \rrbracket_0 \rightarrow \sigma_k = \text{tt}$ and $\llbracket f_2 \rrbracket_k \rightarrow \sigma_{ \sigma } = \text{tt}$
$\llbracket f^* \rrbracket = \text{tt}$	iff	if $\sigma$ is finite then exists $l_0, \dots, l_n$ , such that $l_0 = 0$ and $l_n =  \sigma $ and for all $0 \leq i < n$ , $l_i \leq l_{i+1}$ and $\llbracket f \rrbracket_{l_i} \rightarrow \sigma_{l_{i+1}} = \text{tt}$ Else exists $l_0, \dots, l_n$ , such that $l_0 = 0$ and $\llbracket f \rrbracket_{l_n \rightarrow \sigma_{ \sigma }} = \text{tt}$ and for all $0 \leq i < n$ , $l_i \leq l_{i+1}$ and $\llbracket f \rrbracket_{l_i} \rightarrow \sigma_{l_{i+1}} = \text{tt}$

Table 4.3: Semantics of *SPITL* formulae

### 4.3.8 Derived formulae

in this section, the derived formulae for *SPITL* will be introduced and listed in Table 4.4 In order to help us in formulating and constructing a logical argument or proof.

The common derived formulae listed as follow:

$\text{false}$	$\hat{=}$	$\neg \text{true}$	false value
$\bigcirc f$	$\hat{=}$	$\text{skip} ; f$	next
$\textcircled{w} f$	$\hat{=}$	$\neg \bigcirc \neg f$	weak next
$\text{more}$	$\hat{=}$	$\bigcirc \text{true}$	interval with $\geq 2$ phases
$\text{empty}$	$\hat{=}$	$\neg \text{more}$	one phase interval
$\diamond f$	$\hat{=}$	$\text{true} ; f$	sometimes
$\square f$	$\hat{=}$	$\neg \diamond \neg f$	always
$\diamond_{\text{I}} f$	$\hat{=}$	$f ; \text{true}$	some initial subinterval
$\square_{\text{I}} f$	$\hat{=}$	$\neg(\diamond_{\text{I}} \neg f)$	all initial subintervals
$\diamond_{\text{S}} f$	$\hat{=}$	$\text{true} ; f ; \text{true}$	some subinterval
$\square_{\text{S}} f$	$\hat{=}$	$\neg(\diamond_{\text{S}} \neg f)$	all subintervals

Table 4.4: Derived formulae

### 4.3.8.1 Derived constructs

In this part, the concrete derived constructs are introduced in Table 4.5 as follow:

$\text{if } f_0 \text{ then } f_1 \text{ else } f_2$	$\hat{=}$	$(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
$\text{if } f_0 \text{ then } f_1$	$\hat{=}$	$\text{if } f_0 \text{ then } f_1 \text{ else true}$	if then
$\text{fin } f$	$\hat{=}$	$\square(\text{empty} \supset f)$	final phase
$\text{halt } f$	$\hat{=}$	$\square(\text{empty} \equiv f)$	terminate interval when
$\text{keep } f$	$\hat{=}$	$\boxtimes(\text{skip} \supset f)$	all unit subintervals
$\text{while } f_0 \text{ do } f_1$	$\hat{=}$	$(f_0 \wedge f_1)^* \wedge \text{fin } \neg f_0$	while loop
$\text{repeat } f_0 \text{ until } f_1$	$\hat{=}$	$f_0 ; (\text{while } \neg f_1 \text{ do } f_0)$	repeat loop

Table 4.5: Frequently concrete derived constructs

### 4.3.8.2 Expressions derived constructs

In this part, the derived constructs related to expressions are introduced in Table 4.6 as follow:

$A := \text{exp}$	$\hat{=}$	$\circ A = \text{exp}$	assignment
$A \approx \text{exp}$	$\hat{=}$	$\square(A = \text{exp})$	equal in interval
$A \leftarrow \text{exp}$	$\hat{=}$	$(\text{fin } A) = \text{exp}$	temporal assignment
$A \text{ gets } \text{exp}$	$\hat{=}$	$\text{keep}(A \leftarrow \text{exp})$	gets
$\text{stable } A$	$\hat{=}$	$A \text{ gets } A$	stability
$\text{len}(\text{exp})$	$\hat{=}$	$\exists I \bullet (I = 0) \wedge (I \text{ gets } I + 1) \wedge (I \leftarrow \text{exp})$	interval length

Table 4.6: Frequently derived constructs related to expressions

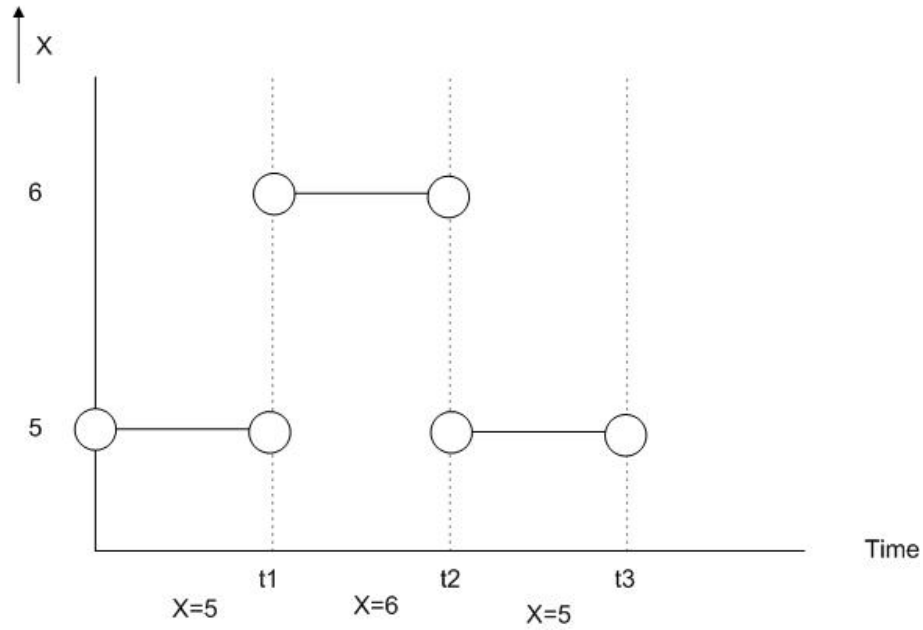


Figure 4.7: Discrete changes Example

### 4.3.9 Discrete and Continuous changes Examples

In order to explain the Discrete and Continuous changes, we will use these simple examples to do that:

#### 4.3.9.1 Discrete Examples

the value of the variable  $X$  has the value 5 on the first and last phases and 6 on the second phase as illustrated in 4.7.:  $(t_1 : \text{empty} \wedge X = [0 : 5, 1 : 5 >); \text{skip};$

$(t_2 : \text{empty} \wedge X = [0 : 6, 1 : 6 >); \text{skip};$

$(t_3 : \text{empty} \wedge X = [0 : 5, 1 : 5 >).$

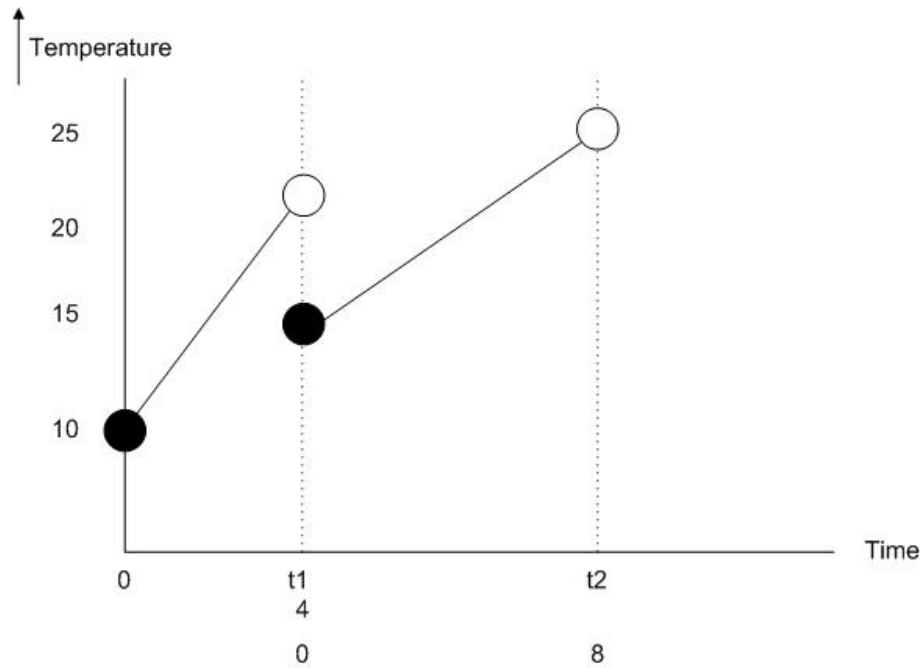


Figure 4.8: Continuous changes Example

#### 4.3.9.2 Continuous Examples

$(4 : \text{empty} \wedge \text{Temperature} = [0 : 10, 1 : 20 >) ; \text{skip} ; (4 : \text{empty} \wedge \text{Temperature} = [0 : 15, 1 : 25 >)$

specifies two phases where the first phase has duration 4 and in which the value of the Temperature changes linearly from 10 to 20 and the second phase has duration 4 and in which the value of the Temperature changes linearly from 15 to 25 as illustrated in 4.8.

## 4.4 Spline example

In this section we will illustrate the use of SPITL to specify and reason about hybrid system properties. This illustration is based on a well known Gas burner example from [90]. We will consider a simple version of the Gas burner example, with only a gas valve and a flame sensor. It will be expressed by a timing constraints and three different phases:

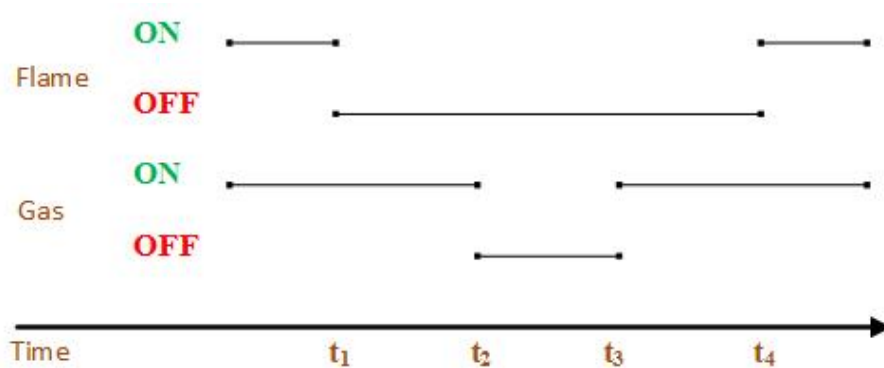


Figure 4.9: Leaking gas burner example

- Burning phase:  
both the flame and the gas are off.
- Idle phase:  
both the flame and the gas are on.
- Leaking phase:

The Digram 4.9 below shows:

- Case 1 (leaking):  
two leak periods ( $leak = Gas \wedge \neg Flame$ ) from  $t_1$  to  $t_2$  as well as from  $t_3$  to  $t_4$ .
- Case 2 (No leaking):  
there is no leak period from  $t_2$  to  $t_3$ .



- Case 3 (No leaking):  
non-leaking period must be long enough in order to minimize the dangerous level of unburned gas.

Therefore, this can be expressed in SPITL as follow:

- Case 1 (leaking):  
 $(t_2 - t_1 : \text{empty} \wedge \neg \text{Flame} \wedge \text{Gas}) ; \text{skip}(t_4 - t_3 : \text{empty} \wedge \neg \text{Flame} \wedge \text{Gas})$ .
- Case 2 (No leaking):  
 $(t_1 : \text{empty} \wedge \text{Flame} \wedge \text{Gas}) ; \text{skip}(t_4 : \text{empty} \wedge \text{Flame} \wedge \text{Gas})$ .
- Case 3 (No leaking):  
 $(t_3 - t_2 : \text{empty} \wedge \neg \text{Flame} \wedge \text{Gas})$ .

## 4.5 Summary

In order to provide executable hybrid systems, ITL has been extended to describe behaviours of hybrid system using Spline. This chapter presented the language which will be used to describe behaviours of hybrid system. First, brief introduction about Interval Temporal Logic was explained alongside with its syntax. After that, the choice of Splines was justified. Spline Interval Temporal Logic was explained in detail alongside with its syntax, formal semantics. There is a need for this extension in order to extend the executable subset Tempura. Tempura should allow us provide an executable hybrid systems model. Iso, The automatic function to Inject assertion points using AnaTempura considered to be one of the main major contributions. moreover, using s-function Anatempura became more powerful as now there is no need to insert the assertion manually.

# Chapter 5

## Runtime verification of hybrid system Framework

### *Objectives:*

---

- To provide an overview of the proposed framework
  - To Describe the proposed framework architecture
  - To discuss how the framework components interact
-

## 5.1 Introduction

Today, whether it is computer hardware, auto mobile systems, air planes, washing and even small devices like a temperature control system, the importance of verification as an integral part of flow design in systems cannot be overemphasized. For instance, a temperature control system comprises of the heating element and a thermostat, so that the variables that will be included to model such a system are the room temperature and the mode of operation of the heater as to whether it is on or off.

For the temperature control system to be effective, a coupling between the continuous and discrete variables must be established so that, for example, the mode of operation of the system will be switched to ON if the temperature of the room decreases below a certain value. However, most of the dynamical systems such as computers, cars, airplanes, and washing machines etc. can be considered as hybrid systems. Despite the advent of these hybrid systems, extant literature suggests that dynamic modelling are conducted either completely continuous or completely discrete. Recent developments has called for the development of hybrid systems which integrates both discrete and continuous dynamic systems for verification of tasks in a number of mission-critical applications, given that the interaction between continuous and discrete systems in technological problems of today have become greatly important.

For example, the loss of the Ariane 5 launcher on the 4th of June, 1996, that plunged into self-destruction mode just thirty seven (37) seconds after takeoff was blame on software error by investigators. However, in actual sense, what changed was the continuous dynamical system upon which the system operates which was integrated into the physical structure of the new launcher whose size has been increased considerably in comparison to its predecessor. Due to this change in physical

environment, the existing code led into a disaster.

In the light of the above, the development of hybrid systems that can adapt to various situation by relying on adequate verification protocols have become increasingly important given the increasing role of computers in the control of physical processes. Assuring the correctness of systems, is a difficult task due to the complexity and size of the system under consideration as well as the various degrees of requirements to be satisfied. Accordingly, the verification of the level of correctness of a given set of codes that interact with continuous environment has become vital.

The development of a hybrid system for verification purposes is complex and every systems has its own unique nature and characteristics. Despite these complexity, research efforts is currently being geared towards the development of hybrid system for verification tasks through modelling of processes of diverse complexity occurring within the system, as precisely as possible. But given that hybrid systems undergo changes including alterations and extensions due to their dynamic nature, it is difficult modelling them for verification purposes with impeccable precision. The quest to find a way around these complexities prompts the embrace of approximate representations of observed events in hybrid systems. These approximate representations assist in providing insight, even if not completely adequate, to gain appreciable understanding of the underlying law(s) leading to the observed events.

The first attempt at most approximations of any physical system is an illustrative description and understanding of the system being studied. Consequently, this helps in establishing the phenomena of logical assumptions necessary to limit the boundary of complexity of the system under consideration. With the assumptions in mind, inferences regarding the relationships between the system under observation and certain parameters as well as factors of interest, can be drawn. The goal of initial observations of the system and identification of appropriate assumptions is to provide the foundation and principles for the mathematical and computational frameworks

of the underlying phenomena being investigated. Consequently, these frameworks impose what can be described as a vast array of input-output relationships on certain variables in relation to others.

In Chapter two, the limitations of the existing research efforts relating to the development of hybrid system were discussed. The pathway which the current research adopted was also presented. Specifically, it was highlighted, that new verification protocols and techniques are required to improve the overall quality of a verification endeavor. The formal verifications of such systems based on specifications that can guarantee their behaviour is very important especially as it pertains to safety-critical applications as highlighted above. Accordingly, this chapter therefore presents a detailed description of the framework denoted as Runtime Verification of Hybrid Systems framework (RVHSF) in terms of its underlying principles and hardware components, including system architecture and structure, system requirements and system outputs. Also provided in chapter is information on how each component of the proposed framework interacts with each other within the overall model of the hybrid system.

