Lower Bound for the Duration of Event Sequences of Given Length in Timed Discrete Event Systems

Viraj Vinod Mestry

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Electrical and Computer Engineering) at

Concordia University

Montréal, Québec, Canada

August 2022

© Viraj Vinod Mestry, 2022

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Viraj Vinod Mestry**

Entitled: **Lower Bound for the Duration of Event Sequences of Given Length in Timed Discrete Event Systems**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

____ Chair
Dr. R. Selmic

____ External Examiner
Dr. F. Nasiri

____ Examiner
Dr. R. Selmic

____ Supervisor
Dr. S. Hashtrudi Zad

Approved by ____
Dr. Yousef Shayan, Chair
Department of Electrical and Computer Engineering

_______________ 2022

____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Lower Bound for the Duration of Event Sequences of Given Length in Timed Discrete Event Systems

Viraj Vinod Mestry

The Supervisory Control Theory (SCT) of Discrete Event Systems (DES) provides a framework for synthesizing a DES supervisor to ensure a DES plant satisfies its design specification. In SCT, supervisor synthesis is performed *offline* before the functioning of plant. Generally, the size of the plant and the specifications models are large resulting in supervisors that need huge computer memory for storage – usually unavailable in embedded systems. A solution to this problem proposed in the literature is Limited Lookahead Policy (LLP). In LLP, the supervisory control commands are calculated *online* during the plant operation. After the occurrence of each event, the next control command is calculated based on the plant behaviour over a limited number of events into the future. In practice such frequent LLP computation would not be feasible as multiple events can occur consecutively over a short duration, not leaving enough time for LLP computation between them. To tackle this issue, a method is proposed called LLP with Buffering where the supervisory control commands are calculated online and buffered in advance for a predefined window of events in future. Determining the correct size of the buffer is crucial in order to achieve a trade-off between the on-board memory requirement and the computational resources and also ensuring that new supervisor commands are computed before the buffer runs out empty.

The size of the buffer primarily depends on (1) the execution time of the code for supervisor calculation and (2) the (fastest) rate of event generation in the plant. This thesis focuses on the second factor. Previously, the minimum execution duration of event sequences have been calculated experimentally. The experimental approach is not exhaustive and thus results in an overestimate

in the value of the minimum execution duration of event sequences. In this thesis, a model-based approach to the computation of the minimum duration is proposed which begins by transforming the untimed model of the plant under supervision to a timed automaton (TA) by incorporating timing information to the events. Next, an exhaustive symbolic matrix-based search algorithm is proposed where all the event sequences from every mode of the TA model are traversed to determine the minimum execution duration of the event sequences. The proposed method avoids the reachability analysis of TA needed to determine the reachable clock regions for each mode. The number of these regions is exponential in the number of events. Instead the method uses a reachability on the graph of the untimed model (polynomial in the number of events). This algorithm runs faster but provides an underestimate for the minimum execution duration of event sequences.

Next, a two-degree-of-freedom solar tracker system is used as plant to analyse the timing behaviour of the events and the implementation of LLP with buffering. In this study, the model-based and experimental methods have been used together to choose a suitable buffer size. The resulting LLP supervisor with buffering has been successfully implemented.

Acknowledgments

First and foremost, I would sincerely like to thank my supervisor, Dr. Shahin Hashtrudi-Zad for providing this fascinating opportunity during the difficult time of pandemic. This research would not have been possible without his immense support, constant guidance and encouragement over the past two years.

I would like to thank my colleagues, Mr. Kiarash Aryankia and Mr. Faiz Ur Rehman for their technical assistance and mentorship during the development of this thesis.

Special thanks to Concordia University and Concordia University Library services for invaluable literature and resources.

Finally, I would like to express gratitude to my parents, grandparents and my family members for their blessings, prayers and emotional support. I would also like to thank all my friends and Mr. Parikshit Tonge in particular for always motivating me throughout this journey.

I dedicate this thesis to my late grandfather Mr. Janardan Mestry.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**

1.1 Supervisory Control of Discrete Event Systems 2

1.2 Literature Review 3

1.2.1 Limited Look-ahead Policies 3

1.2.2 Timed Discrete Event System 6

1.2.3 Execution Time of Program 10

1.3 Research Objectives and Contributions 11

1.3.1 Objectives 11

1.3.2 Contributions 12

1.4 Organization 13

2 Background **14**

2.1 Discrete Event System (DES) 14

2.1.1 Languages 14

2.1.2 Operation on Languages 15

2.1.3 Automata 16

2.1.4 Operations on Automatons 17

2.2 Supervisory Control Theory (SCT) 18

2.2.1 Conventional Supervisory Control . . . 19
2.2.2 Limited Lookahead Policy Based Supervisory Control . . . 22
2.2.3 State-based Limited Look-ahead Policy . . . 24
2.3 Limited Look-ahead Policy With Buffering . . . 25
2.3.1 Validity of Supervisor for LLP With Buffering . . . 26
2.3.2 Buffer Size Selection . . . 28
2.4 Timed Automata . . . 30
2.4.1 Semantics of TA . . . 31
2.4.2 Execution of TA . . . 32
2.5 Clock Regions . . . 34
2.6 Clock Zones and Difference Bound Matrices . . . 36
2.6.1 Operations on DBMs . . . 39
2.7 Summary . . . 45

3 Model-Based Estimation of the Duration of Event Sequences 46
3.1 Problem Formulation and Solution Overview . . . 47
3.2 Adding Timing Information to the Untimed Model . . . 51
3.3 Calculating a Lower Bound for $T_{min}(\delta)$ . . . 55
3.4 Finding Minimum Execution Duration of Event Sequences Using DBM . . . 65
3.5 Algorithm for Calculating $T'_{min,low}(\delta)$ . . . 69
3.6 Time Complexity . . . 76
3.7 Summary . . . 81

4 Experimental Setup 82
4.1 Dual Axis Solar Tracker System . . . 82
4.1.1 Schematic Diagram . . . 82
4.1.2 System Design Requirements . . . 87
4.2 Untimed DES Model . . . 87
4.2.1 Components . . . 88
4.2.2 Component Interactions . . . 97

4.2.3 Specifications 100
4.2.4 Supervisor 103
4.3 Summary 105

5 Case Study: A Solar Tracker System **106**
5.1 LLP with Buffering parameters 107
5.2 Determination of $T_{min}(\delta)$ Using Experiments 108
5.2.1 Tests performed 108
5.2.2 Procedure for Calculating $T_{min,exp}(\delta)$ 110
5.3 Timing Information of the Events 112
5.3.1 Battery SOC Related Events 113
5.3.2 PV Panel Related Events 115
5.3.3 Command Events 117
5.3.4 Sensor Events 117
5.3.5 Other Remaining Events 118
5.4 Theoretical Results 119
5.5 Computation Time Analysis of LLP with Buffering 121
5.6 Selection of Buffering Parameters 122
5.7 Summary 125

6 Conclusion **126**
6.1 Summary 126
6.2 Future Work 127

Bibliography **129**

List of Figures

Figure 1.1 Interaction between plant and supervisor. 3

Figure 2.1 Supervisor control feedback loop [1]. 19

Figure 2.2 Limited lookahed $N-$level tree [2]. 22

Figure 2.3 Plant and two-level expansion. 24

Figure 2.4 VLP-S expansion. 25

Figure 2.5 LLP with expanded window size [3]. 26

Figure 2.6 Tree Expansion [3]. 27

Figure 2.7 Timeline of LLP with Buffering [3]. 29

Figure 2.8 Timed Automaton Example G_{TA}^1 [4]. 33

Figure 2.9 Clock Regions of G_{TA}^1 35

Figure 3.1 LLP with Buffering for the window size N_w [3]. 47

Figure 3.2 TA A_1. 49

Figure 3.3 Reachable clock regions of TA A_1. 50

Figure 3.4 Transition from mode x to x' 51

Figure 3.5 Untimed automaton G example. 53

Figure 3.6 Timed automaton G_{TA} for G 54

Figure 3.7 Example 3.2. TA G_{TA_1}. 62

Figure 3.8 Example 3.3. TA G_{TA_2}. 64

Figure 3.9 Procedure for calculating minimum execution duration of event sequence using DBM. 66

Figure 3.10 State trajectory $\mathcal{T}$ of G_{TA}. 67

Figure 3.11 Counter automaton G_{ev}. 70

Figure 3.12 Unfolded automaton G_{un} 71

Figure 4.1 Two degree-of-freedom solar tracker system. 83

Figure 4.2 Schematic diagram of solar tracker system [5]. 84

Figure 4.3 Coordinate system for solar tracker system. 86

Figure 4.4 Block diagram for the solar tracker system. 88

Figure 4.5 Battery SOC automaton [5] 89

Figure 4.6 PV panel automaton [5] 90

Figure 4.7 Azimuth motor automaton [5]. 91

Figure 4.8 Elevation motor automaton [5]. 92

Figure 4.9 Servomotor delay automaton [5]. 94

Figure 4.10 Azimuth position range automaton [5]. 95

Figure 4.11 Elevation position range automaton [5]. 96

Figure 4.12 Master controller automaton [5]. 97

Figure 4.13 Azimuth and elevation motion as a function of battery SOC. 98

Figure 4.14 Battery SOC events as a function of azimuth and elevation motion [6]. . . . 99

Figure 4.15 Battery as a function of PV panel illumination automaton [5]. 100

Figure 4.16 Azimuth motor motion range specification automaton [5] 101

Figure 4.17 Azimuth motor polling range specification automaton [5] 101

Figure 4.18 Elevation motor polling range specification. 102

Figure 5.1 Number of modes of TA_{STS} with same $t'_{min}((x, \boldsymbol{c}_{l,x}), 8)$. 120

Figure 5.2 Comparison between $C_{max}(N_w)$, $C_{avg}(N_w)$ and $C_{min}(N_w)$ where $N_w = N_{min} + \Delta + \delta$, and $N_{min} = 6$. 122

Figure 5.3 Comparison between $C_{max}(N_w)$, $T'_{min,low}(\delta)$ and $T_{min,exp}(\delta)$, with $N_w = N_{min} + \Omega$ and $N_{min} = 6$. 123

List of Tables

Table 4.1 Legend for schematic diagram of solar tracker system. 84
Table 4.2 States of the battery SOC automaton. 89
Table 4.3 Event list of battery SOC with thresholds over events [5]. 89
Table 4.4 States of the PV panel automaton. 90
Table 4.5 Event list of PV panel with thresholds voltages over events [5]. 90
Table 4.6 States of the azimuth motor automaton. 91
Table 4.7 Event list of azimuth motor automaton with current readings (mA) [5]. . . . 92
Table 4.8 States of the elevation motor automaton. 92
Table 4.9 Event list of elevation motor automaton with current readings (mA) [5]. . . . 93
Table 4.10 States of the servomotor delay automaton. 93
Table 4.11 Event list of servomotor delay automaton [5]. 94
Table 4.12 States of the azimuth angle automaton. 95
Table 4.13 Event list of azimuth angle automaton in degrees [5]. 95
Table 4.14 States of the elevation angle automaton. 96
Table 4.15 Event list of elevation angle automaton in degrees [5]. 96
Table 4.16 Event list of master controller automaton [5]. 97
Table 4.17 Plant, specification and supervisor automatons [3]. 104
Table 5.1 Example for procedure of calculating $T_{min,exp}(4)$. 111
Table 5.2 Experimental $T_{min,exp}(\delta)$ for $\delta = 1, \ldots, 9$. 111
Table 5.3 Battery SOC events time bounds. 114
Table 5.4 PV Panel events time bounds. 116

Table 5.5 Controllable events time bounds. 117
Table 5.6 Sensor events time bounds. 118
Table 5.7 Other remaining events time bounds. 119
Table 5.8 Theoretical $T'_{min,low}(\delta)$ (ms) for $\delta = 1, \ldots, 9$. 121
Table 5.9 $C_{max}(N_{min} + \Omega)$ for LLP with Buffering in ms ($\Omega = \delta + \Delta$). 122
Table 5.10 Number of LLP computations and supervisor size with respect to N_w. 124

Chapter 1

Introduction

A Continuous Variable Dynamic System (CVDS) is a system that is described using differential equations. Generally, man-made dynamic systems with complex behaviour are difficult to describe using differential equations [7]. Examples of such systems are queuing systems, computer/communication systems, manufacturing systems, traffic systems, where the state evolution of the system depends upon the occurrence of the iscrete events over time. Such systems are called Discrete Event Systems (DES). Contrary to the CVDS, DES have a discrete state space and *event-driven* state transition mechanism.

The supervisory control theory (SCT) of the DES [8] provides a formal approach for automatically synthesizing a supervisor for controlling the behaviour of a plant modelled as DES. The supervisor is synthesized *offline* on the basis of the DES model of the plant and its design specifications. One of the major problems in SCT is state space explosion. Large plant and specification models result in a supervisor with a large state space, requiring computer memory larger than what is usually unavailable in embedded system applications. An alternative approach known as Limited Lookahead policies (LLP) [2] has been proposed in the literature where the supervisory control commands are computed *online* (or on-the-fly) after every event occurrence during the functioning of the plant. Implementing LLP requires less memory since only at every iteration it needs a sub-model of plant and specifications. However, this approach imposes unrealistic timing constraints where every supervisory commands must be computed before the occurrence of the next event.

This issue of timing constraint is resolved by an innovative approach called LLP with Buffering [3] where the supervisory commands are calculated online and buffered. However, in this approach an important decision is to choose the buffer size which affects the reliability of this approach. In this thesis, we specifically focus on this step of design of LLP with Buffering.

In this chapter, we briefly review the SCT for DES. Then, a detailed literature review of LLP followed by timed automata is provided. Finally, an overview of the thesis objectives and contributions is discussed.

1.1 Supervisory Control of Discrete Event Systems

DES is a dynamic system which contains a discrete state space where transition from one state to another follows the occurrences of discrete events. Similar to other types of systems, a complex DES plant can be easily modelled by first modeling the components of the plant as DES. These components can be integrated together based on their interactions to describe the complete behaviour of the plant. One of the common formalisms of modelling DES is using finite state automaton.

The plant does not necessarily generate safe event sequences (safe behaviour). To prohibit a plant from generating unsafe event sequences, Ramadage and Wonham introduced a formal approach for supervisory control of DES in [8]. To ensure the safety of the plant, safety specifications are designed as a set of safe event sequences which can also be modelled as automaton. A supervisor ensures that the plant behaviour does not go outside its design specifications. The supervisor observes the events that are generated in the plant (Fig. 1.1). It is assumed that the supervisor can only control a subset of plant events called *controllable events* in the form of enabling or disabling the controllable events. Example of such controllable events are issuing a command to open a value, motor actuation and broadcasting a message. The events which the supervisor cannot disable are referred to as the *uncontrollable events*. For instance, sensor data readings, emergency startup/shutdown operator commands and failure emergency signals are some of examples of uncontrollable events.

In addition to the safety properties of the plant, the supervisor must also ensure that the plant is non-blocking i.e. free from livelocks or deadlocks [9].

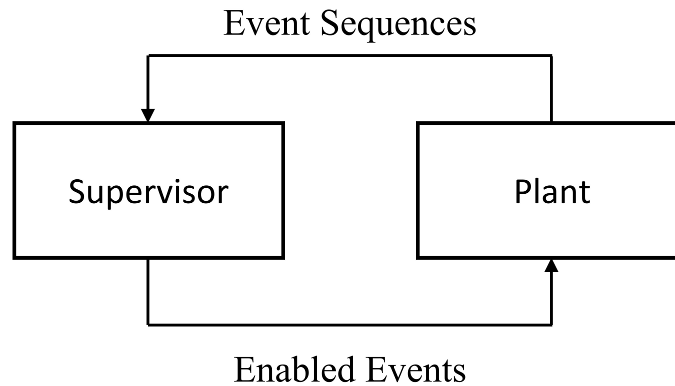


Figure 1.1: Interaction between plant and supervisor.

Similar to the supervisory control of DES, model-based programming for supervisory control and decision of autonomous systems in space based on different supervisor synthesis approach is studied in [10] and [11]. [10] describes Livingstone which is a kernel for model-based reactive self-configuring autonomous system demonstrated in flight on the Deep-Space One mission. In [11], model-based executive called Titan is used to calculate control commands based on the plant observations and configuration goals. Titan is a superset of the Livingstone system which focuses particularly on insufficient modularity and the property of robustness in space applications.

1.2 Literature Review

1.2.1 Limited Look-ahead Policies

Ramadage and Wonham framework [8] has proven to be essential in terms of computing a supervisor that is minimally restrictive and non-blocking. The supervisor is synthesized offline based on the DES models of the plant and specifications. As stated previously, an offline supervisor will be very large for real-world systems. Embedded systems consist of limited on-board memory and although not impossible, it would be usually infeasible to store such large supervisor in the on-board memory. In addition, the plant itself can be time varying (since the behaviour of the components changes time to time) and in such cases computing an offline supervisor would be impractical since the entire model of the plant is not available at the time of supervisor synthesis.

To overcome these problems, Chung et al. devised an approach called Limited Lookahead Policies (LLP) in [2]. In LLP, the supervisory commands are computed online during the functioning of the plant. In this way only the models of components and specifications are stored in the on-board memory of the system, thus reducing the memory consumption. The online memory requirement is reduced at the expense of higher online computation. In addition, LLP supervisory computation does not require complete information of the plant thereby resolving the issue of supervisory computation when the plant is time varying.

In order to synthesize a supervisor in LLP, only a portion of the plant model is explored for calculating control commands. After the occurrence of every event, the plant model will be explored from the current state up to N_w events into the future. Here, N_w is called the look-ahead window. The next supervisory control commands is chosen based on the plant window within the LLP window. In LLP, two different attitudes can be chosen for the unexplored region of the plant to compute the LLP supervisor: *optimistic* and *conservative*. The LLP supervisory commands as well as the size of LLP supervisor are greatly influenced by the attitude selected during the LLP supervisory computation. The LLP supervisor is said to be optimal if the LLP supervisory commands matches the optimal offline supervisory commands at any given moment. The look-ahead window must explore the plant sufficiently enough into the future in order to guarantee optimality of the LLP supervisor.

LLP reduces memory consumption while providing optimal supervisory commands [2]. Two major concerns of LLP are the following: (1) the supervisory computation is repeated after the occurrence of each event, and (2) the supervisory commands must be available before the occurrence of the next event. In other words, the LLP approach increases the online computational (time) complexity along with imposing a timing constraint for completing the LLP supervisory computations (i.e. the occurrence of next event). Therefore, attempts have been made to reduce the computation time of LLP supervisor in [12], [13] and [14].

In [13], a recursive computational method which uses backward dynamic programming to compute the LLP supervisor is proposed. This method uses the results of the previous lookahead window for the computation of the current lookahead window thereby reducing the computation time.

In [12], an approach called Variable Lookahead policy (VLP) is implemented to improve the efficiency of online supervisory calculations. The VLP algorithm uses a forward search technique which can terminate whenever the control decision is made explicitly even before completely exploring the portion of the plant for the window of N_w. Therefore, computation time of a VLP supervisor is definitely less than the computation time of an LLP supervisor. VLP is further improved in [14] by including state information to reduce the computations. This new approach of including state information in variable lookahead policy is named VLP-S. In [14], the VLP-S algorithm assigns a cost to each state and does not explore the states again whose cost is already known thus reducing the exploration of repeated states and loops. Both LLP and VLP take a linguistic approach to synthesize valid online supervisors, however, VLP-S take a state-based approach to synthesize valid online supervisors.

Further improvements in the area of LLP using different LLP supervisory calculation are as follows. In LLP, a conservative or an optimistic attitude is required to be selected by the supervisor for the pending traces. The supervisor with a conservative attitude may result in a restrictive control policy and the supervisor with an optimistic attitude may result violating the design specifications. In [15] an approach called Extension based Limited Lookahead Policy (ELL) is proposed. The main advantage of this approach is that ELL avoids the notion of pending traces and the need to select an attitude for the supervisor. In ELL, the supervisory commands are computed by extending the behaviour of the plant by adding all finite length event sequences beyond the N-step projection of the plant behaviour. In [16] an estimate-based Limited Lookahead policy is proposed which studies the case when the specification is described as a closed language. In [17], a method is presented for estimating the state-space of the LLP lookahed tree based on the window size N_w. The authors in [18] synthesized an optimal LLP supervisor based on reinforcement learning by using several local systems where the supervisor learns the evaluation for control pattern from the rewards obtained from the local systems.

In addition to the theoretical developments, the framework of LLP is incorporated in numerous applications solving specific problems. In [19], the authors synthesized an LLP supervisor using Petri-nets model of the robot system in order to chose a "qualitative" supervisory command for navigating the mobile robot in a building. In [20], the framework of LLP is used to dynamically

allocate task assignments in real-time to a team of robots. In [21], LLP is used to actively monitor software to predict possible faults in advance and take appropriate actions in order to prevent the software from violating some given property. In [22], a symbolic version of LLP is used for dynamic Computing Resource Management.

LLP and VLP approaches for computing supervisory commands are very promising to be implemented for controlling DES; these approaches however will surely fail if events occur back to back in rapid succession. Therefore, to tackle this situation, in [3], LLP with Buffering is introduced where supervisory commands are calculated online and buffered in advance for a predefined window size. The control commands will be always available as required, and sufficient time will be available to perform the forthcoming LLP supervisory computations. In LLP with Buffering, selecting an appropriate buffer size greatly impacts the reliability of the approach. Smaller buffer size results in an LLP supervisor with smaller state space, while larger buffer size may result in an LLP supervisor with state space equal to the offline supervisor. The buffer size mainly depends on two factors: (1) the computation time of the LLP supervisor, and (2) the minimum execution duration of event sequences. In [5], LLP with Buffering is further improved by implementing all supervisory control algorithms in C language for faster computation time.

1.2.2 Timed Discrete Event System

A DES model describes event sequences without necessarily providing any information about the time at which the event occurs. These models can be enriched to provide timing information. Such systems are referred to timed discrete event systems (TDES) where the models posses temporal information. TDES is generally partitioned into two categories depending upon how the time is modelled:

(1) discrete time models where the passage of time is presented as discrete clock tick events;

(2) continuous time models where the time is traced using real-valued clock variables.

The Timed transition Model (TTM) described in [23] is a more common formalism for modelling discrete-time models in TDES. In TTM, a special event *tick* is introduced which represents the passage of time. In [23], each event is associated with an upper time bound and a lower time bound in

terms of *tick* restricting the time of occurrence of the events in the system. The passage of time between two consecutive events is calculated by counting the number of *tick* events between them [24]. One of the advantages of using discrete time model is their simplicity as existing algorithms for untimed DES can be extended to these discrete-time models. However, the major disadvantage is that continuous time is approximated with the error depending on the resolution chosen for the tick size a priori. In practice, choosing smaller resolution for tick size may represent the real-time system accurately but can in turn result in blow up in the state space of the model.

Timed automata (TA) introduced by Alur and Dill in [25] and [26] are well established as continuous-time models for TDES. TA models employ auxiliary real valued variables called *clocks*. The value of each clock increases at a constant rate as time progresses. A state of the TA consists of a discrete state and the valuation of the clocks. Therefore, TA is a dense-time model which can accurately represent a real-time system.

Model Checking

Model checking algorithms are used to verify properties of DES systems. Existing model checking algorithms can be easily extended to discrete-time models of TDES. As mentioned previously, despite their simplicity, these models are not very accurate for modeling real-time systems [27]. On the other hand, TA models are extensively used in automatic verification (model checking) of real-time systems especially in the area of hardware and software verification. Model checking is one of the most successful methods of automatic verification to check whether the finite state model of a system satisfies a given specification (or property). The specification is generally a safety requirement or a liveliness requirement for the system given as temporal logical formula. In [24], the model checking methods are extended to check real-time systems modelled by TA. The state space of the TA is infinite and it would be impractical to check whether the specification is satisfied by all states in the reachable subgraph of the system. Therefore, using a clever method a TA is abstracted into a finite automaton (FA) known as region automaton based on the abstraction described in [26]. The model checking algorithm is performed on the reachable states of the region automaton in [24] since region automaton has a finite state space. However, the complexity of the model checking algorithm is exponential in the number of clocks and the largest time constant in region automaton,

and exponential in the number of discrete states resulting through the parallel composition of the component models [28] [27]. In [28] and [27] symbolic model checking techniques are proposed where the set of states that satisfy the specification are computed symbolically instead of constructing the region automaton (region graph). Symbolic model checking algorithms are successfully implemented in model checking tools such as UPPAAL [29] and Kronos [30] with TA at the core in these model checking tools. Although finite, the state space of the region automaton can be very large and further abstraction to minimize the state space is possible (if finite partitioning of the state space of region automaton is not necessary in determining the correctness of the system) by using zone-based abstraction [26]. Such zone-base abstraction of the TA is implemented in the verification algorithm of real-time system described in [31] and in the model checking tool UPPAAL.

Supervisory Control of TDES

Enormous research has been conducted in the area of supervisory control using TDES models. Brandin and Wonham [32] used the discrete-time TTM [23] to synthesis a supervisor based on the Ramadage and Wonham framework. As previously mentioned, TTM models suffers from the problem of "state space explosion" due to the introduction of *tick* events. Therefore, even for a moderate sized plant, the size of the supervisor can be extremely large. In addition, the discrete-time models limit the accuracy of modeling real-time systems due to the discretization of continuous time. In an effort to reduce the state space of the supervisor, a minimization algorithm is proposed in [33].

TA for synthesizing a supervisor in supervisory control was first proposed by Wong-Toi and Hoffmann in [34]. Since the state space of TA is infinite, in [34], the authors synthesized the supervisor based on the region automaton of the plant and the specifications. Finally, timing details are incorporated in the supervisor to transform the supervisor into a TA. The complexity of the timed supervisor synthesis algorithm is exponential in the number of clocks and the largest clock constant of region automaton. Zone-based abstraction of the TA definitely minimizes the state space but does not provide the information required to synthesize a supervisor [35]. A state of the region automaton corresponds to a discrete state (mode) and a clock region. In region automaton, the state transition occurs when there is an execution of a discrete event or when the clock region changes as time

progresses while dwelling in a discrete state. Therefore, the region automaton is non-deterministic since from every state an event transition can occur at different time moments [36]. To overcome this problem, in [36] two events *tock* and *tack* are introduced to make the region automaton deterministic and refer to region automaton as τ-region automaton. In a τ-region automaton, the *tock* event is defined at a state when there are no forcible events active at the state, while the *tack* event is defined at a state if there are forcible events active at the same state. In a sense, the *tock* event is uncontrollable and *tack* event is controllable. Based on the Ramadage and Wonham framework, the supervisor is synthesized for the τ-region automaton. The approach described in [36] however is applied only to particular type of problems.

In order to minimize the state space explosion caused by performing the untiming operation of a TA into a region automaton, a method is proposed in [37], which transforms the TA into a minimal FA called SetExp Automaton by using two special events *Set* and *Exp* in additions to the events of the TA. The event *Set* corresponds to the set and reset of the clock and the event *Exp* corresponds to the expiration of the clock. The procedure to synthesize a supervisor based on the SetExp Automaton of the plant and the specification is described in [38] by using the concept of forcible events of TDES. Forcible events are the events that can preempt time and other events eligible at that time [9]. The concept of forcible events provides the model a complete description of the real-time system. One setback of the supervisor synthesis procedure in [38] is that the computed supervisor takes the form of a SetExp Automaton and it is not known so far how to transform the SetExp automaton back to a TA.

In all of the above-mentioned approaches, for synthesizing a supervisor based on the Ramadage and Wonham framework, region-based abstraction of the TA is necessary which suffers from the state space explosion. So far only in [35] an attempt is made to synthesize a supervisor without any abstraction to cope with the problem of state space explosion. [35] uses an approach to synthesize a supervisor according to the Ramadage and Wonham framework of SCT to avoid *bad-states*, which are referred to as blocking states and the states that uncontrollably lead to a bad state. In the work of [35], a timed supervisor is synthesized (without any abstraction) by modifying the invaraints and the guards in the semantics of the TA. Modification to the guards of the edges labeled by controllable events results in the supervisor that prevents bad states. In addition, modification to the invariants

of the states in the TA results in the supervisor which uses forcible events to preempt time whenever needed to prevent bad states.

Apart from the Ramadage and Wonham framework, there is another framework called game-based synthesis for synthesizing a supervisor based on a winning strategy for certain timed game automaton modelled by a TA as described in [39–41]. A strategy for the supervisor is supposed to be winning if the game stays in a subset of winning states. In a timed safety game, there are two players who can perform some actions: supervisor (referred to as controller in the literature) and environment. In game-based synthesis, the consequence on the timed game is not only influenced by the action taken by the supervisor and the environment but also by the time at which the actions are taken. The objective is to design a winning strategy which informs the supervisor to take an appropriate action so that regardless of the environment performing any action at any time, the timed game stays in a subset of winning states. While the Ramadage and Wonnham framework focuses on synthesizing a minimally restrictive supervisor, the game-based synthesis synthesizes a supervisor based on a winning strategy which might not be always minimally restrictive.

1.2.3 Execution Time of Program

As mentioned in Section 1.2.1, in order to determine suitable buffer size in LLP with Buffering, two things are needed: (1) the (worst-case) execution time of supervisory control computations, and (2) the fast rate of event generation in the DES plant. Estimation of these two values is similar to the estimation of the execution time of programs for hard real-time systems. In the following, we briefly review the main approaches for estimating a program's execution time.

In practice *dynamic timing analysis* is used to determine the *best-case execution time* (BCET) and the *worst-case execution time* (WCET) of a program, where these bounds are estimated by end-to-end execution time of the program for a subset of test cases [42]. The estimates are usually an overestimate for BCET and an underestimate for WCET which in most cases are not a right choice for hard real-time systems. Moreover, there are other better methods used such as the *static* methods and *measurement-based* methods for determining the BCET and WCET for a program [42]. In static method, the program is not executed on the hardware or the simulator to determine the BCET or WCET, but instead the code of the program is combined with the abstract model of the system

along with some annotations to determine BCET or WCET. The static method results in bounds on the execution time that can underestimate BCET and overestimate WCET. Another method is the measurement based method where the program is executed on the hardware or the simulator, for some set of inputs and the execution time is measured. This method results in the estimate of the BCET and WCET which sometimes can be unsafe for hard real-time systems. The static method prioritizes safety of the system. On the other hand, the measurement method prioritizes the ease of determining the estimates for the execution time. As mentioned in [42], the static method always predicts the bounds on the execution time far lower than the ones produced by measurement-based method.

1.3 Research Objectives and Contributions

In this section we will explain the objectives and the contributions of this thesis.

1.3.1 Objectives

In this thesis we focus on bridging the gap between theoretical results and practical implementation of LLP on a real-time system by further studying the practicality of LLP with Buffering. To the best of our knowledge, there has not been any practical implementation of LLP on any physical system yet. This may be, in part, due to the unrealistic constraints of computing the LLP supervisor commands after the occurrence of each event in the system. In LLP with Buffering, however, the supervisory commands are calculated online in a timely fashion for a certain window of events in the future and buffered. In this way, the supervisory commands are readily available in the buffer after the execution of every event in the system. Determining the accurate size of the buffer is extremely important for two reasons: (1) if the size of the buffer is very small, then there will be frequent LLP supervisory computation requiring high computation resources and (2) if the size of the buffer is large, then the computed LLP supervisor will be large thus requiring higher computer memory. The buffer size depends upon the minimum duration (i.e. fastest execution rate) of event sequences in the plant. In addition, it is absolutely necessary that the computation time of the LLP supervisor must be completed before the buffer runs empty in order to avoid supervisor falling behind plant

event sequences.

Previously, the minimum duration of the event sequences has been determined experimentally by time stamping every occurring event. However, it is not possible to perform exhaustive experimental tests to search for all event sequences that transpires in the system. A theoretical method is proposed in [5] where the TTM of the solar tracker system is used to extract the timing information of the event sequences. However, due to the issue of state space explosion caused by the *tick* events, the search of the timing details of event sequences is restricted only to the partially explored TTM of the system.

In this thesis, our objective is to make the LLP with Buffering to be a feasible control strategy for any real-time system. Moreover, our objective is to develop a theoretical method for performing timing analysis by using continuous-time model (particularly TA) which can exhaustively search all the event sequences and extract the minimum execution duration from every mode of the TA model of the system while avoiding state space explosion.

1.3.2 Contributions

This thesis proposes a method to transform the untimed automaton of the system to a TA in order to extract the timing information of the event sequences. Most of the work done so far is to transforming a TA to FA as reported in the literature [1, 26, 34, 36, 37]. An algorithm is proposed to determine the minimum execution duration of the event sequences in the timed model using matrix-based symbolic analysis introduced in [4]. This algorithm performs exhaustive search by traversing all the event sequences up to any given length from every state of the timed automaton.

In this thesis, timing analysis of the previously implemented LLP with Buffering on the two-degree-of-freedom solar tracker system is studied. The objective of this system is to search for the light source in the surrounding. The minimum execution duration of the events sequences in the solar tracker system is obtained using the proposed theoretical method by transforming the untimed model of the solar tracker system to a TA. Numerous experiments are performed on the solar tracker system to acquire the minimum duration of the event sequences experimentally. The minimum execution duration of event sequences of different length obtained through the theoretical method and experimental method are compared and in all of the cases the theoretical method proves to be

more effective and reliable in determining the size of the buffer.

In several tests performed on the solar tracker system, the buffer size determined theoretically results in the LLP supervisor with Buffering in which all supervisory commands are calculated without missing the deadlines.

1.4 Organization

The thesis is organized as follows. In Chapter 2, we review the details of DES along with the supervisory control of DES, followed by a detailed review of Limited Lookahead Policies with Buffering, followed by formal definitions of TA together with the relevant concepts. The procedure of transforming the DES model to a TA and the methodology to extract the minimum execution duration of the event sequences from the TA is proposed in Chapter 3. To verify the results, the proposed theoretical methodology is applied on the TA model of the solar tracker system. In Chapter 4, the modelling of the solar tracker system (used in the thesis) as DES, design and implementation of the offline method of SCT. In Chapter 5, the method proposed in Chapter 3 is applied to the design of an LLP supervisor with Buffering for the solar tracker system. The results for this case study is used to refine the procedure for choosing the buffer size for LLP supervisor. In Chapter 6, the thesis is summarized and the direction for future research is suggested.

Chapter 2

Background

In this chapter, we will review some of the required preliminaries. Section 2.1 introduces languages and automaton. In Section 2.2, the supervisory control theory is discussed. In Section 2.3 the Limited Look-ahead Policies (LLP) with Buffering (proposed in [3]) is reviewed. In Sections 2.4 and 2.5, the timed automaton (TA) model and clock regions of TA are discussed. Finally, in Section 2.6, the clock zones of TA and their representation as difference bound matrices (DBM) along with some operations on DBM are discussed.

2.1 Discrete Event System (DES)

A Discrete Event System (DES) has a discrete state set and its dynamics are described in terms transitions among states. A DES is an event-driven system where the occurrence of discrete events over time causes transitions from one state to another. Thus, these sequences of events show the behaviour of the system represented by DES. There are many formal ways that can a represent DES, for example, Petri-nets [43] and automatons [1]. In this section, DES models are represented using languages and finite-state automatons as discussed in [1].

2.1.1 Languages

A finite non-empty set of distinct symbols each representing an event in DES forms an **alphabet**, denoted here by Σ. A sequence of events over Σ is called as a **word**, **string** or **trace**. The symbol ϵ

denotes an empty string which contains no events. A **language** is a set of strings over an alphabet Σ. The language Σ^+ contains the set of all finite strings excluding the empty string over the an alphabet Σ. In other words

$$\Sigma^+ := \{\lambda_1\lambda_2 \ldots \lambda_i \mid i \geq 1, \lambda_i \in \Sigma\}.$$

The set of finite strings over the alphabet Σ including the empty string ϵ is denoted by Σ^*:

$$\Sigma^* := \Sigma^+ \cup \{\epsilon\}.$$

The length of a string s is shown by $|s|$. If $s = \epsilon$, then $|s| = |\epsilon| = 0$. Moreover, if $s = pqr$ with $p, q, r \in \Sigma^*$, then p is a **prefix** of s, q is a **substring** of s and r is the **suffix** of s.

2.1.2 Operation on Languages

Let M and P be two languages over Σ.

The **union** of the two languages M and P is the set of strings that either belongs to M or P:

$$M \cup P := \{u \mid u \in M \text{ or } u \in P\}.$$

The **intersection** of the two languages M and P is the set of strings common to M and P:

$$M \cap P := \{u \mid u \in M \text{ and } u \in P\}.$$

The **complement** of the language M denoted by M^{co} contains the strings present in Σ^* but not in M:

$$M^{co} := \Sigma^* - M = \{u \in \Sigma^* \mid u \notin M\}.$$

The **concatenation** of the two languages M and P denoted by ***MP*** contains the strings which is the concatenation of a string in M with a string in P:

$$\textbf{\textit{MP}} := \{uv \in \Sigma^* \mid u \in M \text{ and } v \in P\}.$$

The **prefix-closure** of the language M, denoted by $\overline{M}$, consists of all the prefixes of every string in M:

$$\overline{M} := \{u \in \Sigma^* \mid \exists v \in \Sigma^* \, [uv \in M]\}.$$

Generally, $M \subseteq \overline{M}$; however, if $M = \overline{M}$, then M is known as prefix-closed.

The **Kleene-closure** of the language M denoted by M^* is formed by the concatenation of all the finite strings of M, including the empty string, and is given as

$$M^* := \{\epsilon\} \cup M \cup MM \cup MMM \cup \dots.$$

Definition 2.1 ([2]). *The **post-language** of M after u is denoted by M/u is the set of all suffixes of the string u contained in M:*

$$M/u := \{\exists v \in \Sigma^* \mid uv \in M\}.$$

□

Definition 2.2 ([2]). *The **truncation** of M up to $N \in \mathbb{N}$ denoted by $M|_N$, is the set that consists of all strings in M which have a length not more than N*

$$M|_N := \{u \in \Sigma^* \mid |u| \leq N\}.$$

□

2.1.3 Automata

The languages discussed in Section 2.1.1 can be represented by finite state deterministic automatons. A deterministic automaton takes the form of a five-tuple

$$G = (X, \Sigma, \eta, x_0, X_m),$$

where X is the finite state set, Σ is the finite set of events of G, $\eta : X \times \Sigma \to X$ is a partial transition function, x_0 is the initial state ($x_0 \in X$) and X_m is the set of marked states ($X_m \subseteq X$).

Consider the automaton $G = (X, \Sigma, \eta, x_0, X_m)$. The language generated by the automaton G is called the **closed language**: $L(G) := \{s \in \Sigma^* \mid \eta(x_0, s)!\}$ ($\eta(x_0, s)!$ means string "s" can be occur starting from state x_0). It follows from the definition that $L(G)$ is a prefix-closed language $L(G) := \overline{L(G)}$. The **marked language** by the automaton G is $L_m(G) := \{s \in L(G) \mid \eta(x_0, s) \in X_m\}$. The marked language $L_m(G)$ is the set of strings whose execution from the initial state ends in some marked state. Therefore, it is obvious that $L_m(G) \subseteq L(G)$.

2.1.4 Operations on Automatons

Modelling a complex DES using a single automaton is difficult as it would be cumbersome to capture the interactions between all the components of the system operating concurrently. An alternative approach is to model, the components of the complex system as automatons that can be interconnected together to form the complex system. Therefore, *product* and *synchronous product* (parallel composition) operations are defined for automatons to model these interconnections between the component automatons.

Definition 2.3 ([1])**.** *Consider $G = (X, \Sigma, \eta, x_0, X_m)$. The* ***reachable part*** *of G is the state set X_r defined as*

$$X_r = \{x \in X \mid \exists s \in L(G) \text{ such that } \eta(x_0, s) = x\}.$$

□

The reachable states are the states in G that can be reached from the initial state x_0 by at least one string $s \in L(G)$. The *reachable* sub-automaton of G is denoted as reach$(G) = (X_r, \Sigma, \eta_r, x_0, X_{mr})$, where $\eta_r : X_r \times \Sigma \to X_r$ and $X_{mr} = X_m \cap X_r$.

Definition 2.4 ([1])**.** *Consider the two automatons $G_1 = (X_1, \Sigma_1, \eta_1, x_{0_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, \eta_2, x_{0_2}, X_{m_2})$, the* ***product*** *of G_1 and G_2 is defined as follows:*

$$G_1 \times G_2 := \textit{reach}(X_1 \times X_2, \Sigma_1 \cap \Sigma_2, \eta, (x_{0_1}, x_{0_2}), X_{m_1} \times X_{m_2}).$$

The transition function η is defined as follows. For $x_1 \in X_1$ and $x_2 \in X_2$, and $\lambda \in \Sigma$

$$\eta((x_1, x_2), \lambda) := \begin{cases} (\eta_1(x_1, \lambda), \eta_2(x_2, \lambda)) & \text{if } \eta_1(x_1, \lambda)! \text{ and } \eta_2(x_2, \lambda)! \\ \text{not defined} & \text{otherwise .} \end{cases}$$

□

Definition 2.5 ([1])**.** *Consider the two automatons $G_1 = (X_1, \Sigma_1, \eta_1, x_{0_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, \eta_2, x_{0_2}, X_{m_2})$. The* ***synchronous product*** *of G_1 and G_2 is defined as follows:*

$$G_1 || G_2 = \textit{reach}(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, \eta, (x_{0_1}, x_{0_2}), X_{m_1} \times X_{m_2}),$$

where for $x_1 \in X_1$*,* $x_2 \in X_2$ *and* $\lambda \in \Sigma$

$$\eta((x_1,x_2),\lambda) := \begin{cases} (\eta_1(x_1,\lambda),\eta_2(x_2,\lambda)) & \textit{if } \lambda \in \Sigma_1 \cap \Sigma_2 \textit{ and } \eta_1(x_1,\lambda)! \textit{ and } \eta_2(x_2,\lambda)!, \\ (\eta_1(x_1,\lambda),x_2) & \textit{if } \lambda \in \Sigma_1 - \Sigma_2 \textit{ and } \eta_1(x_1,\lambda)!, \\ (x_1,\eta_2(x_2,\lambda)) & \textit{if } \lambda \in \Sigma_2 - \Sigma_1 \textit{ and } \eta_2(x_2,\lambda)!, \\ \textit{not defined} & \textit{otherwise .} \end{cases}$$

□

The synchronous product of automatons can also be obtained by the product operation. Consider the two automatons G_1 and G_2. Let G'_1 denote the automaton obtained by adding the self-loops of events in $\Sigma_2 - \Sigma_1$ G_1 and let G'_2 denote the automaton obtained by adding the self-loops of events in $\Sigma_1 - \Sigma_2$ to G_2. Then the synchronous product of G_1 and G_2 is given as

$$G_1||G_2 = G'_1 \times G'_2.$$

2.2 Supervisory Control Theory (SCT)

Ramadge and Wonham first introduced the supervisory control theory (SCT) of DES [8, 44]. SCT provides a modeling framework for automatically synthesizing a supervisor based on the models of plant and design specifications (also called legal behavior). The plant and the specifications are modeled as DES and the resulting supervisor is also modelled as a DES. The objective of the supervisor is to prevent illegal behavior in the plant avoiding blocking (i.e. deadlocks and livelocks) in the plant. It is assumed that the supervisor can only disable and prevent a subset of events referred to as the controllable. However, it cannot prevent the rest of events, referred to as uncontrollable events. The supervisor representing the control logic for the operation of the plant can be calculated before the operation starts. This “offline” calculations results in the supervisor which we refer to as the conventional supervisor. In another approach, supervisory control commands can be calculated “online” when the plant is operational and events occur one after the other. The resulting supervisory policy is called LLP. We will review both conventional and LLP supervisory control.

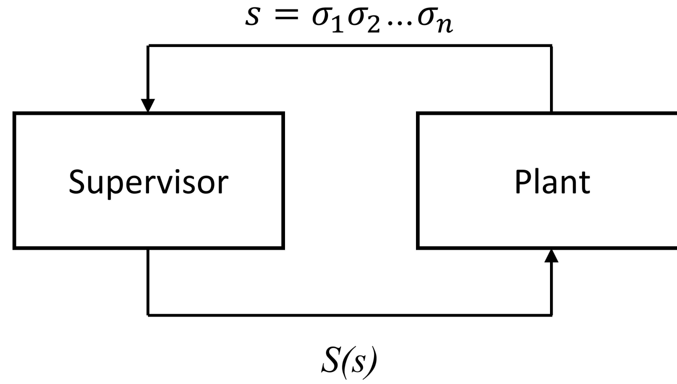


Figure 2.1: Supervisory control feedback loop [1].

2.2.1 Conventional Supervisory Control

Let G and H denote the DES models of the plant and the specification respectively. Let Σ be the event set of G. The alphabet Σ has the partition

$$\Sigma = \Sigma_c \,\dot{\cup}\, \Sigma_{uc},$$

where the disjoint subsets Σ_c and Σ_{uc} denote the controllable and uncontrollable events respectively. Let $L(G)$ and $L_m(G)$ be the closed and the marked language of the plant G respectively. The *supervisory control (S)* for G is any map

$$S : L(G) \to 2^{\Sigma}.$$

Definition 2.6 ([1]). *Consider the automaton $G = (X, \Sigma, \eta, x_0, X_m)$. The set of* ***defined*** *(active) events at a state x is given by*

$$\Gamma(x) := \{\sigma \in \Sigma \mid \eta(x, \sigma)!\}.$$

□

As illustrated in Fig. 2.1, string $s \in L(G)$ is generated by the plant and at the current state $x = \eta(x_0, s)$ the supervisor permits a subset of events, $S(s)$, to occur in the plant. Therefore, the set of

events that remain active at the current state of x of G is

$$S(s) \cap \Gamma(x).$$

At state x, the supervisor S can disable some controllable events; however, it cannot disable uncontrollable events defined at x. Consequently, the supervisor must include the set of uncontrollable events defined at x in $S(s)$:

$$\Sigma_{uc} \cap \Gamma(\eta(x_0, s)) \subseteq S(s).$$

Such a supervisor is known as an admissible supervisor. The plant G under the supervision S is denoted by S/G.

Definition 2.7 ([1]). *For the plant G, the language generated by S/G is denoted by $L(S/G)$ and is defined as*

(1) $\epsilon \in L(S/G)$,

(2) $[s \in L(S/G)$ *and* $s\sigma \in L(G)$ *and* $\sigma \in S(s)] \iff [s\sigma \in L(S/G)]$.

□

Definition 2.8 ([1]). *For the plant G, the language marked by S/G is denoted by $L_m(S/G)$ and is defined as*

$$L_m(S/G) := L(S/G) \cap L_m(G).$$

□

Now, let us define the supervisory control problem.

Problem 2.1. *Consider the DES G and the marked legal language $K \subseteq L_m(G)$ and $K \neq \emptyset$. Find a supervisor such that*

(1) $L_m(S/G) \subseteq K$ *(safety property),*

(2) $\overline{L_m(S/G)} = L(S/G)$ *(non-blocking property).*

□

The solutions to Problem 2.1 are characterized using the properties of controllability and $L_m(G)$-closure.

Definition 2.9 ([1]). *Consider the DES G and the language $K \subseteq L(G)$. Then K is said to be controllable with respect to $L(G)$ and Σ_{uc} if*

$$\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K}.$$

□

The class of controllable sub-languages of K is given as

$$C(K) = \{L \mid L \subseteq K \text{ and } \overline{L}\Sigma_{uc} \cap L(G) \subseteq \overline{L}\}.$$

It can be shown that $C(K) \neq \emptyset$ and has a supremal element denoted by $K^{\uparrow}$:

$$K^{\uparrow} = supC(K).$$

Definition 2.10 ([1]). *Consider a language $K \subseteq L_m(G)$. K is said to be **$L_m(G)$-closed** if $K = \overline{K} \cap L_m(G)$.* □

The theorem below gives the necessary and sufficient conditions for the existence of a solution to Problem 2.1.

Theorem 2.1 ([1]). *Consider the DES G and a non-empty sub-language $K \subseteq L_m(G)$. If K satisfies the following conditions:*

(1) K is controllable with respect to G and Σ_{uc}, and

(2) K is $L_m(G) - closed$.

Then there exists a solution S to the supervisory control problem such that $L_m(S/G) = K$ and $L(S/G) = \overline{K}$ and vice versa. □

Note that if all states of G are marked, then $L_m(G) = L(G)$ and condition 2 will be satisfied.

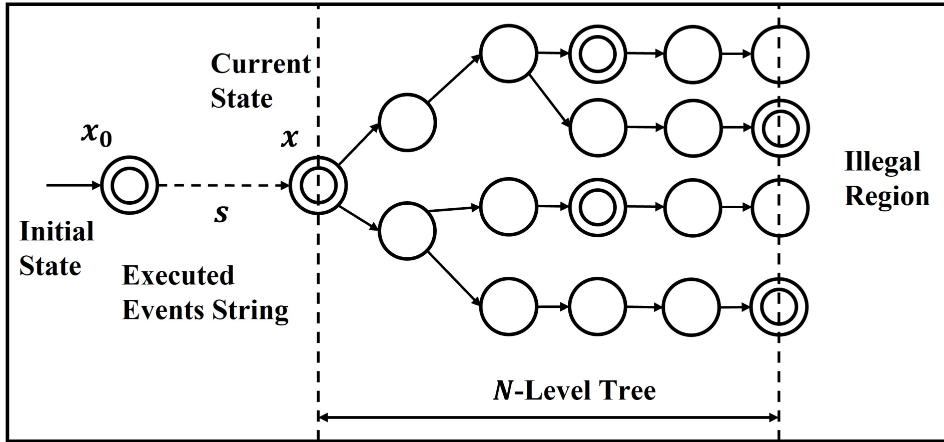


Figure 2.2: Limited lookahed $N-$level tree [2].

2.2.2 Limited Lookahead Policy Based Supervisory Control

A conventional supervisor is a monolithic supervisor which is synthesized *offline* on the basis of the plant and the specification models [45]. A conventional supervisor, although guaranteed to be correct, can be infeasible to be implemented in some real-world applications. For instance, the plant and the specification could have a large state space or be time-varying, or the specifications may not be described as a DES model easily. In [2] Chung et al. address these problems by providing a formal methodology known as Limited Look-ahead Policies (LLP) to synthesize a supervisor, i.e. *online* decision making.

In LLP supervisory control, the supervisory control action is computed based on the behavior of the *N*-level expansion of DES from the current state which takes a form of an *N*-level tree (Fig. 2.2). In the *N*-level expanded tree of the DES, the traces at the Nth level are assumed to be *pending* if they depend on the future behavior. In [2], in order to compute the supervisory control action, the pending traces can choose either one of the two different attitudes: (1) a *conservative* attitude, where the pending traces are assumed to be illegal and (2) an *optimistic* attitude, where the pending traces are assumed to be legal. Further improvements in LLP can be found elsewhere [13, 15, 16].

In this section we shall review LLP. Only the conservative attitude will be discussed. Consider the DES G over the alphabet Σ. Suppose, during the online LLP computation the trace s has been executed and the current state of G is $\eta(x_0, s) = x$. The control action $\gamma^N(s)$ that must be taken at x is given by the following steps.

Step (1) : Initially G is expanded N steps beyond trace s.

Step (2) : The supervisor then determines the illegal traces in the N-level tree obtained in Step (1).

Step (3) : The supervisor with conservative attitude modifies all pending traces of length N obtained in Step (2) as *illegal*.

Step (4) : Then, the supremal controllable sub-language of the marked language obtained in Step (3) with respect to the marked language $L_m(G)/s|_N$ and Σ_{uc} is calculated.

Step (5) : Finally, the supervisory control action $\gamma^N(s)$ is calculated at the current trace s by just allowing the first level of the tree output obtained in Step (4) and the active set of uncontrollable events that can be executed in G after the trace s.

There are two notions **validity** and **run-time error** (RTE), which are needed to be examined for the implementation of the above-mentioned algorithm.

Definition 2.11 ([2]). *A control policy γ^N for a LLP supervisor is said to be valid if*

$$L(G, \gamma^N) = \overline{K^{\uparrow}}.$$

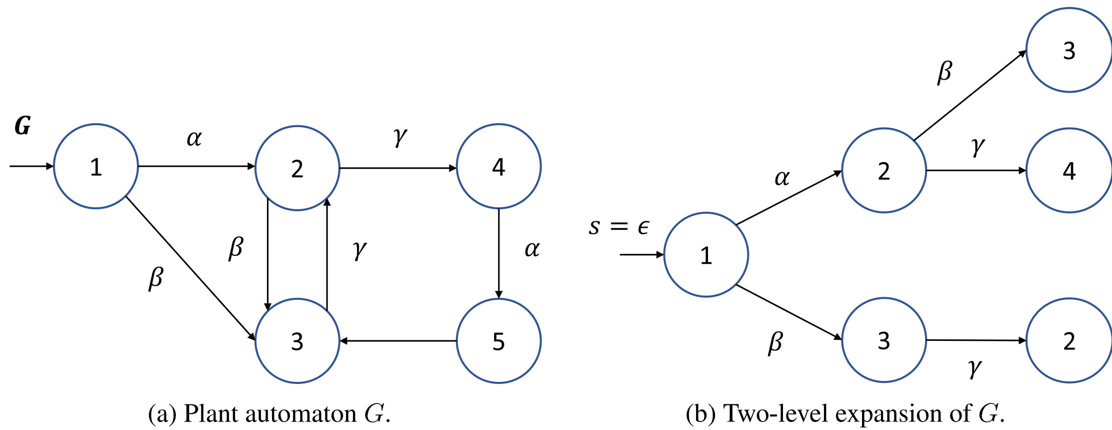
□

The above definition states that the valid LLP supervisor has the same effect as a minimally restrictive conventional supervisor.

Definition 2.12 ([2]). *For any trace $s \in L(G, \gamma^N)$ during LLP computation, if $f^N(s) = \emptyset$, then there is a RTE at trace s in $L(G, \gamma^N)$. A special case, if $s = \epsilon$, then RTE is called **starting error** (SE).* □

There will be a RTE or SE, when at the current state of the system the LLP supervisor cannot guarantee safe action such that blocking or driven uncontrollably to an illegal region cannot be avoid.

As seen in Step (3) of LLP supervisor algorithm, the control action depends on the conservative attitude adopted by the supervisor for the pending traces. Therefore, for conservative attitudes it is necessary to determine the bounds on the look-ahead window N that can guarantee validity and no RTE in the resulting LLP supervisor. [2] provides sufficient conditions under which there exists a minimum length for the expansion window, to guarantee LLP supervisor validity. Here, we denote this number by N_{min}. It should be noted in some cases that these sufficient conditions do not hold and no finite expansion can result in a valid supervisor. In such cases, a state-based LLP can resolve the problem.



(a) Plant automaton G. (b) Two-level expansion of G.

Figure 2.3: Plant and two-level expansion.

Example 2.1. *Consider the automaton of plant G shown in Fig. 2.3a. The two-level lookahead window at the empty string for plant G is shown in Fig. 2.3b. Notice that in LLP each node of the expanded tree corresponds to exactly one trace.* □

2.2.3 State-based Limited Look-ahead Policy

The LLP supervisor in [2] is an event-based LLP, since the plant model is expanded in a $N-$level tree into the future from the current state and is used to calculate supervisory commands. As mentioned previously, in some cases no finite length window can result in an optimal supervisor. (For instance, when the plant contains a loop of uncontrollable events). Consequently, in such cases, it would not be feasible to obtain a valid supervisor. To solve this problem, an alternate approach

known as variable lookahead policies with state information (VLP-S) is proposed in [14]. In VLP-S, the tree expansion in LLP is replaced with expansion of the subautomaton of the plant model reachable from the current state. This approach is is called *variable*. A window of N in these cases contains all reachable states with a sequence with at most N events. In this case, if the offline supervisory control is solvable, then the state-based online will also be solvable and gives an optimal solution with window $N_{min} \leq |X| - 1$ ($|X|$ is the number of plant's states), i.e. validity will be guaranteed.

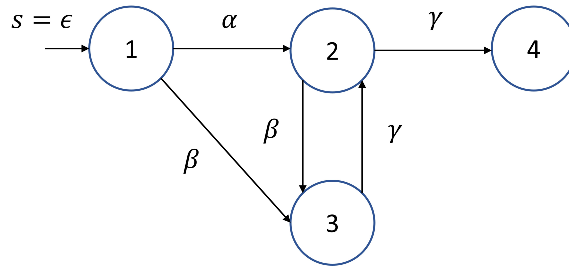


Figure 2.4: VLP-S expansion.

Example 2.2. *Consider the automaton of plant G shown in Fig. 2.3a. The two-level VLP-S expansion for the plant G is a reachable subautomaton of G at the empty string (state 1 in Fig. 2.4).* □

Note that the expansion of VLP-S always terminates since the number of plant's states is finite

2.3 Limited Look-ahead Policy With Buffering

In LLP [2], the supervisory commands are calculated online based on the *N-step* truncated behaviour of the plant and the legal behaviour [46]. However, computing the supervisory commands online requires some time (and high computation power). Such frequent computation might not be feasible in practical implementations. Since multiple events could transpire in the system consecutively at a very rapid pace, there might not be enough time available between two successive events to calculate the control decision online.

Therefore to tackle this issue, in [3] an innovative methodology is proposed called LLP with Buffering. In LLP with Buffering the supervisory commands are calculated online in a timely fashion for finite horizon of Δ events into the future and buffered, to be used in the next event and

the following few events in the future. Here, similar to LLP, the model of the plant is expanded for the lookahead window of length but with a length $N_w \geq N_{min} + \delta + \Delta$ (Fig. 2.5). N_{min} is the minimum window length for the LLP supervisor to become valid.

2.3.1 Validity of Supervisor for LLP With Buffering

In this thesis we only focus on state-based LLP since in this case, validity of the online supervisor can be achieved for a finite window. Let $G = (X, \Sigma, \eta, x_0, X_m)$ denote the plant and $H = (X_H, \Sigma, \eta_H, x_0, X_{mH})$ denote the finite-state automaton marking the design specification. Without loss of generality we assume H is a subautomaton of G.

The lookahead window size should be at least N_{min} for the LLP supervisor to be a valid supervisor. N_{min} is be always bounded and $N_{min} \leq N$, where N is the number of states of the plant (when the specification is the sub-automaton of the plant) [5].

In LLP with Buffering the supervisory commands are calculated for the immediate next event and Δ events in the future. The computations of Δ commands must be performed before the commands

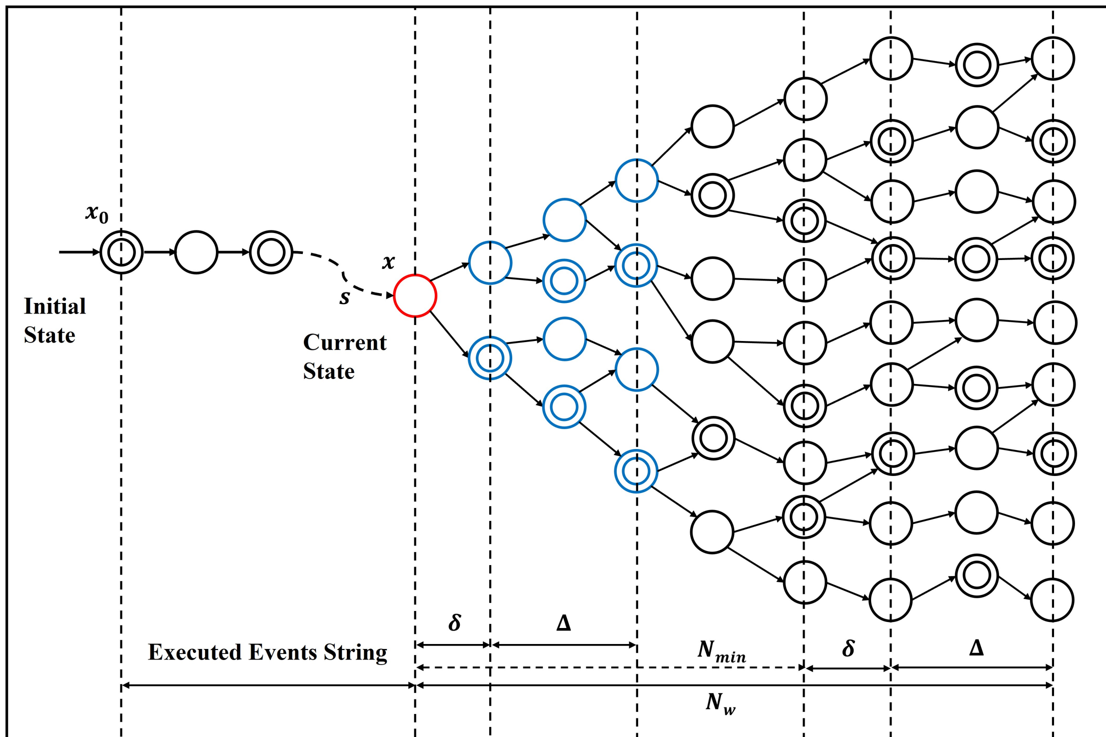


Figure 2.5: LLP with expanded window size [3].

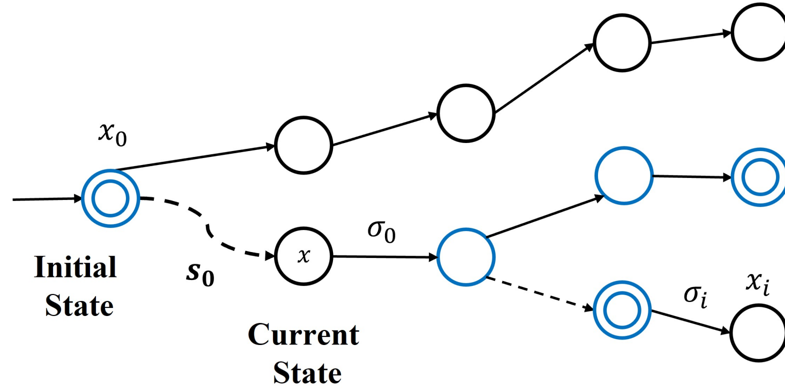


Figure 2.6: Tree Expansion [3].

are needed. Suppose these calculations start δ events before they are needed. Thus for LLP with Buffering the minimum window size is $N_w = N_{min} + \delta + \Delta$. The plant is expanded is by N_w to compute the supervisor which provides the control policy up to $\Delta + \delta$ events in the future (Fig. 2.7). It is known that for the window size N_{min} the supervisor is valid and optimal for immediate subsequent event. However, in LLP with Buffering the look-ahead window is expanded by additional $\delta + \Delta$ events in order to have the supervisory commands available for $\delta + \Delta$ events in the future. Therefore, it is imperative to have these $\delta + \Delta$ commands valid. To discuss the validity of these buffered events, let us consider the following theorem.

Theorem 2.2 ([3]). *If* $s \in \overline{K^{\uparrow}}$, *then*

$$(K^{\uparrow})/s = (K/s)^{\uparrow}.$$

□

In the above theorem, $K^{\uparrow}$ is the supremal controllable sub-language with respect to $L(G)$ and Σ_{uc}. Theorem 2.2 states that, the post-language of the supremal controllable sub-language $(K^{\uparrow})$ is equivalent to the supremal controllable sub-language of the post language (K/s).

Let us consider an example where a small portion of the expanded plant tree is shown in the Fig. 2.6. Suppose, from the initial state x_0 in plant G, the string s_0 is executed transitioning to the current state x. The string s_0 is permitted by the optimal supervisor. According to Theorem 2.2, after i steps permitted by the supervisor, any event σ_i also belongs to the supremal controllable

sub-language of the post language of s_0 [3].

Plant Depth

Plant Depth (PD) is defined as the minimum size of the look-ahead window for which the expanded model of the plant is equal to the plant model [3]. The value of PD depends on the state where the expansion is performed. For a plant, PD is the maximum of PD of all reachable states. In LLP with Buffering, the parameter N_w plays a very important role. If the parameter N_w is small, then the size of the expanded plant model will be small which will result in a short available computation time for LLP and small required memory. It is to be noted that PD depends on the characteristics of the system and cannot be adjusted like a parameter. PD provides a measure of efficiency for LLP with Buffering. If the parameter N_w has its value close to PD, then complete model of the plant is explored in LLP and the computed LLP supervisor will be similar to the conventional supervisor [5]. Thus in these situations, the efficiency of LLP implementation declines (requiring longer computation time and larger computer memory).

In order to take full advantage of LLP, the ratio of N_w to PD should be as small as possible. The efficiency of LLP is given as

$$\frac{N_w}{PD} \quad (N_w \geq N_{min}).$$

In [3], the PD for the solar tracker system is 11 and N_{min} is 6. As a result the efficiency of LLP for the solar tracker system is

$$\frac{N_{min}}{PD} = \frac{6}{11} = 0.54.$$

2.3.2 Buffer Size Selection

During LLP with Buffering, $\delta + \Delta$ number of events are buffered for future use, which takes care of the situation when events transpire in the system at a very rapid pace. Meanwhile, the CPU is available to perform another LLP with Buffering supervisory computation for the next window. However, there is a time constraint such that the computation must be completed before the buffer runs out empty. If the computation is not completed within the time constraint, then this could induce some delay which could be unacceptable in some systems. This case is similar to the periodic

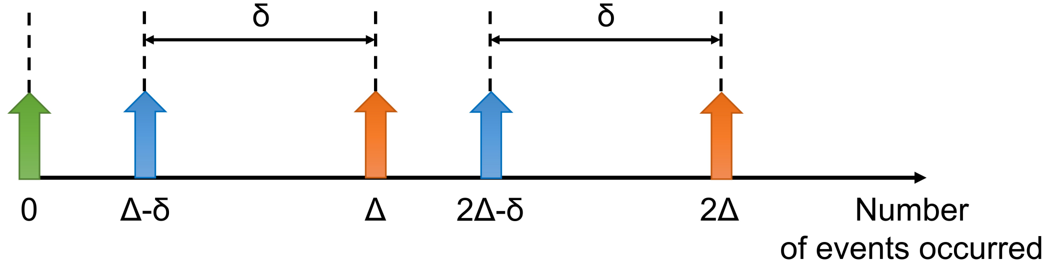


Figure 2.7: Timeline of LLP with Buffering. Taken and reproduced from [3].

processes in the real-time systems where the execution of the code of periodic process must be completed before the deadline or the period before the process repeats again. Let us first provide two definitions which are used to find a suitable buffer size.

Definition 2.13 ([5]). *Consider the automaton $G = (X, \Sigma, \eta, x_0, X_m)$. The function $T_{min}(n)$ returns the **minimum execution duration** of any sequence in $L(G)$ of length $n \in \mathbb{Z}^+$ and is given as*

$$T_{min} := \min\{T(v) \mid |v| = n \textit{ and } (\exists u \in \Sigma^*, uv \in L(G))\},$$

where $T(v)$ is the execution time of sequence v. □

Definition 2.14 ([5]). *$C_{max}(N_w)$ is the **maximum time** required for **LLP computation** over the window size of N_w.* □

The computation of LLP with Buffering must be performed in a timely fashion and within the time constraint in order to have the supervisory commands available whenever required without causing any unnecessary delay. To better explain this statement, consider the timeline of LLP with Buffering as illustrated in the Fig. 2.7.

Step (1) : In the initial step when $n = 0$ (shown by green arrow in the Fig. 2.7), the LLP supervisor is computed for a window $N_w = N_{min} + \Delta$. Following Theorem 2.2, Δ events are valid and stored in the buffer.

Step (2) : Let us denote δ to be the number of events left in the buffer of Δ events. After $\Delta - \delta$ events have passed (shown by blue arrow in the Fig. 2.7), the LLP supervisory computation commences for the window $N_w = N_{min} + \delta + \Delta$, calculating $\delta + \Delta$ number

of events for the interval $n = \Delta - \delta$ to $n = 2\Delta$. For this next LLP computation to be successful, it is required that the LLP computation for the window size N_w must be completed within δ number of events before n reaches Δ in the timeline. Therefore, the constraint

$$C_{max}(N_{min} + \delta + \Delta) \leq T_{min}(\delta) \quad \text{and} \quad \Delta > \delta.$$

Step (3) : When $n = \Delta$ (shown by orange arrow in the Fig. 2.7), the LLP supervisor is updated by the newly computed LLP supervisor consisting of supervisory commands available for the interval of $n = \Delta$ to $n = 2\Delta$. Similar to the Step (2), the LLP supervisor is again calculated at $n = 2\Delta - \delta$ (blue arrow), pre-calculating the control commands for next the interval followed by the Step (3) and so on.

In [3], LLP with Buffering is successfully implemented for two-degree-of-freedom solar tracker system.

2.4 Timed Automata

Timed automata (TA) is a dense-time and well-established models for modelling real-time systems [25, 26]. A TA has a set of real-valued clocks that essentially express the timing behaviour of the system. All clocks increase at an identical rate.

Let $C = \{c_1, \ldots, c_{n_e}\}$ be the finite set of clocks. Let $\mathbb{R}^+$ denote the set of non-negative real numbers. A **clock** is a real-valued function $\mathbb{R}^+ \rightarrow \mathbb{R}^+$ denoted by c. The dynamics of clock c is

$$\frac{dc(t)}{dt} = 1.$$

Thus at current time $t \in \mathbb{R}^+$, $c(t)$ is the value of c since the last time c was reset ti zero. The clock c can be reset to zero whenever a certain transition occurs in TA, thus tracking the time elapsed since the last time the clock was reset to zero.

A **clock constraint** is the conjunction of the atomic constraints over the clocks. Each constraint

takes the form of $c \sim k$, where $c \in C$, $\sim \in \{<, >, \leq, \geq, =\}$ and k is an non-negative integer constant. Let Φ_C denote the set of clock constraints over C.

2.4.1 Semantics of TA

In this section we will discuss the TA model as discussed in [1]. Our discussion of TA is not concerned with the marking of the modes therefore we omit the marking of the modes. A TA takes the form of a six-tuple

$$G_{TA} = (X, \Sigma, C, Inv, T, x_0),$$

where

- X is a finite set of modes;
- Σ is a finite set of events;
- C is a finite set of clocks;
- $Inv : X \to \Phi_C$ is a clock invariant function assigned to each mode $x \in X$;
- $T \subseteq X \times \Sigma \times \Phi_C \times 2^C \times X$ is the set of transitions. A transition from a mode x to x' upon the occurrence of event $\sigma \in \Sigma$ is represented as $\theta = (x, \sigma, g, u, x') \in T$, where $g \in \Phi_C$ is the guard for the transition θ and $u \subseteq C$ is the set of clocks to be reset to zero with this transition;
- $x_0 \in X$ is the initial mode.

The value of the clocks at time t is denoted by vector $\boldsymbol{c} = \{c_1, \ldots, c_{n_e}\}$, $\boldsymbol{c} \in (\mathbb{R}^+)^{n_e}$, where n_e is the total number of the clocks in G_{TA}. The state of the TA at time $t \in \mathbb{R}^+$ is the pair $q = (x(t), \boldsymbol{c}(t))$, where $x \in X$ is the mode at time t and $\boldsymbol{c}$ is the clock vector at time t. The n_e-dimensional unit vector is denoted by $\mathbf{1} = (1, 1, \ldots, 1)$. Therefore, for $\boldsymbol{c} \in (\mathbb{R}^+)^{n_e}$ and $d \in \mathbb{R}$, $\boldsymbol{c} + d \cdot \mathbf{1}$ is denoted by $\boldsymbol{c} + d$. In the initial mode x_0 all clocks are set to zero at time $t = 0$. Therefore, the initial state of G_{TA} is $q_0 = (x_0, \mathbf{0})$, where $\mathbf{0}$ is the zero vector. The transition $\theta = (x, \sigma, g, u, x')$ from x to x' is represented with the transition label as (σ, g, u), where $\sigma \in \Sigma$, $g \in \Phi_C$ and $u \subseteq C$.

g is a clock constraint. If θ has no clock constraint, then the guard is omitted and replaced by "-" in the figure. Similarly, if no clock is reset as the result of transition, u is empty and replaced with "-".

The absence of the Inv condition at a mode means that the invariant of the mode is always *true* at any point of time when the control is at that mode and TA can halt its operation and remain at that mode for an arbitrarily long time and remain indefinitely. For the sake of simplicity, the bounds on the clocks that appear in the guard conditions and the clock invariant at the modes are non-negative integers. It is to be noted that the state space of G_{TA} is infinite (since the clock are real-valued).

In the expression of the invariant of mode x, the constraint on a clock $c \in C$ is of the form $c < k$ (or $c \leq k$) which specifies the *upper time bound* for c such that the mode x should be left when c reaches k. On the other hand, the invariant of x with the constraint of the form $c > k$ (or $c \geq k$) specifies the *lower time bound* on c such that the mode x can be left when c is greater than (or greater than or equal to) k. Similarly, a guard condition g of a transition can take either of the above forms for the clock constraints. Note that the guard condition can have the clock constraint as $c = k$ stating that the transition can occur at the particular moment when the value of the clock c equals k. The TA model needed in the thesis is a subclass of the standard definition of the TA in [26].

2.4.2 Execution of TA

Let us now discuss the dynamical evolution of TAs. For the rest of this section, consider a TA G_{TA}. G_{TA} has two types of state change starting from the initial state $q_0 = (x_0, \mathbf{0})$.

(1) *Timed Action*: In timed action, for a duration $d \in \mathbb{R}^+$, the time is progressed from $\boldsymbol{c}$ to $\boldsymbol{c'} = \boldsymbol{c} + d, d > 0$ and the state of G_{TA} changes from $(x, \boldsymbol{c})$ to $(x, \boldsymbol{c'})$. Since, there is no execution of event, the mode remains unchanged and only the time has progressed. Thus updating the state of the G_{TA}. The timed action is given as

$$(x, \boldsymbol{c}) \xrightarrow{d} (x, \boldsymbol{c'}).$$

A timed action at mode x is admissible iff $\boldsymbol{c'}$ satisfies $Inv(x)$.