The overall goal is to generate executable models for hybrid systems using Interval Temporal logic (ITL). Whereas other formal approaches adopts hybrid automata, (i.e. HTL), the current work adopts ITL by leveraging on its executable subset known as Tempura which has the competitive edge of being able to define more complex temporal features to describe complex systems such as a hybrid system. In the subsections that follows, a full description of the individual components of the overall framework is presented.

## 5.2 General overview of the framework

Figure 5.1 shows a pictorial representation of the overall framework upon which the verification protocol. As stated earlier, hybrid systems are a crossbreed of continuous (real-time) dynamics and discrete events that not only coexist but interact with each other with changes occurring both in response to discrete instantaneous events as well as in response to dynamics as described by the difference equation in time. Given the non-restrictive definition of the term hybrid systems, they are better described using a specific and unique framework to indicate the key issues which the system seeks to address. In the current work, the overall goal is the extension of standard program analysis techniques for systems with the view to verify the correctness of their input versus output parameters.

## 5.3 System specifications (SPITL)

As indicated in Figure 5.1, the first most important step in the overall framework is the specification protocol which is developed based on a new logic termed SPLINE Interval Temporal Logic (SPITL), an extension of Interval Temporal Logic (ITL), which is the hallmark of the current work. SPITL was adopted for the specification of hybrid system in this research because it is particularly suited for modeling change over time in the form of splines that interpolate the discrete time points to describe changes in perception [9]. Additionally, SPITL is endowed with temporal operators that can model behavioral pattern induced by changes in observations over time in the form of splines. However, a fundamental setback of the SPITL sub-framework is that it is not executable, as such it has to be made executable using appropriate means within the overall framework.

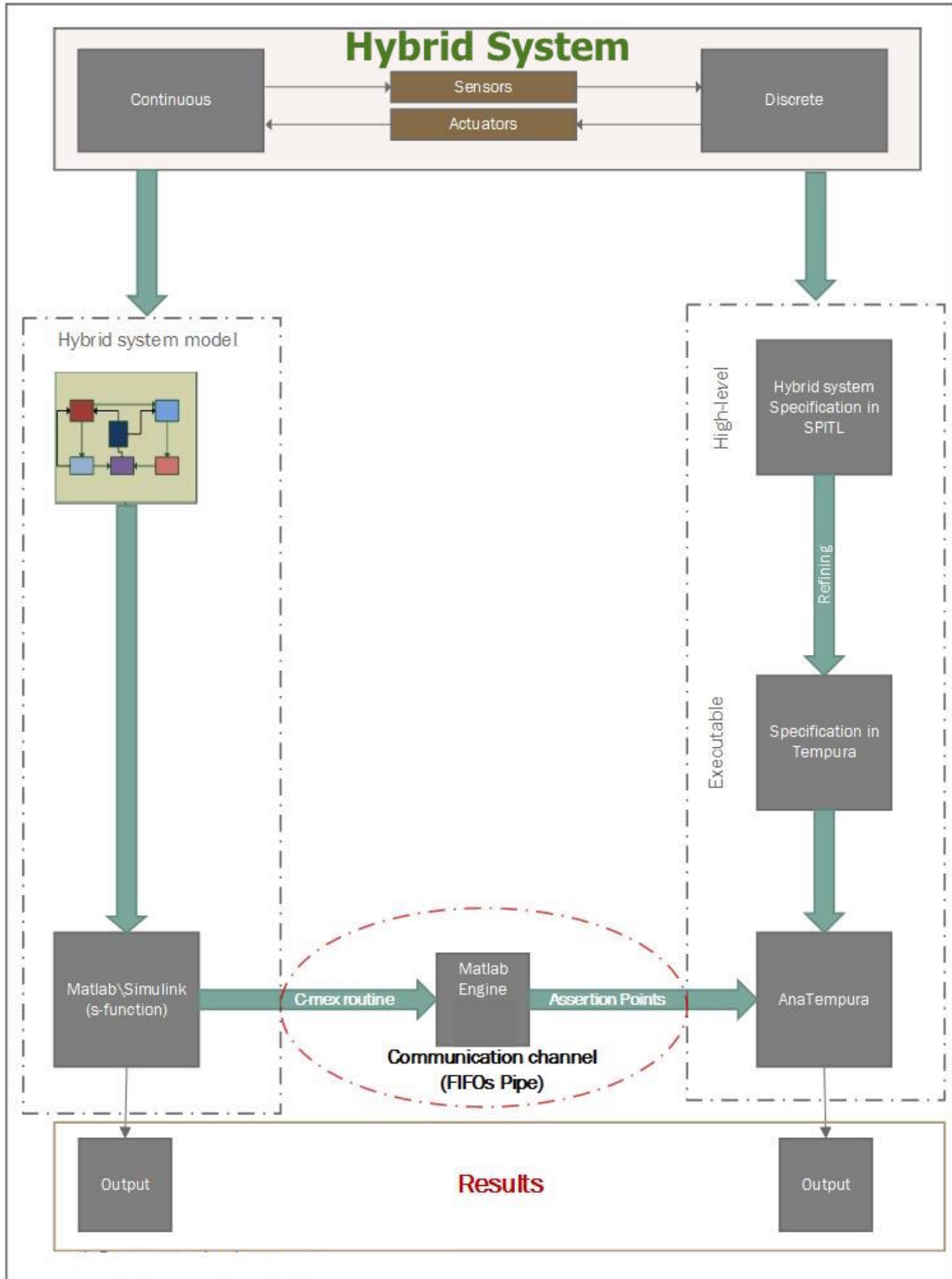


Figure 5.1: General framework

## 5.4 Modelling specifications in Tempura

In the light of the setback highlighted in section 5.3, Tempura - a temporal logic programming language which is an executable subset of ITL was used to carry out the execution procedures for the purpose of specification and verification of system under consideration. Given that one of the key demerits in the field of verification is that different programming languages have been adopted for the development of program codes, their associated properties and other attributes, it is important to incorporate a programming technique that can interface between different programming languages because it will be difficult to adopt the same language in each case. Tempura therefore assist in actualizing this within the context of our framework. Tempura offers an approach to directly execute temporal logic specifications that are suitable for the hybrid system. Given that every statement within Tempura programming language is a temporal formula, it is possible to adopt the whole temporal logic formalism as assertion language and semantics (Moszkowski, 1986). This will be explained as part of the overall description of the framework in subsequent sections.

## 5.5 Matlab/Simulink (s-function)

To improve the efficiency of designs and verification framework for the hybrid system under consideration after execution within ITL/Tempura, it is important to simulate the entire system in terms of its underlying software and hardware platform using appropriate simulation tools for validation. This was achieved as part of the overall framework in this work using Simulink - a graphical or visual programming environment within Matlab programming tool, developed by MathWorks for the modeling, simulation and analysis of dynamic systems in a multi-domain



environment. With its primary interface as a graphical block diagramming tool and customizable set of libraries, Simulink can be adapted for the verification of hybrid systems. Essentially, the use of ITL and its executable subset (i.e. Tempura) linked with Matlab/Simulink provides a formal approach for specifying and modeling hybrid systems and for the acceleration of test-bench procedures. In particular, the ability of Simulink/Matlab to integrate with other programming languages and third-party applications makes it stand out. Accordingly, Simulink/Matlab within the framework adopted in this research was used for data generation and analysis. To achieve this, the use of S-function was introduced within Matlab/Simulink.



Figure 5.2: The integration of ITL/Tempura within MATLAB/Simulink using C-MEX S-function

The S-functions is generally considered as the computer language description of the Simulink block which provide a convenient mechanism for the implementation of custom control blocks that has the capability to interchange run-time data with equation solvers within Simulink. It allows for the development of codes that allows a C function to be called from Matlab. In this framework, the S-function is developed in C programming language and it is rightly tagged C- MEX S-function, allowing for the creation of custom blocks within multiple input and output ports with the capability of handling any form of signal generated within the Simulink model. The use of S-function during the design and development phase of the hybrid system under consideration hugely facilitates the simulation speed and accuracy

of the results. Accordingly, the developed C-Mex allows for direct interchange of data between different optimization routines whilst avoiding the use of the Matlab programming environment.

The implementation of verification protocols within a hybrid system can be time-consuming when standard or conventional development tools are adopted. In this work, such tasks was implemented based on ITL/Tempura framework, which provides some layers of abstraction for the description of underlying hardware within the hybrid system. A pipe was efficiently designed in C –programming language to establish communication between strategic interfaces within the hybrid system. The pipe connects one interface between Matlab/Simulink (S-function) through C-MEX routine to Matlab engine and connects the other interface to AnaTempura (discussed in the section that follows) through the injection of assertion points. Figure 5.3 illustrates how S-function is integrated with ITL/Tempura framework.

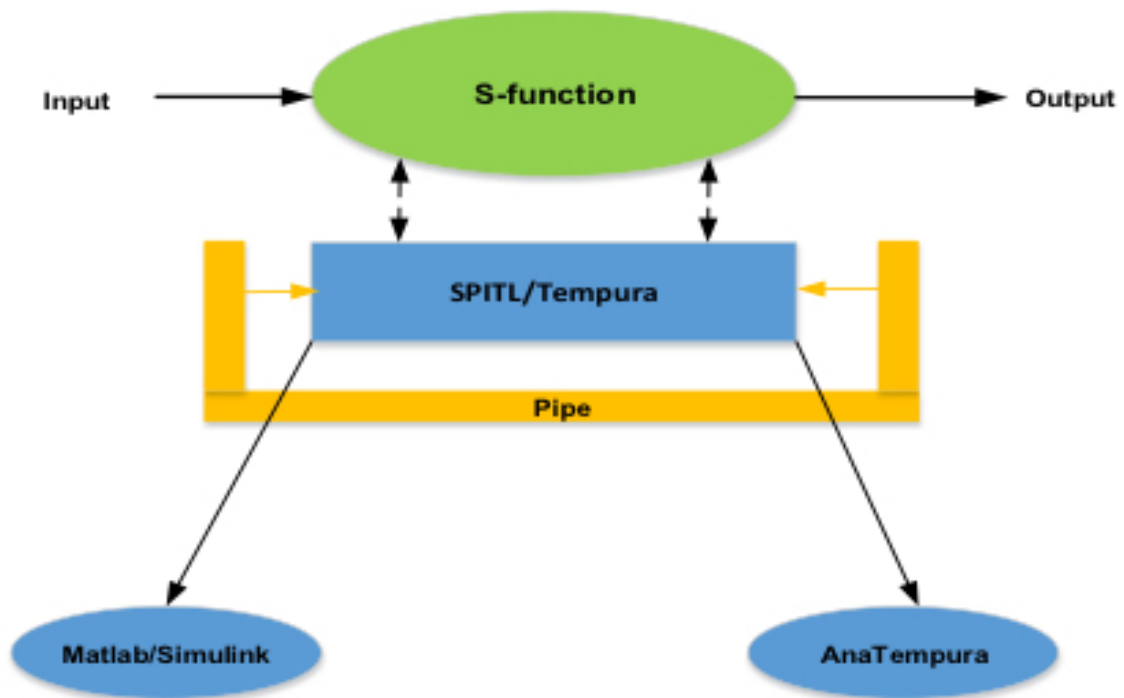


Figure 5.3: Illustration of how S-function is integrated with ITL/Tempura framework.

A setback with the use of AnaTempura is that it is not compatible with Matlab, as such a pipe is needed to establish link between AnaTempura and Matlab using C-programming language given its bi-directional compatibility with both languages. Also, given that Simulink is a simulation tool within Matlab, a C-MEX within Matlab is required to aid the simulation of the model under consideration. This then allow for the injection of assertion points within a Simulink simulation environment. Accordingly, AnaTempura can read and write as well as receive data from the C- MEX which is compatible with Simulink. After establishing linkages and communications using the C-pipe developed, results can then be obtained from each AnaTempura and Simulink from which expected results can be checked for validation in terms of whether the simulated results is in agreement with the defined reference model output. This is achieved using a subroutine developed in C as part of the overall model.

## **5.6 An Automatic function to Inject assertion points using AnaTempura**

As highlighted above, in order to ascertain that the model developed within Simulink is functional or not, a runtime verification tool of ITL is employed within the framework to establish all the desirable properties of the system under test including functionalities, timing and allocation of computing resources. They are equally used for the insertion of assertion points at suitable places in the source code within the overall code of the hybrid system model. Assertion points are employed as a way of managing changes within a hybrid system and they developed at source code level. Figure 5.4 illustrates the mechanism of assertion points within the framework

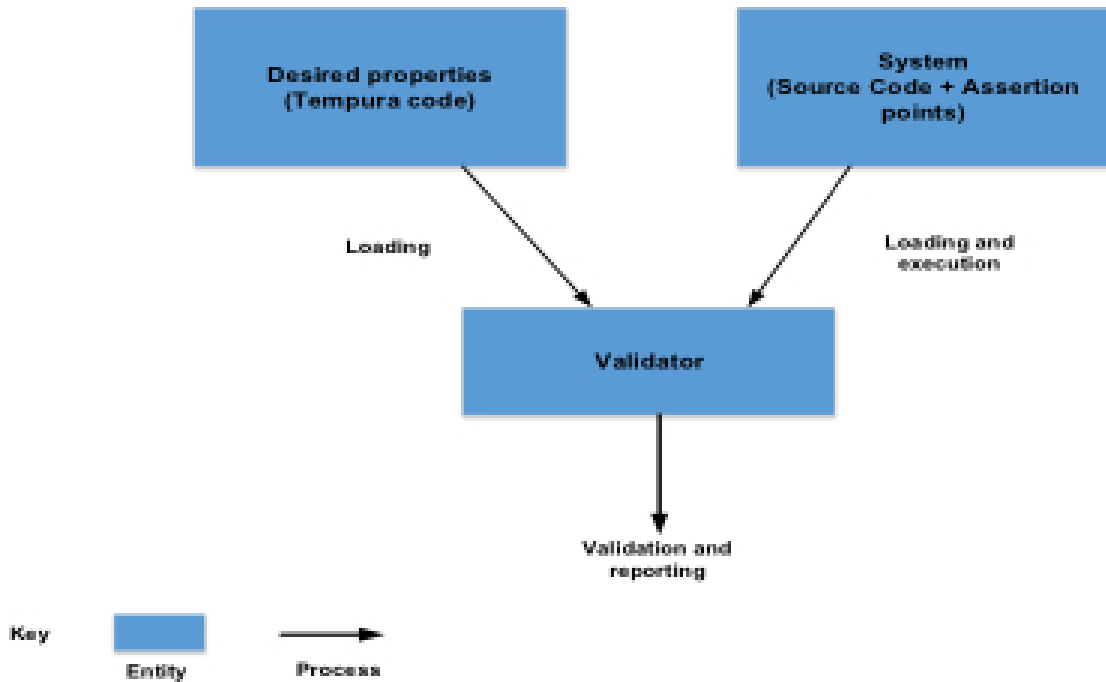


Figure 5.4: Illustration of assertion points (Adapted from kun thesis)

AnaTempura is the programming concept used to ascertain whether the behavior generated or observed satisfies the desired properties. The location of assertion points within the overall code is an important step that is determined by the variables used in defining the property of interest. This is achieved through simple search within the source code so as to locate all places where the defined variables are changing. Those identified places of change therefore represents the assertion points and will ensure that the observed behavior during the runtime of the system is deemed the desired and correct behavior. For instance, as depicted Figure Figure 5.4, for a given property (i.e. desired property) of the hybrid system that is to be validated, the assertion points embedded within code is used to ensure such validation of the desired property during execution time. Such desired properties are formulated and expressed based on Tempura code. Generally speaking, assertion points are added based on the property of interest at an instance of time and they comprise of two components, one for the generation of information and the other

is the mechanism for the processing of the information. This mechanism captures as well as interprets information generated by assertion points but also ensure the validation of such properties which is tagged validator as indicated in Figure 5.4.