(2) *Event transition*: Consider the transition $\theta = (x_1, \sigma, g, u, x_2) \in T$ from mode x_1 to x_2. Event σ is executed iff $\boldsymbol{c}_1 \in g$ is evaluated to *true* (the guard of θ is satisfied). On taking the transition, the set of clocks in u are reset to zero while leaving rest of the clocks unchanged resulting in $\boldsymbol{c}_2$. The event transition is given as

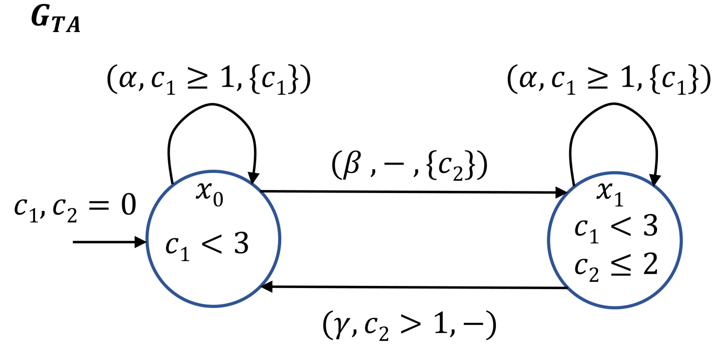


Figure 2.8: Timed Automaton Example G^1_{TA} [4].

$$q_1 = (x_1, \boldsymbol{c}_1) \xrightarrow{\sigma} q_2 = (x_2, \boldsymbol{c}_2).$$

Definition 2.15. *A* ***run*** *(execution) of the TA G_{TA} starting from the initial state $q_0 = (x_0, \mathbf{0})$ is the sequence of timed actions and event transitions of the form*

$$(x_0, \mathbf{0}) \xrightarrow{d_1} (x_0, \boldsymbol{c}'_1) \xrightarrow{\sigma_1} (x_1, \boldsymbol{c}_1) \cdots \xrightarrow{\sigma_n} (x_n, \boldsymbol{c}_n).$$

□

Example 2.3. *Let us take a simple example of TA illustrated in Fig. 2.8. In this system, there are two modes x_0 and x_1, three events α, β and γ and two clocks c_1 and c_2. The invariant at x_0 is defined for c_1 and the invariant at x_1 is defined for both c_1 and c_2. In both modes, the constraint on c_1 is such that α is issued repeatedly with interval between two consecutive events being greater than or equal to 1 and strictly less than 3. The constraint involving clock c_2 in mode x_2 is such that as soon β event occurs, γ should occur within a duration of length strictly greater than 1 and less than or equal to 2. The execution of system starts at mode x_0 at time $t = 0$, with both clocks at $c_1 = 0$ and $c_2 = 0$. It is to be noted that in both the modes there are two transitions and at any time t only one transition can occur. In mode x_0, β can occur anytime since its guard condition is always true. Consider the following run of the G^1_{TA} given below. The clock vector is shown as $[c_1, c_2]$.*

$$(x_0, [0,0]) \xrightarrow{1.1} (x_0, [1.1, 1.1]) \xrightarrow{\alpha} (x_0, [0, 1.1]) \xrightarrow{0.6} (x_0, [0.6, 1.7]) \xrightarrow{\beta} (x_1, [0.6, 0])$$
$$\xrightarrow{0.4} (x_1, [1, 0.4]) \xrightarrow{\alpha} (x_1, [0, 0.4]) \xrightarrow{0.8} (x_1, [0.8, 1.2]) \xrightarrow{\gamma} (x_0, [0.8, 1.2]). \quad (1)$$

In a run of TA, if we remove the information about the timed action and only keep the information of event labels and their time of occurrence in the run, then the resulting sequence will be a timed trace [1]. The timed trace for the run in 1 is

$$(\alpha, 1.1), (\beta, 1.7), (\alpha, 2.1), (\gamma, 2.9). \quad (2)$$

□

2.5 Clock Regions

Most of the analysis algorithms performed on automaton require that the automaton have a finite state space. However, the state space of TA is infinite and it is not possible to build an automaton whose states are the states of the TA. However, if two states of the TA have the same mode that agree on the integral part of all the clock values as well as on the ordering of the fractional part of the clock values, then the run of the TA obtained from these two states is closely similar [25]. The integral part of the clocks is used to determine if any enabling condition is met whereas, the fractional part of the clocks is used to determine which of the clock will reach its integral part first. The values of the clocks are clustered into finitely many equivalence classes such that the states that are equivalent behave similarly [4]. This equivalence is known as *region equivalence*.
Let k_i denote the largest integer constant that the clock variable c_i is compared with in the atomic constraints that appears in the guards or invariant condition in TA G_{TA}.

Definition 2.16 ([4]). *Given a TA G_{TA}, two clock valuations* **c** *and* **c′** *of G_{TA} are* ***region-equivalent*** *if and only if:*

(1) for every $i \in \{1, \ldots, l\}$, and for any $d \in \mathbb{Z}^+$ with $0 \leq d \leq k_i$, $c_i = d$ iff $c'_i = d$, and $c_i < d$

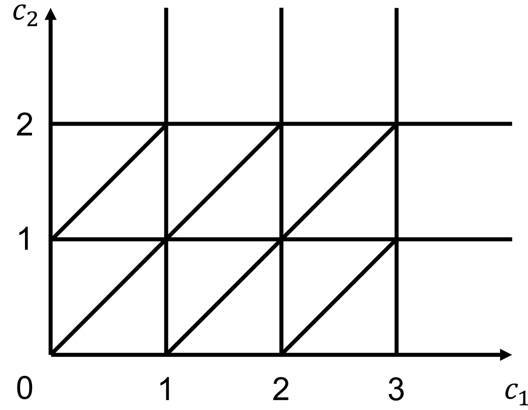


Figure 2.9: Clock Regions of G^1_{TA}

iff $c'_i < d$, *and*

(2) for every $i, j = 1 \ldots l$, *with* $c_i \leq k_i$ *and* $c_j \leq k_j$, *the fractional part of* c_i *is less than equal to the fractional part of* c_j *if and only if the fractional part of* c'_i *is less than equal to the fractional part of* c'_j.

□

If two clock valuations belong to the same partition, then they are *region-equivalent*. Region equivalence is extended to the points in the space $(\mathbb{R}^+)^{n_e}$ such that two states of the TA are region-equivalent if their corresponding modes are same and the values of the clocks are region-equivalent. A *clock region* is a partition that consist of all region-equivalent clock valuations. Note that there are only a finite many regions. Specifically, the number of possible clock regions is at most

$$2^{|C|} \cdot |C|! \cdot \prod_{c} (2k_c + 2). \tag{3}$$

(See Lemma 4.5 in [26]). This number increases exponentially with the number of clocks used in the TA (i.e., $|c|$)and also increases proportionally to the number of constraint constants used (i.e., k_i).

Example 2.4. *Recall the TA* G^1_{TA} *from Fig. 2.8.* G^1_{TA} *consist of two clocks* c_1 *and* c_2 *with* $k_1 = 3$ *and* $k_2 = 2$. *The clock regions for* G^1_{TA}, *shown in Fig. 2.9, are:*

- *12 Corner points: e.g.* $[(1, 0)]$*;*

- *30 Open line segments: e.g.* $[(0 < c_1 < 1), (c_2 = 0)]$*;*

- *18 Open regions: e.g.* $[(0 < c_2 < c_1 < 1)]$.

□

Using region-equivalence, the infinite space of clocks is partitioned into finitely many clock regions. For example in [26], the clock regions are used to form the region automaton which describes the behaviour of the TA using a finite state space. In [4] the search using the region-equivalence is used to determine if a particular property is satisfied by the TA.

2.6 Clock Zones and Difference Bound Matrices

In this section we shall briefly discuss the concept of clock zones. More details can be found elsewhere in [4, 31, 47, 48]. Until now, we have seen that for a TA, there are finitely many clock regions and the number of clock regions is bounded by expression (3). Individual states are grouped as clock regions and these clock regions are called *abstract-states*. However such abstract states can grow exponentially in the number of clocks used and proportionally with the constants appearing in atomic constraints. Further abstraction is possible by efficiently representing a cluster of clock regions by a *clock zone* which provide a coarser and compact representation of clock regions.
A clock zone Z is any convex polyhedron in $(\mathbb{R}^+)^{n_e}$, where n_e is the number of the clocks. A clock zone is the set represented by a set of clock constraints each of which can take one of the following forms [31]:

- $c \sim k$ where $c \in C$, $\sim \in \{\leq, <, \geq, >\}$ and k is an integer constant,

- $c - c' \leq k$ or $c - c' < k$ where $c, c' \in C$ and k is an integer constant.

A clock zone Z consists of integer constraints on clock values and on difference between two clocks. Suppose TA has n_e clocks. Then we shall introduce a fictitious clock c_0 that has the constant value 0. Thus the set of the clocks is $C_0 = C \cup \{c_0\}$. A fictitious clock is used for uniform representation of clock constraints as upper bound on the difference between two clock values. For example, the

clock constraint $c \leq 3$ can be represented as $c - c_0 \leq 3$. A clock zone Z is represented by a square matrix R of dimension $(n_e + 1) \times (n_e + 1)$ with rows and columns indexed from 0 to n_e, with index 0 used for c_0. The upper bound on the difference between the clocks c_i and c_j $(c_i - c_j)$ is represented by the (i, j)th entry in the matrix R and denoted by R_{ij}.
The entries in first column R_{i0} provides the upper bound on the clocks c_i and the entries in the first row provides the upper bound on the clocks $-c_i$. Thus, the clock zone Z is effectively represented by the matrix R which is a data structure called *difference bound matrices* (DBMs) [49, 50].

Canonicalization of DBMs

Every clock zone can be represented by DBM. It is possible that many DBMs represent the same clock zone because the upper bound entries in the DBM are not tight. Consider a DBM R that represents a clock zone Z. The upper bound on the constraint $(c_i - c_j)$ of Z is given by the entry R_{ij} and the upper bound on the difference between the clocks $(c_i - c_m)$ and $(c_m - c_j)$ is given by the entries R_{im} and R_{mj} respectively. The sum of the entries $(R_{im} + R_{mj})$ is referred as the *inferred* upper bound on the difference $(c_i - c_j)$. If the sum $(R_{im} + R_{mj})$ is less than the entry R_{ij}, then the upper bound R_{ij} can be replaced with the inferred upper bound $(R_{im} + R_{mj})$ [4]. If all the entries of the DBM R are tight then DBM R is a **canonical** matrix. This structure is important for testing emptiness of a clock zone.
DBM R is canonical if and only if

$$\forall i, j, m = 1, \ldots, e, \quad (R_{im} + R_{mj}) \geq R_{ij}.$$

The canonical matrix for Z can be obtained by applying an all-pairs shortest path algorithm on any arbitrary matrix for Z [49]. Algorithm 2.1 is the classical Floyd-Warshall shortest-path algorithm which is an all-pairs shortest path algorithm for computing the canonical matrix [51].
Consider an example, where $n_e = 3$ and the clock zone is given by the constraints as follows

$$(2 \leq c_1 \leq 4) \wedge (c_2 \geq 2) \wedge (0 \leq c_3 \leq 4) \wedge (c_2 - c_3 = 2) \wedge (c_2 - c_1 \geq 3)\,. \tag{4}$$

The clock zone described above can be represented by DBM R as follows

Algorithm 2.1 Algorithm for Canonicalization of DBMs [4]

```
1: procedure CANONICAL(R)
2:     Input: (n_e + 1) × (n_e + 1) DBM R with entries in ℤ ∪ {∞}.
3:     Output: Empty if R is empty, otherwise canonical version of DBM R.
4:     for m = 0 to n_e do
5:         for i = 0 to n_e do
6:             for j = 0 to n_e do
7:                 R[i, j] := min(R[i, j], R[i, m] + R[m, j]);
8:             end for
9:         end for
10:    end for
11:    if R[i, i] < 0 then
12:        return Empty;
13:    end if
14:    return R.
15: end procedure
```

$$R = \begin{bmatrix} 0 & -2 & -2 & 0 \\ 4 & 0 & -3 & \infty \\ \infty & \infty & 0 & 2 \\ 4 & \infty & -2 & 0 \end{bmatrix}.$$

Note that constraints on some $c_i - c_j$ are absent. For example, no constraint on $c_1 - c_3$ is in (4). Thus we can say $c_1 - c_3 \leq \infty$ and the corresponding entry in R is ∞. The canonical DBM obtained by applying Algorithm 2.1 to the above DBM R through is given as:

$$R_{can} = \begin{bmatrix} 0 & -2 & -5 & -3 \\ 3 & 0 & -3 & -1 \\ 6 & 4 & 0 & 2 \\ 4 & 2 & -2 & 0 \end{bmatrix}.$$

Here is the procedure to obtain the canonical DBM of a clock zone. Consider a clock zone Z with the constraint $c_i \leq k$. Then DBM R representing the clock zone Z is constructed as follows

- For the constraint $(c_i - c_0) \leq k$ the entry $R_{i0} = k$ to reflect the upper bound on the clock constraint;
- The difference between the clocks $(c_j - c_j)$ is always 0 such that the entry $R_{jj} = 0$ for all $j = 0, \ldots, n_e$;

- The implicit assumption that all clocks are always positive $(c_j) \geq 0$ holds i.e $(c_0 - c_j) \leq 0$ such that the entry $R_{0j} = 0$ for all $j = 0, \ldots, n_e$;
- Finally, for all the remaining clock differences that are unbounded in Z the entries $R_{jm} = \infty$.

DBM R is converted to a canonical DBM R by using Algorithm 2.1.

2.6.1 Operations on DBMs

Clock zones are represented using DBMs. In order to use clock zones to study the dynamics of TA, operation on clock zones and the corresponding DBMs are required. In this section we shall review some of the operations performed on DBMs [4].

Intersection

Consider two canonical DBMs R and R' representing clock zones Z and Z'. The intersection between two zones $Z \cap Z'$ represented by the DBM A is constructed by setting the entry of A_{ij} to the result of the minimum between R_{ij} and R'_{ij}, i.e., for all $0 \leq i, j \leq n_e, A_{ij} = \min(R_{ij}, R'_{ij})$. The resulting DBM A is tested for emptiness and is made canonical if it is not empty.

Time Elapse

Consider the canonical DBM R that represents the clock zone Z. The passage of time for the clock zone Z due to the timed action is obtained by removing all the constraints of the form $c_i \leq k$ (since these upper bounds permits the passage of time indefinitely) by setting the corresponding entry $R_{i0} = \infty$ for all $i = 1, \ldots, n_e$. During timed action, the entries in R_{0i} for each $1 \leq i \leq n_e$ and the entries in R_{ij} for each $1 \leq i, j \leq n_e$ remain the same.

Resetting of Clocks

Suppose there are n_e clocks in the TA. Then the set of *reset actions* is $\mathcal{U}(n_e)$, which are functions from $(\mathbb{R}^+)^{n_e}$ to $(\mathbb{R}^+)^{n_e}$ corresponds to resetting some of the clocks to 0. For each $u \in \mathcal{U}(n_e)$, there is a set of indexes $I_u \subseteq \{1, \ldots, n_e\}$ such that

Algorithm 2.2 Algorithm for Resetting Clock Variables

```
1: procedure RESETCLOCK(DBM D,I_u)
2:     Input: (n_e + 1) × (n_e + 1) DBM D with entries in ℤ ∪ {∞}.
3:     Input: I_u reset index set.
4:     Output: (n_e + 1) × (n_e + 1) canonical DBM R.
5:     R := D;
6:     for c_{σ_i} ∈ I_u do
7:         R[i, 0] = 0;
8:         R[0, i] = 0;
9:         for j := 1 to n_e do
10:            R[i, j] := R[0, j];
11:            R[j, i] := R[j, 0];
12:        end for
13:    end for
14:    return R.
15: end procedure
```

$$\forall \boldsymbol{c} \in (\mathbb{R}^{+})^{n_e}, \forall i = 1, \ldots, n_e, \quad u(c_i) = \begin{cases} 0 & \text{if } i \in I_u, \\ c_i & \text{otherwise .} \end{cases}$$

For a canonical DBM R if clock c_i is reset to zero, then $R_{i0} := 0$, $R_{0i} := 0$. Also, the constraints of the form $c_i - c_j \leq R_{ij}$ and $c_j - c_i \leq R_{ji}$ are replaced with the upper and lower bounds of c_j such that

$$R_{ij} := R_{0j}, R_{ji} := R_{j0} \quad \forall j \neq i, 0.$$

Algorithm 2.2 performs the resetting of clocks. Consider a canoncal DBM R given below

$$R = \begin{bmatrix} 0 & -2 & -5 & -3 \\ 3 & 0 & -3 & -1 \\ 6 & 4 & 0 & 2 \\ 4 & 2 & -2 & 0 \end{bmatrix}.$$

After resetting the clock $c_2 = 0$, R will be updated by applying Algorithm 2.2 to R. This results in the DBM given below:

$$R = \begin{bmatrix} 0 & -2 & 0 & -3 \\ 3 & 0 & 3 & -1 \\ 0 & -2 & 0 & -3 \\ 4 & 2 & 4 & 0 \end{bmatrix}.$$

Now, we shall discuss the transformation of a zone and its corresponding DBM as a result of the timed action and transition from a mode. A symbolic state of the TA is a pair (x, R), where x is the mode and R is DBM representing a zone Z. We will denote this symbolic state as zone to avoid confusion with the state of the TA. For a zone (x, R) there are two possible successors: (1) the effect of time elapse as a result of timed action in the mode, and (2) the successor clock zone to Z as a result of transition from a mode.

In the first case, DBM R corresponding to clock zone Z is updated by setting R_{i0} to ∞ except R_{00} to reflect the time elapse. Then a separate DBM R' is formed according to the $Inv(x)$ (atomic constraints at mode x). Next, intersection between R and R' is taken. Finally, the resulting DBM is tested for emptiness and is made canonical if not empty.

Let us consider our previous example. For our running example recall DBM R_{can}. The canonical DBM R_{can} is updated by letting time elapse as shown by above matrix R.

$$R = \begin{bmatrix} 0 & -2 & -5 & -3 \\ \infty & 0 & -3 & -1 \\ \infty & 4 & 0 & 2 \\ \infty & 2 & -2 & 0 \end{bmatrix}.$$

Suppose, the invariants at the mode only involves the clock constraint $c_1 \leq 5$. The entry R'_{10} is set to 5. Since, there are no clock invariants involving clocks c_2 and c_3, the corresponding entries R'_{20} and R'_{30} will be replaced with ∞ shown by the DBM R'.

$$R' = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ \infty & 0 & 0 & 0 \\ \infty & 0 & 0 & 0 \end{bmatrix}.$$

The DBMs R and R' are intersected which results in the matrix given below.

$$R \cap R' = \begin{bmatrix} 0 & -2 & -5 & -3 \\ 5 & 0 & -3 & -1 \\ \infty & 4 & 0 & 2 \\ \infty & 2 & -2 & 0 \end{bmatrix}.$$

Algorithm 2.3 Algorithm for time successors corresponding to clock invariant conditions

```
1: procedure CLOCKINVARIANT(DBM R, Inv(x))
2:     Input: (n_e + 1) × (n_e + 1) DBM D with entries in ℤ ∪ {∞}.
3:     Input: Inv(x) Clock Invariant conditions at mode x.
4:     Output: (n_e + 1) × (n_e + 1) DBM R.
5:     for i = 1 to n_e do
6:         if (c_{σ_i} ∈ Inv(x)) then
7:             R[i, 0] := u_{σ_i};
8:         else
9:             R[i, 0] := ∞;
10:        end if
11:    end for
12:    return R.
13: end procedure
```

Finally, the DBM is made canonical.

$$E = \begin{bmatrix} 0 & -2 & -5 & -3 \\ 5 & 0 & -3 & -1 \\ 9 & 4 & 0 & 2 \\ 7 & 2 & -2 & 0 \end{bmatrix}.$$

We present a slight modification to capture the time elapse due to the invariants at the mode. Instead of following the above mentioned procedure, in the DBM R the upper bounds on the clocks in the first column are directly modified as

$$\forall i = 1, \ldots, n_e, \quad R_{i0} := \begin{cases} k_i & \text{if } c_i \in Inv(x), \\ \infty & \text{otherwise }, \end{cases} \tag{5}$$

where k_i is the integer constant appearing in the invariants of the mode. The rest of all the entries in R remain same. Then R is checked for emptiness and if not empty, then R is made canonical. Algorithm 2.3 performs the modification on R according to equation (5). Algorithm 2.3 combines the two steps of forming a separate DBM for the invariants and then intersecting it with R.

For our running example, consider the same clock constraint $c_1 \leq 5$ and canonical DBM R_{can}. Suppose R is given as the matrix below with the modifications in the first column (shown in red) when Algorithm 2.3 is applied.

$$R = \begin{bmatrix} 0 & -2 & -5 & -3 \\ 5 & 0 & -3 & -1 \\ \infty & 4 & 0 & 2 \\ \infty & 2 & -2 & 0 \end{bmatrix}.$$

It can be observed that the entry R_{10} is set to 5. Rest all the entries in the first column of R (except the first entry in the first column of R) are ∞ due to the absence of clock constraints on their corresponding clocks. Then R is canonicalized resulting in DBM E'. It can be noted that canonical DBM E' given by below matrix is the same canonical DBM E as seen previously.

$$E' = \begin{bmatrix} 0 & -2 & -5 & -3 \\ 5 & 0 & -3 & -1 \\ 9 & 4 & 0 & 2 \\ 7 & 2 & -2 & 0 \end{bmatrix}.$$

For the second case, the clock zone and its corresponding DBM are updated due to the transition at a mode. Suppose R is the DBM at a mode and R' is the DBM that captures clock constraint in the guard of the corresponding transition. In the next step, DBMs R and R' are intersected. If the intersection is not empty, then the resulting DBM is made canonical. This also means that the corresponding transition is taken. To illustrate this procedure, recall the canonical DBM R_{can} from our running example.

$$R_{can} = \begin{bmatrix} 0 & -2 & -5 & -3 \\ 3 & 0 & -3 & -1 \\ 6 & 4 & 0 & 2 \\ 4 & 2 & -2 & 0 \end{bmatrix}.$$

Now, suppose that the transition consist of a guard which involves the clock constraint $c_1 \geq 3$. This clock constraint is translated to the DBM R'_{can}.

$$R'_{can} = \begin{bmatrix} 0 & -3 & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{bmatrix}.$$

The intersection of R_{can} and R'_{can} results in the matrix:

$$R_{can} \cap R'_{can} = \begin{bmatrix} 0 & -3 & -5 & -3 \\ 3 & 0 & -3 & -1 \\ 6 & 4 & 0 & 2 \\ 4 & 2 & -2 & 0 \end{bmatrix}.$$

After canonicalization:

$$F = \begin{bmatrix} 0 & -3 & -6 & -4 \\ 3 & 0 & -3 & -1 \\ 6 & 4 & 0 & 2 \\ 4 & 2 & -2 & 0 \end{bmatrix}.$$

There is an alternate way to simplify the above operation. Consider a DBM R. The entry in R corresponding to the clock constraint that appears in the guard of the transition can be replaced as

$$R_{0i} := \min(R_{0i}, -k_i) \tag{6}$$

where k_i is the integer constant corresponding to the clock c_i appearing in the guard of a transition. After this modification, R is made canonical if not empty. This eliminates the step of forming another DBM just to capture the guard of the transition. We illustrate this modification with an example. Recall canonical DBM R_{can} and the guard condition $c_1 \geq 3$. The modifications according to equation (6) is performed directly on DBM R_{can} (highlighted in red in the matrix below) resulting in the DBM R shown by the matrix below.

$$R = \begin{bmatrix} 0 & -3 & -5 & -3 \\ 3 & 0 & -3 & -1 \\ 6 & 4 & 0 & 2 \\ 4 & 2 & -2 & 0 \end{bmatrix}.$$

It can be observed that only R_{01} is set to -3 and rest all the entries in R remain same. After canonicalization:

$$F' = \begin{bmatrix} 0 & -3 & -6 & -4 \\ 3 & 0 & -3 & -1 \\ 6 & 4 & 0 & 2 \\ 4 & 2 & -2 & 0 \end{bmatrix}.$$

It can be observed that canonical DBM F' is same as canonical DBM F. These modifications are conducted with the objective to eliminate the formation of another separate DBM that captures the clock constraints and to eliminate the step of carrying intersection of the two DBMs. Operations on clock zones (and the corresponding DBM operations) can be used in the reachability analysis of TAs. This is an important tool for us in the next chapter.

2.7 Summary

In this chapter we reviewed DES, supervisory control of DES, LLP and LLP with Buffering. We also reviewed TA, clock regions, clock zones and their DBMs and operations on DBM. In the next chapter we will present a transformation procedure to augment an untimed automaton to a TA. We will also present specific clock adjustments made to the TA in order to calculate $T_{min}(\delta)$ and an algorithm to calculate $T_{min}(\delta)$.

Chapter 3

Model-Based Estimation of the Duration of Event Sequences

As discussed in Section 2.3 and following the procedure described there, in order to choose a suitable buffer size for LLP, the fastest rate of event generation in the plant is needed. Specifically, for any positive integer δ, the shortest duration of generation of δ events in the plant, $T_{min}(\delta)$, is required. The function $T_{min}(\delta)$ can be obtained experimentally by running tests on the plant. The tests cannot be exhaustive and therefore, can provide an upper bound for $T_{min}(\delta)$

If the timing information of the plant events is available, then one could incorporate this information in the plant model to obtain a timed automaton model (TA) and then use the TA to find $T_{min}(\delta)$. This procedure, however, entails enumerating through all the clock regions from every mode of the TA, a process which has a time complexity exponential in the number of events. In this chapter we propose an algorithm which involves a reachability analysis linear in the number of plant events; however, the algorithm provides a lower bound for $T_{min}(\delta)$.

In Section 3.1, presents the problem formulation. Section 3.2 discusses a procedure to transform an untimed automaton to TA when the timing details of the events is available. In Section 3.3 develops a matrix-based symbolic analysis algorithm that finds a lower bound value for $T_{min}(\delta)$. In Section 3.4 presents an example to illustrate the calculation of $T_{min}(\delta)$.

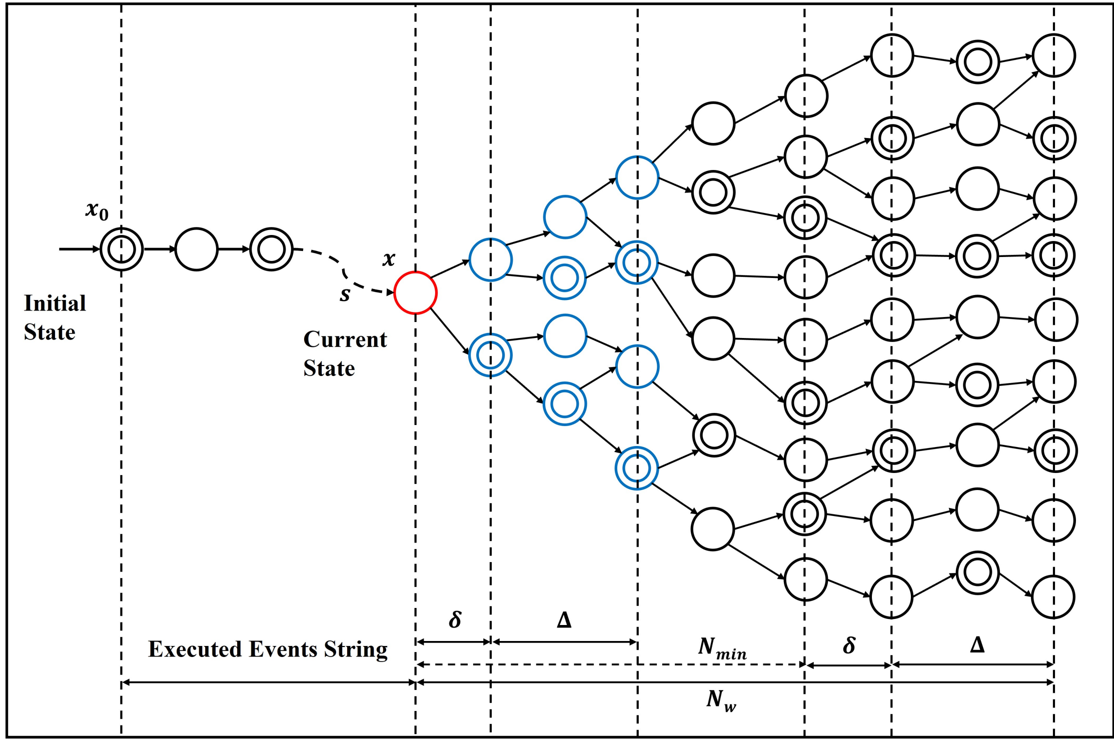


Figure 3.1: LLP with Buffering for the window size N_w [3].

3.1 Problem Formulation and Solution Overview

In Section 2.3, we reviewed LLP with Buffering. Fig. 3.1 illustrates the calculation of supervisory commands in LLP with Buffering with N_w denoting the window size. It can be observed that the model of the plant is expanded from the current state for the window size of $N_w = N_{min} + \Delta + \delta$, where N_{min} is the minimum depth of the plant that should be expanded for the validity of the LLP supervisor. Δ is the number of command events to be buffered and δ is the number of events left in the buffer ($\delta < \Delta$).

In a plant certain events can be generated at a very rapid pace such that there might not be sufficient time available between two successive events to calculate the supervisory commands. Therefore, in LLP with Buffering, Δ number of supervisory commands are precalculated online in a timely fashion and are buffered for future use. In order to calculate the supervisory commands, it is sufficient to respect the constraint $C_{max}(N_w) \leq T_{min}(\delta)$. In other words, the supervisory computation for the window size of N_w must be completed within the time for the fastest generation of δ events in the system.

$T_{min}(\delta)$ can be calculated experimentally by running tests on the physical system. One major issue in calculating $T_{min}(\delta)$ experimentally is the impossibility of performing exhaustive test to determine $T_{min}(\delta)$. Generally, the experimentally calculated $T_{min}(\delta)$ is an overestimate of the actual $T_{min}(\delta)$. Alternatively, if information regarding the time bounds of the events, namely, lower time bound and upper time bound is available, then one other possible solution is to incorporate this timing information into the untimed model and construct a TA model for the physical system. In this TA, each mode corresponds to a state of the untimed model and each clock corresponds to an event of the untimed model. This TA model of the system can then be used to determine $T_{min}(\delta)$ theoretically (i.e. a model-based approach).

A straightforward approach to determine $T_{min}(\delta)$ theoretically is by exploring δ number of events from each mode of the TA and then determining which mode has the minimum execution time (of δ events). This approach, although straightforward, is nevertheless challenging the set of clock values reachable at every mode needs to be found to determine the minimum execution time of δ events. The required calculations are very complex as explained in the following. One possible approach is to use the region-based abstraction of the TA which permits partitioning of the infinite space of clock values into finitely many clock regions [4]. A *region* specifies a mode and a clock region. Using regions, one can perform region-based search algorithm for determining minimum execution duration of δ events from every reachable regions of the TA. Although, clock regions for the TA are finite, the number of clock regions grow exponentially with the number of events and proportionally to the product of the largest constant appearing in the guards or invariants of the TA [4]. In addition, the search algorithm would require to perform two tasks: (1) enumerate through all the reachable regions from the initial mode of TA and then explore δ events through those reachable regions, and (2) find a particular reachable region (which consist of a specific mode and a specific clock region) such that exploring δ events from that particular reachable region would result in the minimum execution of δ events and hence theoretical $T_{min}(\delta)$. Therefore, calculating $T_{min}(\delta)$ using the region-based abstraction (could become very complex) since the number of clock region are exponential in the number of the events.

To better illustrate this point, let us discuss an example Example 3.1. To avoid region-based abstraction, in Section 3.3 we propose a specific setting for the clock variable of each event at each

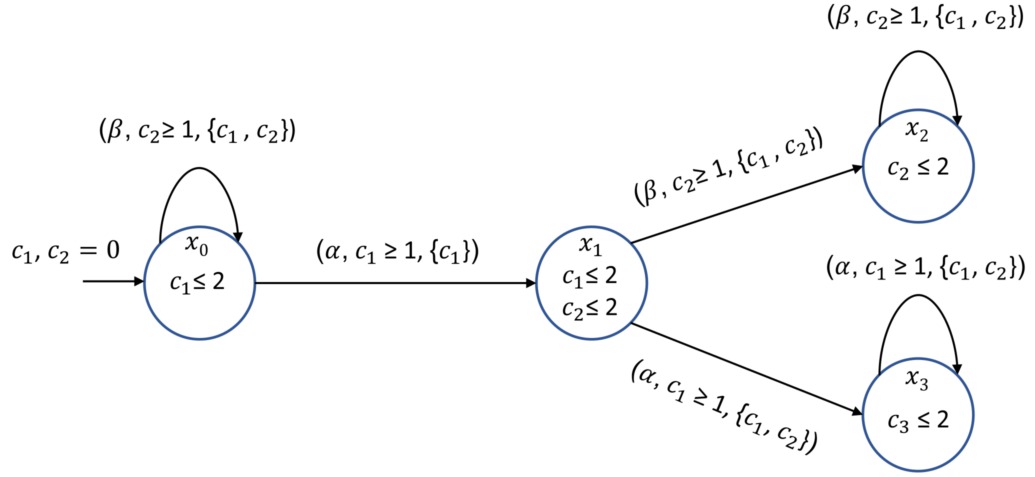


Figure 3.2: TA A_1.

mode based on which a lower bound for $T_{min}(\delta)$ is calculated. In Section 3.5 we propose a zone-based search algorithm for calculating $T_{min}(\delta)$ which uses the modifications made to the clock variables at every mode of the TA. In Section 3.6, we show that the proposed zone-based search algorithm requires time complexity which is polynomial in the number of modes of TA (as a result of the modifications made to the clock variables) for calculating the lower bound for $T_{min}(\delta)$.

Example 3.1. *Consider a TA A_1 shown in Fig. 3.2. TA A_1 consists of four modes and two events. Mode x_0 is the initial mode. The time bounds of the events are given as $(\alpha, 1, 2)$ and $(\beta, 1, 2)$. The clocks for events α and β are c_1 and c_2 respectively. Edges labelled with the symbol τ corresponds to the successor regions due to timed actions. Fig. 3.3 illustrates all the reachable clock regions for TA A_1. In this example, there are four modes and 46 clock regions resulting in a total of 184 possible regions. However, a depth-first search shows only 21 regions are reachable (Fig. 3.3). In order to calculate $T_{min}(\delta)$ from every reachable region; δ number events must be explored. If the initial region $[x_0, c_1 = c_2 = 0]$ is selected, then δ number of event sequences are required to be explored with the clock setting as $(c_1 = c_2 = 0)$ and minimum execution duration amongst all event sequences should be selected. Similarly, this process must be repeated for every reachable region in Fig. 3.3. Finally, a particular region from all the reachable regions with minimum execution duration of δ events provides $T_{min}(\delta)$. In this example, the $T_{min}(\delta)$ is obtained from initial region*

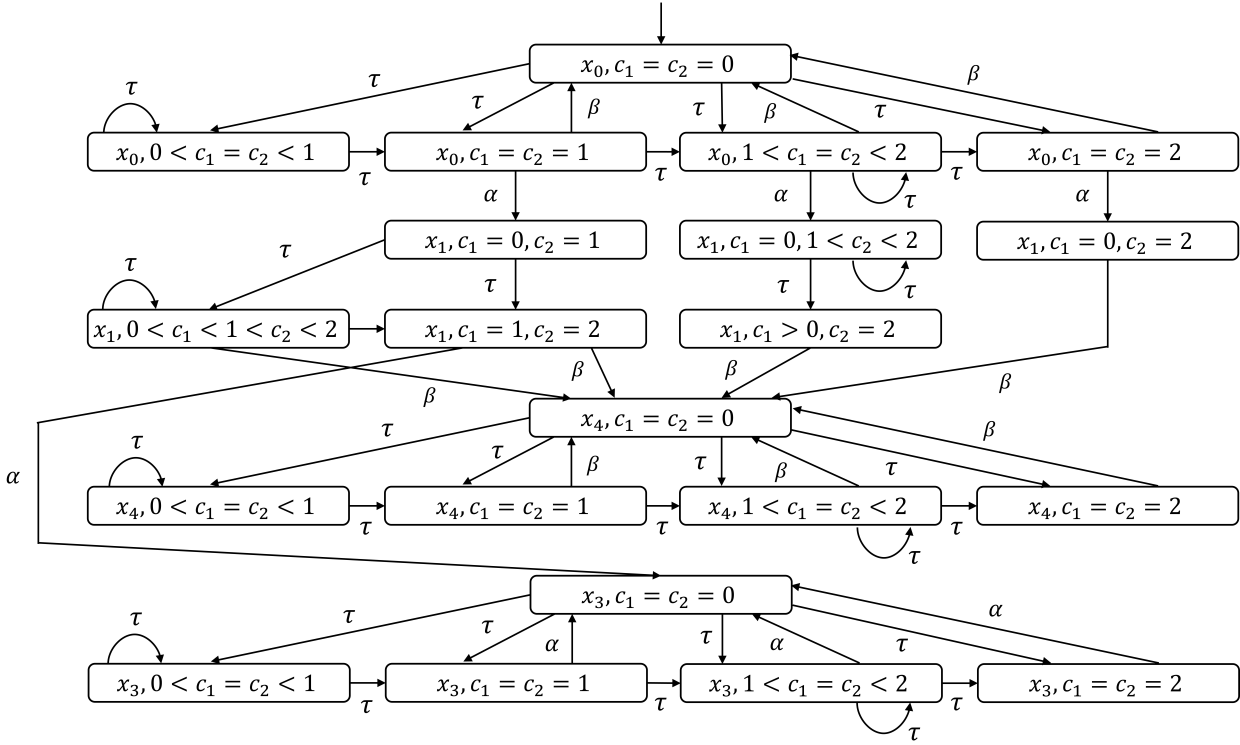


Figure 3.3: Reachable clock regions of TA A_1.

which results from the event sequences $\alpha\beta\beta^*$ and is given as

$$T_{min}(\delta) = \begin{cases} 0, & \delta = 1, 2 \\ \delta - 2. & \delta \geq 3 \end{cases}$$

□

It can be observed from the Fig. 3.3 that just for a four state TA A_1 there are 21 regions. The regions increases drastically if a TA consists of large number of modes. In addition, the number of clock regions increases in the worst case, exponentially in the number of clocks. Even though, $T_{min}(\delta)$ obtained using region-based abstraction can be very accurate, the time complexity for calculating $T_{min}(\delta)$ is exponential in the number of clocks and thus becomes infeasible when the TA has a large number of modes and events.

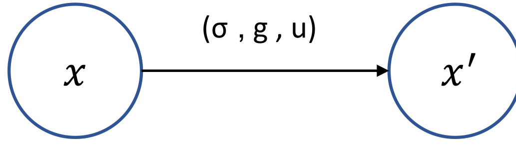


Figure 3.4: Transition from mode x to x'

3.2 Adding Timing Information to the Untimed Model

In this section, we will discuss a procedure to transform the untimed automaton model of the plant to a TA. Here we will assume that lower and upper time bounds for the events of plant are available. The resulting TA will have specific guard conditions, invariants and clock update rules and hence will form a subclass of TA discussed in Chapter 2.

Consider an untimed automaton $G = (X, \Sigma, \eta, x_0, X_m)$. Suppose, the timing information of the events is known. For each event $\sigma \in \Sigma$ a *lower time bound* $l_\sigma \in \mathbb{N}$ and an *upper time bound* $u_\sigma \in \mathbb{N} \cup \{\infty\}$ are known.

The untimed automaton is enriched to a specific type of TA by including the timing information of the events. The following discussion is not concerned with the marked states of the untimed automaton and therefore, we do not consider the marking of states in the TA. The TA takes the form of a six-tuple given as

$$G_{TA} = (X, \Sigma, C, Inv, T, x_0). \tag{7}$$

Each mode $x \in X$ of G_{TA} corresponds to a state of the untimed automaton G. The initial mode x_0 of G_{TA} is the same as initial state in G. The finite event set Σ of G_{TA} is the event set of the untimed automaton G. For each event $\sigma \in \Sigma$, there exists a unique clock $c_\sigma \in C$. For any event σ, c_σ is a function from $\mathbb{R}$ to $\mathbb{R}$ and at any time t, $c_\sigma(t)$ is the time elapsed since the last time clock c_σ was initialized. (Clock initialization will be discussed shortly). In every mode, the dynamics of c_σ is:

$$\frac{dc_\sigma(t)}{dt} = 1.$$

Suppose there are n_e events in Σ and thus n_e clocks in C. We arrange the clock variable in a fixed order and represent them as a vector $\boldsymbol{c} = (c_1, \ldots, c_{n_e})$ where each c_i is the clock variables of a unique event. The state q of G_{TA} at time t is a pair $(x(t), \boldsymbol{c}(t))$, where $x \in X$ is the mode at time

t and $c(t)$ is the clock vector at time t. In mode x_0 and initially at $t = 0$, the values of clocks are $c_i(0) = 0$ and therefore the initial state of G_{TA} is $q_0 = (x_0, \mathbf{0})$, where $\mathbf{0}$ is the zero vector. It is to be noted that the state set of the TA is infinite.

T is the set of transitions of G_{TA}. A transition in TA from mode x to mode x' with event σ (Fig. 3.4) is of the form $\theta = (x, \sigma, g, u, x')$, where g and u are the guard condition and the set of clocks that are reset after transition. For a transition with event σ, the guard condition g is of the form

$$g := c_\sigma(t) \geq l_\sigma.$$

The transition can occur only when g is satisfied (i.e., the clock of σ is at or larger then its lower time bound). From now on,for simplicity, we remove time t and write the guard as $c_\sigma \geq l_\sigma$. In this transition, u is the set of clocks that are reset when the transition occurs. Next, we explain which clocks are reset to zero and then we will provide a formal description of the set u.

Once again consider transition $\theta = (x, \sigma, g, u, x')$ shown in Fig. 3.4. Whenever an event occurs, the corresponding clock is reset to zero. Therefore, for transition $\theta = (x, \sigma, g, u, x')$, u includes c_σ. Next consider an event $\lambda \neq \sigma$. Four cases with respect to transition $\theta = (x, \sigma, g, u, x')$ are possible.

Case (1) λ is not defined at x but becomes defined at x': c_λ is reset to zero.

Case (2) λ is defined at x but becomes undefined at x': c_λ is reset to zero.

Case (3) λ is defined in both x and x': c_λ is not reset to zero.

Case (4) λ is defined neither in x not in x': c_λ is reset to zero.

The clock update set u in transition θ is formally defined as follows:

- $c_\sigma \in u$.

- For $\lambda \neq \sigma$

(1) not $\eta(x, \lambda)!$ and $\eta(x', \lambda)! \implies c_\lambda \in u$,

(2) $\eta(x, \lambda)!$ and not $\eta(x', \lambda)! \implies c_\lambda \in u$,

(3) $\eta(x, \lambda)!$ and $\eta(x', \lambda)! \implies c_\lambda \notin u$,

(4) not $\eta(x, \lambda)!$ and not $\eta(x', \lambda)! \implies c_\lambda \in u$.

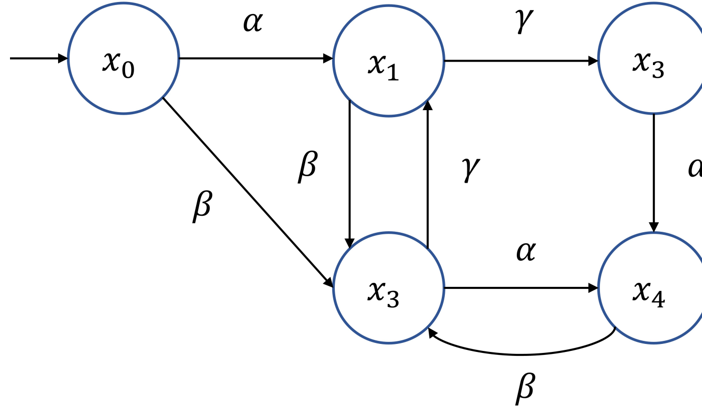


Figure 3.5: Untimed automaton G example.

The maximum allowable time that can be spent at a mode is specified by the clock invariant conditions. The clock invariant conditions are assigned to each mode of the TA and are the conjunction of the expressions on the upper bounds of clocks of the events that are defined at the corresponding state of G. For $u_\sigma = \infty$, the expression is $c_\sigma \le u_\sigma$. For $u_\sigma = \infty$, the expression is $c_\sigma < u_\sigma$. But since the clock invariant condition $c_\sigma < u_\sigma$ is always *true*, it can be omitted. At a mode x, if there is no event with finite upper time bound defined at x, then the clock invariant condition is *true*. Formally the clock invariant condition for each $x \in X$ is defined as:

$$Inv(x) := \begin{cases} true & \text{if } \{\sigma \mid \sigma \in \Gamma(x) \text{ and } u_\sigma \neq \infty\} = \emptyset, \\ \wedge\{c_\sigma \le u_\sigma \mid \sigma \in \Gamma(x) \text{ and } u_\sigma \neq \infty\} & \text{otherwise.} \end{cases}$$

For the TA G_{TA} constructed from the above procedure, we can make the following observations.

(P1) : G_{TA} does not have any input or output channels.

(P2) : Since G is assumed deterministic, G_{TA} is deterministic in the sense that for two transitions $\theta_1 = (x, \sigma_1, g_1, u_1, x_1')$ and $\theta_2 = (x, \sigma_2, g_2, u_2, x_2')$ originating from a mode x, if $x_1' \neq x_2'$, then $\sigma_1 \neq \sigma_2$ and thus the guard conditions g_1 and g_2 are on different events.

It is possible that in G_{TA}, consecutive transitions occur back to back instantaneously (no elapsed time in between transitions). However, we assume only a finite number of such consecutive transitions can occur. More generally we assume G_{TA} is *non-zeno*:

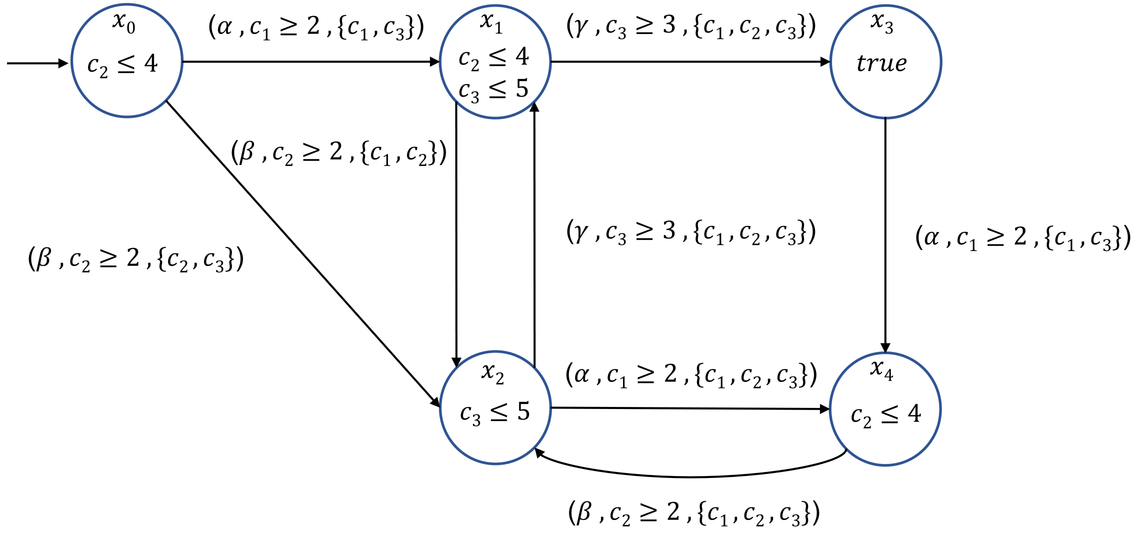


Figure 3.6: Timed automaton G_{TA} for G

(A1) : In TA G_{TA}, over any finite interval of time, only a bounded number of transitions can take place.

A sufficient condition for (A1) to hold that is relatively easy to verify is as follows.

Proposition 3.1. *If in the untimed automaton G, for any cycle, there exists a transition $x \xrightarrow{\sigma} x'$ in the cycle for some $\sigma \in \Sigma$ such that the lower time bound of σ is non-zero: $l_\sigma \neq 0$, then in the TA G_{TA}, assumption (A1) holds.*

Proof. By contradiction, suppose assumption (A1) does not hold. Since the lower time and upper time bound bounds on events are integers, the only trajectories that can occur over a finite interval of time and contain an infinite number of transitions must have a tail of transitions from mode to mode with zero duration. But since the set of modes is finite, this tail includes a cycle of mode with transitions of zero duration. But this contradicts the assumption of Proposition 3.1. □

Consider the untimed automaton G shown in Fig. 3.5. G contains five states, with x_0 as the initial state. The event set is $\Sigma = \{\alpha, \beta, \gamma\}$. Suppose, the timing information for the events is $(\alpha, 2, \infty)$, $(\beta, 2, 4)$ and $(\gamma, 3, 5)$. Then TA G_{TA} for G is obtained according to the above-mentioned procedure and is illustrated in Fig. 3.6. In the examples for TA, the modes are represented by circles and the transition from the source mode to the destination mode is represented by arrows, labelled

with the event, the guard condition and the clock update set. The invariants of each mode are indicated inside the mode.

3.3 Calculating a Lower Bound for $T_{min}(\delta)$

As previously discussed in Chapter 2, $T_{min}(\delta)$ is the minimum execution time for the occurrence of δ consecutive events. To determine $T_{min}(\delta)$ in a timed system obtained by transforming an untimed automaton to a TA, one needs to explore all the regions in the TA by performing a reachability analysis on regions. Performing reachability analysis has exponential time complexity in the number the events (even though the clock regions are finite) since the number of clock regions grows exponentially with the number of clocks in the TA (see Lemma 4.5 in [26]). The reachability analysis is needed to find the range of clock variables reachable in each mode. To avoid the reachability, we will determine a particular clock setting for each mode such that starting from that particular clock setting results in the shortest time for generation of δ events. As will be shown, this can be used to find a lower bound for $T_{min}(\delta)$. Before presenting the modifications to the clock variables, let us provide some useful definitions.

Definition 3.1. *Consider a TA $G_{TA} = (X, \Sigma, C, Inv, T, x_0)$. The set of* ***eligible events*** *at a state $q = (x, \boldsymbol{c})$ of the TA is defined as*

$$\Gamma_{elig}(q) := \{\sigma \mid \sigma \in \Gamma(x) \textit{ and } (l_\sigma \le c_\sigma \le u_\sigma \ (\textit{if } u_\sigma \neq \infty) \textit{ and } l_\sigma \le c_\sigma \ (\textit{if } u_\sigma = \infty))\}.$$

□

In Def. 3.1 the set of eligible events at a state of the TA are the events defined in the current mode that have the clock variables satisfying the invariants at the mode and the guard condition in the respective transitions.

Definition 3.2. *A* ***state trajectory*** *involving p transitions ($p \in \mathbb{Z}^+$) starting from a state q is a sequence of the form*

$$\mathcal{T} := q = q_0 \xrightarrow{(\sigma_1,t_1)} q_1 \xrightarrow{(\sigma_2,t_2)} q_2 \cdots \xrightarrow{(\sigma_p,t_p)} q_p$$

where for $i = 1, \ldots, p$, $\sigma_i \in \Sigma$ and $t_i \in \mathbb{R}^+ \cup \{0\}$ is the time dwelt in q_{i-1} before the occurrence of event σ_i. The trajectory is assumed to satisfy all the invaraints and transition guard conditions. That is for $0 \leq i \leq p$, $Inv(x_i)$ are satisfied for the duration of time TA is at mode x_i and also for transition $\theta_i = (x_{i-1}, \sigma_i, g_i, u_i, x_i)$, g_i is satisfied. □

A state trajectory starting from a state q in the TA is the sequence of events along with their dwell times in each corresponding mode. If the state trajectory $\mathcal{T}$ includes p transitions till state q_p, then the length of the state trajectory $\mathcal{T}$ is p.

Definition 3.3. *Consider a state trajectory $\mathcal{T}$ with the length $p \geq 1$ and dwell times $t_1, \ldots, t_p$, from a state q to another state q'. The **duration** of the state trajectory is*

$$\mathcal{D}(\mathcal{T}) := \sum_{i=1}^{p} t_p.$$

□

The duration of the state trajectory is the total time taken for the execution of the state trajectory. The amount of dwell time in the final state of trajectory $\mathcal{T}$ q' is not included in $\mathcal{D}(\mathcal{T})$.

Definition 3.4. *Consider a TA G_{TA} and a state $q = (x, \boldsymbol{c})$. Then for $p \in \mathbb{Z}^+$, $\mathcal{S}(q, p)$ denotes the set of state trajectories of length p starting from state q.* □

Definition 3.5. *Consider a TA G_{TA}. For a state $q = (x, \boldsymbol{c})$ and a positive integer $\delta \in \mathbb{Z}^+$, the **minimum execution duration** of state trajectories of δ events starting from q is*

$$t_{min}(q, \delta) := min\{\mathcal{D}(\mathcal{T}) \mid \mathcal{T} \in \mathcal{S}(q, \delta)\}.$$

□

According to Def. 3.5, $t_{min}(q, \delta)$ is the minimum execution time over the set of trajectories of length $\delta, \delta \in \mathbb{Z}^+$ starting from state q .

Definition 3.6. *Consider a TA G_{TA}. For every reachable state q and positive integer $\delta \in \mathbb{Z}^+$, the minimum execution duration is of δ events starting from q is*

$$T_{min}(\delta) := \min\{t_{min}(q, \delta) \mid q \text{ is reachable in } G_{TA}\}.$$

□

It is to be noted that $T_{min}(\delta)$ obtained for the TA depends on the set of reachable modes and the values of the clock variables when a mode is reached each time. Since each mode may be entered with different clock values, direct calculation of $T_{min}(\delta)$ becomes complex. Theorem 3.1 shows that a particular choice of clock variables, as defined in the theorem, results in the smallest $t_{min}(q, \delta)$ which can later be used to obtain a lower bound for $T_{min}(\delta)$. Before presenting Theorem 3.1, we need the results of Lemma 3.1 and Lemma 3.2.

Lemma 3.1. *Consider the TA G_{TA} and a state $q = (x, \mathbf{c})$. Suppose for some $\sigma^* \in \Sigma$, the clock variable at state q satisfies $l_{\sigma^*} \leq c_{\sigma^*} \leq u_{\sigma^*}$. Let $q' = (x, \mathbf{c'})$ be another state corresponding to the same mode x with a different clock vector $\mathbf{c'}$. Assume all clock variables of $\mathbf{c}$ and $\mathbf{c'}$ are the same except for σ^* where $c'_{\sigma^*} = l_{\sigma^*}$. Then for any $\delta \in \mathbb{Z}^+$, $t_{min}(q', \delta) \leq t_{min}(q, \delta)$.*

Proof. Consider a state trajectory $\mathcal{T}$ of length $p = \delta$ starting from q.

$$\mathcal{T} : q = (x, \boldsymbol{c}) \xrightarrow{(\sigma_1,t_1)} q_1 = (x_1, \boldsymbol{c}_1) \xrightarrow{(\sigma_2,t_2)} \cdots \xrightarrow{(\sigma_p,t_p)} q_p = (x_p, \boldsymbol{c}_p),$$

We show by induction that there exists another state trajectory $\mathcal{T}'$ containing the same sequence of events $\sigma_1 \ldots \sigma_p$ that can be executed from q' with the same dwell times $t_1, \ldots, t_p$. Furthermore, at any time t the corresponding clock vectors $\boldsymbol{c}_i$ and $\boldsymbol{c}'_i$ are identical except for event σ^* where $c'_{\sigma^*} \leq c_{\sigma^*}$ at each state q_i and q'_i on the state trajectories $\mathcal{T}$ and $\mathcal{T}'$.

$$\mathcal{T}' : q' = (x, \boldsymbol{c'}) \xrightarrow{(\sigma_1,t_1)} q'_1 = (x_1, \boldsymbol{c}'_1) \xrightarrow{(\sigma_2,t_2)} \cdots \xrightarrow{(\sigma_p,t_p)} q'_p = (x_p, \boldsymbol{c}'_p).$$

- Induction Base: At the initial state q, σ_1 occurs after a duration of t_1 in state trajectory $\mathcal{T}$.
  - If $\sigma_1 = \sigma^*$, since $c'_{\sigma^*} = l_{\sigma^*}$, $\sigma_1 = \sigma^*$ can also occur from q' after a duration of t_1. Upon transition, the clock of σ^* is reset to zero in both state trajectories. Thus $\boldsymbol{c}_1 = \boldsymbol{c}'_1$.
  - If $\sigma_1 \neq \sigma^*$, still σ_1 can occur in $\mathcal{T}'$ starting from q' after the same duration of t_1 since all clock variables in $\boldsymbol{c'}$ at q' other than c_{σ^*} have the same initial value as clock variables in $\boldsymbol{c}$ at q. Furthermore, since initially $c'_{\sigma^*} \leq c_{\sigma^*}$ and at time t_1, σ^* did not preempt σ_1 in trajectory $\mathcal{T}$, then in trajectory $\mathcal{T}'$, σ^* does not have to preempt σ_1 at t_1 either.

- Induction Step: Suppose state trajectory $\mathcal{T}'$ can occur for the first k steps, with $1 \leq k < p$. We show that step $k+1$ can also occur in $\mathcal{T}'$. We consider two cases.

Case (1) : Over the first k steps of state trajectory $\mathcal{T}$, the clock of σ^* has not been reset to zero. In this case, the corresponding clock variables of $\mathcal{T}$ and $\mathcal{T}'$ at step k are all identical, except for σ^* for which

$$l_{\sigma^*} \leq c'_{\sigma^*} \leq c_{\sigma^*} \leq u_{\sigma^*}. \tag{8}$$

In this case, using an argument similar to the base case, we can show that σ_{k+1} can occur at state q'_k in $\mathcal{T}'$ after a time t_{k+1}.

Case (2) : If after some k' transitions, with $1 \leq k' \leq k$, the clock of σ^* resets to zero, then after $\sigma_{k'}$: $c'_{\sigma^*} = c_{\sigma^*}$. This means that the clock variables on both trajectories will become identical: $\boldsymbol{c'_{k'}} = \boldsymbol{c}_{k'}$. Since after $\sigma_{k'}$ the state of the TA is the same on both state trajectories, the rest of events $\sigma_{k'+1} \dots \sigma_p$ can occur on $\mathcal{T}'$ with dwell times $t_{k'+1}, \dots, t_p$.

Thus by induction we showed that if a state trajectory $\mathcal{T}$ starting from q can occur in the TA G_{TA}, then the state trajectory $\mathcal{T}'$ which has the same events and dwell times as $\mathcal{T}$ can also occur in TA G_{TA} from q'. Thus $\mathcal{D}(\mathcal{T}) = \mathcal{D}(\mathcal{T}')$. Then if follows from Def. 3.5 that $t_{min}(q', \delta) \leq t_{min}(q, \delta)$. $\square$

According to Lemma 3.1, suppose at $q = (x, c)$ the clock variables of the events whose values are above the lower time bound are reduced to the lower time bound values (while leaving other clocks values same), resulting in a new state q^* with the same mode x. Then by repeatedly applying Lemma 3.1, we can conclude that $t_{min}(q^*, \delta) \leq t_{min}(q, \delta)$.

Consider a TA $G_{TA} = (X, \Sigma, C, Inv, T, x_0)$. Let $q = (x, \boldsymbol{c})$ be a state of G_{TA} satisfying the invariants at mode x. We would like to obtain a lower bound for $t_{min}(q, \delta)$ ($\delta \in \mathbb{Z}^+$). We begin by examining events defined at mode x and find the one with the largest lower time bound

$$l_{\sigma,max} := \max\{l_\sigma \mid \sigma \in \Gamma(x)\}. \tag{9}$$

Next we raise the clock values of the events defined at mode x by $l_{\sigma,max}$ and obtain a state $q' = (x, \boldsymbol{c'})$. That is the clock variables at $\boldsymbol{c'}$

$$c'_\sigma = \begin{cases} c_\sigma + l_{\sigma,max} & \sigma \in \Gamma(x), \\ c_\sigma & \sigma \notin \Gamma(x). \end{cases} \tag{10}$$

This increase may result in some clocks exceeding their upper time bounds. To resolve this, the upper time bounds of the events defined at mode x are increased by $l_{\sigma,max}$:

$$\forall \sigma \in \Gamma(x) \quad u'_\sigma = u_\sigma + l_{\sigma,max}. \tag{11}$$

Let us denote the TA with the new upper time bounds u'_σ as $G'_{TA,x}$. $G'_{TA,x}$ is identical to G_{TA} except for the time bounds. For simplicity we use the same labels for the modes of $G'_{TA,x}$ (i.e. x). Also, for $G'_{TA,x}$ the shortest execution time function will be denoted by t'_{min}.

Lemma 3.2. *Consider state $q = (x, \boldsymbol{c})$ in G_{TA} and state $q' = (x, \boldsymbol{c'})$ in $G'_{TA,x}$ as defined in equations (10) and (11). For any $\delta \in \mathbb{Z}^+$, $t'_{min}(q', \delta) \leq t_{min}(q, \delta)$.*

Proof. Consider a state trajectory $\mathcal{T}$ of length $p = \delta$ starting from q in G_{TA}.

$$\mathcal{T} : q = (x, \boldsymbol{c}) \xrightarrow{(\sigma_1, t_1)} q_1 = (x_1, \boldsymbol{c}_1) \xrightarrow{(\sigma_2, t_2)} \cdots \xrightarrow{(\sigma_p, t_p)} q_p = (x_p, \boldsymbol{c}_p).$$

We show that in $G'_{TA,x}$ there exists a state trajectory $\mathcal{T}'$ of length p starting from $q' = (x, \boldsymbol{c'})$ which has the same event sequence $\sigma_1 \ldots \sigma_p$ with the same dwell times $t_1, \ldots, t_p$.

$$\mathcal{T}' : q' = (x, \boldsymbol{c'}) \xrightarrow{(\sigma_1, t_1)} q'_1 = (x_1, \boldsymbol{c'}_1) \xrightarrow{(\sigma_2, t_2)} \cdots \xrightarrow{(\sigma_p, t_p)} q'_p = (x_p, \boldsymbol{c'}_p).$$

We will use induction to prove the claim.

- Induction Base: At state q' for any event σ defined at x, the clock variable c'_σ satisfies

$$l_\sigma \leq l_{\sigma,max} \leq c'_\sigma \quad \text{and} \quad c'_\sigma = c_\sigma + l_{\sigma,max} \leq u_\sigma + l_{\sigma,max} = u'_\sigma. \tag{12}$$

Therefore, the invariants of mode x are satisfied in q' and all defined events are eligible. In mode x, as time passes, since clock variables change at the same rate, the equality $c'_\sigma = c_\sigma + l_{\sigma,max}$ will remain valid. Thus, if c_σ satisfies its invariant in x in G_{TA} (i.e. $c_\sigma \leq u_\sigma$), then c'_σ will also satisfy its invariant in x in $G'_{TA,x}$ (i.e. $c'_\sigma \leq u'_\sigma$). Since on state trajectory $\mathcal{T}$, σ_1 occurs from state q in G_{TA} after dwell time t_1, then σ_1 can also occur from state q' in $G'_{TA,x}$ after the same dwell time t_1.

- Induction Step: Suppose in state trajectory $\mathcal{T}'$, k steps occur up to the transition σ_k with state of $G'_{TA,x}$ entering q'_k. We show that σ_{k+1} can occur in $G'_{TA,x}$ after the dwell time of t_{k+1}. For event σ_{k+1} there are two possible cases.

Case (1) : Event σ_{k+1} was reset to zero (at least once) in the first k steps of $\mathcal{T}$. Let k' with $1 \leq k' \leq k$ be the last transition where event σ_{k+1} occurred in which the corresponding clock variable was reset to zero. Since $\mathcal{T}$ and $\mathcal{T}'$ have the same sequence of events and same corresponding dwell times up to step k, then the clock variable corresponding to event σ_{k+1} was also reset at step k' in the trajectory $\mathcal{T}'$. After that step the clock variable of event σ_{k+1} will have identical values in $\mathcal{T}$ and $\mathcal{T}'$ up to and including step k. Thus, σ_{k+1} can occur at mode x_k in trajectory $\mathcal{T}'$ after the same dwell time t_{k+1} identical to $\mathcal{T}$.

Case (2) : Event σ_{k+1} was not reset to zero in the first k steps of $\mathcal{T}$. In this case the clock variable for this event in G_{TA} and $G'_{TA,x}$ will satisfy $c'_{\sigma_{k+1}} = c_{\sigma_{k+1}} + l_{\sigma,max}$ from the beginning of the trajectories up to mode x_k and thus σ_{k+1} can occur from state q'_k in $G'_{TA,x}$ after the same dwell time of t_{k+1}.

Therefore, for every $\mathcal{T}$ of length δ in G_{TA}, there exists a trajectory $\mathcal{T}'$ from q' in $G'_{TA,x}$ with $\mathcal{D}(\mathcal{T}) = \mathcal{D}(\mathcal{T})'$. Therefore, by induction $t'_{min}(q', \delta) \leq t_{min}(q, \delta)$. □

According to Lemma 3.2, we make two adjustments (1) we raise the clock values of all the events defined at x by $l_{\sigma,max}$ and, (2) we increase the upper time bounds of the events defined at mode x. With these two adjustments any sequence of events that can be generated in the G_{TA} can be generated in the new system with the same dwell times from the same mode. This results

$t'_{min}(q', \delta)$ is a lower bound for $t_{min}(q, \delta)$. Lemma 3.1 and Lemma 3.2 can be combined to prove the final result which is Theorem 3.1. We need one more definition before proving the theorem that provides a lower bound for $T_{min}(\delta)$.

Consider the TA $G'_{TA,x}$. Next consider state $q_{l,x} = (x, \boldsymbol{c}_{l,x})$ in $G'_{TA,x}$ corresponding to mode x and clock variable for events defined at x set to the respective lower time bounds and zero otherwise:

$$\boldsymbol{c}_{l,x} = (l_1, \ldots, l_{n_e}), \tag{13}$$

where $\Sigma = \{\sigma_1, \ldots, \sigma_{n_e}\}$ and

$$l_i := \begin{cases} l_{\sigma_i} & \sigma_i \in \Gamma(x) \\ 0 & \sigma_i \notin \Gamma(x). \end{cases}$$

Definition 3.7. *Consider the TA G_{TA} in equation (7). For any mode x reachable in untimed automaton G, let $G'_{TA,x}$ be the TA constructed as in equations (9) - (11). Also let clock vector $\boldsymbol{c}_{l,x}$ be defined as in equation (13). For any $\delta \in \mathbb{Z}^+$, define*

$$T'_{min,low}(\delta) := \min\{t'_{min}((x, \boldsymbol{c}_{l,x}), \delta) \mid x \textit{ is reachable in } G\},$$

□

The following theorem summarizes the main result for this chapter. It will show that for any mode x, $t'_{min}((x, \boldsymbol{c}_{l,x}), \delta)$ provides a lower bound for shortest execution time of δ events out of x, and thus $T'_{min,low}(\delta)$ as defined in Def. 3.7 provides a lower bound for $T_{min}(\delta)$.

Theorem 3.1. *Consider the TA G_{TA}. For any $\delta \in \mathbb{Z}^+$, $T'_{min,low}(\delta) \leq T_{min}(\delta)$.*

Proof. Let $q = (x, \boldsymbol{c})$ be a state of G_{TA} and $q' = (x, \boldsymbol{c'})$ be the corresponding state of $G'_{TA,x}$ described by equations (9) - (11). Also, let $q_{l,x} = (x, \boldsymbol{c}_{l,x})$ be another specific state of $G'_{TA,x}$ as described in equation (13). Applying Lemma 3.1 to $G'_{TA,x}$ and state $q' = (x, \boldsymbol{c'})$ and all events defined at mode x results in:

$$t'_{min}(q_{l,x}, \delta) \leq t'_{min}(q', \delta). \tag{14}$$

Using Lemma 3.2

$$t'_{min}(q',\delta) \leq t_{min}(q,\delta). \tag{15}$$

Using equation (14) and equation (15) results in:

$$t'_{min}(q_{l,x},\delta) \leq t_{min}(q,\delta). \tag{16}$$

In other words, at any mode x, the shortest execution time calculated from a specific state $q_{l,x}$ in $G'_{TA,x}$ is a lower bound for the shortest execution time in G_{TA} from *any* state q in mode x. By Def. 3.7, $T'_{min,low}(\delta)$ is the lowest value of $t'_{min}(q_{l,x},\delta)$ over all the reachable modes of $G'_{TA,x}$. It follows from equation (16) that

$$\begin{aligned} T'_{min,low}(\delta) &\leq \min\{t_{min}(q,\delta) \mid q = (x,\boldsymbol{c}) \text{ with } \boldsymbol{c} \text{ satisfying invaraints at } x\}, \\ &\leq \min\{t_{min}(q,\delta) \mid q \text{ is reachable in } G_{TA}\}, \\ &= T_{min}(\delta). \quad \text{(By Def. 3.6)} \end{aligned}$$

This completes the proof. □

The following examples illustrates Theorem 3.1.

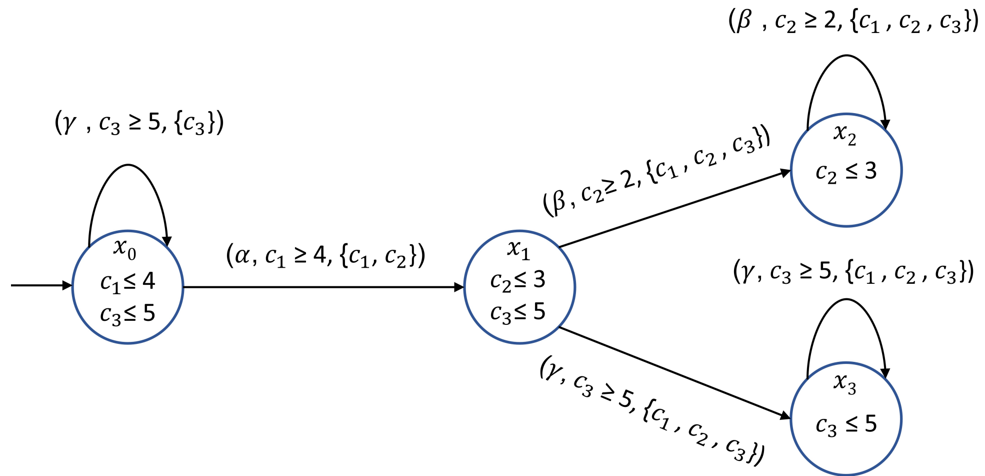


Figure 3.7: Example 3.2. TA G_{TA_1}.

Example 3.2. *Consider the TA shown in Fig. 3.7. G_{TA_1} contains four modes and three events. Mode x_0 is the initial mode. The time bounds of the events are $(\alpha, 4, 4), (\beta, 2, 3)$ and $(\gamma, 5, 5)$. The clocks for events α, β and γ are c_1, c_2 and c_3 respectively. Suppose we would like to calculate $t_{min}(x_1, \delta)$. Initially the clock variables are zero: $c_1 = c_2 = c_3 = 0$. At $t = 0$, the execution of G_{TA_1} starts. At $t = 4$, α occurs which resets clocks c_1 and c_2 according to the update set of the transition. Event γ at state x_0 requires at least 5 time units and therefore never occurs. After the α transition, G_{TA_1} enters mode x_1 with $c_3 = 4$. From this mode we want to calculate $t_{min}((x, \boldsymbol{c}_1), \delta)$ with $c_1 = (0, 0, 4)$. Time passes and at $t = 5$, γ occurs and G_{TA_1} enters mode x_3 and generates γ events only afterwards. At x_1, β event never happens since at least 2 time units have to pass in mode x_1 before β can happen. While at mode x_1:*

$$\begin{aligned} &c_1(t) = c_2(t) = t - 4 && 4 \leq t \leq 5, \\ &c_3(t) = t && 4 \leq t \leq 5. \end{aligned} \tag{17}$$

Equation (17) defines the set of reachable states in mode x_1. Specifically, at $t = 5$ in mode x_1 when γ occurs at $t = 5$, the values of the clocks are $c_1(5) = c_2(5) = 1$ and $c_3(5) = 5$. Let us denote this state as $q^* = (x_1, (1, 1, 5))$. State q^* yields the smallest values for $t_{min}((x_1, \boldsymbol{c}), \delta)$ for all states in mode x. From q^*, $t_{min}(q^*, 1) = 0$, $t_{min}(q^*, 2) = 5$ (since $l_\gamma = 5$) and more generally

$$t_{min}(q^*, \delta) = 5(\delta - 1) \qquad (\delta \geq 1).$$

Next we find $t'_{min}(q_{l,x_1}, \delta)$ with $q_{l,x_1} = (x_1, \boldsymbol{c}_{l,x_1})$ state. Consider G'_{TA_1,x_1} which is essentially same as G_{TA_1} but with different upper bounds for the events defined at mode x_1. In mode x_1, β and γ are defined. Therefore,

$$l_{\sigma,max} = \max\{2, 5\} = 5.$$

The upper time bounds of β and γ at mode x are increased by 5. Therefore the event time bounds in G'_{TA_1,x_1} are given as $(\alpha, 4, 4)$, $(\beta, 2, 8)$ and $(\gamma, 5, 10)$. Therefore, $q_{l,x_1} = (x_1, (0, 2, 5))$. At q_{l,x_1}, both β and γ are eligible to occur. Since β has a smaller lower time bound it will determine $t'_{min}(q_{l,x_1}, \delta)$:

$$t'_{min}(q_{l,x_1}, \delta) = 2(\delta - 1) \qquad (\delta \geq 1).$$

□

We see that $t'_{min}(q_{l,x_1}, \delta)$ is a lower bound for $t_{min}(q, \delta) = 5(\delta - 1)$. As this example shows, the difference between $t'(q_{l,x_1}, \delta)$ and $t_{min}(q, \delta)$ could be large. $t'_{min}(q_{l,x}, \delta)$ is used in the calculation of $T'_{min,low}(\delta)$. It would be important to know how significant the difference between $T_{min}(\delta)$ and its lower estimate $T'_{min,low}(\delta)$ is. It would be useful to have a way of assessing the difference. Experimental measure of $T_{min}(\delta)$ denoted by $T_{min,exp}(\delta)$ provide an upper bound for $T_{min}(\delta)$. If $T_{min,exp}(\delta)$ and $T'_{min,low}(\delta)$ are not reasonably close, then further experiments or a more detailed theoretical analysis are required. This will be explored in Chapter 5 in the application of the above procedure to an experimental setup.