Within the hybrid system's overall framework, the inclusion of assertion points occurs in the transformation process based on Tempura code to executable code (e.g. C). Given that assertion points are based on information such as variable, value and time stamp, they are programmed as a function or a subroutine to capture the variable under consideration with the appropriate value under the right time stamp. Figure 5.5 depicts how AnaTempura is adopted for runtime verification. The verification procedure can be summarised as follows:

1. Establishment of all the desired systems properties including functional, timing and resource attributes.
2. Insertion at appropriate places within the source code of the system assertion points.
3. The use of AnaTempura to ascertain whether the generated behaviour meets the criteria defined for the desired properties.

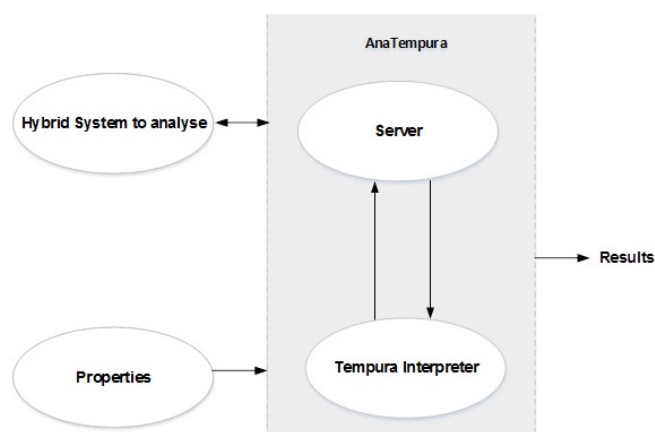


Figure 5.5: Illustration of runtime verification based on AnaTempura

## 5.7 Chapter summary

In Chapter four, the SPITL and its extension were extensively introduced for the rationale and specification of hybrid systems based on SPITL. This chapter presents the detailed description of the building blocks of the underlying framework of the hybrid system modelled in this thesis. It describes all components of the framework, identifying how they interlink with each other. In the chapter that follows, detailed description of the implementation of the hybrid system is provided. Also, The automatic function to Inject assertion points using AnaTempura considered to be one of the main major contributions. moreover, using s-function Anatempura became more powerful as now there is no need to insert the assertion manually. Next chapter will discuss how we can implement this framework.

# Chapter 6

## Design and Implementation

### *Objectives:*

---

- Provide the reasons of selection of tools for the implementation.
  - Provide the architecture of selected components.
  - Present the implementation of each component.
-

## 6.1 Overview

This chapter presents the implementation of the proposed Hybrid System verification system which forms the basis of the current work. The verification components of the system include AnaTempura, Assertion point, Matlab Engine, S Function routine, FIFO Pipe and the Simulink Model. Also presented is an explanation of the functioning of the Simulink and Model implementation in terms of how it operates within the Simulink and AnaTempura verification System.

The chapter introduces the main components of the Simulink and Model based Implementation, providing insight into the system design using class and sequence diagrams which illustrate the system structure and processes respectively. The chapter concludes with an explanation of the mapping between AnaTempura and Matlab, as well as an explanation of the implementation of the framework of the system.

## 6.2 Simulink and Model based Implementation

Simulink[140] is the model simulation engine developed by mathworks; it works by simulating a model presented by a system of differential equations in continuous time or difference equations in discrete time. The way systems of equations are presented is not necessarily explicit; Simulink provides a graphical user interface with a library of blocks and a canvas to instantiate the blocks and connect them together. The composition of blocks describes the time differential of the system with state variables created as necessary as indicated by blocks.

The focus of this research is hybrid systems, which is a combination of continuous and discrete time. Therefore these systems can be defined by combination of differential and difference equations [120]. The state of a hybrid system either changes with a flow (continuous) or in discrete steps, therefore this interaction of discrete



and continuous change make the hybrid systems interesting and challenging to perform formal analysis. The study based on mathematical modelling is long present for the individual discrete and continuous system; however the formal analysis of hybrid systems is fairly new and can be traced to Maler et.al [122] in the field of computer science.

A good example of a hybrid system is a Digital controller, now let suppose a computer scientist is given a design, how are they going to ensure that the design meets the specification. The first task is to have a method to define formal specification requirements of the system. Since Interval Temporal language does support discrete timed logic but lacked support for the hybrid systems, therefore this work is meant to extend the capability of the ITL to support the hybrid systems.

### **6.3 AnaTempura**

Since AnaTempura is the key formal verification tool used here, this was a requirement to produce a model of a hybrid system, and then can have an online transaction between the model and the formal verification engine such as AnaTempura. Since, Simulink is a powerful modelling tool and have required blocks and libraries to model a complex hybrid system, therefore this was chosen as a modelling tool. Since this sort of modeling and verification needs interaction between two different applications therefore an inter process communication such as pipes. The details of this implementation is discussed in detail in this chapter in the following sections

### **6.4 Steps to compiling the Design**

One of the implementation challenges for modeling the Hybrid system is to maintain the information flow between the two applications i.e. the Simulink model and

the Anatempura runtime environment. Other than the Anatempura, Matlab and Simulink engines are used and the whole process flow is divided into four major steps, which are then explained further. The case study is based on the modeling of a mine pump digital controller.

- Firstly, we need to generate the Tempura code that is run by tempura interpreter(AnaTempura), and is used to validate the mine pump controller outputs.
- Secondly, the EXE file compiled from C source which is called from AnaTempura and which in turn calls the Matlab engine.
- The Simulink is running the hybrid system model that simulates the environment, controller and the sensors. An S-function which provides the interlink between the Simulink model and the Mex compiled C code.
- the MEX file compiled from C source which is called by our and provide the link to the Anatempura run time environment.

All four components have different runtime environments in terms of access to console and standard C library; including both C based executables such that MEX environment does not provide access to all API functions as limited by Matlab's own C compiler.

## 6.5 AnaTempura and Assertion point

The first step in interaction between the AnaTempura which validates the model is the insertion of the assertion points. The assertion points are used in the code to check the validity of a certain property at run time. For evaluating the behaviour of the Digital controller of the hybrid system modelled in simulink. Anatempura

code is used to validate the different properties, while the assertion points in the code are used to pass the parameters to and from the C-Mex files generated from the Simulink model. The assertion points are basically used to serve two purposes, firstly to generate information and secondly to process the information received.

The processing step is the part where the properties of the system are validated. The assertion points gathers and emit data from the binary level.

The location of the assertion points are chosen carefully, for e.g. to find the current state of the variables, which are changing in the background Simulink process, an assertion point is used to read a variable. Now when this is received and based on the properties and condition of the received value this is processed accordingly, and then a result is generated, which connects to an actuator in the Simulink model in form of a variable.

```
assertion("MethanePresent",MethanePresent[0]?1:0);
assertion("WaterLow",WaterLow[0]?1:0);
assertion("WaterHigh",WaterHigh[0]?1:0);
if(MethanePresent[0]){
    XD[0] = false;  assertion("XD",0);
} else {
    if(WaterLow[0]) {XD[0] = false; assertion("XD",0);}
    if(WaterHigh[0]) {XD[0] = true; assertion("XD",1);}
}
}
```

This code first assert to read the status of different variables from the Simulink model for example "MethanePresent". Based on the outcome of this variable, the tempura code make a decision about the motor turn on or off and this value is again passed to the Matlab engine the S-function using the assertion command. The output provided is a binary output either true or false.

## 6.6 Matlab Engine

Engine applications are standalone C/C++ that allow to call MATLAB from inside the user defined C/C++ program, and can utilize MATLAB as a computational environment. To run a Matlab Engine, a MATLAB installation is required on the machine and it will not work with the run time Matlab compiler. The Matlab engine can be called by creating an Engine object with the handle “Engine” and all its prototypes are available in the “engine.h” Matlab files.

(Engineopen) function is used to open the Matlab engine, the function must have a NULL parameter if operated in Windows. The (EngineEval) String function is used to evaluate the string present in the function for the Matlab Engine for example:

```
”engEvalString (eng, ”mex -c minePumpcontroller.c  
minePumpcontoller_wrapper.c ”);”
```

This instruction code will execute the “C” files using the mex compiler. Similar Strings can be evaluated and executed using the (evalstring) function.

## 6.7 S Function

S-functions [190] also known as system function offers a powerful and useful mechanism for the extension of the functional capabilities of the Simulink environment. An S-function is a form of a programming language that offers the description of a Simulink block and is written in many programming languages including C,C++, Matlab or even Fortran. Within an S-function, C, C++ and Fortran are compiled as MEX files using mex utility (see for instance the Build MEX-File).

In accordance to other MEX files, S-functions are regarded as a subroutine that are dynamically linked with Matlab so that the Matlab interpreter can be loaded or

executed in an automatic fashion. S-functions adopt a special calling syntax known as the S-function API which allows the programmer to interact with the Simulink engine.

This interaction is almost the same with the interaction occurs between the engine and built-in Simulink blocks. As such, S-functions follow a general form and can work perfectly well for continuous, discrete and hybrid systems. In this work, the S-function is linked with the (minepumpcontroller C file).

The S-function uses a standard template from the `sfuntmpl doc c`, the user needs to customize it for its particular usage. This files provide code template for initialization, defining the size and number of input and output ports, i.e. the number of S-function parameters. Also the Sampling times are setup to determine the number of times the S-function needs interaction between the host program and the Simulink model.

The S-function has standard Simulink out and in ports which provides the input and the output to the remaining model. The in port provides the input variables from the Simulink model and is read through the C mex executable and through the assertion points to the AnaTempura environment which are then used to validate and generate a corresponding output to be given back to the Simulink model to control any actuators in the model.

## 6.8 FIFO Pipe

Now the big question is that how the data is being passed between the Simulink model to the C Mex environment and to the AnaTempura executable environment. In this case a Fifo pipe is used to read data to the AnaTempura. A pipe is an OS based mechanism to communicate between different running processes. One of the processes writes to the pipe and the other process may read through it. There are

several ways of implementing pipes, in this case study a Fifo pipe is used. A FIFO pipe is a first in and first out mechanism, thus the data written first is first read by the other processes. This data is picked up by the EXE file after the call chain returns.

The process to read and write follows:

- A C executable is used to, create a named pipe using the "mkfifo" command, command, for example: "mkfifo" *MYPIPE*. Normally in Windows or OS environments this can be created by a C executable.
- In the directory where this was done a new file named *MYPIPE* appears.
- Redirect the output of C Mex process to that pipe.
- Inside Matlab, a file open and reading function can be used from the pipe exactly like reading from an ordinary file

## 6.9 Simulink Model

The actual mine pump model (our hybrid system case study) is implemented using the Simulink library and the functions. A mine pump problem can be defined as water infiltrating a mine and the water is collected in a sump with the view to be flushed out of the mine, see Figure ??.

This is a classical hybrid system where the water discharge rate and the methane level can be modelled using a continuous differential function, while on the other hand the water low and high level are discrete quantities.

Readings from all sensors, and a record of the operation of the pump, must be logged for later analysis. In the Simulink model the Low and high water levels are monitored by the relational operators and also the methane and air flow is simulated by a sinusoidal function. These low and high water level and the methane

level is passed as a binary input to the S-function, these values are passed to the Anatepura executable through the C mex named pipe. The Anatepura verify the behaviour and take the appropriate decision and send the result as a binary output to enable or disable the motor, which controls the water pump, to extract additional water.

Sensors D and E are sensors that monitors water levels and have actuators that can detect when the water level rises above or below a defined threshold. A pump controller activates the pump when the water level reaches a high and deactivates it when it goes below the defined threshold. In the case of a failure in the pump such that water can no longer be pumped out, then the mine must be evacuated within one hour. The mine has other sensors (A,B,C) that are configured to monitor other measurements such as airflow, carbon monoxide and methane levels.

Within the overall setup, the system is configured such that an alarm is activated to alert the operator as quickly as possible about the criticality of the measurements under observation within the system so that appropriate and timely evacuation procedure of the mine is commenced within the shortest period of time of roughly one hour.

To prevent explosion risks, the operation of the pump must be embarked upon only when the level of methane is below a critically defined level. The operation of the pump can be influenced by human factors but it must be within certain defined limits. The pump can be switched on or off by an operator the water level is between the low and high thresholds that are already defined. A special operator, or possibly the supervisor, can activate or deactivate the pump when necessary without this restriction. In every case, it must be ensured that the methane level is always below the defined threshold whenever the pump is to be operated.

The description highlighted above is a classical hybrid system where the water discharge rate and the methane level can be modelled using a continuous differential

function, with level of water either low or high representing discrete quantities. Within such systems, measurements from all sensors, and the details of the operation of the pump, must be logged for later analysis.

In the Simulink model, the Low and high water levels are monitored by the relational operators and also the methane and air flow is simulated by a sinusoidal function. These low and high water levels as well as the level of the methane is passed as a binary input to the S-function, which are then passed to the AnaTempura executable through the C mex named pipe. The AnaTempura verify the behaviour and take the appropriate decision and send the result as a binary output to enable or disable the motor, which controls the water pump, to extract additional water.

## 6.10 Summary

This chapter describes the case study adopted in this thesis and gives an overview of its overall implementation detailing the interaction between the hybrid system being modelled in the Simulink, and the S-function that is used to pass the variables from the model through a named pipe to the AnaTempura, which has the classic model of the system and can be adopted in run time to verify the behaviour of the hybrid system and validate the values generated. In the chapter that follows a further detailed discussion of the model in terms of evaluation and implementation of the framework is presented.



# Chapter 7

## Case study and Evaluation

### *Objectives:*

---

- Overview
  - case study Design
  - Running the case study
  - case study Evaluation
-

## 7.1 Overview

A number of investigations have been carried out to examine the theory of this research. Substantial progress has been made over the past years in the field of real-time control system development. However, the modern development methods, languages, and tools are still not mature enough to solve essential problems in this area. A real-time control system should be described as a reactive system with predictable behaviour, including the timing domain. There is, for instance, no development language, method, or tool, which supports the specification, the correctness analysis, and the simple development and maintenance of systems with timing constraints.

## 7.2 Mine pump system (the case study)

The Case study chosen in this work is a control of a simple mine pump control system to control excess water flow in a mine. It controls the water pump to discharge any excess water considering water levels, methane level and the air flow in the mine. Figure 7.1 illustrates the Mine pump system.

### 7.2.1 Case Study Description

The mine pump has basically four sensors, in our study only three has been considered. The air flow sensor has been ignored, while water low, high and methane sensors are used. These sensors provide continuous input to the mine pump controller, which continuously monitor these levels. The system also needs to monitor the state of the pump, which is either turned off or on. The purpose of the pump is to control the flow of the water. When pump is on this means its discharging the excess water out of the mine. However in dangerous condition or when water levels

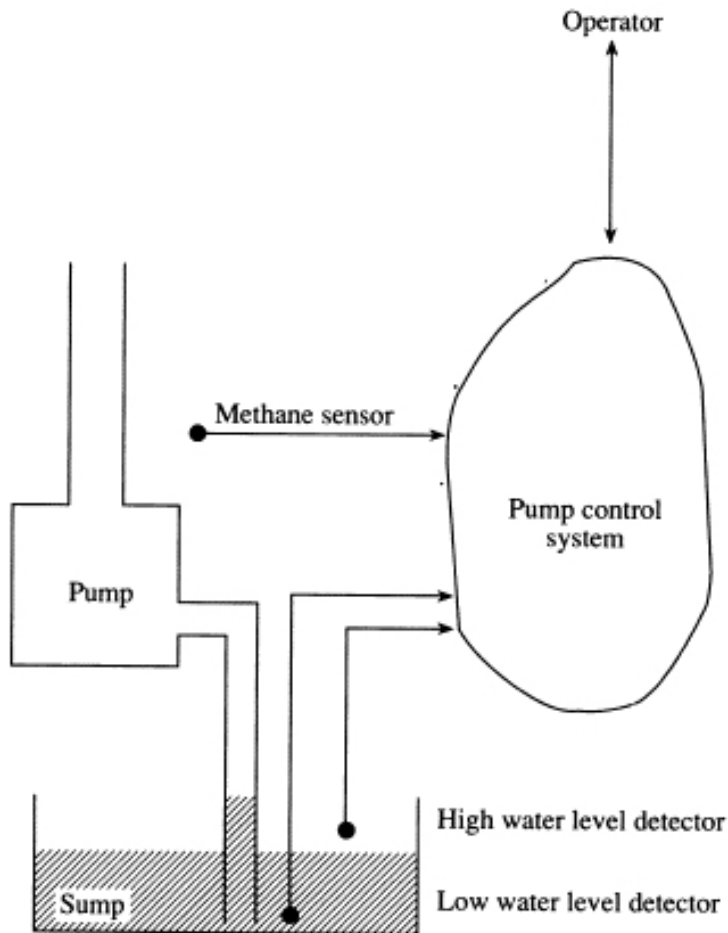


Figure 7.1: Mine Pump System modified from [120]

or in normal or low levels the pump need to be stored.