Example 3.3. *Consider the TA G_{TA_2} in Fig. 3.8. This example shows why it is necessary to raise the upper time bounds of some events in Lemma 3.2. We see that if the upper time bounds are not raised, some sequences could be eliminated in G_{TA_2}. In this example there are five modes and three events with their timing information as $(\alpha, 2, 4)$, $(\beta, 2, 2)$ and $(\gamma, 4, 5)$ along with the clocks c_1, c_2 and c_3 respectively. In mode x_0, at $t = 2$, β occurs. After the β transition, G_{TA_2} transitions to*

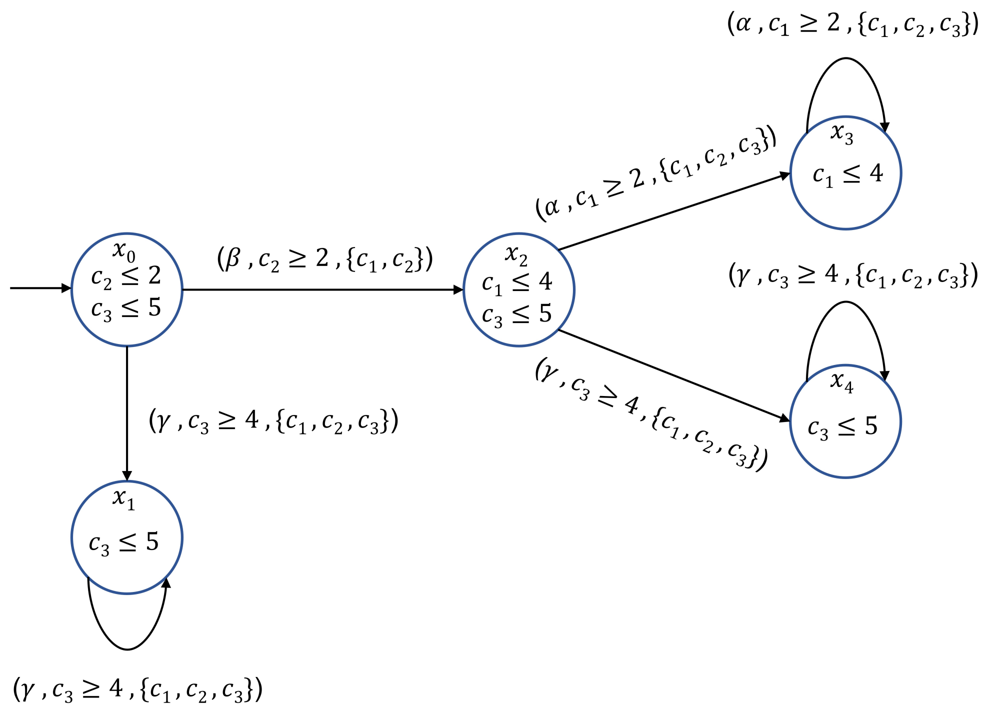


Figure 3.8: Example 3.3. TA G_{TA_2}.

mode x_2 where $c_1 = c_2 = 0$ and $c_3 = 2$. So at $t = 4$, either α or γ can occur in mode x_2. Therefore G_{TA_2} can execute both $\beta\alpha$ and $\beta\gamma$.

If we want to find $t'_{min}(q_{l,x_0}, \delta)$ from mode x_0, we set the clock of events β and γ to their lower bounds $c_2 = 2, c_3 = 4$ and initialize c_1 to $c_1 = 0$. If at $t = 0$, β occurs in mode x_0, then upon entering x_2, $c_3 = 4$ and $c_1 = 0$. Hence, event γ occurs between $0 \leq t \leq 1$ and α cannot occur in mode x_2 because clock c_1 never reaches the minimum value of 2 for α to occur. □

3.4 Finding Minimum Execution Duration of Event Sequences Using DBM

In this section, we will explain how minimum execution duration of an event sequence is calculated by setting the clock values according to Theorem 3.1 and by using DBMs for clock regions when entering the first mode in the event sequence. Consider an example shown in Fig. 3.9 which consists of a state trajectory $\mathcal{T}$ of length three with the event sequence $\alpha\beta\alpha$. We would like to calculate the minimum execution duration of $\mathcal{T}$. The timing information for events is $(\alpha, l_\alpha, u_\alpha)$ and $(\beta, l_\beta, u_\beta)$. The clocks for events α and β are c_1 and c_2 respectively. We introduce a fictitious clock c_{fict} in order to calculate minimum execution duration of event sequences. This fictitious clock c_{fict} is similar to the clock *Now* as mention in [52]. The clock c_{fict} is never reset, except at the initial mode x_0 and in the update set of all the first transitions at the mode x from where $t'_{min}((x, \boldsymbol{c}_{l,x}), \delta)$ is required to be determined. It is to be noted that the clock c_{fict} is included in DBM which changes the dimension of DBM to $(n_e + 2)$. Extracting the lower bound value of c_{fict} from DBM at δ transition results in $t'_{min}((x, \boldsymbol{c}_{l,x}), \delta)$ value. We assume that DBMs for clock zones from R_0 to R_6 are calculated and are not empty.

Initially, clock zone for R_0 is represented according to Theorem 3.1 by the following DBM:

$$R_0 = \begin{bmatrix} 0 & 0 & -l_\alpha & 0 \\ \infty & 0 & \infty & \infty \\ l_\alpha & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{bmatrix}.$$

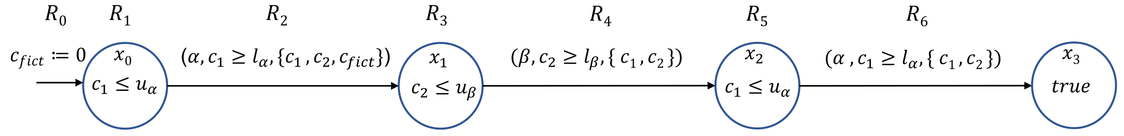


Figure 3.9: Procedure for calculating minimum execution duration of event sequence using DBM.

Next, the DBM for clock zone R_0 will be canonicalized in order to tighten the bounds of the matrix. Notice that the entries $(1, 0)$ and $(0, 1)$ in DBM R_0 correspond to the upper bound and lower bound value of clock c_{fict} respectively. The DBMs for the clock zones R_1, R_3 and R_5 are to check whether the clocks satisfy the invariants at the corresponding modes. The DBMs for the clock zones R_2, R_4 and R_6 are to check whether the clocks satisfy the guard conditions in the respective transitions for the corresponding event to occur. If DBMs for R_1 and R_2 are not empty then event α has occurred. Following Def. 3.5, the calculation of minimum execution duration of event sequence starts after the occurrence of first event which in this case is α transition from mode x_0. After α transition from mode x_0; clock c_{fict} is reset and the corresponding entries in DBM R_2 are reset to zero. For rest of the transitions in $\mathcal{T}$, c_{fict} is never reset. The value of the clock c_{fict} will keep on increasing as the event transitions of state trajectory $\mathcal{T}$ occurs. Finally, the minimum execution duration of the state trajectory $\mathcal{T}$ is obtained by extracting and negating the lower bound value of c_{fict} at entry $R_6(0, 1)$ of DBM R_6.

Following example explains above-mentioned procedure for calculating minimum executing duration of an event sequence.

Example 3.4. *Consider G_{TA} shown in Fig. 3.6. The timing information for the events is $(\alpha, 2, \infty)$, $(\beta, 2, 4)$ and $(\gamma, 3, 5)$. In this example suppose is $\delta = 3$ and we want to calculate $t'_{min}((x_0, \mathbf{c}_{l,x_0}), 3)$. To illustrate the procedure of calculating $t'_{min}((x_0, \mathbf{c}_{l,x_0}), 3)$, consider one of the state trajectories $\mathcal{T}$ of length 3 of G_{TA} shown in Fig. 3.10 that has the event sequence $\beta\gamma\gamma$. In this case, despite of the mode being x_0, the clocks are not initialized to $\mathbf{0}$, on the other hand, they are initialized to $\mathbf{c}_{l,x_0} = (2, 2, 0)$. Notice that initially in x_0, c_{fict} is zero. In this example the upper time bounds of the events that are defined at the mode x_0 must be raised by $l_{\sigma,max}$. Here, $l_{\sigma,max} = \max(2, 2) = 2$ and therefore, the new timing information for the events is $(\alpha, 2, \infty)$, $(\beta, 2, 6)$ and $(\gamma, 3, 7)$.*

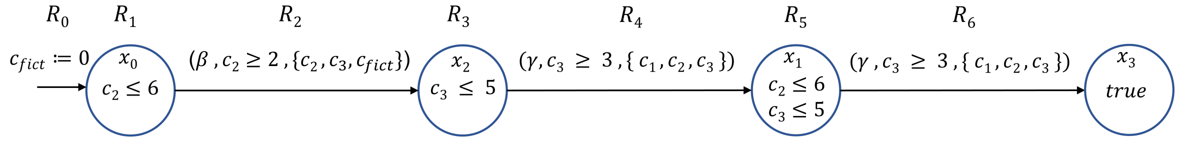


Figure 3.10: State trajectory $\mathcal{T}$ of G_{TA}.

Initially, the clocks are $\mathbf{c}_{l,x_0} = (2, 2, 0)$ *and then the clock zone is given by the following DBM* R_0 *(which includes* c_{fict} *as well).*

$$R_0 = \begin{bmatrix} 0 & 0 & -2 & -2 & 0 \\ \infty & 0 & \infty & \infty & \infty \\ 2 & \infty & 0 & \infty & \infty \\ 2 & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}.$$

The canonical DBM R_0 *is given as*

$$R_0 = \begin{bmatrix} 0 & -2 & -2 & -2 & 0 \\ \infty & 0 & \infty & \infty & \infty \\ 2 & 2 & 0 & 0 & 2 \\ 2 & 2 & 0 & 0 & 2 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}.$$

Upon entry to the mode x_0*, the set of clocks are updated using timed action (i.e. passage of time) in the mode* x_0 *that results in canonical DBM* R_1 *given below.*

$$R_1 = \begin{bmatrix} 0 & -2 & -2 & -2 & -2 \\ 6 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

R_1 *represents the clock zone while at mode* x_0.

DBM R_2 *is formed by updating the clock* c_2 *in* R_1 *according to the guard condition of the* β *transition.* R_2 *describes the set of clock valuations upon entry to mode* x_2*. It can be observed in DBM*

R_1, that the entry corresponding to the clock c_2 in the first row is -2. After taking the transition, clocks c_2 and c_3 are reset. The dwell time at mode x_0 does not contribute in the calculation of the $t'_{min}((x_0, \boldsymbol{c}_{l,x_0}), 3)$. Canonical DBM R_2 is given below.

$$R_2 = \begin{bmatrix} 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 \\ 6 & 6 & 0 & 6 & 6 \\ 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 \end{bmatrix}.$$

The DBM R_3 given below captures the time elapse in mode x_2.

$$R_3 = \begin{bmatrix} 0 & 0 & -2 & 0 & 0 \\ 5 & 0 & -2 & 0 & 0 \\ 11 & 6 & 0 & 6 & 6 \\ 5 & 0 & -2 & 0 & 0 \\ 5 & 0 & -2 & 0 & 0 \end{bmatrix}.$$

The value of c_3 upon entering x_2 is 0 and the guard condition on the γ transition is $c_3 \geq 3$. Thus R_3 is updated according to the guard condition and is tested for emptiness resulting in canonical DBM R_4.

$$R_4 = \begin{bmatrix} 0 & -3 & -5 & -3 & -3 \\ 5 & 0 & -2 & 0 & 0 \\ 11 & 6 & 0 & 6 & 6 \\ 5 & 0 & -2 & 0 & 0 \\ 5 & 0 & -2 & 0 & 0 \end{bmatrix}.$$

Once γ transition occurs, all clocks (except c_{fict}) are reset, thereby updating clock zone R_4.

$$R_4 = \begin{bmatrix} 0 & -3 & 0 & 0 & 0 \\ 5 & 0 & 5 & 5 & 5 \\ 0 & -3 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 \\ 0 & -3 & 0 & 0 & 0 \end{bmatrix}.$$

On entry to mode x_1, clock zone R_4 is updated according to the invariants at x_1 resulting in clock zone R_5 which is represented by the following canonical DBM.

$$R_5 = \begin{bmatrix} 0 & -3 & 0 & 0 & 0 \\ 10 & 0 & 5 & 5 & 5 \\ 5 & -3 & 0 & 0 & 0 \\ 5 & -3 & 0 & 0 & 0 \\ 5 & -3 & 0 & 0 & 0 \end{bmatrix}.$$

For the final γ transition, R_5 is updated according to the guard condition and is made canonical resulting in clock zone R_6.

$$R_6 = \begin{bmatrix} 0 & -6 & -3 & -3 & -3 \\ 10 & 0 & 5 & 5 & 5 \\ 5 & -3 & 0 & 0 & 0 \\ 5 & -3 & 0 & 0 & 0 \\ 5 & -3 & 0 & 0 & 0 \end{bmatrix}.$$

As a result of timed actions at x_1, the entry of c_3 in first row in R_6 will be updated from 0 to 3 to satisfy the guard condition. Consequently, the minimum duration of the trajectory is $\mathcal{D}(\mathcal{T}) = -R_6(0,1) = 6$. □

3.5 Algorithm for Calculating $T'_{min,low}(\delta)$

Based on Def. 3.7 we will provide an algorithm which calculates $T'_{min,low}(\delta)$. For computational speed, we will use the operations performed on DBMs (as discussed in Section 2.6 of Chapter 2) in the algorithm to calculate $T'_{min,low}(\delta)$.

In order to determine $t'_{min}(q_{l,x}, \delta)$ from the state $q_{l,x} = (x, \boldsymbol{c}_{l,x})$ of the TA it is necessary to explore all the state trajectories in $\mathcal{S}(q_{l,x}, \delta)$. A graph traversing algorithm such as *best-first search* (BFS) algorithm [53] can be used to explore all the state trajectories in $\mathcal{S}(q_{l,x}, \delta)$. Given a graph, the BFS algorithm starts from root node and at each stage explores all the neighbouring unvisited node along each branch before backtracking. In BFS algorithm all the vertices are visited at most

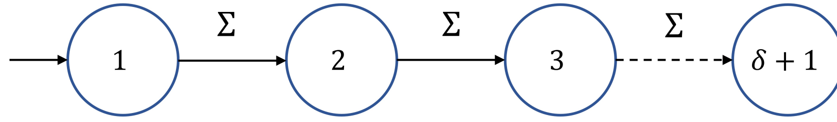


Figure 3.11: Counter automaton G_{ev}.

once. The BFS algorithm maintains a stack data structure to store the states during exploration. For example, recall the untimed automaton G shown in the Fig. 3.5 which has the form of a directed graph. Suppose BFS is used to find states reachable from the initial state x_0. The BFS algorithm starts from the root note which is state x_0 and then explores state x_1 and state x_2 and adds them at the top of the stack in the same order as they were explored. In the BFS algorithm, we are assuming that all the states are explored depending upon the event label; then its corresponding target state is explored first. Then state x_2 is popped from the stack and state x_4 is explored and added at the top of the stack. Next state x_4 is popped from the stack. There is no further exploration required from state x_4, since state x_2 is already been explored. State x_1 is popped from the stack and state x_3 is explored and added at the top of the stack. Now state x_3 is popped from the stack, however, there is no further exploration possible and hence the algorithm terminates. The traversing order for G using BFS algorithm is given as

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_4 \rightarrow x_3.$$

The BFS algorithm (explained above) just provides the information about the states which are reachable by some trajectory; however, it does not traverse through all of the trajectories from the root state. Suppose, in an automaton one needs to traverse through all the trajectories of a length, say δ, from the root state, then one must perform unfolding of all the event sequences up to the length δ. In order to unfold all the event sequences, a counter automaton G_{ev} shown in the Fig. 3.11 can be built by having transitions for all events in Σ counting to δ events.

For our running example if the product of G and G_{ev} is taken then an unfolded automaton is obtained which is given as

$$G_{un} = product(G, G_{ev}).$$

Fig. 3.12 illustrates the unfolded automaton G_{un} for G with $\delta = 3$. The states of G_{un} are deliberately not renamed just to show that the states of G can be visited again. Moreover, it can be observed

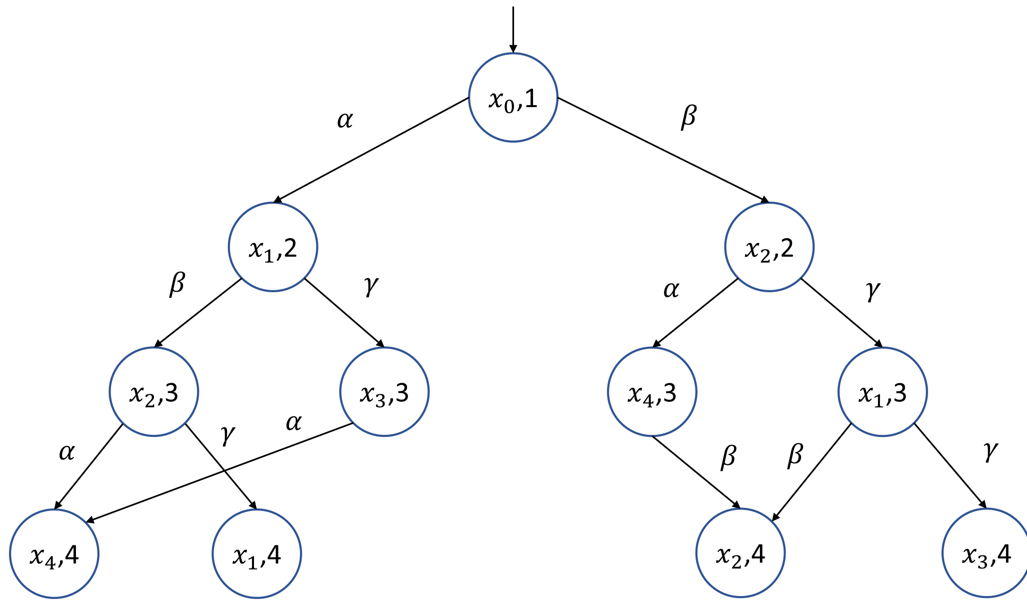


Figure 3.12: Unfolded automaton G_{un}

that all the sequences of events have a length of at most 3 in G_{un}. Since all the sequences of events of length up to δ are need to be explored, a state may be visited more than once. Therefore, to overcome this problem we adjust the BFS algorithm such that the algorithm does not keep track of the visited states and we call it *modified BFS algorithm*. To give a brief explanation for this algorithm, suppose we start the exploration from the root state (x_0, 1). Then the modified BFS algorithm will explore states (x_1, 2) and (x_2, 2) and add them at the top of the stack in the same order as they were explored. State (x_2, 2) is popped from the stack and states (x_4, 3) and (x_1, 3) are discovered from state (x_2, 2) and they are added at top of the stack. State (x_1, 3) is next to be explored and from here state (x_2, 4) and (x_3, 4) are added at the top of the stack. In this way the two event sequences $\beta\gamma\gamma$ and $\beta\gamma\beta$ are completely explored. Since there are no further transitions from states (x_2, 4) and (x_3, 4), the algorithm backtracks and explores state (x_4, 3). From state (x_4, 3), state (x_2, 4) is discovered and added at the top of the stack. Notice that, state (x_2, 4) is visited again for the second time. Since there is no transition from state (x_2, 4), there is no further exploration from state (x_2, 4). In this way, the sequence $\beta\alpha\beta$ is traversed. Now, the algorithm backtracks to state (x_1, 2) and traverses through all the sequences in the left side of the unfolded automaton G_{un} (depicted in Fig. 3.12) in a similar fashion until the stack is empty.

Remark 3.1. *Instead of performing modified BFS algorithm on an unfolded automaton; an alternative way is to modify the unfolded automaton and then apply regular BFS algorithm. While performing the unfolding of an automaton for δ number of events, for every state which has more than one incoming transitions we split that state into mulitple identical states (depending upon the number of incoming transitions) with identical outgoing transitions. For example in Fig. 3.12, state (x_2, 4) has two incoming transitions of event β. Then, state (x_2, 4) can be split into two identical states (x_2, 4, 1) and (x_2, 4, 2) with identical outgoing transitions.* □

Suppose in a deterministic automaton with n_e events. If the untimed automaton is unfolded for the length of δ, then, in the worst-case G_{un}, has

(1) $\frac{n_e^{\delta+1}-1}{n_e-1}$ states,

(2) $\frac{n_e^{\delta+1}-1}{n_e-1} - 1$ event transitions.

The unfolded automaton has the structure similar of a directed tree, resulting in exponential growth of states in δ in G. However to avoid this complexity, such an unfolding operation is not performed in the modified BFS algorithm because the states are explored on-the-fly. In other words, the graph of unfolded automaton is explored piece by piece.

The modified BFS algorithm explained above is used to traverse through all the event sequences in an untimed automaton while ignoring states already visited. Similarly, this algorithm combined with the DBM operations for testing the emptiness of the clock zones can be used to obtain $t'_{min}(q_{l,x}, \delta)$ from any mode x. Before explaining the resulting algorithm, let us first present a minor adjustment made to the zones. Previously in Section 2.6 of Chapter 2, a zone is of the form (x, R) but we will represent the zone as (x, R, D) where $x \in X$ is the mode, R is the non-empty canonical DBM that represents the clock zone Z and D is the depth of the mode which signifies the number of transitions required to reach the mode x (from some initial mode).

Now we discuss the elements used in our algorithm, Algorithm 3.1. First we present the *Image* operator which will be used in our algorithm. Given a mode x and the transition set T, *Image*(x, T) computes the set of modes (successor modes) that can be reached in one transition [54]

$$\textit{Image}(x, T) := \{x' \in X \mid (x, \sigma, g', u', x') \in T\}.$$

The *Image* operator returns the set of modes successor to x or is empty if there are no outgoing transitions from x.

Algorithm 3.1 relies on the following data structures:

(1) The variable X_s is of type set(mode); it stores the set of successor modes obtained by the *Image* operator. Initially, the variable is initialized as **EmptySet** that corresponds to the set being empty.

(2) The variable zoneStack is of type stack(zoneStack). It stores the reachable zones that are explored during the on-the-fly modified BFS algorithm. As usual, the stack follows **Last In First Out** (LIFO) principle. The operations that can be performed on the stack data structure are: (1) The stack is initialized as an **EmptyStack** which corresponds to the stack being empty; (2) the procedure **pop** takes the stack and removes the zone from the top of the stack and updates the stack; (3) the procedure **push** updates the stack by adding the reachable zone that has been explored to the top of the stack.

Algorithm 3.1 is explained as follows

(1) The input to the algorithm is the state $q_{l,x} = (x, \boldsymbol{c}_{l,x})$, a timed automaton $G'_{TA,x}$ and a positive integer δ. In lines 6-8, the initial zone is pushed into zoneStack with mode x and DBM R_l. DBM R_l has its $(0, i)$th entries in the first row as $-l_\sigma$ and $(i, 0)$th entries in the first column as l_σ only for the events defined at mode x. Rest all the entries of R_l are zero. A variable t_p is used to hold the duration of one state trajectory which is then used to compare the duration with another state trajectory. Initially t_p is initialized to ∞.

(2) The loop is iterated until zoneStack is empty. If zoneStack is not empty, then the last zone in the stack is popped. In lines 11 and 12, all the successor modes of x are obtained by the *Image* function and the depth of these successor modes is incremented by 1 to represent the depth from the mode x. If there are no successor modes from x, then another zone is popped from zoneStack and the process repeats. In line 11, T is the transition set of $G'_{TA,x}$.

(3) Lines 13-23 check if the successor modes are reachable from the current mode by forming their corresponding DBMs and testing them for emptiness. For a zone (x, R, D), the effect of

Algorithm 3.1 Algorithm for Calculating $t'_{min}(q_{l,x}, \delta)$

1: **procedure** $t'min(q_{l,x}, G'_{TA,x}, \delta)$
2: **Input:** $G'_{TA,x}$ A timed automaton, δ A positive integer, $q_{l,x} = (x, \boldsymbol{c}_{l,x})$ state of the TA.
3: **Output:** t'_{min}.
4: stack(zoneStack) := EmptyStack;
5: set(mode) x_s := EmptySet;
6: $D_0 := 0$;
7: $t_p := \infty$;
8: push((x, R_l, D_0), zoneStack);
9: **while** (**not** empty(zoneStack)) **do**
10: (x, R, D, t) := pop(zoneStack);
11: X_s := Image(x, T); ▷ T is the transition set of TA $G'_{TA,x}$.
12: $D_s := D + 1$;
13: **if** (**not** empty X_s and $D_s \leq \delta$) **then**
14: R := ClockInvariant($R, Inv(x)$); ▷ Calls Algorithm 2.3.
15: R := Canonical(R); ▷ Calls Algorithm 2.1.
16: **if** (**not** empty R) **then**
17: **for** *every* $(x, \sigma_j, g_j, u_j, x_{s_j}) : x_{s_j} \in X_s$ **do**
18: $R[0, j] := \min(R[0, j], -l_{\sigma_j})$;
19: R_{s_j} := Canonical(R); ▷ Calls Algorithm 2.1.
20: **if** (**not** empty R_{s_j}) **then**;
21: R_{s_j} := ResetClock(R_{s_j}, I_{u_j}); ▷ Calls Algorithm 2.2.
22: push((x_{s_j}, R_{s_j}, D_s), zoneStack);
23: **if** ($D_s = \delta$) **then**
24: $\mathcal{D}(\mathcal{T}_j) := -R_{s_j}[0, 1]$;
25: $t'_{min}(q_{l,x}, \delta) = min(\mathcal{D}(\mathcal{T}_j), t_p)$;
26: $t_p = t'_{min}(q_{l,x}, \delta)$;
27: **end if**
28: **end if**
29: **end for**
30: **end if**
31: **end if**
32: **end while**
33: **return** $t'_{min}(q_{l,x}, \delta)$.
34: **end procedure**

time elapsed is captured by passing the DBM R and the $Inv(x)$ through Algorithm 2.3 which returns the updated DBM R according to $Inv(x)$. As mentioned previously in Section 2.6.1 of Chapter 2, Algorithm 2.3 excepts DBM of dimension $(n_e + 1)$, however, in this case Algorithm 2.3 is called by passing DBM of dimension (n_e+2). This change in the dimension of DBM does not affect output of Algorithm 2.3. DBM R is tested for emptiness and if not empty is made canonical by using Algorithm 2.1. DBM R of dimension $(n_e + 2)$ is passed

Algorithm 3.2 Algorithm for calculating $T'_{min,low}(\delta)$.

```
1:  procedure TMIN(G'_{TA,x}, δ)
2:      Input: G'_{TA,x} = (X, Σ, C, Inv, T, x_0) A timed automaton;
3:      Input: δ A positive integer.
4:      Output: T'_{min,low}.
5:      for all x ∈ X do
6:          q_{l,x} := (x, c_{l,x});
7:          T_val[0, i] := t'min(q_{l,x}, G'_{TA,x}, δ);        ▷ Calls Algorithm 3.1.
8:      end for
9:      T'_{min,low}(δ) := min(T_val);
10:     return T'_{min,low}(δ).
11: end procedure
```

to Algorithm 2.1 which does not affect the output of the algorithm. In line 18, according to the guard g_j, DBM R is updated by setting the $(0, j)$th entry to $\min(R(0, j), -l_{\sigma_j})$. DBM R is tested for emptiness; if not empty, then $G'_{TA,x}$ enters the mode x_{s_j} and the resulting clock zone is made canonical and is denoted by R_{s_j}; otherwise (if R is empty) the clock zones for other outgoing transitions from x to the successor modes are tested for emptiness. Following Def. 3.3, the duration of the state trajectory up to the length δ is given in line 24. Some or all of the entries of DBM R_{s_j} are reset to zero according to the update set u_j of the corresponding transition. The resulting DBM R_{s_j}, the corresponding successor mode x_{s_j} and the depth of the successor mode D_j are pushed to the top of zoneStack. The DBM for all transitions from x to the successor modes are formed and tested for emptiness. The search continues by popping the last successor zone from zoneStack and repeating the same process until zoneStack is empty.

(4) When the depth of the successor modes is equal to δ, the total duration of the state trajectory is compared with the duration of the previous state trajectory and the minimum of the two is stored. In this way, $t'_{min}(q_{l,x}, \delta)$ is calculated. Finally, the algorithm terminates when all state trajectories of length δ are explored from the $q_{l,x}$ and provides $t'_{min}(q_{l,x}, \delta)$.

Algorithm 3.1 uses the framework of the modifided BFS algorithm because during the exploration a mode can be visited more than once. Also, the algorithm will terminate as soon as all of the trajectories in $\mathcal{S}(q_l, \delta)$ are explored.

Algorithm 3.2 calculates $T'_{min,low}(\delta)$ by applying Algorithm 3.1 to every mode of the TA. Following Def. 3.7, Algorithm 3.2 is easy to understand and calculates $t'_{min}(q_{l,x}, \delta)$ from each mode x of $G'_{TA,x}$. The algorithm terminates when $t'_{min}(q_{l,x}, \delta)$ is calculated for every mode of $G'_{TA,x}$ (reachable in G) and returns $T'_{min,low}(\delta)$, for the given $\delta \in \mathbb{Z}^+$.

As previously discussed, a full blown reachability analysis over the timed system requires time complexity exponential in the number of events. We shall see in the next section that polynomial time complexity is required by the Algorithm 3.2 to calculate $T'_{min,low}(\delta)$.

Example 3.4 Continued

Example 3.4 demonstrates the calculation of minimum execution duration for just one event sequence. We shall continue the same example with $\delta = 3$ for calculating $T'_{min,low}(\delta)$ for G_{TA} shown in Fig. 3.6. Initially, Algorithm 3.2 calls Algorithm 3.1 for calculating $t'_{min}((x_0, \boldsymbol{c}_{l,x_0}), 3)$. Then, Algorithm 3.1 unfolds all the event sequences of length 3 of G_{TA} shown in Fig. 3.12. In addition, Algorithm 3.1 performs the calculation of determining minimum execution duration of all trajectories in Fig. 3.12 similar to the calculation demonstrated in Example 3.4 resulting in $t'_{min}((x_0, \boldsymbol{c}_{l,x_0}), 3) = 2$. Algorithm 3.2 repeats the process of calculating $t'_{min}(q_{l,x}, 3)$ (by repeatedly calling Algorithm 3.1) for remaining modes of G_{TA} resulting in:

- $t'_{min}((x_1, \boldsymbol{c}_l), 3) = 4$,
- $t'_{min}((x_2, \boldsymbol{c}_l), 3) = 4$,
- $t'_{min}((x_3, \boldsymbol{c}_l), 3) = 4$.
- $t'_{min}((x_4, \boldsymbol{c}_l), 3) = 4$.

Therefore, Algorithm 3.2 returns $T'_{min,low}(3) = 2$ for G_{TA} (Fig. 3.6).

3.6 Time Complexity

Computational complexity is the measure of the amount of computing resources such as time and space required during the run-time of an algorithm on large inputs [55]. In this thesis, only time

complexity is calculated. The time complexity is an aspect of the computational complexity which is the measure of the amount of computer time taken to run an algorithm on large inputs. To obtain the computational complexity of an algorithm, the asymtotic “big-O" notation is used. The “big-O" notation describes the worst case run-time of an algorithm. We will use the concept of floating-point operation (flop) for calculating $O(\cdot)$ [56]. A flop is the amount computer time required to execute a statement. The flop includes the operations such as one addition, one subtraction, one multiplication, one division and some subscript manipulation. A multiplication or division when coupled with addition or subtraction is one flop. All DBM algorithms as discussed in Chapter 2 involve matrix manipulations. Therefore, to determine the time complexity of an algorithm, it is necessary to determine the total flop count required in the matrix manipulations. Moreover, during the asymptotic analysis for large inputs, we only consider the higher order term and we drop the coefficient of this term and ignore the other lower order term because the higher order term dominants other lower order terms.

The input to Algorithm 2.1 is a $(n_e + 1) \times (n_e + 1)$ square DBM which consists of three loops which runs from 0 to n_e to carry out the matrix manipulation such as the $\min$ operation that calculates the minimum between two entries of a DBM. In the algorithm the loops m, i and j run from 0 to n_e and therefore it takes $O((n_e + 1)^3)$ flops [4]. The highest order of the polynomial is significant and therefore, the worst-case flops for the algorithm is given below

$$O(n_e^3). \tag{18}$$

Algorithm 2.2 for resetting DBM, consists of the input as $(n_e + 1) \times (n_e + 1)$ square DBM and the index set I_u with n_e elements. It is possible that there exists a transition in a TA, such that on taking the transition all the clocks are reset to zero. Therefore the first loop i runs from 1 to n_e. The second loop j also runs from 1 to n_e as well and therefore, the worst-case flop count for this algorithm is

$$O(n_e^2). \tag{19}$$

Algorithm 2.3 for determining the time elapse in the modes has a $(n_e + 1) \times (n_e + 1)$ square DBM and the invariant condition Inv for a mode. To transform the DBM that represents the time elapse

according to the invariants, the loop i runs from 0 to n_e. Consequently, in the worst-case the total flops required are

$$O(n_e). \tag{20}$$

Note that when Algorithm 3.1 calls Algorithm 2.1, Algorithm 2.2 and Algorithm 2.3, the DBM passed through these algorithms has a dimension of $(n_e + 2)$. *However, changes in the dimension will not change the time complexity of Algorithm 2.1, Algorithm 2.2 and Algorithm 2.3.*

Before moving on to the time complexity of Algorithm 3.1 and Algorithm 3.2, let us first discuss the time complexity of the BFS algorithm. Consider a graph with $|V|$ as the number of vertices and $|E|$ as the number of edges between the vertices. Then the time complexity of the BFS algorithm to traverse the graph is $O(|V| + |E|)$ [57].

Consider the TA $G'_{TA,x} = (X, \Sigma, C, Inv, T, x_0)$. Let us define some notations:

- $n_X = |X|$ is the total number of modes of G_{TA};
- $n_e = |\Sigma| = |C|$ the number of events of G_{TA};
- $|T_{out}|$ is the total number of outgoing transitions from a mode $x \in X$.