This problem has been a classical case study for hybrid systems and computer scientists have used this as baseline to test tools, verification processes etc.. The main goal is to somehow verify the model behaviour implemented in an environment of choice, using the ITL constructs, specially using its executable form *Ana tempura*. The implementation details of this model have already been discussed in a previous chapter, the focus here is to get a more in-depth understanding of the model and its verification process.

A mechanism for managing change in a legacy system should be practical, systematic and compositional. A fundamental issue in our approach is the ability to

capture the behaviour of (sub) system. Once the behaviour is captured then we can assert if such behaviour satisfies a given property. And as a property is a set of behaviours, satisfaction is achieved by checking if the captured system's behaviour is an element of this set. We are not dealing here with the formal verification of properties which requires that all possible behaviours of a system satisfy the properties. The formal verification of these properties may also be performed using an ITL verifier.

We are only concerned with validating properties which requires that only interesting behaviours satisfy the properties. The states of a (sub) system to be changed are captured by inserting assertion points at suitably chosen places. These divide the system into several code-chunks. Properties of interests are then validated over this behaviour. A mechanism for managing change in a legacy system should be practical, systematic and compositional. A fundamental issue in our approach is the ability to capture the behaviour of (sub) system.

Once the behaviour is captured then we can assert if such behaviour satisfies a given property. And as a property is a set of behaviours, satisfaction is achieved by checking if the captured system's behaviour is an element of this set. We are not dealing here with the formal verification of properties which requires that all possible behaviours of a system satisfy the properties. The formal verification of these properties may also be performed using an ITL verifier. We are only concerned with validating properties which requires that only interesting behaviours satisfy the properties.

The starting point is formulating all behavioural properties of interest, such as safety and timeliness. These are stored in a Tempura file. An information generating mechanism, namely, Assertion points are stored in a C file. These Assertion Points will generate run-time information (assertion data), such as state values, time stamps, during the execution of the program. The sequence of assertion data will

be a possible behaviour of the system and for which we check the satisfaction of our property. This chapter will cover initially the functional and temporal requirements of the hybrid model. We will look into finding out the different operational conditions and its related outcomes in terms of real-time constraints. Further, use the ITL to put these real-time constraints in for validation purpose.

### **7.2.2 Specification of mine pump system in SPITL**

Spline Interval Temporal Logic (SPITL) is a flexible notation for both propositional and first order reasoning about periods of time found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and timeliness. Choice of ITL in this work is based on the availability of an executable subset of the logic. This offers a flexible and rapid prototyping system, known as Tempura. Its syntax resembles that of SPITL. It has as data structures integers and Booleans and the list construct to build more complex ones.

The verification model and boundaries are defined in AnaTempura using the SPITL syntaxes. The AnaTempura executable will be in continuous communication with the real-time model running in Simulink. The communication between the AnaTempura executable and the Simulink model is performed using the Matlab C-Mex and S-function using FIFO pipes between the different processes. This process is defined later in this chapter and a briefing on it is also available in the Implementation chapter three. Before putting the model specification for validating the model, this is necessary to understand the functional and timing requirements of the mine pump problem.

### 7.2.2.1 Functional requirement

The functional behaviour of the pump can be defined in terms of its two operating conditions i.e. when it's running and when it's off.

Therefore, the pump is running or turned on when

- The water level is higher than the water high mark and the pump is off and the methane level is lower than the critical value.
- This can also be turned on by an operator if the water level is above the low water level and the methane levels are within bounds of the critical values.
- This can also be turned on by a supervisor only in case if the methane levels are less than critical.

For this requirements this make the methane levels as the most critical functional requirement to operate the pump, hence the continuous monitoring of this is the most time critical and safety critical event. The water level monitoring is quiet important however the methane levels need to be checked before any pump switching on operation can take place.

Now, the next thing is to check for the conditions of switching off the pump

- When the Water level is less than or equal to the lower limit of water level to be maintained.
- Or in case when the methane levels go above critical and pump is on then it immediately needs to be shut down.
- The pump can also turn off by an operator or supervisor when the water level is less than the high water level.

The turning off is again a critical function of the level of water plus of methane as it is an explicit requirement to turn off the pump even if methane levels are above

critical levels and water levels are reaching or going above the high water level, then an immediate evacuation must be triggered on.

### **7.2.2.2 Timing requirement**

The other major requirement of the model checking is to monitor the timing of the processes which are taking place and as per safety requirements a late action is no action and is considered a failure. Therefore, the main timing requirements which need to be looked into to verify the mine pump model.

- Frequency or periods of monitoring the sensors i.e. the two water levels and the methane level. Now this can be a bit complex as the monitoring rates must comply to the possible flow rates of the water, pump operations (rate of water pumped out) and the worst case methane flow levels. The values are discussed in the Simulink model which is assumed in this study.
- The other important time line is the evacuation or shut down deadline. The pump needs to be switched once the methane levels increase a certain level to avoid people trapped in and there is a blast. The shutting down of the pump is related to the methane sampling frequency and is directly related to increasing rate of the methane. This relates to the deadline, hence the safety margin needs to be in correspondence with the combination of the period and deadline times the rate of methane flow.

### **7.2.3 Writing the requirement in Tempura**

The main Tempura file which is executed implicates the conditions is shown below. In this test basically the three inputs X, Y, Z are checked for 8 possible cases and verified the behaviour of the Simulink model running in real-time.

- The code first load the Ana tempura library files conversion and exprog.

- Enginecode, which is the C executable which calls the Matlab engine and interacts with the Simulink model. The message is displayed as, when the engine code starts as shown on the figure below:

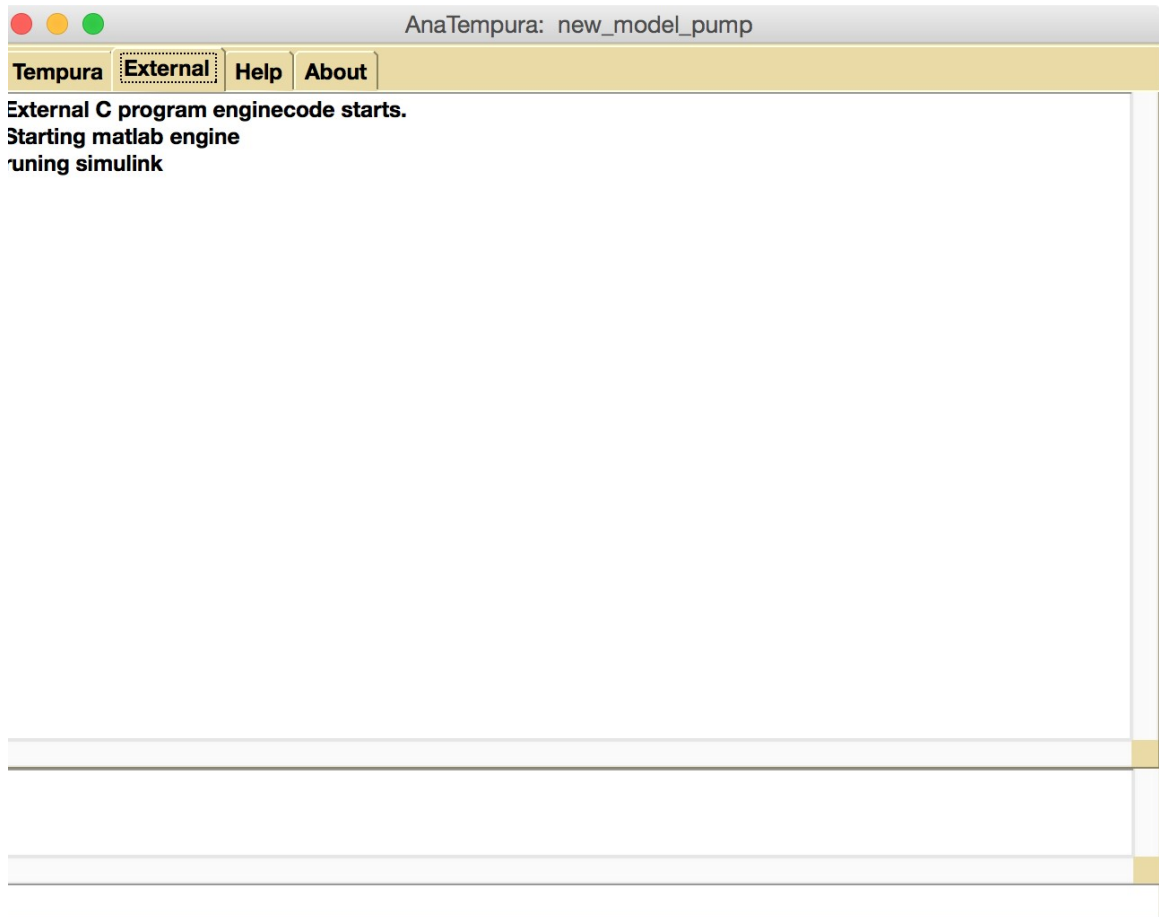
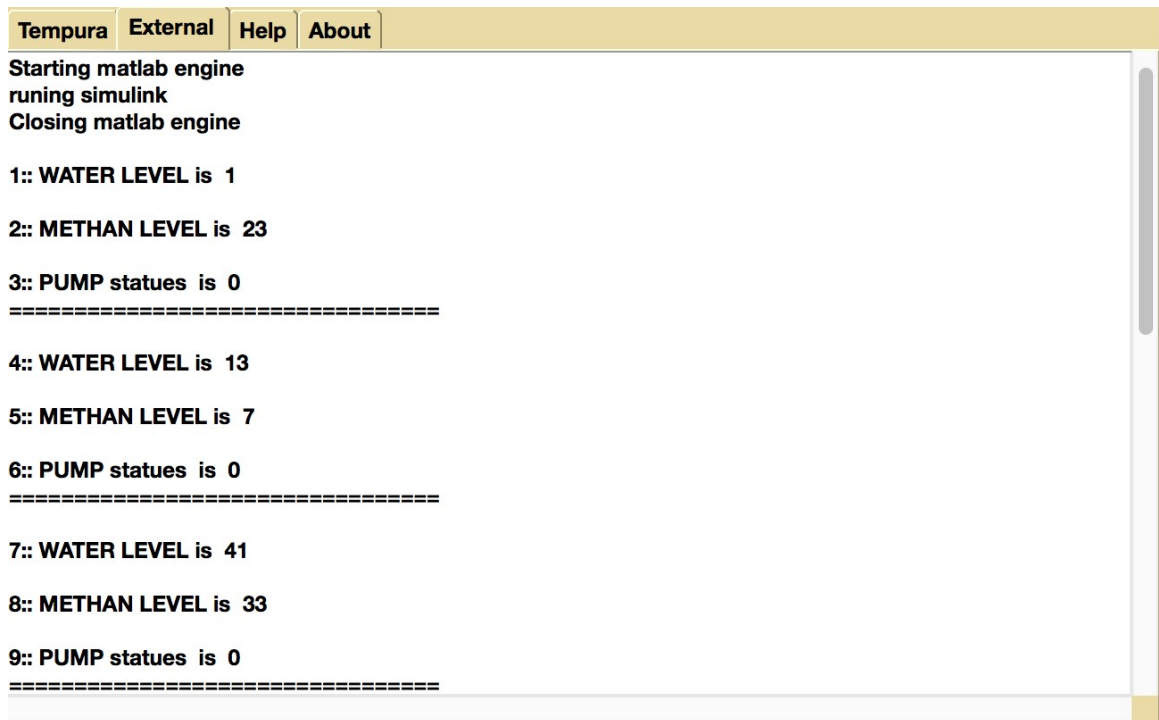


Figure 7.2: Matlab engine Code starting connection in AnaTempura

- Then the program defines the critical levels to which the program shall monitor to take decisions, for example the methane critical level is set as 15, water high level is 30 and low as 10.
- A get function is used to retrieve the variables through the FIFO pipe and then compared for different conditions to make the decisions. The status of all the variables are retrieved and updated on the output as shown on the figure below:





```

Tempura External Help About
Starting matlab engine
runing simulink
Closing matlab engine

1:: WATER LEVEL is 1

2:: METHAN LEVEL is 23

3:: PUMP statues is 0
=====

4:: WATER LEVEL is 13

5:: METHAN LEVEL is 7

6:: PUMP statues is 0
=====

7:: WATER LEVEL is 41

8:: METHAN LEVEL is 33

9:: PUMP statues is 0
=====

```

Figure 7.3: Variable update on the Ana tempura external console

- The two main decision structures are based on the current state of the pump switch status. Lets first analyse the situation when the pump is on
  - If the Methane critical level is less than critical and pump is on means the supervisor has turned it on.
  - If both the Methane level is less than critical and water level above the low level of water then the operator is allowed to turn the pump on, this is level of precedence that only the supervisor is able to turn on the pump at any levels of water if and only if the methane level is less than critical.
  - In case when the Methane level is less than critical and water level is above high level then the system shall intervene and turn on the pump.
- The other structure to look is to turn of the pump in the following conditions.

- If the methane levels are above critical, this shall immediately trigger the pump off operation.
  - Also, when the water level reaches below or equal to the lower limit then it shall trigger the pump to turn off to stop any further drain out of water to keep it in required limits.
  - And, if the water is above lower limit and less than the high then the operator can turn off the pump as required.
- The supervisor can turn off pump at any time and override any other conditions.

below is the AnaTempura code that check mine pump system:

```
load "conversion".
load "exprog".
/* prog enginecode 0 */
set print_states = false.
define critical=15.
define high_water=30.
define low_water=10.
define pump_on=1.
define pump_off=0.
define get_var(X,Y) =
{
  exists T : {
    get2(T) and /*output(etime(T)) and*/
    if avar(T)=X then {Y=strint(aval(T))}
  }
}.

```

```
define water_switch_on(X,Y,Z) = {
    if Y < critical    then {
format("CASE 3::the pump is switched on by the supervisor!\n")
    and
    if X > low_water then
        format("CASE 2::the pump is switched on by the operator!\n")
            else if X > high_water then
                format("CASE 1:: the pump is switched on!\n")
            }
    else
        format("CASE 8::pump cannot be switched on:\n")
    }.
define water_switch_off(X,Y,Z) = {

    if Y > critical    then {
format("CASE 5::the pump is switched off because the methane level
    is becomes critically high \n")
}
    else if X <=low_water then {
        format("CASE 4::the pump is switched off because the water
            level falls below the low water mark\n")
        }
        else if X < high_water then{
            format("CASE 6::the pump is switched off by the operator!\n")
        }
    else
```

```
format("CASE 7::the pump is switched off by the supervisor!\n")
}.

/* run */ define test() = {
  exists WATER_LEVEL,METHAN_LEVEL,PUMP_STATE: {

    for counter<50 do {{skip and get_var("WATER_LEVEL",WATER_LEVEL)
      and get_var("METHAN_LEVEL",METHAN_LEVEL) and
      get_var("PUMP_STATE",PUMP_STATE)}} and

    if PUMP_STATE =pump_off then {
      water_switch_on(WATER_LEVEL,METHAN_LEVEL,PUMP_STATE)
    }
    else if PUMP_STATE =pump_on then {

      water_switch_off(WATER_LEVEL,METHAN_LEVEL,PUMP_STATE)
    }
    else
      format("the Pump is already off as supposed and no need
to do action at this moment\n")
    }
  }
}.
```

### 7.2.4 C-Mex Code and S-function

The source code level checking the mine pump system is the C-Mex file which makes a pipe between the Matlab Simulink and the AnaTempura executable.

S-functions (system-functions) provide a powerful mechanism for extending the capabilities of the Simulink environment. An S-function is a computer language description of a Simulink block written in MATLAB, C, C++, or Fortran. C, C++, and Fortran S-functions are compiled as MEX files using the Mex utility (see Build MEX-File). As with other MEX files, S-functions are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute.

S-functions use a special calling syntax called the S-function API that enables you to interact with the Simulink engine. This interaction is very similar to the interaction that takes place between the engine and built-in Simulink blocks.

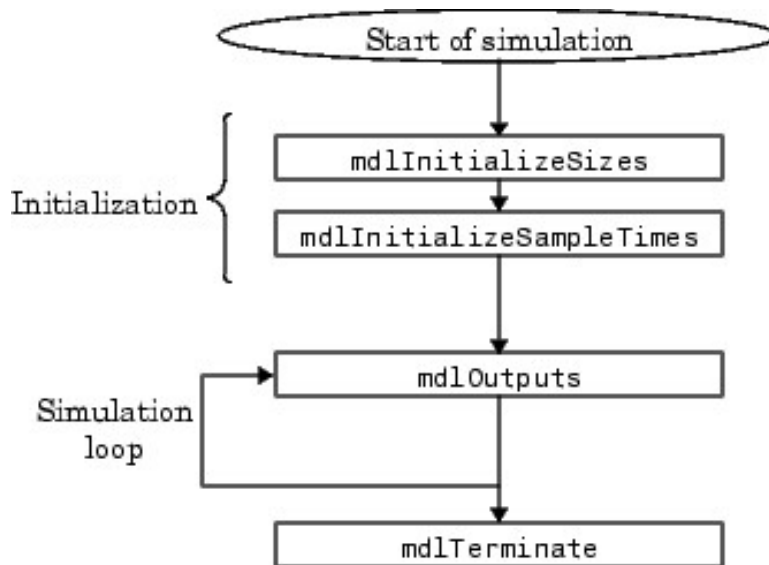


Figure 7.4: Flow of S-Function [113]

S-functions follow as shown in the figure above is a general form and can accommodate continuous, discrete and hybrid systems. In this project the S-function is linked with the mine pump controller (C) file. The S-function uses a standard

template from the (*sfuntmpldoc.c*), the user needs to customize it for its particular usage. This file provide code template for initialization, defining the size and number of input and output ports, i.e. the number of S-function parameters. Also the Sampling times are setup to determine the number of times the S-function needs interaction between the host program and the Simulink model.

The S-function has standard Simulink out and in ports which provides the input and the output to the remaining model. The in port provides the input variables from the Simulink model and is read through the C Mex executable and through the assertion points to the AnaTempura environment which are then used to validate and generate a corresponding output to be given back to the Simulink model to control any actuators in the model.

To us the C-Mex and the S-function, a custom setting is done with an update controller wrapper, which updates the variables from the Simulink model at sampling interval as defined in the model initialization function. The Sampling size is also defined in the model initialization function.

- Initially the Input and output port names are defined and the size of the variables and their dimension. The name of the input and output port must match that as defined in the Simulink model, as well the order in which they are connected to the S-function.
- The sampling time can be defined in the model initialization or can be taken from the Simulink model and in this case it is the sampling time as defined in the model with a parameter (*IN – HERITED<sub>S</sub>AMPLE<sub>T</sub>IME*).
- Once the input, out ports and the Sampling time is initialized a continuous loop will run to update the inputs and output values to and from the Simulink model, the function as de-fined in the Flow diagram is the model output function.

- A Wrapper function is written to define these inputs and output in this case, the assertion function is used for the communication. The assertion only executes only if the assertion holds and returns a true value. The C function given below provides the assertions, it first generates the water level, methane level and pumps status as random quantities and then passes these values to the model, and these are also checked by the AnaTempura executable and validate the output generated by the Simulink model

The s-function routine is illustrated in the listing code below:

```
void minePumpController_Update_wrapper
(const boolean_T *methanePresent ,
                                     const boolean_T *waterLow ,
                                     const boolean_T *waterHigh ,
                                     const boolean_T *motorEnable ,
                                     real_T *xD)
{
    int i , WATERLEVEL,METHANLEVEL,PUMP_STATE;

    for (i = 0; i < 50; i++)
    {
        //hw = GetRand(0 , 50);
        // lw= GetRand(0 , 50);
        WATERLEVEL= GetRand(0 , 50);
        METHANLEVEL= GetRand(0 , 50);
        PUMP_STATE = rand() % 2;
        //int high_water=30;
        // int low_water=5;
        // int crtical_methan=10;
```

```
// break_line("=====");
assertion("WATER_LEVEL",WATER_LEVEL);
text_out2("WATER LEVEL is ",WATER_LEVEL);
assertion("METHAN_LEVEL",METHAN_LEVEL);
text_out2("METHAN LEVEL is ",METHAN_LEVEL);
assertion("PUMP_STATE",PUMP_STATE);
text_out2("PUMP statues is ",PUMP_STATE);
break_line("=====");
// assertion("waterHigh",hw);
}
```

### 7.2.5 Simulink model

As discussed before the mine pump model is implemented by use of a Simulink model, as it has its own certain advantages. It has mathematical and interface blocks to implement the mine environ-ment and case study easily, it provides in form of S-blocks and C-Mex executables to communicate with other processes such as the AnaTempura environment which is running the model checking and using the assertion method can interact with the Simulink model.

The main three inputs to the system and interconnections are as follows:

- The rate of water flow is governed by a first differential of the difference of the pump discharge rate and a water leak constant (0.1); this will simulate a right behaviour of water accumulating to the discharge rate. Once the pump is off this means the water leakage constant will push the water levels high as because of the accumulating integral 1/s block on the Simulink model.
- Then the water level is compared with a relational block to monitor the high and the low level mark which are defined by the two constant blocks.



- The Level of methane in this case is simulated by a sinusoidal function to provide a good time line check to verify the model behavior. This will keep the model in check with pump on conditions as methane levels are fluctuating and accumulating may need to trigger off the pumps.

The figure below show the simulink model block of the mine pump.

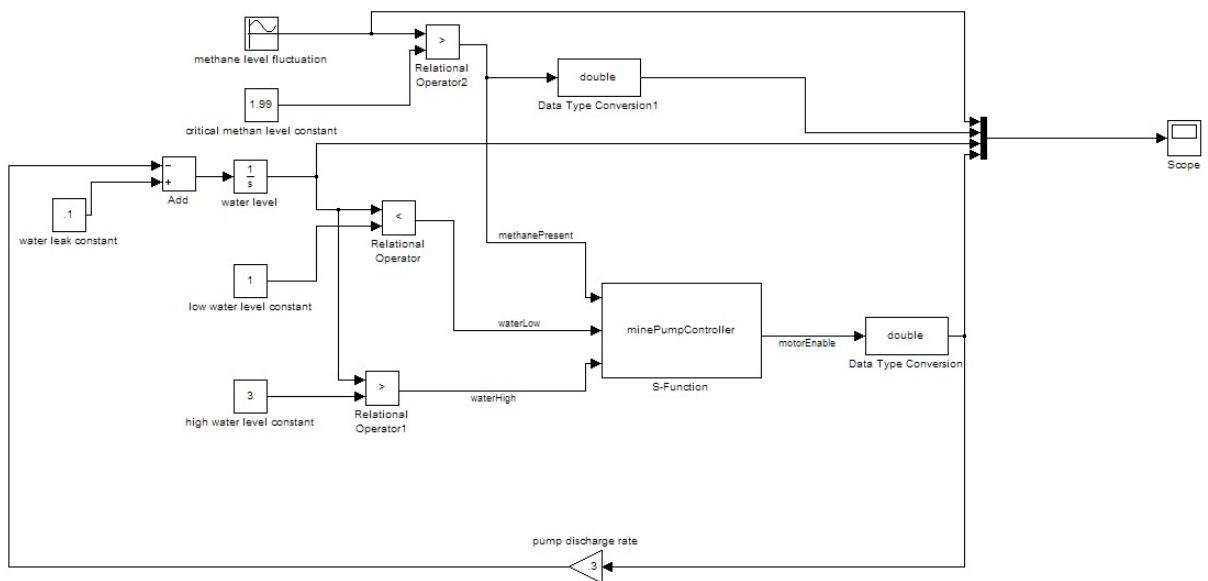


Figure 7.5: The Simulink Block

### 7.2.6 Case study results

The major requirement is to verify a hybrid system (mine pump controller) model in run time. The verification consists of both functional and temporal. For this purpose ITL has been used to verify the system. Before this work ITL was only able to verify the behaviour of the discrete systems and this is by articulating the system at discrete interval. Since for the hybrid system, to verify the continuous element, there is a requirement that the ITL shall be able to execute real-time transaction between the modelled system and the verification system (AnaTempura in this case). A new component SPITL has been introduced which uses the concept of spline to model a

continuous system. The major challenge in the mine pump system is to monitor the water and methane levels in real-time and verify that the system behaviour meets the operating specifications. Failures in the model behaviour must be detected in real-time constraint and then reported, with evasive actions to evacuate the mine in case of the system anomaly.

After running the mine pump on both Anatempura and mmatlab simulink we got results that approved the underling formalism and technique used in this study are successfully worked. The figure below shows the scope plot of the simulink model it can be seen on the graph that the water and methane level satisfied the case study requirements.

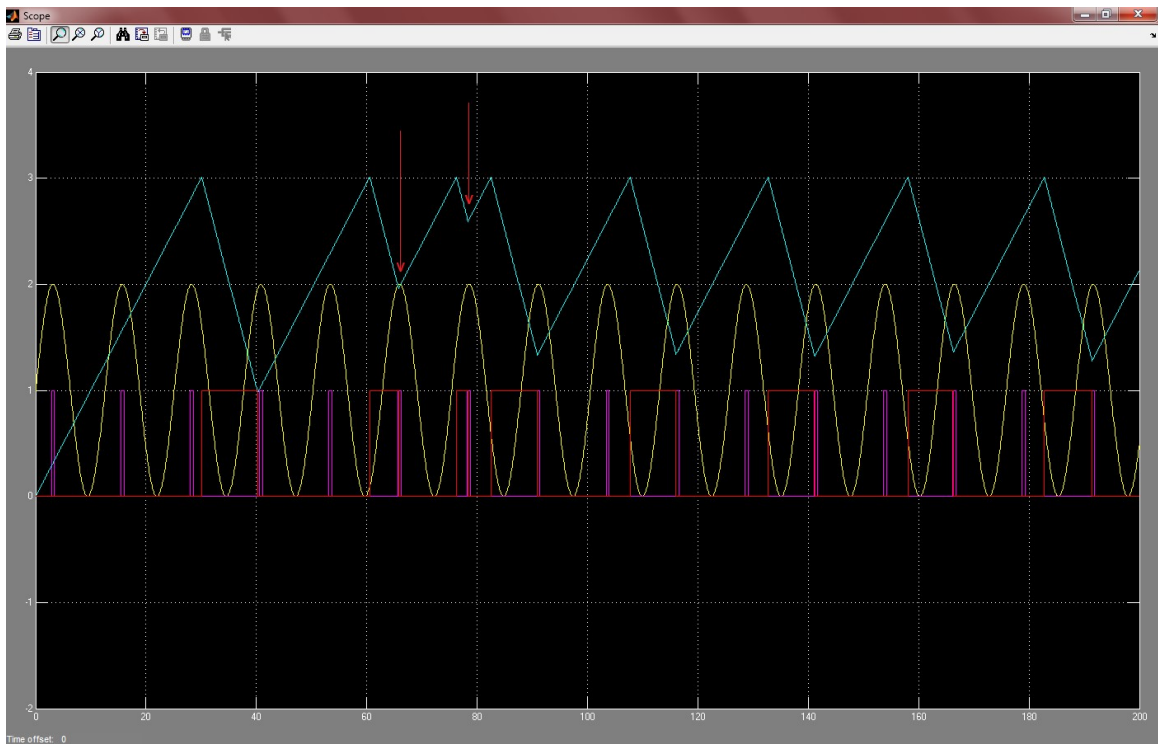


Figure 7.6: the Simulink model scope plot

- The yellow line is the methane level fluctuating (modelled as a sine wave) between 0 and 2
- The purple line is the methane sensor output which is 1 (True) if the methane

level is above 1.99 and 0 (False) if methane level is below 1.99

- The cyan line is the accumulated water level, this is modelled that the water leak as constant and the water level as the integration of (leaked water - water pumped out)
- The red line is the pump motor enable signal from the controller; when the motor is enabled (red line=1) total water level decrease, as the pump discharge water at a constant rate which is faster than water leaking

Therefore, as can be seen from the AnaTempura output results shown the figure below that the assertion points collected the data from the simulink and checked them against the AnaTempura properties test cases and its satisfied as well.

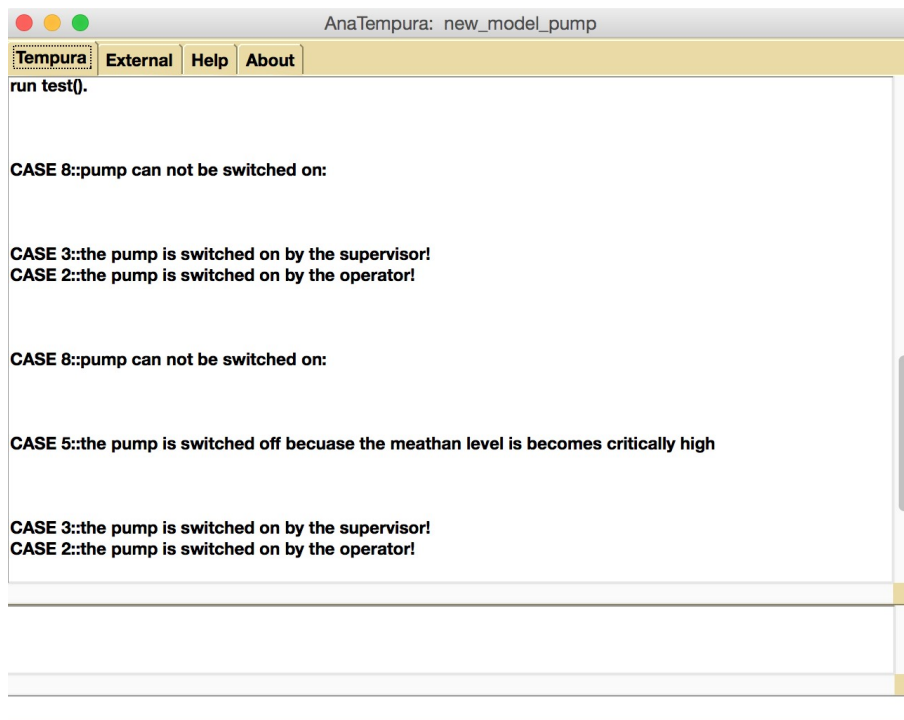


Figure 7.7: Mine pump test cases results in AnaTempura

### 7.3 Summary

In summary, hybrid systems are the focus of this research and deals with the verification of such a system using the ITL syntax and the use of its executable AnaTempura. This chapter dealt with a case study on a hybrid system i.e. a min pump control system. The logical architecture of the mine system is explained and all its functional and timing requirements are provided. These requirements are then translated in terms of ITL syntax.

The validation facility provided by the ITL is used to check a mine pump model being simulated using the Matlab Simulink model. An s function based in the Simulink model is used to communicate using an inter-process FIFO to the Ana tempura executable. A C executable is used to initiate the Matlab model and provide variables to the Simulink model and the AnaTempura executable environment using the assertion method.

Eight different cases are generated based on the different functional requirements and the behaviour of the model is validated using the status of the Water Levels and the Methane level. The Output of the Simulink model and the states are validated, both the expected output and the actual model output is displayed to show the working of the model and validates its behaviour.

# Chapter 8

## Conclusion and Future Work

### *Objectives:*

---

- Summary of Research.
  - Contributions.
  - Success Criteria.
  - Limitations.
  - Future work.
-

## 8.1 Introduction

The goal of this section is to describe how the research aims and objectives listed in Section 1.3 are achieved. To reiterate, the main aim of the current research is to develop a framework that can be used to verify and simulate a computer system's behaviour in terms of safety and liveness properties, using executable subset of Interval Temporal Logic (ITL) and its extension for the development of a hybrid system termed Spline Interval Temporal Logic (SPITL). This entails the use of Tempura with subsequent integration with AnaTempura and Matlab with the view to verify such a hybrid system model done within Simulink. The intended outcome is to improve the interpreter Tempura by merging multiple assertion points thereby making them to receive the points. Against this backdrop, the realization of the research aim could be said to have been achieved due to a number of research procedures and modelling that have been carried out as discussed in the section that follows.