Algorithm 3.1 consists of two loops. The first loop is the *while* loop which is the outer loop and the second loop is *for* loop which is the inner loop. The for loop terminates when all the successor modes from the current mode x are explored. Within the for loop the push and the pop operation on the zones in the stack data structure (in lines 11 and 27 of Algorithm 3.1 respectively) requires $O(1)$ flop each. In the worst case, if all the outgoing transitions to the successor modes from the current mode x within the for loop are possible then pushing the zones of the successor modes of x in the zoneStack requires $|T_{out}| \cdot O(1)$ flops. It is to be noted that $|T_{out}|$ is not the same for every $x \in X$. The *image* operator in line 12 of Algorithm 3.1 requires $O(T)$ flops. Suppose R is the clock zone for the current mode x, then $O(n_e)$ flops is time required to update R using Algorithm 2.3 (ClockInvariant function). Next, the updated R requires $O(n_e^3)$ flops for its canonicalization. In Algorithm 3.1, canonicalization operation is used again at line 21. This canonicalization operation is repeated for T_{out} times from x in order to check the possibility of all the outgoing transitions to

the successor modes, thus in worst case situation requiring $|T_{out}| \cdot O(e^3)$ flops. If all the outgoing transitions to the successor modes from x are possible, then resetting (ResetClock function at line 26 in Algorithm 3.1) of their corresponding DBMs requires $|T_{out}| \cdot O(e^2)$ flops. Therefore, in the worst case, the time complexity required by one mode of the TA is

$$O(1) + O(T) + O(n_e) + O(n_e^3) + |T_{out}| \cdot O(n_e^3) + |T_{out}| \cdot O(n_e^2) + |T_{out}| \cdot O(1). \tag{21}$$

The outer while loop terminates when the stack data structure is empty. Algorithm 3.1 traverses through all the trajectories of length δ from the current mode x resulting in $\frac{n_e^{\delta+1}-1}{n_e-1}$ modes. Therefore, in the worst case, the time complexity for Algorithm 3.1 is given as (equation 21 is simplified by dropping the asymptotic O notation)

$$\begin{aligned} \textit{Time complexity} &= \left(\frac{n_e^{\delta+1}-1}{n_e-1}\right) \cdot [(1) + (T) + (n_e) + (n_e^3) + |T_{out}| \cdot (n_e^3) + |T_{out}| \cdot (n_e^2) \\ &\quad + |T_{out}|], \end{aligned} \tag{22}$$

$$\begin{aligned} &= \frac{1}{n_e-1}(n_e^{\delta+1}[(1) + (T) + (n_e) + (n_e^3) + |T_{out}| \cdot (n_e^3) + |T_{out}| \cdot (n_e^2) \\ &\quad + |T_{out}|] - [(1) + (T) + (n_e) + (n_e^3) + |T_{out}| \cdot (n_e^3) + |T_{out}| \cdot (n_e^2) \\ &\quad + |T_{out}|]), \end{aligned} \tag{23}$$

$$\begin{aligned} &= \frac{1}{n_e-1}([(n_e^{\delta+1}) + (T) \cdot (n_e^{\delta+1}) + (n_e^{\delta+2}) + (n_e^{\delta+4}) + |T_{out}| \cdot (n_e^{\delta+4}) \\ &\quad + |T_{out}| \cdot (n_e^{\delta+3}) + |T_{out}| \cdot (n_e^{\delta+1})] - [(1) + (T) + (n_e) + (n_e^3) \\ &\quad + |T_{out}| \cdot (n_e^3) + |T_{out}| \cdot (n_e^2) + |T_{out}|]), \end{aligned} \tag{24}$$

Substituting $|T_{out}| = n_e$

$$\begin{aligned} &= \frac{1}{n_e-1}([(n_e^{\delta+1}) + (T) \cdot (n_e^{\delta+1}) + (n_e^{\delta+2}) + (n_e^{\delta+4}) + (n_e^{\delta+5}) + (n_e^{\delta+4}) \\ &\quad + (n_e^{\delta+2})] - [(1) + T + (n_e) + (n_e^3) + (n_e^4) + (n_e^3) + (n_e)]). \end{aligned} \tag{25}$$

Substituting $T = (n_X) \cdot (n_e)$

$$= \frac{1}{n_e - 1}([(n_e^{\delta+1}) + (n_X) \cdot (n_e) \cdot (n_e^{\delta+1}) + (n_e^{\delta+2}) + (n_e^{\delta+4}) + (n_e^{\delta+5}) + (n_e^{\delta+4}) + (n_e^{\delta+2})] - [(1) + (n_X) \cdot (n_e) + (n_e) + (n_e^3) + (n_e^4) + (n_e^3) + (n_e)]). \tag{26}$$

Algorithm 3.2 calls Algorithm 3.1 for each mode of the TA. Therefore, equation (25) is multiplied with (n_X) and we get the following time complexity for Algorithm 3.2

$$\textit{Time Complexity} = \frac{(n_X)}{n_e - 1}([(n_e^{\delta+1}) + (n_X) \cdot (n_e^{\delta+2}) + (n_e^{\delta+2}) + (n_e^{\delta+4}) + (n_e^{\delta+5}) + (n_e^{\delta+4}) + (n_e^{\delta+2})] - [(1) + (n_X) \cdot (n_e) + (n_e) + (n_e^3) + (n_e^4) + (n_e^3) + (n_e)]). \tag{27}$$

$$= \frac{1}{n_e - 1}([(n_X) \cdot (n_e^{\delta+1}) + (n_X^2) \cdot (n_e^{\delta+2}) + (n_X) \cdot (n_e^{\delta+2}) + (n_X) \cdot (n_e^{\delta+4}) + (n_X) \cdot (n_e^{\delta+5}) + (n_X) \cdot (n_e^{\delta+4}) + (n_X) \cdot (n_e^{\delta+2})] - [(n_X) + (n_X^2) \cdot (n_e) + (n_X) \cdot (n_e) + (n_X) \cdot (n_e^3) + (n_X) \cdot (n_e^4) + (n_X) \cdot (n_e^3) + (n_X) \cdot (n_e)]), \tag{28}$$

$$= \frac{1}{n_e - 1}([(n_X) \cdot (n_e^{\delta+1}) + (n_X^2) \cdot (n_e^{\delta+2}) + 2 \cdot (n_X) \cdot (n_e^{\delta+2}) + 2 \cdot (n_X) \cdot (n_e^{\delta+4}) + (n_X) \cdot (n_e^{\delta+5})] - [(n_X) + (n_X^2) \cdot (n_e) + 2 \cdot (n_X) \cdot (n_e) + 2 \cdot (n_X) \cdot (n_e^3) + (n_X) \cdot (n_e^4)]). \tag{29}$$

All the lower order terms and the coefficients of the higher order term are disregarded in the equation (28) and we get the following equation in terms of (n_X) and (n_e) as

$$\textit{Time Complexity} = O\left((n_X) \cdot (n_e^{\delta+5})\right). \tag{30}$$

Equation (29) is the time complexity for Algorithm 3.2. It can be observed that Algorithm 3.2 for

determining $T'_{min,low}(\delta)$ requires time complexity which is polynomial in the number of modes. However, an algorithm for determining $T_{min}(\delta)$ using clock regions would require much higher (double exponential) time complexity, because the number of clock region are exponential in the number of clocks (refer to Example 3.1).

3.7 Summary

In this chapter we presented the problem formulation and we discussed about exponential computational complexity required in determining the minimum execution duration of event sequences using clock regions. Then, we presented a transformation procedure to augment an untimed automaton to TA in order to calculate $T_{min}(\delta)$. Then we discussed about specific adjustments that should be made to the clock variables of TA in order to calculate $T_{min}(\delta)$. This adjustment reduces the computational complexity at the expense of lower bound for $T_{min}(\delta)$. We developed an algorithm using DBMs to calculate a lower bound for $T_{min}(\delta)$ by adjusting the clock variables in DBM. Finally, we provided the time complexity for the aforementioned algorithm. In the next section we shall review in detail the hardware, modeling and designing a conventional supervisor for the system of the solar tracker setup used in our case study.

Chapter 4

Experimental Setup

Later in this thesis we will use the algorithm proposed in Chapter 3 to design and implement LLP with Buffering for the supervisory control of solar tracker system.

In this chapter we will discuss the modelling of the solar tracker system. In Section 4.1 we will discuss briefly the schematic diagram and the design requirements of the solar tracker system. Section 4.2 provides a model of the solar tracker system as an untimed DES along with the synthesis of a conventional supervisor according to the specifications. In Chapter 5 we will discuss control using LLP with Buffering.

4.1 Dual Axis Solar Tracker System

The system chosen is a two degree-of-freedom solar tracker system shown in Fig. 4.1. The system was developed in [6] to implement conventional supervisory control [58].

4.1.1 Schematic Diagram

The schematic diagram of the solar tracker system is depicted in the Fig. 4.2. The complete system consists of two main subsystems:

(1) Remote station (Solar tracker system),

(2) Ground station (Computer).

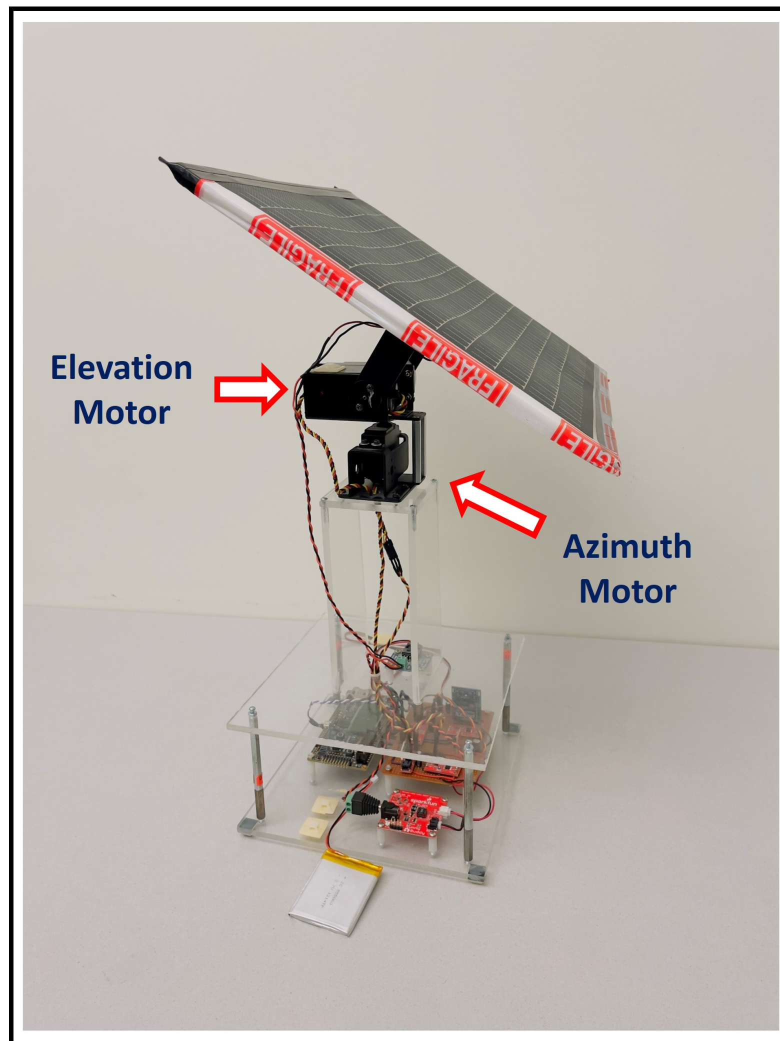


Figure 4.1: Two degree-of-freedom solar tracker system.

Remote Station

The remote station is the dual axis solar tracker system which can be thought of as a subsystem of a satellite (e.g. a cubesat) with the responsibility of providing electrical energy. The solar tracker system is equipped with a photovoltaic (PV) panel, two servo motors, a microcontroller, a battery, and a communication module. The PV panel (PT15-300) is mounted on a pan-tilt assembly and is maneuvered by two position controlled servo motors. The PV panel captures the solar energy and converts it into electrical energy. The power provided by the PV panel is regulated by a maximum power point tracker device (MPPT) to an appropriate operating voltage and then distributed to the components of the system. Excess power that is produced by the PV panel is stored in the on-board

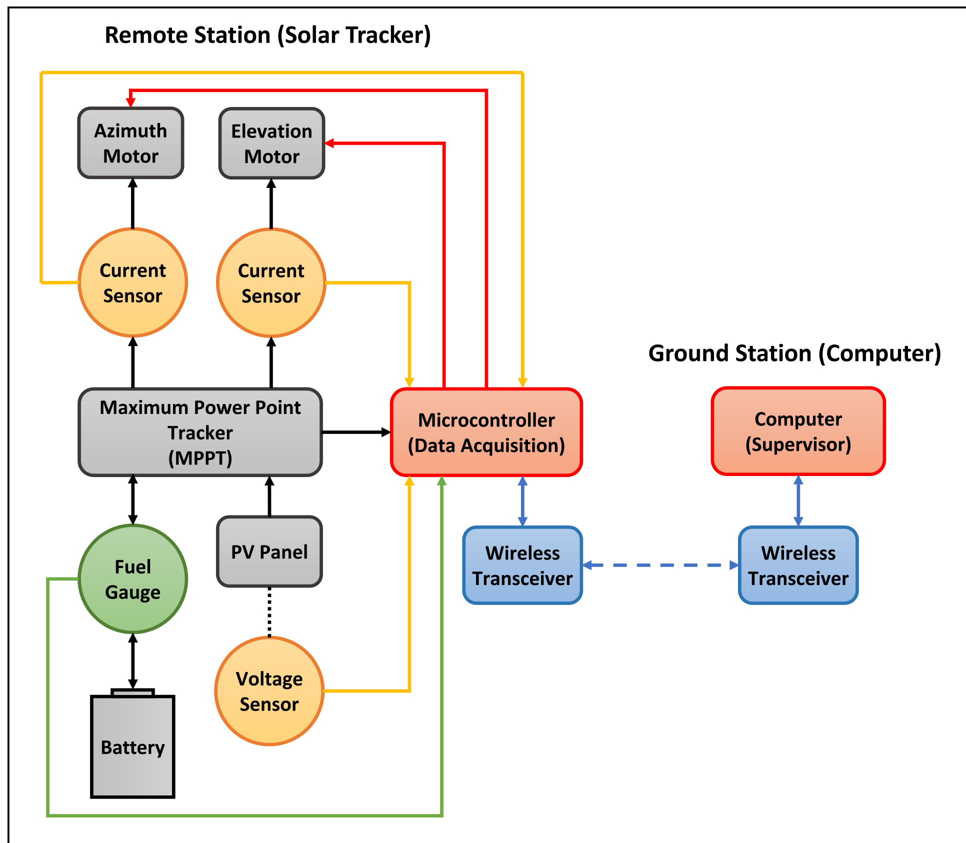


Figure 4.2: Schematic diagram of solar tracker system [5].

Table 4.1: Legend for schematic diagram of solar tracker system.

Legend	
Red line	PWM signal
Green line	I2C serial
Black line	DC power
Yellow line	Analog signal
Blue line	UART serial voltage
Blue dotted line	Wireless RF communication
Black dotted line	Voltage measurement

battery of the system. The on-board battery used is a single cell 3.7 V LiPo battery with a capacity of 2500 mAH supply. When insufficient power is generated by the PV panel, the on-board battery is used to power the system components. We initially assume that the battery is fully charged.[1] The two position controlled servo motors used are azimuth (HS-645MG servo motor) and elevation

[1] This adjustment is made to simplify the tests on the system. In a practical setting this can be adjusted such that the system can start from any initial charge.

(HS-805BB servo motor) servo motors providing two degrees of freedom (2-DOF) for moving the PV panel. Both the servo motors are capable of moving in clockwise (CW) and counterclockwise (CCW) direction with an operating range of $180°$. Fig. 4.3 illustrates the coordinate system for the solar tracker system. Azimuth is measure along the X axis and elevation is measured along the Y axis of the transverse plane of the PV panel. The rotation of the azimuth motor in the CCW direction around the positive X axis corresponds to the PV panel turning left and the rotation of the azimuth motor in the CW direction around negative X axis corresponds to the PV panel turning right. Similarly, the rotation of the elevation motor in the CCW direction around the positive Y axis corresponds to the PV panel moving in upwards and the rotation of the elevation motor in the CW direction around the negative Y axis corresponds to the PV panel moving in downwards.

The elevation angle θ_{EL} is measured with respect to horizon. The elevation motor is mechanically restricted, resulting in a range of $-45° \leq \theta_{EL} \leq 45°$. Initially, θ_{EL} is set to $45°$ (fully CCW). The azimuth motor provides a range of $90°$ to either left or right, resulting in a total $180°$ range. The initial angle for azimuth is chosen to $\theta_{AZ} = 90°$. Thus the range of θ_{AZ} will be $0° \leq \theta_{AZ} \leq 180°$. Both motors are controlled by a microcontroller sending pulse width modulation (PWM) control signals. The servomotors are capable of rotating at a very high speed and could induce damage to the system. Therefore, for the safe operation of the system, the servomotors are limited to a $2°$ rotation for a single step rotation followed by a 2 seconds wait time after every single step rotation. There are four sensors employed for monitoring the state of system. A voltage sensor measures the voltage generated by the PV panel (which depends on to the illumination level of the PV panel). There are two current sensors employed to monitor the current consumption by the servomotors after every movement command from the microcontroller. If and when a current sensor senses that a motor is continuously drawing current, then that motor is assumed to have failed. For simplicity, in this implementation only the elevation motor is assumed to have failure condition. A fuel gauge monitors the state of charge (SOC) of the battery and reports the SOC in percentage to the microcontroller. All sensor data is polled at 50 ms intervals (i.e. 50 ms sampling time) and stored in the local memory of the microcontroller.

The microcontroller used in the solar tracker system is EFM32$^{\text{TM}}$ Leopard Gecko which is a 32-bit

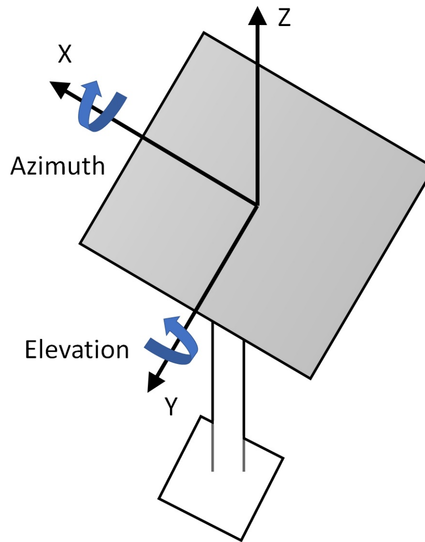


Figure 4.3: Coordinate system for solar tracker system.

ARM Cortex-M3 processor with 48 MHz, 256 kB flash memory and 32 kB RAM. This micro-controller is ideal for low energy consumption battery operated applications. The microcontroller acts as an interface between the remote station and the ground station and is responsible for performing data acquisition and communication. The micro-controller sends and receives control signals from the ground station through a serial communication port (wireless transceiver).

Ground Station

The ground station is a computer equipped with a wireless serial communication port for communicating with the remote station. The ground station receives the sensor data from the microcontroller and performs the supervisory computations and then transmits the appropriate supervisory commands to the remote station. The computer has Intel(R) Core(TM) i5-2400 processor operating at 3.10 GHz and 8 GB RAM.

More details about the hardware implementation can be found in [6,58]. The implementation of the LLP supervisor with Buffering along with software implementation can be found in [3]. More details about the optimized code for the computations of LLP supervisor with Buffering implemented in the computer can be found in [5].

4.1.2 System Design Requirements

The main objective of the solar tracker system is to discover a sufficiently bright light source by scanning the surrounding and holding that position in order to charge the on-board battery from the solar power generated by the PV panel. In this implementation, the operator of ground station acts a master controller (MC) which issues commands to tracker to perform maneuvers to discover the bright light source. This set of maneuvers are initiated by the “Full Sweep" command. Upon reception of this command, the solar tracker system initiates a predetermined path where both the motors rotate in such a way that allows PV panel to sweep the entire surrounding hemisphere. Note that the supervisory control algorithm are implemented in computer. Therefore control commands are calculated in the computer and then transmitted to the solar tracker. During the full sweep, the solar tracker system continuously monitors its voltage sensor to check the illumination levels of the PV panel. The solar tracker system ceases its motor movements when a bright light source is detected and indicates to the MC that the sweep was successful; otherwise, the motors continues to rotate until both the motors reache their respective maximum positions. If no bright light source is detected, then the solar tracker system indicates to the MC that the sweep was a failure. Moreover, during the maneuver if there is an obstruction preventing the elevation motor from moving further (i.e. failure), then the solar tracker system informs the MC that there is a failure in the elevation motor. The responses provided by the solar tracker system to the MC as described above are:

- Bright Detected
- Sweep Failure
- Elevation Motor Failure.

The system has two other safety requirements that will be later discussed in Section 4.2.3.

4.2 Untimed DES Model

The solar tracker system described in the previous section is modelled as a discrete event system (using finite automatons). Initially, all the components of the solar tracker systems along with their interactions are modelled to form the plant. Next, the specifications for the solar tracker plant are

formulated which specify the set of legal behaviours that the plant must adhere to in order to prevent any unwanted behaviour. Finally, the set of controllable and uncontrollable events are defined for the plant and the conventional supervisor is computed. The conventional supervisor is used later in Chapter 5 for comparison with LLP supervisor.

4.2.1 Components

The block diagram of the solar tracker system containing various components of the system is shown in Fig. 4.4.

Battery

The battery automaton illustrated in the Fig. 4.5 represents the battery SOC (measured by the fuel gauge sensor interfaced with the microcontroller). The battery SOC changes when both the servomotors actuate resulting in a drop in the battery voltage. To incorporate this, the battery SOC model consists of three different states: Critical, Safe and Full. The state transition occurs when the

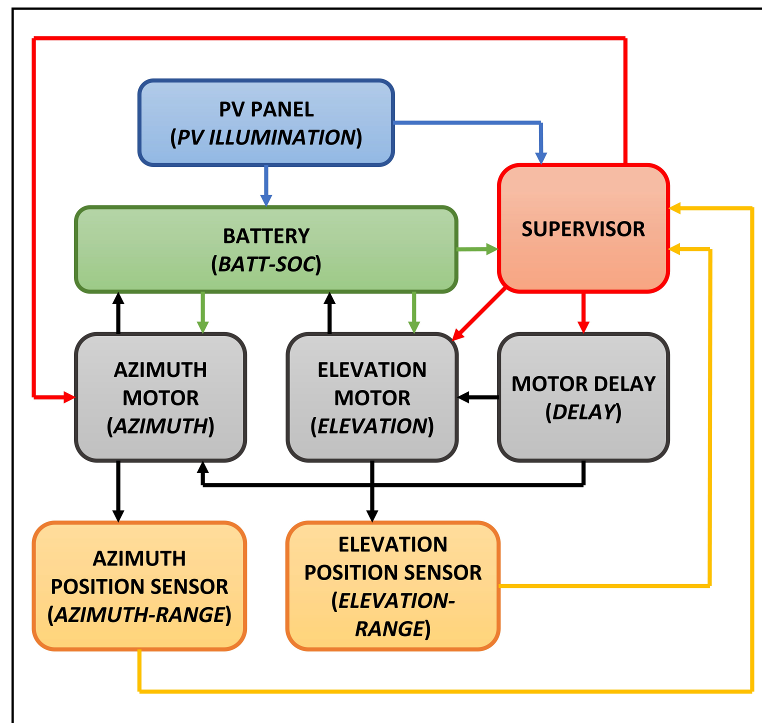


Figure 4.4: Block diagram for the solar tracker system.

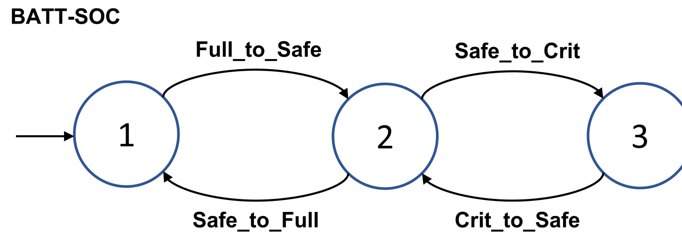


Figure 4.5: Battery SOC automaton [5].

Table 4.2: States of the battery SOC automaton.

State Number	State
1	Full
2	Safe
3	Critical

Table 4.3: Event list of battery SOC with thresholds over events [5].

Event	Battery SOC Threshold (%)	Controllability
Full_to_Safe	90	Uncontrollable
Safe_to_Full	95	Uncontrollable
Safe_to_Crit	50	Uncontrollable
Crit_to_Safe	55	Uncontrollable

battery SOC increases while charging or decreases due to motor movements. Table 4.2 describes the state number associated with the SOC of the battery.

In the battery SOC automaton, there are four uncontrollable events describing the crossing of specific thresholds defined in Table 4.3. The reason for these events to be uncontrollable is because the charging and discharging of the battery cannot be controlled. When either motor or both move, a sudden high amount of current is drawn from the battery and any noise in the sensor readings can lead to unwanted triggering of the events when the readings is around threshold. Therefore, a 5% hysteresis is included for each transition to prevent event triggering due to the noise. For example, Safe_to_Full occurs at 95% threshold whereas Full_to_Safe occurs at 90%.

Photovoltaic (PV) Panel

The automaton for the PV panel component is illustrated in Fig. 4.6. The principle objective of the solar tracker is to find a bright light source during one of its maneuvers. The PV panel generates an output voltage depending upon the intensity and the incidence angle of the light coming from

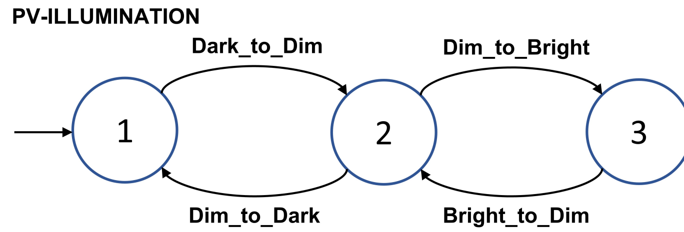


Figure 4.6: PV panel automaton [5].

Table 4.4: States of the PV panel automaton.

State Number	State
1	Dark
2	Dim
3	Bright

Table 4.5: Event list of PV panel with thresholds voltages over events [5].

Event	Threshold Voltage (V)	Controllability
Dark_to_Dim	6	Uncontrollable
Dim_to_Dark	5	Uncontrollable
Dim_to_Bright	16	Uncontrollable
Bright_to_Dim	15	Uncontrollable

light source (which itself depends on the orientation of the PV panel). The maximum output voltage is generated by the PV panel when the light source is perpendicular to the surface of the PV panel. Table 4.4 shows three states of the PV panel automaton which are Dark, Dim and Bright. These states corresponds to the illumination level of the PV panel measured by the voltage sensor by measuring the output voltage generated by the PV panel.

There are total four uncontrollable events which are triggered when the output voltage measured by the voltage sensor crosses a particular threshold voltage. The output voltage of the PV panel can be effected by the noise in the measured voltage and as a result, undesired triggering of events can occur when the output voltage fluctuates around the threshold value. Consequently, the thresholds are separated by a hysteresis of 1 V to avert this undesired situation. Table 4.5 provides the events along with the threshold voltage for triggering of them.

Azimuth Motor

The azimuth motor automaton contains of three states and four events as illustrated in Fig. 4.7. The states of the azimuth motor automaton are given in Table 4.6. The initial state is the Idle state which corresponds to the azimuth motor being idle (i.e. not moving). The microcontroller can command to move the motor in either CW or CCW direction when the motor is idle. To prevent any damage to the PV panel, the movement of the motor is limited to 2° for each movement command. Since these commands are issued by the supervisor, they are considered to be controllable events (depicted in green). During the movement of the motor, the motor draws current from the battery and as soon as the motor movement is complete, the current decreases below 500 mA. When the current decreases below 500 mA, AZ_CW_OK or AZ_CCW_OK event occurs matching the motor movement command issued by the microcontroller. Then the motor returns to the idle state. These "OK" events are uncontrollable events since they depend on the sensor readings. For simplicity it is assumed that the azimuth motor is fault free. All the events, along with their current readings, are shown in Table 4.7.

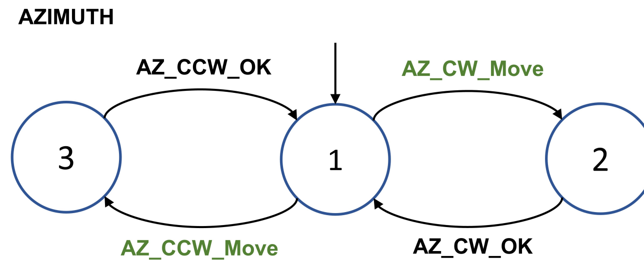


Figure 4.7: Azimuth motor automaton [5].

Table 4.6: States of the azimuth motor automaton.

State Number	State
1	Idle
2	Azimuth Turning CW
3	Azimuth Turning CCW

Table 4.7: Event list of azimuth motor automaton with current readings (mA) [5].

Event	Measured Current (mA)	Controllability
AZ_CW_MOVE	N/A	Controllable
AZ_CCW_MOVE	N/A	Controllable
AZ_CW_OK	≤ 500	Uncontrollable
AZ_CCW_OK	≤ 500	Uncontrollable

Elevation Motor

The elevation motor automaton has a similar description as the azimuth motor automaton with an additional “Elevation Motor Failed" state. The elevation motor enters the failed state whenever there is a obstruction in the motion of motor shaft or due to some electrical problem in the motor. The states of the elevation motor are given in Table 4.8.

As illustrated in Fig. 4.8, the elevation motor can either move in CW (CCW) direction through

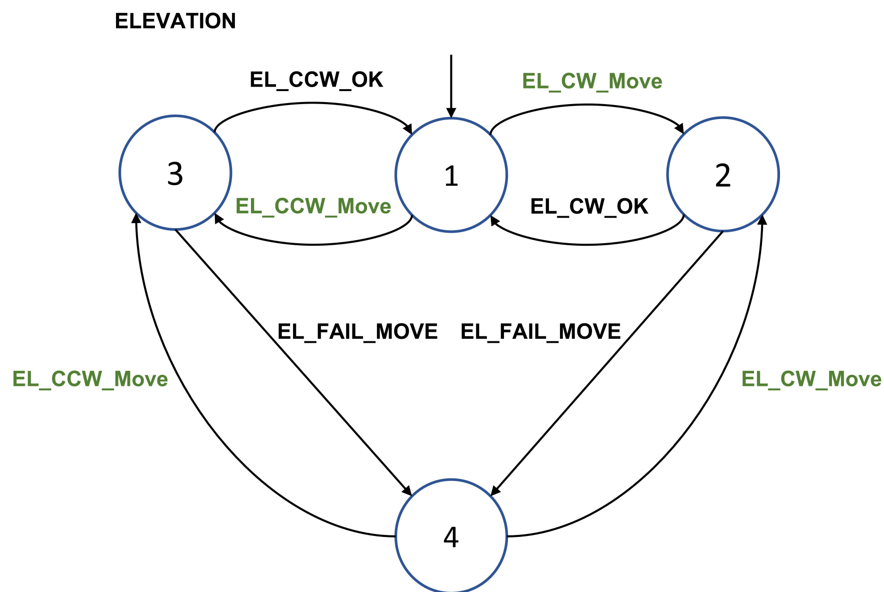


Figure 4.8: Elevation motor automaton [5].

Table 4.8: States of the elevation motor automaton.

State Number	State
1	Idle
2	Elevation Turning CW
3	Elevation Turning CCW
4	Elevation Motor Failed

Table 4.9: Event list of elevation motor with current readings (mA) [5].

Event	Measured Current (mA)	Controllability
EL_CW_MOVE	N/A	Controllable
EL_CCW_MOVE	N/A	Controllable
EL_CW_OK	≤ 500	Uncontrollable
EL_FAIL_MOVE	≥ 500	Uncontrollable
EL_CCW_OK	≤ 500	Uncontrollable

the controllable event EL_CW_MOVE (EL_CCW_MOVE) followed by the uncontrollable event EL_CCW_OK (EL_CCW_OK). After any of the motor motion events, if the average current over a 2 second time frame still remains above 500mA, then EL_FAIL_MOVE event is triggered. The elevation motor has entered the Elevation Motor Failed state. At the Elevation Motor Failed state, the supervisor can issue a move command again for the elevation motor to move in CW or CCW direction which may result in a successful motor movement. Therefore, the elevation motor failed state could be temporary. The above-mentioned events are listed in Table 4.9.

Servomotor Delay Procedure

The safety measure of two-second delay is introduced after every motor movement is completed to ensure that there is no damage to the PV panel and to ensure smooth operation of the system. The delay is implemented in the control software and is modelled here as an automaton shown in Fig. 4.9 with the states for this automaton given in Table 4.10. After an azimuth or elevation motor rotation in either CW or CCW direction is completed, a Wait_2_Sec event is triggered (by the microcontroller software) which ensures that there is a two-second delay followed by a respective OK event for the motor to guarantee that the motor motion is completed and the motors are in the Motor Idle state.

Table 4.10: States of the servomotor delay automaton.

State Number	State
1	Motor Idle
2	In Waiting
3	Checking Current

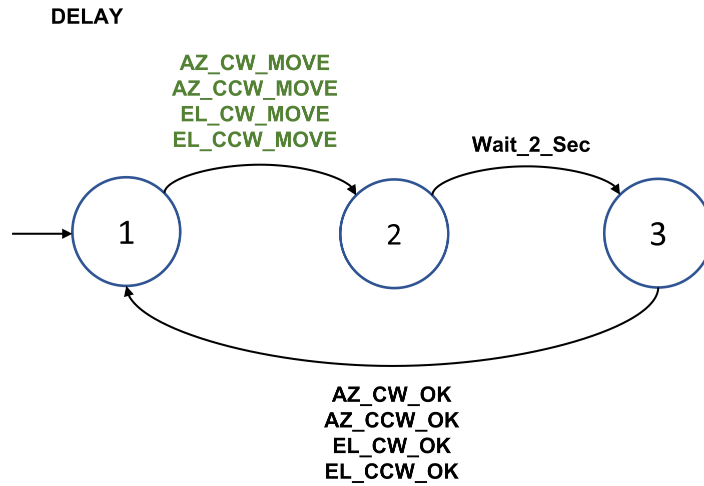


Figure 4.9: Servomotor delay automaton [5].

Table 4.11: Event list of servomotor delay automaton [5].

Event	Controllability
AZ_CW_MOVE	Controllable
AZ_CCW_MOVE	Controllable
EL_CW_MOVE	Controllable
EL_CCW_MOVE	Controllable
Wait_2_Sec	Uncontrollable
AZ_CW_OK	Uncontrollable
AZ_CCW_OK	Uncontrollable
EL_CW_OK	Uncontrollable
EL_CCW_OK	Uncontrollable

Azimuth Position

The azimuth motor operates over a range of $180°$ with the initial position of the motor at $90°$ (center). The current position of the azimuth motor shaft is tracked and stored within the microcontroller's memory. When at the center position, the motor can move in the CW direction (respectively CCW direction) which will increment (respectively decrement) the position by $2°$ after each motor motion is completed. There is a software implemented in microcontroller that tracks the azimuth position range. The DES model shown in Fig. 4.10 models the operation of this software. Table 4.12 describes the states of the automaton. The position of the azimuth motor can be polled by the controllable event AZ_POLL_RANGE, which returns the current position (θ_{AZ}) of the motor. The uncontrollable events AZ_RANGE_OK or AZ_MAX_CW or AZ_MAX_CCW are triggered

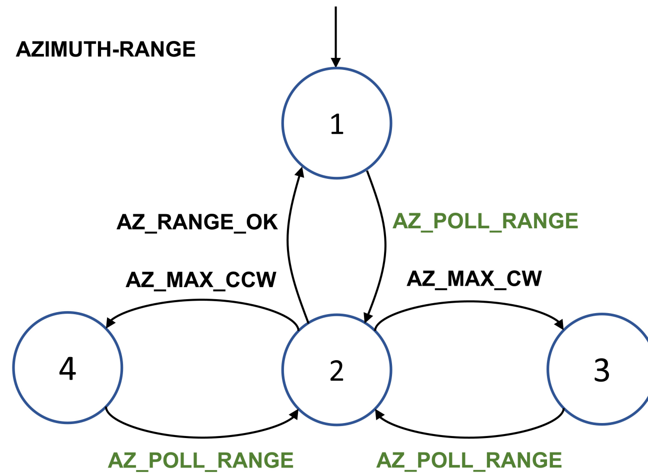


Figure 4.10: Azimuth position range automaton [5].

Table 4.12: States of the azimuth angle automaton.

State Number	State
1	Azimuth in Range
2	Azimuth Polling Range
3	Azimuth Maximum CW
4	Azimuth Maximum CCW

Table 4.13: Event list of azimuth angle automaton in degrees [5].

Event	Angle Range ($\circ$)	Controllability
AZ_POLL_RANGE	N/A	Controllable
AZ_RANGE_OK	$0° < \theta_{AZ} < 180°$	Uncontrollable
AZ_MAX_CW	$\theta_{AZ} = 0°$	Uncontrollable
AZ_MAX_CCW	$\theta_{AZ} = 180°$	Uncontrollable

depending on the value of θ_{AZ}. The four events along with their controllability and range are provided in Table. 4.13.

Elevation Position

The elevation motor is limited to operate over a range of $90°$, with the initial position at $45°$ (maximum CCW). The elevation position automaton is similar to the azimuth position automaton and illustrated in Fig. 4.11. The states for this automaton are given in Table 4.14 and the event list is provided in Table 4.15. Similar to the azimuth position automaton, the elevation position automaton models a software code that tracks the elevation position.

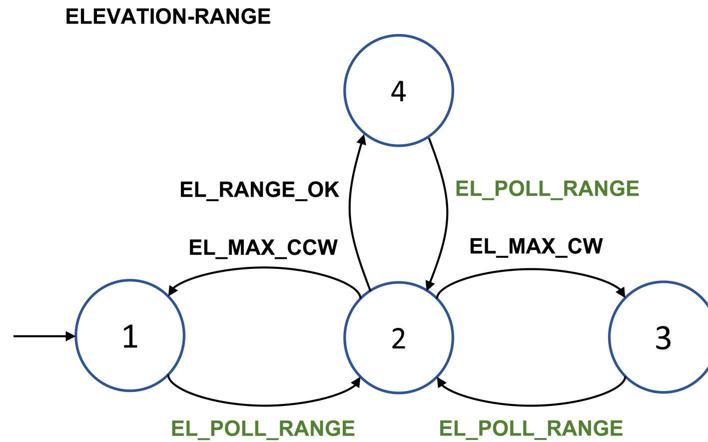


Figure 4.11: Elevation position range automaton [5].

Table 4.14: States of the elevation angle automaton.

State Number	State
1	Elevation Maximum CCW
2	Elevation Polling Range
3	Elevation Maximum CW
4	Elevation in Range

Table 4.15: Event list of elevation angle automaton in degrees [5].

Event	Angle Range (○)	Controllability
EL_POLL_RANGE	N/A	Controllable
EL_RANGE_OK	$-45^{\circ} < \theta_{EL} < 45^{\circ}$	Uncontrollable
EL_MAX_CW	$\theta_{EL} = -45^{\circ}$	Uncontrollable
EL_MAX_CCW	$\theta_{EL} = 45^{\circ}$	Uncontrollable

Master Controller

The master controller models the human operator at the ground station that can issue commands and also receive the feedback from the system. We assume that the human operator at the ground station issues Full_Sweep commands upon which the supervisor is supposed to send appropriate command sequences. Depending upon the result of the Full Sweep maneuver, the supervisor can report back by issuing EL_MOTOR_FAIL or Bright_Detected or Sweep_Failure event. Therefore, from the supervisor's point of view, the Full_Sweep command is an uncontrollable event and the rest of commands are controllable. The master controller automaton shown in Fig. 4.12 is used to include the operator as part of the model of the plant from the supervisor's perspective. The list of

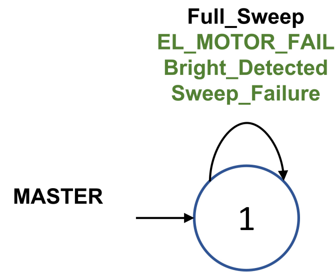


Figure 4.12: Master controller automaton [5].