## 8.2 Summary of Thesis

This thesis describes a study of the problems concerning the handling evolutionary verification of hybrid systems. A systematic research approach has been optimized to engineer hybrid systems and a tool has been developed and implemented based on this approach. The research commenced by addressing the overall process of hybrid systems engineering. Hybrid systems nowadays plays an important role within the broad area of software engineering. Enormous problems of engineering hybrid systems requires solutions, especially as it pertains to problems associated with the handling of verification of hybrid systems.

The current work also shows that there exists little systematic research on en-

engineering hybrid systems. It is well-known that the use of formal methods is fundamental for ensuring the correctness of hybrid systems. However, gap still exist between formal development and run-time evaluation. Therefore, an approach to engineer the evolving hybrid systems in a systematic fashion is addressed in this thesis.

The approach proposed and developed in this thesis is used to tackle the verification of hybrid systems rapidly, efficiently, and above all correctly. The central aim is to develop an integrated framework to deal with verification of hybrid systems. This framework integrates conventional approaches and formal technologies for engineering hybrid systems.

The basic components verification of hybrid systems have been identified and defined in this thesis, providing technical basis for a repeatable, well defined, and managed development process. It first addresses a general architectural methodology of handling verification of hybrid systems. This involves crossing levels of abstraction of time-critical systems, from specification in ITL and its extension in SPITL to source code in c-mex file that enable AnaTempura interfere with Matlab and its modeling tool, Simulink, using s-function routine as pipe.

Hybrid systems behaviours of interest can be analysed and validated in any stage of evolutionary development. The validation and analysis are performed within a single logical framework. The assertion points technique was adopted in the current work to generate run-time data, which fully reflects run-time behaviours of the time-critical systems. The run-time data were then captured and used to validate behaviours of interest with respect to the formal specification of the system. Errors are reported during the system run, i.e., the run-time analysis does not only report an error but also indicate the location of the error.

A set of extendable compositional rules have been adopted as the main guideline for a repeatable and well-managed approach to handle verification of hybrid systems.

A prototype tool has been developed to support the proposed approach. The tool is also used to implement the developed guidelines for guided evolution. A case study was used for experiments with the approach and the prototype system to demonstrate the success and effectiveness of the proposed approach.

### 8.3 Research Question revisited

**How can Interval Temporal logic be extended in order to specify hybrid systems, which integrates both discrete and continuous systems. In order to provide properties that capture the dynamic behaviour of hybrid systems and how can these properties be formally verified at runtime and how can this verification can be inserted in to the hybrid system model in matlab simulink?**

We propose to address the overall research question, a set of research questions that tackle each of the underlying issues.

**RQ 1. What is the appropriate formalism technique that is required for the specification and verification of hybrid systems?**

In order to gain an appreciable knowledge of the hybrid systems under consideration and its associated behaviour/ properties, temporal logics was the choice of formalism upon which the current research draws knowledge from. Our choice was justified by the wide applicability of temporal logic to hybrid systems as shown in extant literature. Amongst various flavours of temporal logics, ITL was selected given its numerous advantages (see details in Chapter three) for the various advantages (Chapter 3) especially its compositional attributes which provides the necessary resources and tool support for runtime verification.



**RQ 2. What are the kind of properties of a hybrid system behaviour that such a formalism can express?**

SPITL inherited its original logic based on ITL. Therefore, safety, liveness and timing properties can be powerfully expressed in SPITL. In thesis evolution (see Chapter 7) such properties were specified in a case study with the view to study and understand those properties.

**RQ 3. Does the formalism have adequate tool support in order to simulate and verify hybrid systems?**

In this thesis it has been established through a case study (see Chapters 6 and 7) that by linking AnaTempura with Matlab/Simulink, hybrid systems can be simulated and verified in an efficient manner.

**RQ 4. How can we describe the behaviour of hybrid systems using Interval Temporal Logic?**

The novelty of the current work lies in the extension of ITL into what is now known as Spline Interval Temporal logic (SPITL), in which not only discrete time behaviour can be expressed, but can also consider the continuous time behaviour over time in form of spline.

**RQ 5. How can we characterise the whole time interval instate of characterising fixed points on the interval?**

In SPITL, a behaviour is a sequence of phases (i.e. states have duration). Furthermore, a phase replaces a sequences of discrete states with a continuous behaviour represented by a spline.

**RQ 6. Can we have new operators in ITL that can deal with states durations?**

We have introduced a semantic model where phases have duration and within a phase (chapter 4).

**RQ 7. Can the proposed extension of ITL be used to reason about hybrid systems?**

It has been established that SPITL can deal with both discrete and continuous systems. Therefore, SPITL has been effectively used to gain some level of reasoning and understanding about hybrid system.

**RQ 8. How do we verify at runtime the behaviour of hybrid system under investigation using our framework?**

There is a growing cognizance that most specification and verification methods are beginning to attain their limits. Model-checking is limited to checking systems of finite size and deductive methods and can handle only systems whose complexity are minimal due to the heavy user interaction required [146]. In contrast to formal verification, practical verification techniques provide a mechanism to verify only properties of interest. In the current work, a framework based on runtime verification technique, using a Tempura interpreter called, AnaTempura is proposed. It was established that proof obligations can be encoded in tempura and then verified against the hybrid system behaviour. The verification can be performed through the injection of assertion points into source code (Chapter 5).

## 8.4 Criteria for Success and Analysis

### 8.4.1 Extended ITL formalism to reason about hybrid systems

Although the extended propositional ITL is based on the original ITL, the semantics of SPITL is much different from ITL. First, the extension to include continuous time behaviour as sequence of phases that integrates both discrete and continuous time

logic.

Second, the extension uses Spline form to model the logic formalism that can make the logic dynamic and use first and second derivative to make ITL more expressive.

### **8.4.2 Extended AnaTempura**

In the thesis, we extended Tempura in several ways. However, the input, output statements, data types declaration statements and pointers are excluded given that there is currently no straight forward way to include these statements in Tempura under the SPITL notation. Accordingly, further research is therefore required to solve these problems.

In this thesis, the temporal semantics of programs within the extended Tempura is investigated under the model theory. Operational and axiomatic semantics of programs still requires further research.

Another very active research field is real time programming. Currently, the extended Tempura is concerned with a sequence of states without absolute time. We could find a way to extend Tempura to use time explicitly so that hybrid systems can be handled by Tempura.

The Tool AnaTempura is designed to support the step-by-step methodology of handling verification of hybrid systems. This tool helps engineers in handling verification of hybrid systems in a comprehensive way. AnaTempura helps the user by performing its functions in an intelligent way. AnaTempura automatically monitors hybrid systems execution and analyses the system's run-time behaviours.

AnaTempura successfully linked with Matlab techniques. Therefore, the tool has become more effective and powerful as well as more friendly user interface. Both AnaTempura and MATLAB are helpful in the analysis of the behaviours of the system and reveal the evolutionary development process of the system. AnaTempura

considers possible error cases comprehensively. It is tolerant to many user errors. The tool checks for the errors, corrects the errors whenever possible, and gives relevant prompt information.

## 8.5 Future Directions

Based on the discussions in former sections, we concluded that the approach has novel ideas and is useful in handling verification of hybrid systems. The resulting tool scales up the approach. In addition, our approach can be easily adapted with hybrid system model tools like simulink or modelica. However, SPITL need to be studied to guarantee a complete proof refinement.

Also, Tempura need to be studied extensively as part of further research with the view to solve some of the timing problems and generate a complete semantic to enhance the AnaTempura to work effectively with external hybrid systems model tools.

# Chapter 9

## Appendix A

This appendix is present Matlab Simulink code. As well as the Matlab Engine Code.

### 9.1 Mine Pump Controller wrapper

```
#if defined(MATLAB_MEX_FILE)
#include "tmwtypes.h"
#include "simstruc_types.h"
#else
#include "rtwtypes.h"
#endif

/* %%-SFUNWIZ_wrapper_includes
_Changes_BEGIN —
EDIT HERE TO _END */
#include <math.h>
#include "assertion.h"
/* %%-SFUNWIZ_wrapper_
```

```
includes_Changes_END ——  
    EDIT HERE TO _BEGIN */  
#define u_width 1  
#define y_width 1  
/*  
    * Create external references here.  
    *  
    */  
/* %%%-SFUNWIZ-wrapper_  
externs_Changes_BEGIN —— EDIT HERE TO _END */  
  
/* %%%-SFUNWIZ-wrapper  
_externs_Changes_END —— EDIT HERE TO _BEGIN */  
  
/*  
    * Output functions  
    *  
    */  
void minePumpController_Outputs_wrappe  
r(const boolean_T *MethanePresent ,  
    const boolean_T *WaterLow ,  
    const boolean_T *WaterHigh ,  
    boolean_T *MotorEnable ,  
    const real_T *XD)  
{  
/* %%%-SFUNWIZ-wrapper_Outputs  
_Changes_BEGIN —— EDIT HERE TO _END */
```

```
MotorEnable[0] = XD[0]; //assertion("MotorEnable",XD[0]==0?0:1);
/* %%%-SFUNWIZ_wrapper_Outputs
_Changes_END — EDIT HERE TO _BEGIN */
}

/*
 * Updates function
 *
 */
void minePumpController_
Update_wrapper(const boolean_T
 *MethanePresent ,
    const boolean_T *WaterLow ,
    const boolean_T *WaterHigh ,
    const boolean_T *MotorEnable ,
    real_T *XD)
{
/* %%%-SFUNWIZ_wrapper_Update
_Changes_BEGIN — EDIT HERE TO _END */
    //text_out("MethanePresent");

assertion("MethanePresent" ,
MethanePresent[0]?1:0);
assertion("WaterLow" ,WaterLow[0]?1:0);
assertion("WaterHigh" ,WaterHigh[0]?1:0);
if(MethanePresent[0]){
    XD[0] = false;  assertion("XD",0);
```





```
#define IN_0_BUS_NAME
#define IN_0_DIMS          1-D
#define INPUT_0_FEEDTHROUGH 1
#define IN_0_ISSIGNED      0
#define IN_0_WORDLENGTH    8
#define IN_0_FIXPOINTSCALING 1
#define IN_0_FRACTIONLENGTH 9
#define IN_0_BIAS          0
#define IN_0_SLOPE         0.125
/* Input Port 1 */
#define IN_PORT_1_NAME     WaterLow
#define INPUT_1_WIDTH      1
#define INPUT_DIMS_1_COL   1
#define INPUT_1_DTYPE      boolean_T
#define INPUT_1_COMPLEX    COMPLEX_NO
#define IN_1_FRAME_BASED   FRAME_NO
#define IN_1_BUS_BASED     0
#define IN_1_BUS_NAME
#define IN_1_DIMS          1-D
#define INPUT_1_FEEDTHROUGH 1
#define IN_1_ISSIGNED      0
#define IN_1_WORDLENGTH    8
#define IN_1_FIXPOINTSCALING 1
#define IN_1_FRACTIONLENGTH 9
#define IN_1_BIAS          0
#define IN_1_SLOPE         0.125
/* Input Port 2 */
```

```
#define IN_PORT_2_NAME      WaterHigh
#define INPUT_2_WIDTH      1
#define INPUT_DIMS_2_COL   1
#define INPUT_2_DTYPE      boolean_T
#define INPUT_2_COMPLEX    COMPLEX_NO
#define IN_2_FRAME_BASED   FRAME_NO
#define IN_2_BUS_BASED     0
#define IN_2_BUS_NAME
#define IN_2_DIMS          1-D
#define INPUT_2_FEEDTHROUGH 1
#define IN_2_ISSIGNED      0
#define IN_2_WORDLENGTH    8
#define IN_2_FIXPOINTSCALING 1
#define IN_2_FRACTIONLENGTH 9
#define IN_2_BIAS          0
#define IN_2_SLOPE         0.125

#define NUMOUTPUTS        1
/* Output Port 0 */
#define OUT_PORT_0_NAME    MotorEnable
#define OUTPUT_0_WIDTH    1
#define OUTPUT_DIMS_0_COL 1
#define OUTPUT_0_DTYPE    boolean_T
#define OUTPUT_0_COMPLEX  COMPLEX_NO
#define OUT_0_FRAME_BASED  FRAME_NO
#define OUT_0_BUS_BASED   0
#define OUT_0_BUS_NAME
```





```
*/
static
void mdlInitializeSizes (SimStruct *S)
{

DECL_AND_INIT_DIMSINFO
(inputDimsInfo);
DECL_AND_INIT_DIMSINFO
(outputDimsInfo);
ssSetNumSFcnParams(S, NPARAMS);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter
                mismatch will be
                reported by Simulink */
    }
ssSetNumContStates(
S, NUMCONTSTATES);
ssSetNumDiscStates
(S, NUMDISCSTATES);

if (!ssSetNumInputPorts
(S, NUMINPUTS)) return;
/*Input Port 0 */
ssSetInputPortWidth
(S, 0, INPUT_0_WIDTH); /* */
ssSetInputPortDataType
(S, 0, SS_BOOLEAN);
```

```
ssSetInputPortComplexSignal(S, 0, INPUT_0_COMPLEX);
ssSetInputPortDirectFeedThrough(S, 0, INPUT_0_FEEDTHROUGH);
ssSetInputPortRequiredContiguous
(S, 0, 1); /*direct input signal access*/

/*Input Port 1 */
ssSetInputPortWidth
(S, 1, INPUT_1_WIDTH); /* */
ssSetInputPortDataType
(S, 1, SS_BOOLEAN);
ssSetInputPortComplexSignal(S, 1, INPUT_1_COMPLEX);
ssSetInputPortDirectFeedThrough(S, 1, INPUT_1_FEEDTHROUGH);
ssSetInputPortRequiredContiguous
(S, 1, 1); /*direct input signal access*/

/*Input Port 2 */
ssSetInputPortWidth
(S, 2, INPUT_2_WIDTH);
/* */
ssSetInputPortDataType
(S, 2, SS_BOOLEAN);
ssSetInputPortComplexSignal(S, 2, INPUT_2_COMPLEX);
ssSetInputPortDirectFeedThrough(S, 2, INPUT_2_FEEDTHROUGH);
ssSetInputPortRequiredContiguous
(S, 2, 1);
/*direct input signal access*/
```

```
if (!ssSetNumOutputPorts
(S, NUMOUTPUTS)) return;
ssSetOutputPortWidth
(S, 0, OUTPUT_0_WIDTH);
ssSetOutputPortDataType
(S, 0, SS_BOOLEAN);
ssSetOutputPortComplexSignal(S, 0, OUTPUT_0_COMPLEX);
ssSetNumSampleTimes(S, 1);
ssSetNumRWork(S, 0);
ssSetNumIWork(S, 0);
ssSetNumPWork(S, 0);
ssSetNumModes(S, 0);
ssSetNumNonsampledZCs(S, 0);

/* Take care when
specifying exception
free code –
see sfuntmpl_doc.c */
ssSetOptions
(S,
(SS_OPTION_EXCEPTION_
FREE_CODE |
SS_OPTION_USE_TLC_WITH_
ACCELERATOR
SS_OPTION_WORKS_
WITH_CODE_REUSE));
```

```
}

# define
MDLSETINPUTPORT
_FRAME_DATA
static void
mdlSetInputPortFrameData
(SimStruct *S,
    int_T port,
    Frame_T frameData)
{
    ssSetInputPortFrameData
    (S, port, frameData);
}

/* Function:
mdlInitializeSampleTimes =====
=====
=====
=====
* Abstract:
*   Specify
the sample time.
*/
static void
mdlInitializeSampleTimes
(SimStruct *S)
{
```



```
    ssSetSampleTime
    (S, 0, SAMPLE_TIME_0);
    ssSetOffsetTime

    (S, 0, 0.0);
}
#define
MDL_INITIALIZE
_CONDITIONS
/* Function:
    mdlInitialize
    Conditions =====
    =====
    =====
    * Abstract:
    *   Initialize
    *   the states
    */
static
void
    mdlInitializeConditions
(SimStruct *S)
{
    real_T *XD
    = ssGetRealDiscStates
    (S);
```

```
        XD[0] = 0;

    }

#define
MDL_SET_I
NPUT_PORT_DATA_TYPE

static
void mdlSetInput
PortDataType
(SimStruct *S,
 int port, DTypeId dType)
{
    ssSetInputPortDataType
    ( S, 0, dType);
}

#define
MDL_SET_OUTPUT_PORT_DATA_TYPE

static
void
mdlSetOutputPortDataType
(SimStruct *S, int port,
 DTypeId dType)
{
    ssSetOutputPortDataType
    (S, 0, dType);
```

```
}

#define
MDL_SET_
DEFAULT_PORT_DATA_TYPES
static
void
mdlSetDefaultPortDataTypes
(SimStruct *S)
{
    ssSetInputPortDataType
    ( S, 0, SS_DOUBLE);
    ssSetOutputPortDataType
    (S, 0, SS_DOUBLE);
}
/* Function: mdlOutputs =====
=====
=====

*
*/
static void mdlOutputs
(SimStruct *S, int_T tid)
{
    const boolean_T
    *MethanePresent
    = (const boolean_T*)
    ssGetInputPortSignal(S,0);
```

```
const boolean_T
  *WaterLow = (const boolean_T*) ssGetInputPortSignal(S,1);
const boolean_T *WaterHigh
  = (const boolean_T*)
  ssGetInputPortSignal(S,2);
boolean_T
  *MotorEnable = (boolean_T *)ssGetOutputPortRealSignal(S,0);
const real_T *XD = ssGetDiscStates(S);

minePump
  Controller_Outputs_wrapper
  (MethanePresent, WaterLow,
   WaterHigh, MotorEnable, XD);
}
#define MDLUPDATE /*
Change to #undef to remove function */
/* Function: mdlUpdate =====
=====
=====

* Abstract:
* This function is called
once for every major
integration time step.
* Discrete states ar
e typically updated here, but this
function is useful
* for performing any tasks
```

```
    that should only take place once per
*    integration step.
*/
static void mdlUpdate(
SimStruct *S, int_T tid)
{
    real_T      *XD
    = ssGetDiscStates(S);
    const boolean_T
*MethanePresent = (const boolean_T*) ssGetInputPortSignal(S,0);
    const boolean_T *WaterLow
    = (const boolean_T*)
    ssGetInputPortSignal(S,1);
    const boolean_T *WaterHigh
    = (const boolean_T*)
    ssGetInputPortSignal(S,2);
    boolean_T      *MotorEnable

= (boolean_T *)ssGetOutputPortRealSignal
(S,0);

    minePumpController_Update_wrapper
(MethanePresent , WaterLow, WaterHigh ,
    MotorEnable , XD);
}
```

```
/* Function: mdlTerminate =====  
=====
```