Table 4.16: Event list of master controller automaton [5].

Event	Controllability
Full_Sweep	Uncontrollable
EL_MOTOR_FAIL	Controllable
Bright_Detected	Controllable
Sweep_Failure	Controllable

these events, along with their controllability status are given in Table 4.16.

4.2.2 Component Interactions

Modelling the interactions between the components of the plant is extremely important since the interaction automatons capture the physical attributes as well as the interactions of the physical components of the plant. For instance, in the solar tracker system, the battery SOC will not increase when the azimuth or elevation motor are in motion. As a result, this limitation of the system should be represented in the interaction model in order to obtain an accurate model of the plant. It is imperative to model these interaction automatons accurately, as removal of any possible state transition will cause an inaccurate plant automaton and consequently, the plant automaton will not match the system. Usually, the interactions describe restrictions on the events of one automaton as a function of the states of other automatons. The DES model is constructed by adding appropriate self-loops of the restricted events to the states of other automatons. The interaction automatons for the above components of the solar tracker system are described in this section.

Motor Motion as a Function of Battery SOC

In the solar tracker system for the successful movement of the servomotors, it is necessary that the battery have a sufficient charge, i.e. the battery SOC should be at least above 50%. The events related to the successful motor movement are permitted in only Full and Safe states of the BATT-SOC automaton as shown in Fig. 4.13. This interaction blocks the CW and CCW successful movement events of the azimuth and the elevation motor in the Critical state when the battery SOC is below 50%.

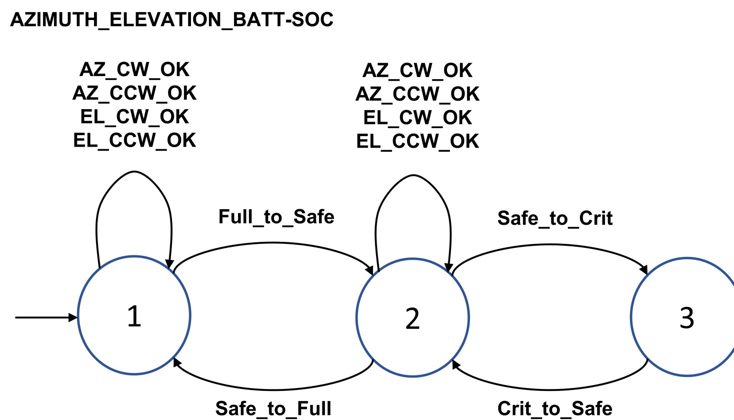


Figure 4.13: Azimuth and elevation motion as a function of battery SOC.

Battery SOC as a Function of Motor Motion

In the solar tracker system, whenever the motors are in motion, they draw a significant amount of current from the battery. According to the specifications of the PV panel, the maximum current generated by the PV panel is less than the no load current operating currents of the azimuth and the elevation motor. Thus, it is obvious that whenever the motors are moving, only the events related to the decrease in the battery SOC are possible. When both motors are in the idle state or when the azimuth motor is idle and the elevation motor is failed, all battery SOC events can occur; hence the battery can either charge or discharge. To represent the battery SOC events as a function of motor motion automaton, the synchronous product of the AZIMUTH automaton (Fig. 4.7) and the ELEVATION automaton (Fig. 4.8) is formed with the appropriate battery related events added as self-loops to the resulting automaton (Fig. 4.14).

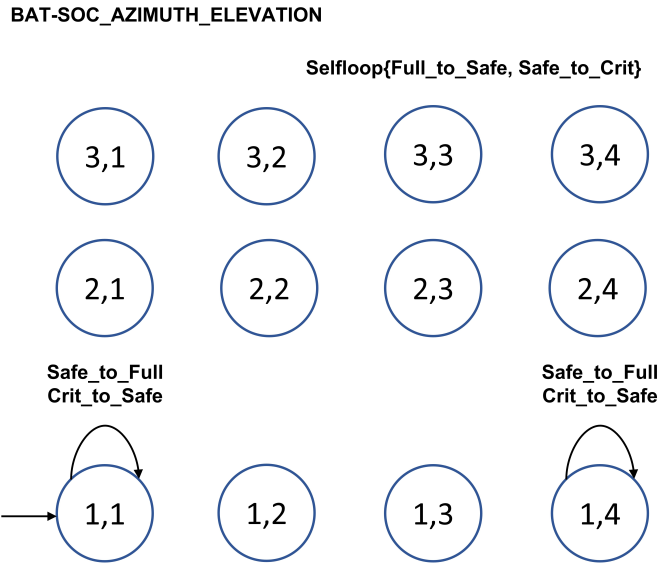


Figure 4.14: Battery SOC events as a function of azimuth and elevation motion [6].

Battery SOC as a Function of PV Panel Illumination

The charging of the battery depends upon the current generated by the PV panel. According to the specifications of the PV panel, the PV panel can generate enough current to charge the battery when it is in Dim or Bright state; thus the events related to the charging of the battery are permitted in these states and in the Dark state. Moreover, motor movements of both motors are possible when the PV panel illumination level is in any of the three states. Motor movement can result in the discharging of the battery. Therefore all the events related to discharging of the battery are permitted in the three states of the PV panel automaton. The automaton modeling battery SOC events as a function of PV Panel Illumination is shown in Fig. 4.15.

The complete model of the solar tracker system ($Plant$) is formed by performing the synchronous product of the component and the interaction automatons.

$$
\begin{aligned}
Plant = sync(&\text{BATT-SOC, PV-ILLUMINATION, AZIMUTH, ELEVATION, DELAY,}\\
&\text{AZIMUTH-RANGE, ELEVATION-RANGE, MASTER,}\\
&\text{AZIMUTH_ELEVATION_BATT-SOC,}\\
&\text{BATT-SOC_AZIMTH_ELEVATION, BATT-SOC_PV-ILLUMINATION}).
\end{aligned}
$$

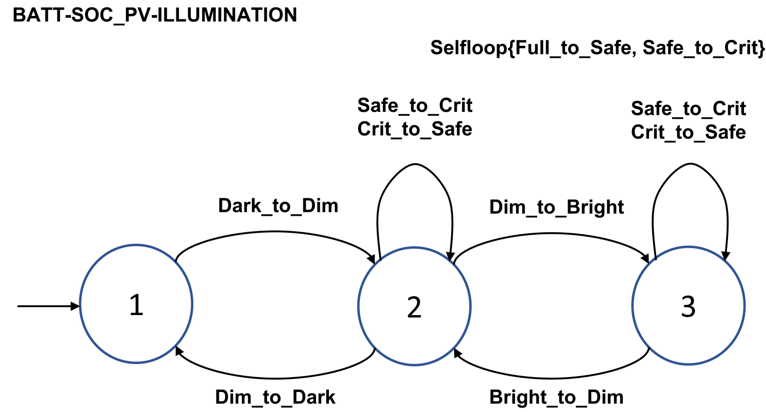


Figure 4.15: Battery as a function of PV panel illumination automaton [5].

$Plant$ contains 1584 states and 16800 transitions. In this thesis, only the safety property of the solar tracker system is taken into the consideration. The non-blocking property of the solar tracker system is not taken into consideration. Hence, all the states of $Plant$ are considered marked.

4.2.3 Specifications

The specifications provide a formal description of the design requirements. In this thesis, only the safety properties of the solar tracker system are taken into the consideration.

Servomotor Motion Range Specification

The servomotor motion range specification is defined to prevent any damage occurring to the servomotors when they are in their maximum positions. The specification for azimuth motor is given in Fig. 4.16. The specification for the elevation motor is similar to the specification for the azimuth motor and is not shown for brevity. In this specification, whenever the azimuth motor is in maximum CW (resp. CCW) state, further movement in the CW (resp. CCW) direction is prohibited. For instance, when the azimuth motor is in the Azimuth Maximum CW state, then only the command to rotate the azimuth motor in the CCW direction should be permitted which is depicted by the AZ_CCW_Move event added as a self-loop in the Azimuth Maximum CW state in Fig. 4.16.

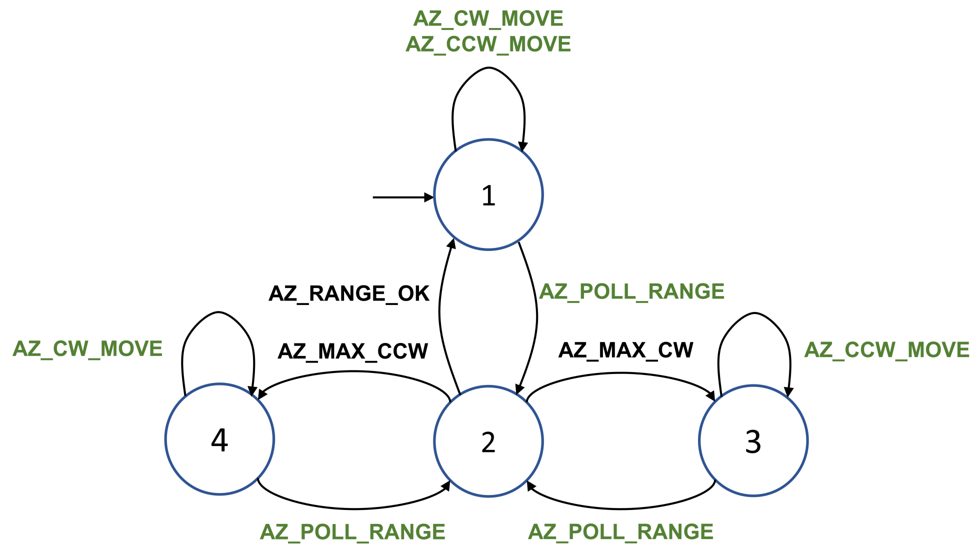


Figure 4.16: Azimuth motor motion range specification [5].

Servomotor Polling Range Specification

The servomotor polling range specification provides restriction on the polling of motor angles. The polling of the servomotors' current position should be done after the events for successful motor motion such as XX_CW_OK or XX_CCW_OK (XX = AZ or EL) transpires which causes the state transition in the servo motor automatons to the Idle state (state 1 in Fig. 4.7 and Fig. 4.8). The azimuth motor polling range specification is shown in Fig. 4.17. Polling should not be done while a move is attempted. A similar specification is used for the elevation motor and is shown in Fig. 4.18.

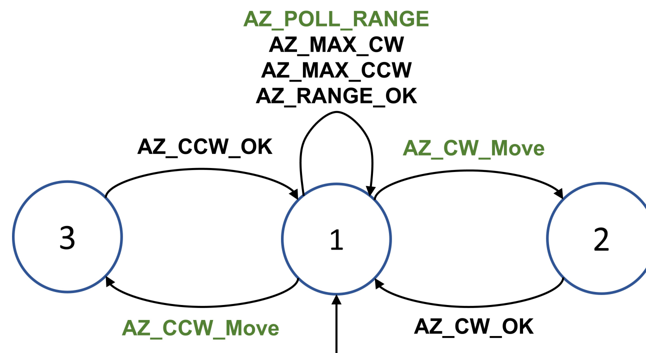


Figure 4.17: Azimuth motor polling range specification [5].

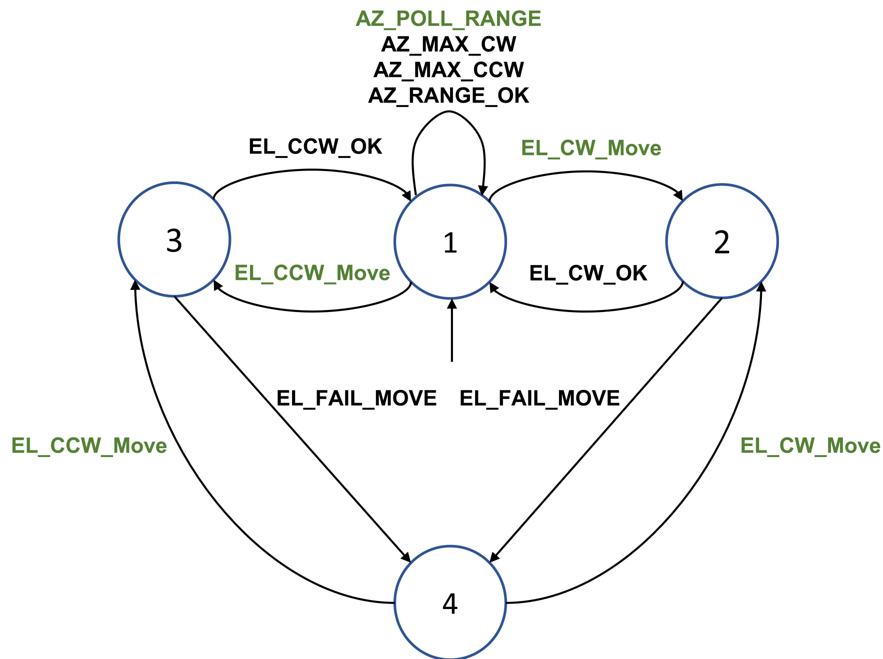


Figure 4.18: Elevation motor polling range specification.

Servomotor Full Sweep Specification

The previously defined specifications ensure the safety of the solar tracker system. In addition to these specifications, it is required to have a specification that describes the sequence of steps (maneuvers) the solar tracker system needs to perform to detect a bright light source. The main design requirement of the solar tracker system is to sweep its front hemisphere to search for a bright light source with sufficient intensity. This specification is referred to as “Full Sweep" and is explained in details in [6].

The operator issues the Full_Sweep command to initiate Full sweep maneuver. Initially, the azimuth motor and the elevation motor are at 90° and 45° respectively. When the solar tracker system receives the Full_Sweep command, the following maneuvers are performed:

- The azimuth motor rotates in the CCW direction from its current position until it reaches the maximum CCW position and then the elevation motor rotates in CCW direction from its current position until it reaches the maximum CCW position.
- The azimuth motor rotates in the CW direction from the maximum CCW position until it reaches the maximum CW position.

- The elevation motor rotates from the maximum CCW position in the CW direction until it reaches the maximum CW position.
- The azimuth motor rotates in the CCW direction from the maximum CW position until it reaches he maximum CCW position.

During the above sequence of maneuvers, if a bright light source is detected, then a Bright_Detected event is sent to the MC, otherwise, if no bright light source is detected at the end of the maneuver, then Sweep_Failure event is generated. Moreover, during the maneuver if there is a failure in the elevation motor, then EL_MOTOR_FAIL event is sent to the ground station, and thus aborting future maneuvers. To prevent the battery from providing insufficient voltage during the motor movements, only one motor is allowed to move at a time. This specification automaton has 58 states and 209 transitions and can be found in [6].

Finally, the complete specification automaton for the solar tracker plant is obtained by adding self-loops of the irrelevant events to each of the specification automatons described above and taking the product of the resulting self-looped automatons. The complete specification (K) automaton contains of 416 states and 4216 transitions.

4.2.4 Supervisor

The Discrete Event Control Kit (DECK) [59] developed in MATLAB [60] is used to design the supervisor using the supervisory control theory (SCT). The solar tracker plant models are represented in DECK as automaton objects ($G = automaton(N, TL, X_m)$) with the following properties

- N: Total number of States,
- TL: State Transition List,
- X_m : List of Marked States.

The procedure for obtaining the supervisor is given below

Table 4.17: Plant, specification and supervisor automatons [3].

Automaton	Number of States	Number of Transitions
Solar Tracker Sytem ($Plant$)	1584	16800
Specification (K)	416	4216
Supervisor (S)	2061	9467
Solar Tracker System Under Supervision	2061	9467

(1) The complete plant for the solar tracker system is formed by using the **sync** function ($[Plant, States] = sync(G_1, \ldots, G_n)$) which obtains the synchronous product of the component automatons and the interactions automatons of the solar tracker system.

(2) The complete specification for the solar tracker plant is formed by using the **product** function ($[K, States] = product(K_1, \ldots, K_n)$) that performs the product of all the specification automatons designed for the solar tracker system.

(3) Finally the supervisor is obtained using the **supcon** function ($S = supcon(K, Plant, \Sigma_{uc})$) where G is the plant model, K is the complete specification model and Σ_{uc} is the list of uncontrollable events.

The supcon function generates the supremal controllable sub-language of the intersection of the marked language of the plant and the marked language of the specification ($L_m(Plant) \cap L_m(K)$) with respect to the closed language of the plant ($L(Plant)$) and the uncontrollable events Σ_{uc} [61]. The supervisor automaton (S) is obtained from the automaton $Plant$ and the specification automaton K. The supervisor S is the *offline conventional supervisor* for the solar tracker system. S contains of 2061 states and 9527 transitions. Table 4.17 summarizes information about the above-mentioned automatons. All of the states of $Plant$, K and S are marked since non-blocking property is not considered in this thesis.

4.3 Summary

In this chapter we reviewed the hardware of the solar tracker setup used in our case study. We also discussed modeling the system and designing a conventional supervisor for the system. In the next chapter, we will apply the algorithm proposed in Chapter 3 to design a supervisor based on LLP with Buffering.

Chapter 5

Case Study: A Solar Tracker System

In this chapter, we present the results of running the algorithms mentioned in Chapter 3 on the timed model of the solar tracker system. The theoretical $T'_{min,low}(\delta)$ value calculated for solar tracker system results in a tighter lower bound value which is not suitable for LLP with Buffering. Therefore, we provide some necessary adjustments for calculating an appropriate value of $T'_{min,low}(\delta)$ such that this value can be used for determining the buffering parameters of LLP with Buffering. Then we explain a practical method for obtaining $T_{min,exp}(\delta)$ value experimentally for solar tracker system. Comparisons between theoretical $T'_{min,low}(\delta)$ and experimental $T_{min,exp}(\delta)$ shows that both the values are reasonably close. Finally, on the basis of the adjustments made to $T'_{min,low}(\delta)$; the values for buffering parameters of LLP with Buffering are selected.

The rest of this chapter is organised as follows. Section 5.1 reviews the details of the buffering parameters of LLP with Buffering. The procedure for calculating the experimental $T_{min,exp}(\delta)$ is explained in Section 5.2. In Section 5.3, we present the timing information for all the events that were explained in the Section 4.2 of Chapter 4 of the solar tracker system. Section 5.4 presents the theoretical $T'_{min,low}(\delta)$ value calculated using the timed model of the solar tracker system. Section 5.5 presents the analysis of computation time for the LLP with Buffering implemented for the solar tracker system. In Section 5.6 the values for the buffering parameters are selected for the solar tracker system.

5.1 LLP with Buffering parameters

In Chapter 4 we explained the modelling and the design specifications for the solar tracker system. In this chapter we are going to explain the design of LLP with Buffering and discuss about how the buffering parameters have to be chosen. In this section we review the buffering parameters used in LLP with Buffering.

The three main parameters for LLP with Buffering are N_{min}, δ and Δ, where N_{min} is the minimum depth of the plant that should be expanded for the validity of the LLP supervisor, Δ is the number of events to be buffered and δ is the number of events left in the buffer, provided that $\delta < \Delta$. Therefore, the LLP supervisory commands are computed for the look-ahead window N_w which is given as

$$N_w = N_{min} + \delta + \Delta.$$

N_{min} is calculated based on the expansion of the product of $Plant$ automaton (solar tracker system) and K automaton (specifications for solar tracker system). From [3], N_{min} for the solar tracker plant is 6. Parameters Ω can be defined as $\Omega = \Delta + \delta$ and thus $N_w = N_{min} + \Omega$. The PD size for the $Plant$ automaton of the solar tracker system is 12. When $N_w = 25$ ($N_{min} = 6$ and $\Omega = 19$) the product of the expanded $Plant$ and the expanded K specification automaton is equivalent to the product of the complete $Plant$ automaton and K automaton. If for a look-ahead window size, the expanded model is the same as the complete model, then the window size is referred to as the *saturation point* (SP) for that model. The SP for the product of the plant and specification is given as

$$SP = N_{min} + 19 = 25.$$

Hence, using a window size N_w of larger than 25 will neither change the supervisor size nor the computation time. If the value $\delta = 0$ and $\Delta = 0$ ($\Omega = 0$) are used, then the look-ahead window size is $N_w = N_{min}$ and regular LLP (no buffering) will be performed since there are no events to be buffered.

5.2 Determination of $T_{min}(\delta)$ Using Experiments

In this section we will explain numerous experimental tests that are performed on the solar tracker system. Moreover we will explain the procedure to determine $T_{min}(\delta)$ experimentally based on the event timing information collected through numerous experimental tests performed on the solar tracker system.

The micro-controller is used for interfacing the ground station (computer) and the solar tracker system (more details can be found elsewhere in [3]). The sampling time for the system is set to 50 ms to measure the feedback from the sensors and detect the occurrences of events. All uncontrollable and controllable events which are detected in the solar tracker system are transmitted to the computer; however, to reduce the effect of communication delay, only the controllable events are transmitted back from the solar tracker system to the computer. In [3], the communication delay is 20 ms. In [5] multi-threaded programming approach is implemented that has one separate thread created for the LLP supervisory computation and another separate thread created for serial communication. The main reason to use multi-threading is that the occurrence of the events in the solar tracker system are sporadic and during the computation of the LLP supervisor if any data is received by the computer then the control loop will miss this data. The events that are transmitted or received are in the form of data packets which on reception in the computer is stored in a First In First Out (FIFO) buffer. Universal Asynchronous Receiver/Transmitter (UART) is used to receive and transmit the data packets. More details about the implementation of this communication link can be found in [5].

5.2.1 Tests performed

In order to determine $T_{min,exp}(\delta)$ experimentally; we perform some experimental tests on the solar tracker system. However, it is infeasible to perform exhaustive experimental test since there are many different possibilities of tests that can be performed. Therefore, we select certain specific sets of tests which are reasonably large and cover the most important cases. First, we provide an overview of the tests that are performed and later we discuss different values that are selected for the buffering parameters for each tests in detail. In all the tests explained below, Full Sweep maneuver

of the solar tracker system is performed and the PV panel is at its initial position (see Section 4.1 in Chapter 4).

- Full Sweep maneuver tests of the solar tracker system are performed with some of the tests resulting in successful maneuver i.e. light source detected and the rest resulting in unsuccessful maneuver i.e. light source not detected (where the light source was removed).
- Different positions of the light source with respect to the PV panel are considered for the tests that resulted in successful Full Sweep maneuver.
- Different initial values of buffering parameters are chosen for both successful and unsuccessful Full Sweep maneuver.
- Initially the battery voltage is maintained at 4.2 V (fully charged) at the start of each Full Sweep maneuver (resulting in successful and unsuccessful runs with different values of buffering parameters). Same tests are performed by maintaining the battery voltage between 4.0 V to 4.2 V at the start of each Full Sweep maneuver.

Now, we describe in details the values selected for LLP buffering parameters during each unsuccessful Full Sweep maneuver due to the absence of light source. Battery voltage is maintained at 4.2 V (fully charged) at the start of the each test runs described below. There are three different sets of tests that are performed using LLP with Buffering by setting the buffering parameters as follows.

(1) In the first set of tests δ is incremented from 2 till 8, and for each δ, Δ is set to $\Delta = \delta + 1$ in order to compute LLP supervisor.

(2) In the second set of tests δ increments from 2 to 9, and for each δ the value of $\Delta = 10$ is fixed in order to compute LLP supervisor.

(3) In the third set of tests δ increments from 2 to 8 and for each δ the value of $\Delta = SP - N_{min} - \delta$ ($SP = 25$ and $N_{min} = 6$) in order to compute LLP supervisor.

Next, we describe in details different positions selected for the light source with respect to the PV panel. In the tests described below the LLP supervisor is computed each time for different buffering

parameters values: δ takes the values 2, 5 and 9, and for each δ, Δ is set at 10. The set of tests performed for each combination of δ and Δ are as follows.

(1) In the first set of tests; the elevation motor range is at $\theta_{EL} = 45°$ and the position light source is fixed when the azimuth motor range reaches $\theta_{AZ} = 0°$.

(2) In the second set of tests; the elevation motor range is at $\theta_{EL} = 45°$ and the position of the light source is fixed when azimuth motor range reaches $\theta_{AZ} = 90°$.

(3) In the third set of tests; the elevation motor range is at $\theta_{EL} = 45°$ and the position of the light source is fixed when azimuth motor range reaches $\theta_{AZ} = 180°$.

(4) In the fourth set of tests; the elevation motor range is at $\theta_{EL} = -45°$ and the position of the light source is fixed when the azimuth motor range reaches $\theta_{AZ} = 90°$.

(5) In the fifth set of tests; the elevation motor range is at $\theta_{EL} = 0°$ and the position of the light source is fixed when the azimuth motor range reaches $\theta_{AZ} = 0°$.

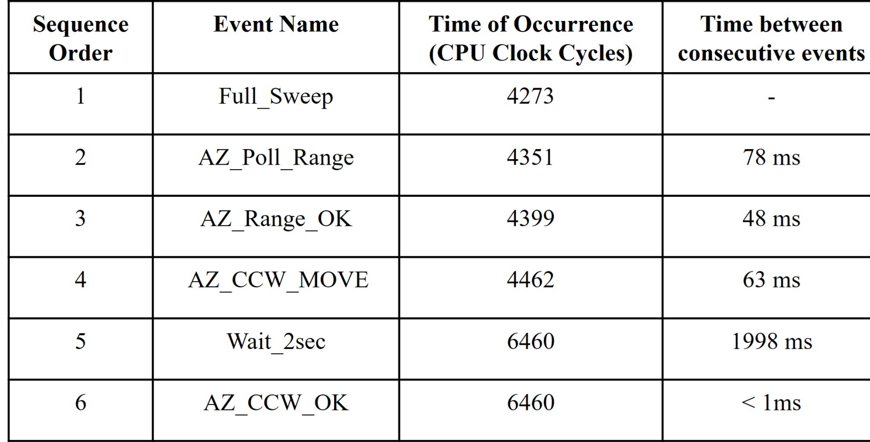
All the above tests runs are repeated again except by maintaining the battery voltage initially between 4.0V to 4.2V. There are another set of tests which corresponds to the failure of elevation motor, however, for brevity, these tests are not performed.

5.2.2 Procedure for Calculating $T_{min,exp}(\delta)$

In order to calculate $T_{min,exp}(\delta)$, a function is developed in C language which time stamps the events received by the computer. The time stamping is performed using CPU clock cycles.

The solar tracker system initiates its Full Sweep maneuver after receiving the Full_Sweep command from the computer. The events occurring in the solar tracker system are transmitted to the computer which on the reception are time stamped using the CPU clock cycles. Thus, every event is associated with its occurrence time. In this way, for the entire Full sweep maneuver, all the events along with their occurrence time are obtained. To demonstrate the procedure of calculating $T_{min,exp}(\delta)$, consider buffering parameters $\delta = 4$, $\Delta = 10$ and $N_{min} = 6$ resulting in the look-ahead window $N_w = 4+10+6 = 20$. Table 5.1 shows first six events that follow after microcontroller of the solar tracker system has received the Full_Sweep command. Only the first six

Table 5.1: Example for procedure of calculating $T_{min,exp}(4)$.



Sequence Order	Event Name	Time of Occurrence (CPU Clock Cycles)	Time between consecutive events
1	Full_Sweep	4273	-
2	AZ_Poll_Range	4351	78 ms
3	AZ_Range_OK	4399	48 ms
4	AZ_CCW_MOVE	4462	63 ms
5	Wait_2sec	6460	1998 ms
6	AZ_CCW_OK	6460	< 1ms

Table 5.2: Experimental $T_{min,exp}(\delta)$ for $\delta = 1, \ldots, 9$.

δ	$T_{min,exp}(\delta)$ (msec)
1	0
2	0
3	0
4	31
5	46
6	93
7	249
8	343
9	406

events are considered here to explain how $T_{min,exp}(\delta)$ is calculated. Since $\delta = 4$, we will calculate the $T_{min,exp}(4)$ for this sequence. In Table 5.1 the total duration on the occurrence of four events is calculated by sliding a window of length four from the first event to the fourth event, then from the second event to the fifth event and so on which is shown by the arrows on the right side of the table. The total duration from the first event Full_Sweep till the fourth event AZ_CCW_Move is calculated by adding the time duration of all intermediate events between these two events which results in the duration of 189 ms (78 ms + 48 ms + 63 ms). Then from the second event AZ_Poll_Range to the fifth event, Wait_2sec, the total duration is 2109 ms and so on. This procedure for calculating the duration of four consecutive events is applied to the entire event sequence that are received in the computer by sliding the window of length $\delta = 4$. Finally, the minimum duration of four events is determined.

For each test runs described in Section 5.2.1, the timed sequences are recorded and the procedure to determine experimental $T_{min,exp}(\delta)$ (previously explained) is followed. Table 5.2 provides the experimental $T_{min,exp}(\delta)$ for $\delta = 1 \ldots 9$. Following Def. 3.6, it can be observed in Table 5.2, $T_{min,exp}(1) = 0$. From Table 5.2, it can be observed that for $T_{min,exp}(2) = 0$ and $T_{min,exp}(3) = 0$. This is because information about one or more than one events that occur in the system rapidly can be transmitted from the solar tracker system to the PC in one information packet.

5.3 Timing Information of the Events

The untimed automaton of the system does not include any timing information of the events. In Chapter 3 a method is proposed to transform an untimed model of the system to a TA assuming the timing information of the events is available. Each event is associated with a lower time bound and a upper time bound. These bounds would usually represent a delay caused due to communication or computation or physical necessity. It is crucial to accurately determine the timing information so that the behaviour of the timed model of the system can match the physical system.

The solar tracker system consists of total 30 events. Out of these 30 events, nine events are controllable and the rest are uncontrollable. The duty of the micro-controller of the solar tracker system is to scan all the sensors periodically with a sampling time of 50 ms. At every sampling time, all the events that are detected by the micro-controller are packed in an information packet and sent to the PC in order to update the state of the component models and the LLP supervisor. In [3], the communication delay from transmitting the events between the micro-controller and the PC was found to be 20 ms. In the LLP implementation, the uncontrollable events, which are detected by the micro-controller, are sent to the PC to be validated as uncontrollable event. Therefore, the time bounds for the uncontrollable events are increased by the communication delay of 20 ms. The controllable events are sent by the LLP supervisor in the PC to the micro-controller and then the implementation of these events are reported back to the PC in order to be validated as controllable events. Consequently, the time bounds for the controllable events are increased by twice the communication delay (40 ms). *Note that the timing information for all the events is in seconds.*

5.3.1 Battery SOC Related Events

The on-board battery used for the solar tracker system is charged via the power produced by the PV panel connected to the MPPT module. The charging of the battery through the MPPT module and the consumption of the battery by the motors are used to determine the lower and the upper time bounds of the BATT-SOC events (Fig. 4.5 and Table 4.3). The maximum current that the PV panel can generate under full sun is 200 mA [62] which can be used for charging the battery. Under no load the current consumption of the azimuth motor is 350 mA according to the datasheet in [63] and the current consumption of the elevation motor is 700 mA according to datasheet in [64]. The current generated by the PV panel depends upon the intensity and the incident angle of the light source. We assume that the intensity of the light source is constant. In addition, the time required for the Full Sweep maneuver in order to find the light source compared with the actual time to charge the battery is small. Therefore, we assume that maximum current is generated by the PV panel for charging the battery which is used to determine the time bounds for the Safe_to_Full and Crit_to_Safe events. The no-load operating current of the motors for discharging the battery is used for determining the time bounds for the Full_to_Safe and Safe_to_Crit events.[1] In addition, we assume that the charging and discharge curves of the LiPo battery are linear. The time bounds for the BATT-SOC events are calculated by using the following formula:

$$\text{charging/discharging time} = \frac{\text{battery capacity}}{\text{charging/discharging current}}. \tag{31}$$

The charging time for the battery is

$$\begin{aligned} \text{charging time} &= \frac{2500}{200} \\ &= 12.5 \text{ hours} \\ &= 45000 \text{ sec.} \end{aligned} \tag{32}$$

There are two charging events in BATT-SOC: Safe_to_Full and Crit_to_Safe based on Table 4.3.

[1] Accurate time bounds for the battery SOC events related to the discharging of the battery can be calculated by using the value of current drawn by the motors when the load is attached.

Table 5.3: Battery SOC events time bounds.

Event	Lower Time Bound (sec)	Upper Time Bound (sec)
Full_to_Safe	642.62	2570.42
Safe_to_Full	2250.02	18000.02
Safe_to_Crit	642.62	10281.62
Crit_to_Safe	2250.02	24750.02

Minimum time for the occurrence of Safe_to_Full event at Safe state is when BATT-SOC enters Safe from state Full with a SOC 90% and then SOC increases to 95% (resulting in Safe_to_Full), i.e. 5% increase in the battery SOC from 90% to 95%. Thus, taking 5% of the charging time in equation (32) (since the charging curve of Lipo battery is assumed to be linear) results in the lower time bound for the Safe_to_Full event as 2250 sec. Suppose, the BATT-SOC automaton enters the Safe state with the SOC as 55% as a result of the Crit_to_Safe event. Then the maximum time required for the Safe_to_Full event to occur is when the battery charges from 55% until 95%. Therefore, by taking 40% of the charging time in equation (32) results in the upper time bound to be 18000 sec. Similarly, the time bounds for Crit_to_Safe event are calculated.

The discharging time for the battery when the azimuth motor is operating is calculated as

$$\begin{aligned} \text{discharging time} &= \frac{2500}{350} \\ &= 7.14 \text{ hours} \\ &= 25704 \text{ sec.} \end{aligned} \tag{33}$$

The discharging time for the battery when the elevation motor is operating is calculated as

$$\begin{aligned} \text{discharging time} &= \frac{2500}{700} \\ &= 3.57 \text{ hours} \\ &= 12852 \text{ sec.} \end{aligned} \tag{34}$$

Next, we shall calculate the time bounds for the discharging events Full_to_Safe and Safe_to_Crit. Suppose, automaton BATT-SOC enters the Full state with a SOC of 95% following the occurrence

of Safe_to_Full event. The minimum time for Full_to_Safe event to occur is when the battery discharges from the 95% to 90% when only the *elevation* motor is operating (due to the higher current consumption by the elevation motor) and azimuth motor is idle. Therefore, the lower time bound is 5% of the discharge time from equation (34) resulting in 642.6 sec. Initially, the battery is assumed to be fully charged with SOC 100%, then the maximum time required for the Full_to_Safe event to occur is when the battery discharges till 90% when only the azimuth motor is moving while the elevation motor is idle. This results in 10% of the discharge time from the equation (33). Therefore, the upper time bound for the Full_to_Safe event is 2570.4 sec. Similarly, the time bounds are calculated for the Safe_to_Crit event. It is to be noted that during the modeling of the time bounds for the events related to the discharging and charging of the battery, the power consumed by either one or both motors are idle and the power consumed by the other components are much smaller as compared to the discharging or charging power and hence are disregarded. The time bounds for the BATT-SOC automaton events are given in Table 5.3. In addition, a communication delay of 20 ms is added to the time bounds of all the events since they are uncontrollable.

5.3.2 PV Panel Related Events

The time bounds for the PV-ILLUMINATION events (Fig. 4.6 and Table 4.5) depend upon the environmental conditions. Factors such as the orientation of the PV panel with respect to the light source, relative motion of PV panel, and the speed of the motor play important roles in deciding the time bounds of the events. The time bounds also depend on the intensity of the light source; if the light source has a high intensity, then PV panel can generate high voltages even when the panel is not facing directly the light source. Ideally, the time bounds for the events of PV-ILLUMINATION must be modeled experimentally during the designing and assembling phase of the system.

The solar tracker system operates within laboratory environment and the ambient lighting of the laboratory is considered in the estimating time bounds. Recall that in Section 4.2.1 of Chapter 4, in the operation of the solar tracker system we assumed that solar tracker system always starts in the Dark state. The lower time bounds for the PV panel events depends upon how fast the threshold voltage is crossed and the speed at which the motor moves. Therefore we resort to experimental test for determining the time bounds of PV panel events. The first event that can occur in the Dark state

Table 5.4: PV Panel events time bounds.

Event	Lower Time Bound (sec)	Upper Time Bound (sec)
Dark_to_Dim	2.02	∞
Dim_to_Dark	2.02	∞
Dim_to_Bright	0.72	∞
Bright_to_Dim	0.72	∞

is the Dark_to_Dim event. Experimentally it is observed that only one step rotation of the motor is enough for the PV panel to generate the voltage above the threshold of 6V in order to trigger the Dark_to_Dim event. Therefore, the lower time bound for the Dark_to_Dim event is 2.02 seconds resulting from a 2° motion of a motor: 2 seconds delay plus an additional 20 ms for communication delay. If the system just enters the Dim state, then the PV panel voltage can drop below the threshold of 5V when the motor moves in the opposite direction with just one step rotation. As a result, the lower time bound for the Dim_to_Dark event is also 2.02 seconds.