*\* Abstract:*

*\* In this function, you should perform any actions that are necessary at the termination of a simulation.*

*For example, if memory was allocated in mdlStart, this is the place to free it.*

```
*/  
static void mdlTerminate(SimStruct *S)  
{  
}
```

*#ifndef MATLAB\_MEX\_FILE*

*/\* Is this file being compiled as a MEX-file?*

*\*/*

```
#include "simulink.c"  
  
/* MEX-file interface mechanism  
  
*/
```

*#else*

```
#include "cg_sfund.h"
```

```
    /* Code generation
       registration function */
#endif
```

## 9.3 Matlab Engine code

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define MAXBUF 1024
#include <string.h>
#include "engine.h"
#include <limits.h>
#define BUFSIZE 256
#define FIFO_NAME "my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main(int argc, char *argv[])
{
    Engine *eng;
    //int res;
    char buffer[BUFFER_SIZE + 1];

    printf("Starting matlab engine\n");
```

```
    if (!(eng = engOpen(NULL)) ) {
        printf("Can't start MATLAB engine\n");
        return 0;
    }

    printf("runing simulink\n");

    engEvalString(eng, "mex -c
minePumpController.c minePumpController_wrapper.c");
    engEvalString(eng, "mex
minePumpController.c minePumpController_wrapper.c");
    engEvalString(eng, "
sim('minePumpModel')");
    engEvalString(eng, "
open_system('minePumpModel/Scope')");

    FILE *fd;
    char line[BUFSIZE];
    fd = fopen("myfifo","r");
    for( int i =20; i>1 ; i--)
    {
        while (fgets(line,256, fd)!=NULL) printf("%s",line);

    }
    fclose(fd);

    printf("Closing matlab engine\n");
```



```
    engClose(eng);
    printf(" Matlab AnaTempura
interface is closed\n");
    return(0);
//  exit(EXIT_SUCCESS);
}
```

## 9.4 Fifo Pipe

```
/* fifo2.c */
/* This file will write
to the fifo a string.*/ #include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <limits.h>
#define FIFO_NAME "my_fifo"
#define BUFFER_SIZE PIPE_BUF
int main()
{
int res;
char buffer[BUFFER_SIZE + 1];
if (access(FIFO_NAME, F_OK) == -1) {

res = mkfifo(FIFO_NAME, 0777); if (res != 0) {
```

```
/* the fifo name */
/* check if fifo already if
   not then, create the fifo*/
fprintf(stderr ,
"Could not create fifo %s\n", FIFO_NAME);
exit(EXIT_FAILURE); }
}
printf(" Process %d opening FIFO\n",
getpid()); res = open(FIFO_NAME, O_WRONLY);
sprintf(buffer , "hello ");
write(res , buffer ,BUFFER_SIZE);
printf(" Process %d result %d\n",
getpid() , res); sleep(5);
if (res != -1) (void)close(res);
printf(" Process %d finished\n", getpid());
exit(EXIT_SUCCESS);
}
```

# Chapter 10

## Appendix B

This appendix is to list the tempura code.

### 10.1 Assertion points

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define FIFO_NAME "my_fifo"
#define BUFFER_SIZE PIPE_BUF
```

```
int myclock()
{

    struct timeval stop, start;
    gettimeofday(&start, NULL);

    gettimeofday(&stop, NULL);
    return stop.tv_usec - start.tv_usec;
}

bool initialized = false;
char * myfifo = "/tmp/fooPipe";
void initialize(void)
{
    initialized=true;

    FILE *fd = fopen(myfifo, "w");

    fclose(fd);
}

void text_out(char *txt)
{
    fprintf(myfifo, "%d:: %s\n", myclock(), txt); fflush(stdout);
}
```

```
void scan_sensor(char *txt, int *temp)
{
    text_out(txt);
    scanf("%d",temp);
}
```

```
extern void assertion(char *aname,
int val)
{
    if(!initialized) initialize();
    fprintf(myfifo,"!PROG: assert
%s:%d:%d:!\n",aname, val, myclock()); fflush(stdout);
}
```

```
void assertion1(char *aname, int val)
{
    if(!initialized) initialize();
    fprintf(myfifo,"!PROG: assert
%s:%d:%d:%d:%d:!\n",aname, val, 1,2,myclock()); fflush(stdout);
}
```

## 10.2 Tempura Code

load "conversion".

load "exprog".

```
load "tcl".
/* prog enginecode 0 */

set print_states = true.

define get_var(X,Y) =
{
    exists T : {
        get2(T) and /
        *output(etime(X)) and*/
        if avar(T)=X then
            {Y=string(aval(T))}
    }
}.

/* run */ define test() = {
exists MotorEnable, MethanePresent,
MaterLow, WaterHigh, XD, counter : {

for counter <20 do {skip and
get_var("MotorEnable",
MotorEnable) and
output(MotorEnable) and
```

```
    get_var("MethanePresent",
MethanePresent)
    and output(MethanePresent) and
get_var("WaterLow",WaterLow)
    and output(WaterLow) and
get_var("WaterHigh",WaterHigh) and output(WaterHigh) and
get_var("XD",XD) and output(XD)}

}
}.
```

# Bibliography

- [1] Verification of digital and hybrid systems. 2(Lics 96):278–292, 2000.
- [2] J. A. Abraham. Introduction to Temporal Logics. *Computer*, pages 1–10, 2010.
- [3] P. E. Abraham. Modeling and Analysis of Hybrid Systems Lecture Notes. 2012.
- [4] L. D. Alfaró and Z. Manna. Verification in Continuous Time by Discrete.
- [5] T. M. Alghamdi. Policy-based Runtime Tracking for E-learning Environments PhD Thesis. 1988.
- [6] J. F. Allen and G. Ferguson. Actions and Events in Interval Temporal Logic. *Journal of Logic and Computation*, 1994.
- [7] A. Z. Almutairi. Context-Aware and Adaptive Usage Control Model. (September), 2013.
- [8] S. Also and S. Also. 9 Temporal Expressions. pages 321–360.
- [9] R. Alur. Formal verification of hybrid systems. *Proceedings of the ninth ACM international conference on Embedded software - EMSOFT '11*, (January):273, 2011.



- [10] R. Alur and D. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
- [11] R. Alur, J. Esposito, M. Kim, V. Kumar, and I. Lee. Formal modeling and analysis of hybrid systems : A case study in multi-robot coordination. *Mechanical Engineering*, 1998.
- [12] R. Alur, T. A. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.
- [13] J. Anderson, R. N. M. Watson, D. Chisnall, K. Gudka, I. Marinos, and B. Davis. TESLA: Temporally Enhanced Security Logic Assertions. *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, pages 1–14, 2014.
- [14] K. Androutsopoulos. Specification and verification of reactive systems with rds. (June 2004), 2004.
- [15] P. J. Antsaklis and N. Dame. Hybrid Systems : Review and Recent Progress. 2003.
- [16] P. J. Antsaklis, X. Koutsoukos, and J. Zaytoon. On hybrid control of complex systems: A survey. *IEEE Trans. Autom. Control*, vol:32no9—10pp1023—1045, 1998.
- [17] S. Arun-Kumar. Introduction to Logic for Computer Science. page 97, 2002.
- [18] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying industrial hybrid systems with MathSAT. *Electronic Notes in Theoretical Computer Science*, 119(2):17–32, 2005.
- [19] F. Bacchus, J. Tenenber, and J. A. Koomen. A Non-Red Temporal Logic  
1 Introduction 2 A Non-Red Temporal Logic. (Focs 1989):1–19.

- [20] R. J. R. Back. Atomicity Refinement in a Refinement Calculus Framework. *Computer*, pages 1–43, 1993.
- [21] H. Barringer, M. Fisher, D. Gabbay, G. Gough, and R. Owens. MetateM : A Framework for Programming in Temporal Logic. *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, 3096:94–129, 1990.
- [22] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–64, 2011.
- [23] A. A. Bayazit and S. Malik. Complementary use of runtime validation and model checking. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2005:1049–1056, 2005.
- [24] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Computing Surveys*, 32(1):12–42, 2000.
- [25] J. A. Bergstra and C. A. Middelburg. Process algebra for hybrid systems. *Theoretical Computer Science*, 335(2-3):215–280, 2005.
- [26] J. A. Bergstra and C. A. Middelburg. Continuity controlled hybrid automata. *Journal of Logic and Algebraic Programming*, 68(1-2):5–53, 2006.
- [27] S. Bisanz, U. Hannemann, and J. Peleska. Executable Semantics for Hybrid Systems - The Hybrid Low-Level Framework. *2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 64–67, 2008.
- [28] H. Bowman, H. Cameron, P. King, and S. Thompson. Mexitl: Multimedia in executable interval temporal logic. *Formal Methods in System Design*, 22(1):5–38, 2003.

- [29] H. Brandl, M. Weiglhofer, and B. K. Aichernig. Automated conformance verification of hybrid systems. *Proceedings - International Conference on Quality Software*, pages 3–12, 2010.
- [30] M. S. Branicky and M. S. Branicky. *Studies in Hybrid Systems : Modeling , Analysis , and Control* by by. 1995.
- [31] M. S. Branicky and M. S. Branicky. *Studies in Hybrid Systems: Modeling, Analysis, and Control. Electrical Engineering*, 1995.
- [32] C. Brzoska. Programming in metric temporal logic. *Theoretical computer science*, 202(1-2):55–125, 1998.
- [33] T. Bultan. CMPSC 267 Class Notes Introduction to Temporal Logic and Model Checking Chapter 1 Introduction.
- [34] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Angiovanni-Vincentelli. Languages and Tools for Hybrid Systems Design. *Foundations and Trends® in Electronic Design Automation*, 1(1/2):1–193, 2006.
- [35] R. Carter. Verification of Liveness Properties on Hybrid Dynamical Systems. 2013.
- [36] A. Cau. Interval Temporal Logic A not so short introduction. 2009, 2009.
- [37] A. Cau. Interval Temporal Logic A not so short introduction Features of ITL Features of ITL ITL ’ s Influence ITL ’ s Influence Part I : Propositional Logic Part II : Propositional ITL. (2), 2009.
- [38] A. Cau, C. Czarnecki, and H. Zedan. Designing a provably correct robot control system using a ‘lean’ formal method. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1486:123–132, 1998.

- [39] A. Cau, H. Janicke, and B. Moszkowski. Verification and enforcement of access control policies. *Formal Methods in System Design*, 43(3):450–492, 2013.
- [40] A. Cau and B. Moszkowski. Interval Temporal Logic Proof Checker 1. 1997.
- [41] A. Cau, B. Moszkowski, and H. Zedan. Interval temporal logic. *URL: <http://www.cms.dmu.ac.uk/> . . .*, pages 1–27, 2006.
- [42] A. Cau and H. Zedan. Refining interval temporal logic specifications. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1231:79–94, 1997.
- [43] A. Cau, H. Zedan, N. Coleman, and B. Moszkowski. Using ITL and Tempura for large-scale specification and simulation. *Pdp*, 1996.
- [44] Y. F. Chen and Z. M. Liu. Integrating temporal logics. *Integrated Formal Methods, Proceedings*, 2999(Dc):402–420, 2004.
- [45] Z. Chen, A. Cau, H. Zedan, X. Liu, and H. Yang. A Refinement Calculus for the Development of Real-Time Systems. *Proceedings 1998 Asia Pacific Software Engineering Conference (Cat. No.98EX240)*, 1998.
- [46] S. Colin and L. Mariani. Runtime Verification. *Lecture Notes in Computer Science*, 55:525–555, 2005.
- [47] W. Damm, H. Dierks, S. Disch, W. Hagemann, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. *Science of Computer Programming*, 77(10-11):1122–1150, 2012.
- [48] J. M. Davoren and A. Nerode. Logics for hybrid systems. *Proceedings of the IEEE*, 88(7):985–1010, 2000.

- [49] C. De Boor, C. De Boor, C. De Boor, and C. De Boor. *A practical guide to splines*, volume 27. Springer-Verlag New York, 1978.
- [50] V. D. Dimitriadis. Modelling , Safety Verification and Design Continuous Discrete / Processing Systems of. (April), 1997.
- [51] a. Donzé. Trajectory-Based Verification and Controller Synthesis for Continuous and Hybrid Systems. *Analysis*, 2007.
- [52] J. Dorsey. Continuous and Discrete Control Systems. 85782895(M), 2002.
- [53] D. Dranidis, E. Ramollari, and D. Kourtesis. Run-time verification of behavioural conformance for conversational web services. *ECOWS'09 - 7th IEEE European Conference on Web Services*, pages 139–147, 2009.
- [54] Z. Duan and N. Zhang. A Complete Axiomatization of Propositional Projection Temporal Logic. *2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 271–278, 2008.
- [55] A. M. El-kustaban. Studying and Analysing Transactional Memory Using Interval Temporal Logic and AnaTempura. 1988.
- [56] K. D. Emanuele and G. Pace. Runtime Validation Using Interval Temporal Logic.
- [57] E. A. Emerson and J. Y. Halpern. “sometimes” and “not never” revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.
- [58] A. Estrin and M. Kaminski. The expressive power of Temporal Logic of Actions. *Journal of Logic and Computation*, 12(5):839–859, 2002.

- [59] G. E. Fainekos and G. J. Pappas. Robustness of temporal logic specifications for continuous time signals. *Theoretical Computer Science*, 410(42):4262–4291, 2009.
- [60] D. T. Fakultat and A. Cau. COMPOSITIONAL VERIFICATION AND SPECIFICATION OF REFINEMENT FOR REACTIVE SYSTEMS IN A DENSE TIME Dissertation Antonio Cau Zum Druck genehmigt ∴ (August), 1995.
- [61] C. D. Fensel and F. Fischer. Propositional Logic ©. pages 1–73, 2010.
- [62] M. Finger and D. Gabbay. Combining Temporal Logic Systems. *Notre Dame Journal of Formal Logic*, 37(2):204–232, 1996.
- [63] T. Finin. Introduction to Logic Programming and Prolog. *Most*.
- [64] M. Fisher. A Survey of Concurrent M ETATE M — The Language and its Applications.
- [65] M. Fisher. A Resolution Method for Temporal Logic. *Ijcai*, 91:99–104, 1991.
- [66] M. Fisher. An introduction to executable temporal logics. *Knowledge Engineering Review*, 11(01):43, 1996.
- [67] M. Fisher. Implementing temporal logics: Tools for execution and proof. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3900 LNAI:129–142, 2006.
- [68] M. Fisher. *An Introduction to Practical Formal Methods Using Temporal Logic*. 2011.
- [69] M. Fr and M. R. Hansen. A Robust Interpretation of Duration Calculus. *Symposium A Quarterly Journal In Modern Foreign Literatures*.

- [70] P. Fritzson and A. Pop. Meta-Programming and Language Modeling with. 2011.
- [71] P. Fulfillment. A Theory of Run-time Verification for Safety Critical Reactive Systems. (May), 2005.
- [72] C. Furia and M. Rossi. Integrating discrete-and continuous-time metric temporal logics through sampling. *Formal Modeling and Analysis of Timed Systems*, pages 215–229, 2006.
- [73] C. A. Furia and M. Rossi. A theory of sampling for continuous-time metric temporal logic. *ACM Transactions on Computational Logic*, 12(1):1–40, 2010.
- [74] A. Galton. Temporal Logic. 2004, 2003.
- [75] V. Goranko, A. Montanari, P. Committee, E. Franconi, and P. H. Groningen. Interval temporal logics. *Theoretical Computer Science*.
- [76] M. Gordon. From LCF to HOL: a short history. *Proof, Language, and Interaction*, pages 1–16, 2000.
- [77] D. P. Guelev. Probabilistic Interval Temporal Logic. (144), 1998.
- [78] R. Hale. Using Temporal Logic for Prototyping: The Design of a Lift Controller. *Temporal Logic in Specification*, 398:374–408, 1987.
- [79] R. Hale and B. Moszkowski. Parallel Programming in T e m p o r a l Logic.
- [80] S. Hanneke, W. Fu, and E. Xing. Discrete Temporal Models of Social Networks. *Electronic Journal of Statistics*, 4:585–605, 2010.
- [81] M. R. D. o. I. T. T. U. o. D. Hansen and Z. I. I. f. S. T. N. U. Chaochen. Duration Calculus : Logical Foundations. *Formal Aspects of Computing - Springer*, 9(3):283–330, 1997.

- [82] I. A. N. Hayes, R. Colvin, D. Hemer, and P. Strooper. A Refinement Calculus for Logic Programs arXiv : cs / 0202002v1 [ cs . SE ] 4 Feb 2002.
- [83] W. Heise. An efficient model checker for Duration Calculus. *Mathematical Modelling*, 2010.
- [84] T. a. Henzinger, P.-h. Ho, and H. Wong-toi. HYTECH: a model checker for hybrid systems. 1(1997):110–122, 2001.
- [85] T. a. Henzinger and P. W. Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221(1-2):369–392, 1999.
- [86] T. a. Henzinger and H. Wong-toi. from the HYTECH. *Analysis*, (December), 2001.
- [87] P. Hespanha. Discrete Event , Hybrid Systems. *Science*.
- [88] Y. Hirshfeld and A. Rabinovich. An expressive temporal logic for real time. *Mathematical Foundations of Computer . . .*, pages 492–504, 2006.
- [89] Y.-c. Ho. CS780 Discrete-State Models. 77(1):3–6, 1989.
- [90] B. E. Hons, M. Eng, and D. Heffernan. A Monitoring Approach to Facilitate Run-time Verification of Software in Deeply Embedded Systems Author. *Methods*, (March), 2010.
- [91] D. V. Hung. From Continuous Specification to Discrete Design. *Sensors And Actuators*, 2002.
- [92] D. V. Hung and Z. Chaochen. Probabilistic Duration Calculus for Continuous Time. *Formal Aspects of Computing*, 11:21–44, 1999.
- [93] D. V. Hung, H. Zedan, and A. Cau. A Formal Design Technique for Real-Time Embedded Systems Development using Duration Calculus.



- [94] M. R. A. Huth and M. D. Ryan. *Modelling and reasoning about systems*. 2000.
- [95] L. Imsland, P. Kittilsen, and T. S. Schei. Model-based optimizing control and estimation using Modelica models. *Modeling, Identification and Control*, 31(3):107–121, 2010.
- [96] A. Informatica. Parallel composition of assumption-commitment specifications Semantic analysis. 176:1–24, 1996.
- [97] H. Janicke. ITLTracer: Runtime Verification of Properties expressed in ITL. 356, 2010.
- [98] H. Janicke, F. Siewe, K. Jones, A. Cau, and H. Zedan. Analysis and Run-time Verification of Dynamic Security Policies. *Defence Applications of Multi-Agent Systems*, 3890:92–103, 2006.
- [99] S. Jiang and R. Kumar. Failure diagnosis of discrete event systems with linear-time temporal logic specifications. pages 1–26.
- [100] K. Johannisson. *Thesis for the Degree of Doctor of Philosophy Formal and Informal Software Specifications*. 2005.
- [101] A. A. Julius. *On Interconnection and Equivalence of Continuous and Discrete Systems A Behavioral Perspective*. 2005.
- [102] T. Kaye, S. Kholgade, J. Knutz, D. Lannoye, and J. Sartori. Applying Software Engineering Principles in Developing Safety-Critical Software Systems : A Class Project.
- [103] R. V. Kharche. MATLAB Automatic Differentiation using Source Transformation. (August 2011), 2012.
- [104] D. Kincaid and W. Cheney. Numerical Analysis, Brooks, 1991.

- [105] D. Kortenkamp, R. Simmons, T. Milam, and J. L. Fernández. A Suite of Tools for Debugging Distributed Autonomous Systems. *Formal Methods in System Design*, 24(2):157–188, 2004.
- [106] B. Krogh. Recent advances in discrete analysis and control of hybrid systems. *Sixth International Workshop on Discrete Event Systems, 2002. Proceedings.*, pages 2–5, 2002.
- [107] Y. Kwon and G. Agha. LTLC: Linear temporal logic for control. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4981 LNCS:316–329, 2008.
- [108] L. Lamport. What Good is Temporal Logic?, 1983.
- [109] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
- [110] J. A. Larsen, R. Wiśniewski, and J. D. Grunnet. *Combinatorial hybrid systems*. 2008.
- [111] E. a. Lee and H. Zheng. Operational semantics of hybrid systems. *Hybrid Systems: Computation and Control (HSCC), volume LNCS 3414*, pages 25–53, 2005.
- [112] M. Leucker. Teaching runtime verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7186 LNCS:34–48, 2012.
- [113] X. Li. Specification and Simulation of a Concurrent Real-time System.
- [114] F. Lin. Analysis and synthesis of discrete event systems using temporal logic. *Proceedings of the 1991 IEEE International Symposium on Intelligent Control, (August)*:140–145, 1991.

- [115] X. Liu, Z. Chen, H. Yang, H. Zedan, and W. Chu. A Design Framework for System Re-engineering. *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, pages 342–352, 1997.
- [116] G. Lowe and H. Zedan. Re nement of Complex Systems : A Case Study 1 Introduction 2 The Temporal Agent Model. 1995.
- [117] F. M. *An Introduction to Pratical formal Methods Using Temporal Logic*. 2011.
- [118] Y. Ma, Z. Duan, X. Wang, and X. Yang. An interpreter for framed tempura and its application. 2007.
- [119] M. Mäkelä. T-79 . 231 Parallel and Distributed Digital Systems Temporal Logic Temporal logic. 2003.
- [120] O. Maler, D. Nickovic, and A. Pnueli. Checking temporal properties of discrete, timed and continuous behaviors. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4800 LNCS:475–505, 2008.
- [121] Z. Manna and A. Pnueli. Verifying hybrid systems. *Lecture Notes In Computer Science*, 736:4–35, 1993.
- [122] N. Markey, I. Course, and C. Section. Expressiveness of temporal logics. *Computer*, 2006.
- [123] J. Melorose, R. Perroy, and S. Careas. No Title No Title. *Statewide Agricultural Land Use Baseline 2015*, 1(2001), 2015.
- [124] M. Mergency. a Framework for. 22(8):1–23, 2011.

- [125] C. A. Middelburg. Truth of Duration Calculus Formulae in Timed Frames. (82), 1996.
- [126] S. Mitra. TECHNOLOGY A Verification Framework for Hybrid Systems. (September), 2007.
- [127] A. Mok and D. Stuart. Simulation vs. verification: getting the best of both worlds. *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*, pages 12–22, 1996.
- [128] B. Moszkowski. Executing temporal logic programs. *Seminar on Concurrency*, (February 2000), 1986.
- [129] B. Moszkowski. A hierarchical analysis of propositional temporal logic based on intervals. *arXiv preprint cs/0601008*, 2:1–45, 2006.
- [130] B. Moszkowski. Using Temporal Logic to Analyse Temporal Logic: A Hierarchical Approach Based on Intervals. *Journal of Logic and Computation*, 17(2):333–409, 2007.
- [131] B. Moszkowski. for specification of concurrent systems. 2010.
- [132] B. Moszkowski. Interconnections between classes of sequentially compositional temporal formulas. *Information Processing Letters*, 113(9):350–353, 2013.
- [133] B. Moszkowski. Compositional reasoning using intervals and time reversal. *Annals of Mathematics and Artificial Intelligence*, 71(1-3):175–250, 2014.
- [134] B. Moszkowski, D. Guelev, and M. Leucker. Guest editors' preface to special issue on interval temporal logics. *Annals of Mathematics and Artificial Intelligence*, 71(1-3):1–9, 2014.

- [135] B. C. Moszkowski. An Automata-Theoretic Completeness Proof for Interval Temporal Logic ( Extended Abstract ). 1853(Icalp):223–234, 2000.
- [136] P. Naldurg, K. Sen, and P. Thati. A temporal logic based framework for intrusion detection. *Formal Techniques for Networked and ...*, 2004.
- [137] S. Nidhra and J. Dondeti. How to Write a Literature Review. 2(2):29–50, 2012.
- [138] M. Orgun and W. Ma. An Overview of Temporal and Modal Logic Programming. *Proceedings of the 1st International Conference on Temporal Logic - Lecture Notes in Artificial Intelligence*, pages 445–479, 1994.
- [139] G. Ossimitz and M. Mrozek. The basics of system dynamics: Discrete vs. continuous modelling of time. ... *the System Dynamics Society*], *System ...*, pages 1–8, 2008.
- [140] O. Özgün and Y. Barlas. Discrete vs . Continuous Simulation : When Does It Matter ? *27th International Conference of The System Dynamics Society*, (06):1–22, 2009.
- [141] D. K. Pace. Modeling and simulation verification and validation challenges. *Johns Hopkins APL Technical Digest (Applied Physics Laboratory)*, 25(2):163–172, 2004.
- [142] P. K. Pandya. Interval duration logic: Expressiveness and decidability. *Electronic Notes in Theoretical Computer Science*, 65(6):241–259, 2002.
- [143] H. Peter and M. Bernhard. Towards an Algebra of Hybrid Systems. pages 121–133, 2006.

- [144] Q. C. Pham. Analysis of discrete and hybrid stochastic systems by nonlinear contraction theory. *2008 10th International Conference on Control, Automation, Robotics and Vision, ICARCV 2008*, (December):1054–1059, 2008.
- [145] A. Pnueli, C. Science, L. Zuck, and N. Haven. In and Out of Temporal Logic. 1993.
- [146] A. Rabinovich. Automata over continuous time. *Theoretical Computer Science*, 300(1-3):331–363, 2003.
- [147] A. Rabinovich and B. A. Trakhtenbrot. From Finite Automata toward Hybrid Systems ( Extended Abstract ). (5).
- [148] A. Rao, A. Cau, and H. Zedan. Visualization of interval temporal logic. *Proc. 5th Joint Conference on Information Sciences*, pages 687–690, 2000.
- [149] S. Reengineering. Software Reengineering for Evolution 3.1. pages 23–52.
- [150] T. Reinbacher, J. Geist, P. Moosbrugger, M. Horauer, and A. Steininger. Parallel runtime verification of temporal properties for embedded software. *Proceedings of 2012 8th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications, MESA 2012*, pages 224–231, 2012.
- [151] C. H. Reinsch. Smoothing by spline functions. *Numerische mathematik*, 10(3):177–183, 1967.
- [152] G. Rosu. Temporal Logic.
- [153] P. Schnoebelen. The Complexity of Temporal Logic Model Checking. *World*, 4:1–44, 2002.
- [154] E. Shafie. Runtime Detection and Prevention for Structure Query Language Injection Attacks. 2013.

- [155] F. Siewe, A. Cau, and H. Zedan. A compositional framework for access control policies enforcement. *Proceedings of the 2003 ACM Workshop on Formal Methods in Security Engineering, FMSE'03, Oct 30 2003*, pages 32–42, 2003.
- [156] F. Siewe and D. V. Hung. Deriving real-time programs from duration calculus specifications. (222):92–97, 2000.
- [157] F. Siewe, H. Janicke, and K. Jones. Dynamic access control policies and web-service composition. *93*, 2005.
- [158] F. Siewe. A Compositional Framework for the Development of Secure Access Control. page 225, 2005.
- [159] B. I. Silva, K. Richeson, B. Krogh, and A. Chutinan. Modeling and verifying hybrid dynamic systems using CheckMate. *Proceedings of 4th International Conference on Automation of Mixed Processes*, pages 323–328, 2000.
- [160] M. Solanki. A compositional framework for the specification, verification and runtime validation of reactive web services. *Framework*, 2005.
- [161] M. Solanki. Tesco-s: A framework for defining temporal semantics in owl enabled services. ... *on Frameworks for Semantics in Web Services*, pages 1–6, 2005.
- [162] M. Solanki, A. Cau, and H. Zedan. Introducing Compositionality in Webservice Descriptions.
- [163] B. A. Trakhtenbrot. Automata, circuits and hybrids: Facets of continuous time (Invited Talk). pages 754–755, 2001.
- [164] S. Troncale and J.-p. Comet. A Temporal Logic with Event Clock Automata for Timed Hybrid Petri Nets. *Event (London)*, (April), 2007.

- [165] G. Tsai and B. Mcmillin. ENSURING THE SATISFACTION OF A TEMPORAL SPECIFICATION AT RUN-TIME. 1995.
- [166] A. Tuzhilin. Programming reactive systems in temporal logic. 1990.
- [167] P.-b. R. Verification. Mohamed Khalefa Sarrab. (March), 2011.
- [168] V. Vishal, S. Gugwad, and S. Singh. Modeling and Verification of Agent Based Adaptive Traffic Signal using Symbolic Model Verifier. 53(3), 2012.
- [169] B. Wadge. A hybrid predicate calculus.
- [170] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines A Practical Approach*. 2006.
- [171] T. Wilke. Specifying Timed State Sequences in Powerful Decidable Logics and Timed Automata. *Notes*, 863:694–715, 1994.
- [172] Q. Xu, A. Cau, and P. Collette. On unifying assumption-commitment style proof rules for concurrency. *CONCUR'94: Concurrency Theory*, pages 267–282, 1994.
- [173] H. Yang, X. Liu, and H. Zedan. Tackling the Abstraction Problem for Reverse Engineering in A System Re-engineering Approach. *International Conference on Software Maintenance (Cat. No. 98CB36272)*, (November 1998):284–293, 1998.
- [174] A. C. Zedan. The Systematic Construction of Information Systems. *Systems Engineering for Business Process Change*, NA:1–16, 2000.
- [175] H. Zedan, A. Cau, and B. Moszkowski. Compositional Modelling  $\therefore$  *Syntax And Semantics*, pages 19–44.



- [176] J. Zhang and B. H. C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software*, 79(10):1361–1369, 2006.
- [177] S. Zhou, A. Cau, H. Zedan, and A. Cau. Run-time Analysis of Time-critical Systems. *Journal of System Architecture*, 51(5):331—345, 2005.
- [178] S. Zhou, H. Zedan, and A. Cau. A framework for analysing the effect of 'change' in legacy code. *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99)*. 'Software Maintenance for Business Change' (Cat. No.99CB36360), pages 411–420, 1999.