The time bounds for Dim_to_Bright and Bright_to_Dim events is obtained experimentally. The minimum time for the Dim_to_Bright event to occur is when the system is in the Dim state and the slightest movement of the PV panel towards the light source causes the system to enter the Bright state. In order to calculate the lower time bound for the Dim_to_Bright event; the PV panel is at its initial position and the light source is fixed at a position directly facing the PV panel. Next, full sweep command for solar tracker system is issued and the event sequence along with their timing details are recorded in the PC. Then, after the occurrence of Dark_to_Dim event in the recorded event sequence, the minimum time required for the Dim_to_Bright event to occur is measured. Therefore, the lower time bound for Dim_to_Bright is 700 ms plus an additional 20 ms for the communication delay. If the PV panel moves in the opposite direction then the Bright_to_Dim event can occur with the same lower time bound of 720 ms. The Table 5.4 provides the time bounds for all the PV-ILLUMINATION automaton events. It is to be noted that the upper time bound for all the events in Table 5.4 is infinity. This is to account for the situation when an event may not occur or occur after an arbitrarily long time. For instance, in the tests where there is an absence of the light source and the solar tracker system is in a dark environment; then neither Dark_to_Dim and nor Dim_to_Bright event will occur in full sweep manoeuvre of the system. However, these event will not be prevented indefinitely when the ambient lighting conditions around the solar tracker system

Table 5.5: Controllable events time bounds.

Event	Lower Time Bound (sec)	Upper Time Bound (sec)
AZ_CW_MOVE	0.09	∞
AZ_CCW_MOVE	0.09	∞
AZ_POLL_RANGE	0.09	∞
EL_CW_MOVE	0.09	∞
EL_CCW_MOVE	0.09	∞
EL_POLL_RANGE	0.09	∞
Bright_Detected	0.09	∞
Sweep_Failure	0.09	∞
EL_MOTOR_FAIL	0.09	∞

changes.

5.3.3 Command Events

Table 5.5 provides the time bounds for all command events generated by the supervisor. These events are controllable. As mentioned previously, the sampling time of the system is 50 ms and the communication delay is 20 ms. Therefore, the lower time bound for all the controllable event is 90 ms (50 ms + 2 $\times$ 20 ms), since anything happening within 50 ms cannot be detected in the system. The upper time of the controllable event is infinity because no assumption is made on how quickly the supervisor may issue a specific command.

5.3.4 Sensor Events

The events mentioned in Table 5.6 do not depend upon the environment. The lower time bound for the successful motor movements and maximum motor range events is just one sampling time plus the communication delay i.e. 70 ms (50 ms + 20 ms). According to data sheets [63] and [64], the 2° rotation for the azimuth and elevation motors requires 6 ms and 4.6 ms respectively. However, the event triggering logic for he successful motor movements as mentioned in [6] does not depend upon the time required to complete 2° rotation but on the 2 sec wait time. Therefore, we do not include the time required by the motors to complete 2° rotation. In addition, the time required to complete 2° rotation is smaller than the sampling time of the system. In [3], during every sampling time of the system, the occurrence of each event is checked twice. Once new events are detected by

Table 5.6: Sensor events time bounds.

Event	Lower Time Bound (sec)	Upper Time Bound (sec)
AZ_CW_OK	0.07	0.07
AZ_CCW_OK	0.07	0.07
AZ_MAX_CW	0.07	0.07
AZ_MAX_CCW	0.07	0.07
AZ_RANGE_OK	0.07	0.07
EL_CW_OK	0.07	0.07
EL_CCW_OK	0.07	0.07
EL_MAX_CW	0.07	0.07
EL_MAX_CCW	0.07	0.07
EL_RANGE_OK	0.07	0.07
EL_FAIL_MOVE	0.07	0.07

the micro-controller, they are immediately reported to PC. Therefore, the upper time bound for the successful motor movements and maximum motor range events is also 70 ms.

5.3.5 Other Remaining Events

There are two events that are not part of the events that are described above. Full_Sweep event is related to the operator and Wait_2_Sec event is related to the hardware of the system.

The Full_Sweep event initiates the Full Sweep maneuver of the solar tracker system. The time bounds of the Full_Sweep event completely depend on the operator. The lower time bound for this event is taken as 5 seconds with an additional 20 ms for the communication delay, as it is assumed the operator does not repeatedly initiate the command after every short interval of time interval. The upper time bound is set to infinity to account for the situation as the operator may take a long time to issue a Full_Sweep command.

The Wait_2_Sec event has a lower time bound and upper time bound of 2 seconds with additional 20 ms of communication delay since the event represents 2 seconds delay after every 2° rotation of the motor.

The timing bounds for Full_Sweep and Wait_2_Sec events is given in Table 5.7.

Table 5.7: Other remaining events time bounds.

Event	Lower Time Bound (sec)	Upper Time Bound (sec)
Full_Sweep	5.02	∞
Wait_2_Sec	2.02	2.02

5.4 Theoretical Results

As mentioned in Section 3.3 of Chapter 3, the theoretical $T'_{min,low}(\delta)$ is a *lower bound* for $T_{min}(\delta)$. To obtain $T'_{min,low}(\delta)$, the untimed model of the solar tracker system under supervision (see Table 4.17) is transformed to a TA using the guidelines mentioned in Section 3.2. Let us denote the TA of the solar tracker system under supervision as TA_{STS}. TA_{STS} is the input for Algorithm 3.1 and Algorithm 3.2, which calculates $T'_{min,low}(\delta)$. However, $T'_{min,low}(\delta)$ may result in a very conservative value. This could lead to the selection of large value for buffering parameters for LLP and thus, larger look-ahead window size N_w in turn resulting in an LLP supervisor equivalent to the offline supervisor. Therefore, one must choose the buffering parameters meticulously in order to utilize the advantages of LLP with Buffering. For this purpose we have imposed a threshold criterion that permits disregarding $t'_{min}(q', \delta)$ values calculated from some modes of the TA that appear to be very conservative.

Typically, similar practice is followed when determining the BCET and the WCET for programs as they may result in optimistic or conservative scheduling policies [42].
In a sense, $T_{min}(\delta)$ can be viewed as BCET of δ number of events in a system. The algorithm to compute $T'_{min,low}(\delta)$ is similar to the static method which uses zone-based abstraction and thus results in the lower bound for $T_{min}(\delta)$ and the procedure to calculate $T_{min,exp}(\delta)$ is similar to the measurement-based method which results in an overestimate for $T_{min}(\delta)$. In this case we will consider the overestimate as the upper bound for $T_{min}(\delta)$. To explain this better, consider Fig. 5.1 which illustrates the distribution of $t'_{min}(q_l, 8)$ from each mode x of the TA_{STS}. For each $x \in X$ of TA_{STS}, $q_{l,x} = (x, \boldsymbol{c_{l,x}})$. From Table 5.2, $T_{min,exp}(8) = 343$ ms and therefore, in Fig. 5.1 the values of $t'_{min,low}(q_l, 8)$ that are above 343 ms can be disregarded as they do not contribute in the calculation of the $T'_{min,low}(\delta)$. The values that are under 343 ms are the ones that contribute for $T_{min}(\delta)$ and have more importance. The small value for $t'_{min}((x, \boldsymbol{c}_{l,x}), 8)$ are 230 ms. But we

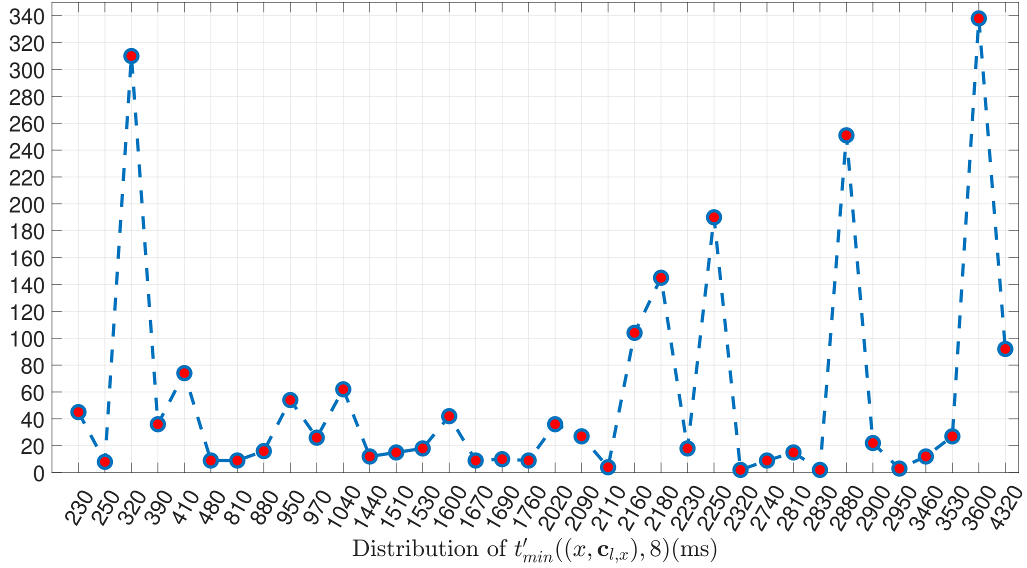


Figure 5.1: Number of modes of TA_{STS} with same $t'_{min}((x, \boldsymbol{c}_{l,x}), 8)$.

observe that this value occur for a small number of modes (45). To quantify “not frequent" a 5% threshold is selected in this thesis for the solar tracker system that: if the number of modes that result in $t'_{min}(q_l, \delta)$ lower than $T_{min,exp}(\delta)$ are less than 5% of the total modes of the TA, then $t'_{min}(q_l, \delta)$ from those modes are disregarded.[2] Given the fact that $t'_{min}(\delta)$ is used to obtain a lower estimate for $T_{min}(\delta)$, we may consider ignoring low values of $t'_{min}(\delta)$ if they occur in a “small" number of modes. In other words, if the modes that generate $t'_{min}(q_l, \delta)$ are lower than the experimental value and are not frequent, then they are ignored. The TA_{STS} has 2061 modes and 5% threshold translates to approximately 104 modes. The reasons to select 5% threshold are as follows.

(1) When the solar tracker system is moving in these 2061 modes, in 5% of the cases, the value of $T'_{min,low}(\delta)$ might be far lower than the actual $T_{min}(\delta)$ and even if the solar tracker system reaches any of these modes, it is guaranteed that the actual $T_{min}(\delta)$ will likely be higher since $T'_{min,low}(\delta)$ is a lower bound for $T_{min}(\delta)$.

(2) If all the modes are considered, then the LLP design will be too conservative which is not desirable.

[2]The exact threshold also depends on how critical the application is. The more critical the system, lower the threshold should be.

Table 5.8: Theoretical $T'_{min,low}(\delta)$ (ms) for $\delta = 1, \ldots, 9$.

δ	$T'_{min,low}(\delta)$ (ms)
1	0
2	0
3	0
4	0
5	0
6	90
7	230
8	320
9	390

From Fig. 5.1, we observe that $t'_{min}(\delta)$ values of 230 ms and 250 ms occur in 2.5% of the modes. The value of 320 ms is obtained for 15.04% of modes. Therefore, based on the 5% rule, $t'_{min}(\delta)$ values of 230 ms and 250 ms, are ignored. Thus $T'_{min,low}(\delta) = 320$ ms. Finally, the same threshold is applied for determining $T'_{min,low}(\delta)$, for other values of δ from 2 to 9 and reported in Table 5.8.

5.5 Computation Time Analysis of LLP with Buffering

In LLP with Buffering, the product of solar tracker plant and specification is expanded for a window of $N_w = N_{min} + \Omega$. Different combinations of δ and Δ can result in same Ω value. For instance, $\Omega = 7$ is the same in both the cases when $\delta = 3$ and $\Delta = 4$ and $\delta = 2$ and $\Delta = 5$; thus, the expanded plant model will be the same in both the cases. The only difference is that the number of LLP supervisory computations will change. The computation time however would be the same. In other words, regardless of the different values of δ and Δ, if Ω is the same for different combinations of δ and Δ, then $C_{max}(N_w)$ for LLP supervisor for the look-ahead window size N_w would be the same.

The parameter δ provides the information about how many events are left in the buffer before supervisory calculations must be performed for the next window. It is crucial to complete the computation of the LLP supervisor commands for the window size of N_w within the occurrence of δ events. A sufficient condition is $C_{max}(N_w) \leq T_{min}(\delta)$. For the solar tracker system $N_{min} = 6$. The supervisory control computation times $C_{max}(N_w)$ is called $2 \leq \delta \leq 9$ and $\Delta = \delta + 1$ in

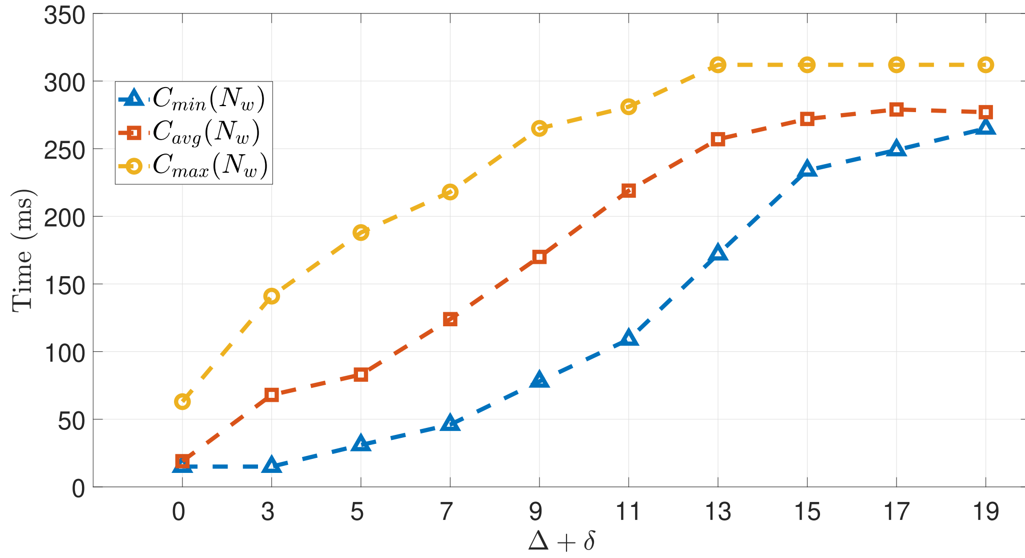


Figure 5.2: Comparison between $C_{max}(N_w)$, $C_{avg}(N_w)$ and $C_{min}(N_w)$ where $N_w = N_{min}+\Delta+\delta$, and $N_{min} = 6$.

Table 5.9: $C_{max}(N_{min} + \Omega)$ for LLP with Buffering in ms ($\Omega = \delta + \Delta$).

$\boldsymbol{\Omega}$	**0**	**3**	**5**	**7**	**9**	**11**	**13**	**15**	**17**	**19**
$\boldsymbol{C_{max}}(N_w)$	63	156	188	203	218	250	265	312	312	312

Table 5.9.[3] Table 5.9 also includes the computation time for the regular LLP supervisor when δ and Δ are both zero. There are two reasons to select $\delta = 1, \ldots, 9$ and $\Delta = \delta + 1$: (1) we would like to know the minimum $C_{max}(N_w)$ for each value of δ and; (2) going beyond $SP = 19$ will not change the $C_{max}(N_w)$ since the expansion of the product of the plant and the specifications will result in the complete model [5]. The minimum, average and the maximum computation times for LLP supervisor with buffering for the solar tracker system is illustrated in the Fig. 5.2.

5.6 Selection of Buffering Parameters

Selecting the correct buffering parameters plays a crucial role in LLP with Buffering involves tradeoffs:

(1) If the size of the expansion N_w becomes SP, then LLP supervisor is same as the offline

[3] $C_{max}(N_w)$ is determined by measuring end-to-end execution time required for computing LLP supervisor in PC. The LLP supervisory computations are performed on a standard desktop PC with Intel(R) Core(TM) i5-2400 processor operating at 3.10 GHz and 8 GB RAM.

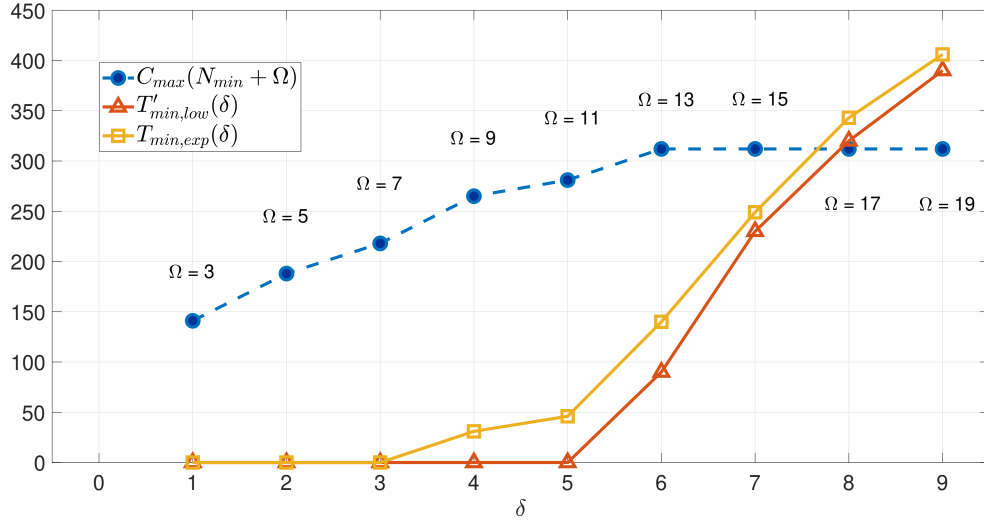


Figure 5.3: Comparison between $C_{max}(N_w)$, $T'_{min,low}(\delta)$ and $T_{min,exp}(\delta)$, with $N_w = N_{min} + \Omega$ and $N_{min} = 6$.

supervisor. Consequently, more memory is consumed due to the expansion of the complete plant and the specifications.

(2) If N_w has a value close to N_{min}, then the advantages of LLP with Buffering are not utilized since frequent LLP supervisor computation will be required. As a result, this will take considerable CPU time which may result in other tasks being preempted [5].

Rigorous experiments are performed to obtain the correct buffering parameters. While there are a plethora of combinations for the value of δ and Δ, we need to determine only those values of the buffering parameters that for the resulting look-ahead window N_w, $C_{max}(N_w)$ can be computed within the deadline of δ events.

Fig. 5.3 compares the value of $T'_{min,low}(\delta)$, $T_{min,exp}(\delta)$ and $C_{max}(N_w)$ from the Table 5.2, Table 5.8 and Table 5.9 respectively, which provides the maximum time for LLP computations for any given frame size Δ. If we aim for

$$C_{max}(N_w) < T'_{min,low}(\delta) < T_{min,exp}(\delta), \quad (35)$$

then from Fig. 5.3 we conclude that δ can have three values $7, 8$ and 9. For $\delta = 7$, equation (35) will be satisfied for Ω values $3, 5$ and 7. But with $\delta = 7$, $\Omega = 7$, $\Delta = \Omega - \delta = 0$. So, $\delta = 7$ is

Table 5.10: Number of LLP computations and supervisor size with respect to N_w.

Look-ahead Window Size N_w ($N_{min} = 6$)	**Number of LLP Computations**	**Supervisor**	
		States	*Transitions*
$\delta = 0, \Delta = 0$ ($N_w = 6$)	1153	91	330
$\delta = 8, \Delta = 9$ ($N_w = 23$)	158	2001	9159
$\delta = 9, \Delta = 10$ ($N_w = 25$)	144	2061	9467

unacceptable. If we choose $\delta = 8$, then equation (35) is satisfied for $3 \leq \Omega \leq 15$. For $\delta = 7$ and $\Omega = 15$, we get $\Delta = \Omega - \delta = 8$ which is acceptable. It can be observed from the Fig. 5.3 that for the buffering parameters $\delta = 8$ and $\Delta = 9$ ($\Omega = 17$)

$$C_{max}(23) = 312\, ms < T'_{min,low}(8) = 320\, ms < T_{min,exp}(8) = 343\, ms.$$

For $\delta = 9$, equation (35) holds for $3 \leq \Omega \leq 19$. Only $\Omega = 19$ results in an acceptable $\Delta = \Omega - \delta = 10$. For $\delta = 9$ and $\Delta = 10$, the computation time of the LLP supervisor is well under the $T'_{min,low}(8)$. However in this case, the LLP supervisor size is the same as the size of the offline supervisor. Therefore, for the solar tracker system we select the buffering parameters $\delta = 8$ and $\Delta = 9$.

Table 5.10 provides the number of LLP supervisory computations and the supervisor size in terms of the states and transitions when N_w is 6, 23 and 25. From the Table 5.10, when $\delta = 8$ and $\Delta = 9$, the size of the LLP supervisor is almost equal to the size of offline supervisor (see Table 4.17). However, there is still LLP with Buffering performed on the product of the expanded plant and expanded specifications is not completely explored; nevertheless, the memory consumption will be high. On the contrary, the number of LLP computations during the “Full Sweep" maneuver for the solar tracker plant is reduced considerably. In comparison with the regular LLP ($N_w = 6$), there is a significant 86.30% reduction in the number of LLP computations. For $\delta = 0$, $\Delta = 0$, $C_{max}(6) = 60$ ms and the *computation time* $= 63 \times 1153 = 72639$ ms. However, for $\delta = 8$, $\Delta = 9$, $C_{max}(23) = 312$ ms and the *computation time* $= 312 \times 158 = 49296$ ms. It can be noticed that there is a significant 32% reduction in the total computation time when $\delta = 8$ and $\Delta = 9$ is chosen.

The condition $C_{max}(N_w) < T_{min}(\delta)$ could be too conservative and strict. This depends on the

control system, for instance how many times it is acceptable to miss the completion of LLP supervisory computations within the deadline of δ events. Therefore, using these buffering parameter values will provide a starting point and the values can be adjusted further until they are acceptable for any given control system.

Lower values of buffer parameters can be selected by using faster CPU and by implementing efficient algorithm for computing LLP supervisor. There are many software tools which uses efficient algorithms for computing supervisors. Supremica [65] is one of the most complete tool among the existing supervisory tools which implements compositional abstraction-based techniques [66], as well as binary decision diagram base methods for synthesizing supervisors. Supremica requires 0.8s to synthesize a modular supervisor for an agv plant containing $2.57 \cdot 10^7$ reachable states.[4] (See Table 1 in [65]). Using such efficient algorithms for computing LLP supervisor will definitely contribute in selecting lower values for buffer parameters.

5.7 Summary

In this chapter we reviewed buffering parameters used in LLP with Buffering. Then, we discussed the sets of tests performed on solar tracker system and the procedure to calculate $T_{min}(\delta)$ experimentally. We also discussed the selection of timing information of all events (that were previously mentioned in Chapter 4). Then, we compared the results of theoretically obtained $T'_{min,low}(\delta)$ with experimentally obtained $T_{min,exp}(\delta)$ for selecting the buffering parameters for LLP supervisor of solar tracker system. In the next chapter, we will discuss conclusion and future work of this thesis.

[4] All the experiments were performed on a standard desktop PC using a single 3.3 GHz microprocessor and not more than 2 GB of RAM.

Chapter 6

Conclusion

6.1 Summary

This thesis examines LLP with Buffering. In particular, it examines the process of choosing the buffer size. A key factor in this decision is the rate of event generation in the plant. In this thesis, a model-based approach is developed for finding the minimum execution duration of TA.

Firstly, we assume that the timing information of all events such as lower time bound and upper time bound is available. Secondly, we assume that all events are observable. Then, a procedure with a detailed set of guidelines is presented to augment the untimed model of the system under supervision and to transform it as a specific class of TA by incorporating the timing information of the events. Next, all the event sequences (of any given length) are traversed from each mode of TA through an exhaustive matrix-based symbolic analysis algorithm. The infinite state space of the TA and the region-based abstraction to RA are disregarded and instead the clock zone-based study of the TA is used which is efficiently represented by DBM data structure in the algorithm. In addition, the ambiguity of deciding which reachable clock valuations of each mode would result in smallest $T_{min}(\delta)$ is avoided by making all the events eligible by setting their respective clocks to the lower time bounds value of the defined events. The proposed method is conservative and provides a conservative lower bound on the value of $T_{min}(\delta)$. The theoretically obtained $T_{min}(\delta)$ can be too conservative which is handled by implementing an application specific threshold criterion to

disregard very low values of minimum execution duration of event sequences. Since, the region-based abstraction is avoided, the required reachability analysis requires polynomial time complexity in the number of events of the TA.

As a proof of concept, the theoretical analysis is performed on the TA model of a two-degree-of-freedom solar tracker system. Using both the theoretically calculated $T_{min}(\delta)$ (an underestimate) and experimentally calculated $T_{min}(\delta)$ (an overestimate), results in theoretical analysis being more reliable for selecting the value of buffer parameters for LLP. The theoretical analysis guarantees that the selected buffering parameters are correct in the sense that LLP with Buffering supervisor can always compute the necessary commands before the buffer runs out empty. The model of the solar tracker system and the offline supervisor for it has 1584 states and 2061 states respectively. If the speed of LLP with Buffering supervisory code is improved, then LLP with buffering can be implemented in larger and more complex systems. Some possible directions for speeding up the code are discussed in the next section.

6.2 Future Work

The algorithm proposed in this thesis involves DBM calculations. DBMs can be represented symbolically as binary decision diagrams (BDDs) as described in [67, 68] which can be used in our algorithm for reducing the time complexity. In addition, the canonicalization of DBM represented by BDD as described in [69] can also be used in our algorithm for reducing the time complexity. Moreover, the speed of the LLP with Buffering supervisory code computation can be improved by using BDDs as used in symbolic supervisory control (e.g. [70, 71]) and by using recursive calculations as mentioned in [13]. By reducing LLP with Buffering supervisory code computation time, the size of the buffer and memory consumption will be reduced.

In this thesis, we assumed constant size for the buffer. However, the buffer size can be made variable by using the values of supervisory code computation time (i.e. $C_{max}(N_w)$) and minimum sequence duration (i.e $T_{min}(\delta)$) at the current state of the system. As a result, for instance, during LLP computation, if the execution time expected for the events remaining in the buffer is sufficiently long for computing the upcoming LLP supervisory commands, then the size of the next buffer can

be increased. There are two ways we can extend this:

(1) During the LLP computation, the value of $T_{min}(\delta)$ from the current state of the system model and for the remaining number of events in the buffer can be calculated online.

(2) The exhaustive algorithm calculates the minimum execution duration of event sequences from every mode of the TA model of the system. Thus, one can calculate beforehand (offline) the minimum execution duration of event sequences of different length from every mode of the TA. Then, the minimum execution duration of the event sequences along with their length from each mode can be stored in a look-up table. Therefore, depending upon the current state (respective mode) and the number of events left in the buffer at any instance of LLP execution, the size of the buffer can be varied by choosing an appropriate value for the buffer size from the look-up table.

Implementing the above-mentioned improvements in LLP with Buffering will definitely reduce the values of buffering parameters thereby reducing the memory consumption for storing the LLP with Buffering supervisor and reduce the computation time of the code for LLP with Buffering.

Bibliography

[1] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2008.

[2] S.-L. Chung, S. Lafortune, and F. Lin, "Limited lookahead policies in supervisory control of discrete event systems," *IEEE Transactions on Automatic Control*, vol. 37, no. 12, pp. 1921–1935, 1992.

[3] E. Ghaheri, "Limited lookahead supervisory control with buffering in discrete event systems," Master's thesis, Concordia University, August 2018.

[4] R. Alur, *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[5] F. U. Rehman, "Extension and implementation of look-ahead supervisory control with buffering," Master's thesis, Concordia University, July 2020.

[6] K. Searle, "Microcontroller based supervisory control of a solar tracker," Master's thesis, Concordia University, December 2016.

[7] Y.-C. Ho, "Introduction to special issue on dynamics of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 3–6, 1989.

[8] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM journal on control and optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[9] W. M. Wonham and K. Cai, *Supervisory Control of Discrete-Event Systems*. Springer, 2019.

[10] B. C. Williams and P. P. Nayak, "A model-based approach to reactive self-configuring systems," *Proceedings of the National Conference on Artificial Intelligence*, pp. 971–978, 1996.

[11] B. C. Williams, M. D. Ingham, S. H. Chung, and P. H. Elliott, "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 212–237, 2003.

[12] S.-L. Chung, S. Lafortune, and F. Lin, "Supervisory control using variable lookahead policies," *Discrete Event Dynamic Systems*, vol. 4, no. 3, pp. 237–268, 1994.

[13] ——, "Recursive computation of limited lookahead supervisory controls for discrete event systems," *Discrete Event Dynamic Systems*, vol. 3, no. 1, pp. 71–100, 1993.

[14] N. B. Hadj-Alouane, S. Lafortune, and F. Lin, "Variable lookahead supervisory control with state information," *IEEE Transactions on Automatic Control*, vol. 39, no. 12, pp. 2398–2410, 1994.

[15] R. Kumar, H. M. Cheung, and S. I. Marcus, "Extension based limited lookahead supervision of discrete event systems," *Automatica*, vol. 34, no. 11, pp. 1327–1344, 1998.

[16] S. Takai, "Estimate based limited lookahead supervisory control for closed language specifications," *Automatica*, vol. 33, no. 9, pp. 1739–1743, 1997.

[17] C. Winacott, B. Behinaein, and K. Rudie, "Methods for the estimation of the size of lookahead tree state-space," *Discrete Event Dynamic Systems*, vol. 23, no. 2, pp. 135–155, 2013.

[18] H. Umemoto and T. Yamasaki, "Optimal llp supervisor for discrete event systems based on reinforcement learning," *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 545–550, 2015.

[19] K. Kobayashi and T. Ushio, "An application of llp supervisory control with petri net models in mobile robots," *IEEE International Conference on Systems, Man and Cybernetics*, vol. 4, pp. 3015–3020, 2000.

[20] A. Tsalatsanis, A. Yalcin, and K. P. Valavanis, "Dynamic task allocation in cooperative robot teams," *Robotica*, vol. 30, no. 5, pp. 721–730, 2012.

[21] C. Zhao, W. Dong, and Z. Qi, "Active monitoring for control systems under anticipatory semantics," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 318–325.

[22] N. Berthier, H. Marchand, and É. Rutten, "Symbolic limited lookahead control for best-effort dynamic computing resource management," *IFAC-PapersOnLine*, vol. 51, no. 7, pp. 112–119, 2018.

[23] J. S. Ostroff and W. M. Wonham, "A framework for real-time discrete event control," *IEEE Transactions on Automatic Control*, vol. 35, no. 4, pp. 386–397, 1990.

[24] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pp. 414–425, 1990.

[25] R. Alur and D. Dill, "Automata for modeling real-time systems," *International Colloquium on Automata, Languages, and Programming*, pp. 322–335, 1990.

[26] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[27] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and Computation*, vol. 111, no. 2, pp. 193–244, 1994.

[28] K. G. Larsen, P. Pettersson, and W. Yi, "Compositional and symbolic model-checking of real-time systems," *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp. 76–87, 1995.

[29] ——, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1, pp. 134–152, 1997.

[30] S. Yovine, "Kronos: A verification tool for real-time systems," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 123–133, 1997.

[31] H. Wong-Toi, "Symbolic approximations for verifying real-time systems," Ph.D. dissertation, Stanford University, 1995.

[32] B. A. Brandin and W. M. Wonham, "Supervisory control of timed discrete-event systems," *IEEE Transactions on Automatic Control*, vol. 39, no. 2, pp. 329–342, 1994.

[33] A. Saadatpoor and W. Wonham, "Supervisor state size reduction for timed discrete-event systems," *2007 American Control Conference*, pp. 4280–4284, 2007.

[34] H. Wong-Toi and G. Hoffmann, "The control of dense real-time discrete event systems," Stanford University, Tech. Rep. STAN-CS-92-1411, 1992.

[35] A. Rashidinejad, M. Reniers, and M. Fabian, "Supervisory control synthesis of timed automata using forcible events," *arXiv preprint arXiv:2102.09338*, 2021.

[36] A. Gouin, L. Libeaut, and J.-L. Ferrier, "Supervisory control of timed automata," *1999 European Control Conference (ECC)*, pp. 543–550, 1999.

[37] A. Khoumsi and L. Ouedraogo, "A new method for transforming timed automata," *Electronic Notes in Theoretical Computer Science*, vol. 130, pp. 101–128, 2005.

[38] A. Khoumsi and M. Nourelfath, "An efficient method for the supervisory control of dense real-time discrete event systems," *Proceedings of the 8th International Conference on Real-Time Computing Systems (RTCSA)*, 2002.

[39] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems," *Annual Symposium on Theoretical Aspects of Computer Science*, pp. 229–242, 1995.

[40] E. Asarin, O. Maler, and A. Pnueli, "Symbolic controller synthesis for discrete and timed systems," *International Hybrid Systems Workshop*, pp. 1–20, 1994.

[41] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis, "Controller synthesis for timed automata," *IFAC Proceedings Volumes*, vol. 31, no. 18, pp. 447–452, 1998.

[42] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, pp. 1–53, 2008.

[43] J. Moody and P. J. Antsaklis, *Supervisory Control of Discrete Event Systems Using Petri Nets*. Springer Science & Business Media, 1998, vol. 8.

[44] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE*, vol. 77, no. 1, pp. 81–98, 1989.

[45] R. H. Moulton, A. J. Marasco, and K. Rudie, "Limited lookahead policies for the control of discrete-event systems: A tutorial," *arXiv preprint arXiv:2006.06514*, 2020.

[46] F. Boroomand and S. Hashtrudi-Zad, "A limited lookahead policy in robust nonblocking supervisory control of discrete event systems," *2013 American Control Conference*, pp. 935–939, 2013.

[47] R. Alur, C. Courcoubetis, D. L. Dill, N. Halbwachs, and H. Wong-Toi, "An implementation of three algorithms for timing verification based on automata emptiness," *IEEE International Real-Time Systems Symposium*, pp. 157–166, 1992.

[48] J. Bengtsson and W. Yi, "Timed automata: semantics, algorithms and tools," *Advanced Course on Petri Nets*, pp. 87–124, 2003.

[49] D. L. Dill, "Timing assumptions and verification of finite-state concurrent systems," *International Conference on Computer Aided Verification*, pp. 197–212, 1989.

[50] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[51] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[52] M. M. Bersani, M. Rossi, and P. San Pietro, "On the initialization of clocks in timed formalisms," *Theoretical Computer Science*, vol. 813, pp. 175–198, 2020.

[53] K. H. Rosen, *Handbook of Discrete and Combinatorial Mathematics*. CRC Press, 2017.

[54] S. Miremadi, Z. Fei, K. Åkesson, and B. Lennartson, "Symbolic representation and computation of timed discrete-event systems," *IEEE Transactions on Automation Science and Engineering*, vol. 11, no. 1, pp. 6–19, 2013.

[55] M. Sipser, *Introduction to the Theory of Computation*. ACM New York, NY, USA, 1996, vol. 27, no. 1.

[56] B. N. Datta, *Numerical Linear Algebra and Applications*. SIAM, 2010, vol. 116.

[57] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[58] K. Searle and S. Hashtrudi-Zad, "Microcontroller based supervisory control of a solar tracker," *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–6, 2017.

[59] S. Hashtrudi Zad, S. Zahirazami, and F. Boroomand, "Discrete event control kit (deck)," November 2013. [Online]. Available: http://users.encs.concordia.ca/~shz/deck/

[60] MATLAB, *9.4.0.813654 (R2018a)*. Natick, Massachusetts: The MathWorks Inc., 2018.

[61] W. M. Wonham and P. J. Ramadge, "On the supremal controllable sublanguage of a given language," *SIAM Journal on Control and Optimization*, vol. 25, no. 3, pp. 637–659, 1987.

[62] Flex solar cells. [Online]. Available: http://www.flexsolarcells.com/

[63] Hs-645mg high torque, metal gear premium sport servo. [Online]. Available: https://www.robotshop.com/media/files/pdf/hs645mg.pdf

[64] Hs-805bb mega giant scale servo. [Online]. Available: https://cdn.sparkfun.com/datasheets/Robotics/hs805.pdf

[65] R. Malik, K. Åkesson, H. Flordal, and M. Fabian, "Supremica–an efficient tool for large-scale discrete event systems," *The 20th World Congress of the International Federation of Automatic Control, Toulouse, France*, pp. 9–14, July 2017.

[66] S. Mohajerani, R. Malik, and M. Fabian, "A framework for compositional synthesis of modular nonblocking supervisors," *IEEE Transactions on Automatic Control*, vol. 59, no. 1, pp. 150–162, 2013.

[67] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi, "Algebric decision diagrams and their applications," *Formal Methods in System Design*, vol. 10, no. 2, pp. 171–206, 1997.

[68] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large boolean functions with applications to technology mapping," *Proceedings of the 30th International Design Automation Conference*, pp. 54–60, 1993.

[69] F. Balarin, "Approximate reachability analysis of timed automata," *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 52–61, 1996.

[70] C. Ma and W. M. Wonham, "Nonblocking supervisory control of state tree structures," *IEEE Transactions on Automatic Control*, vol. 51, no. 5, pp. 782–793, 2006.

[71] S. Miremadi, K. Akesson, and B. Lennartson, "Symbolic computation of reduced guards in supervisory control," *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 4, pp. 754–765, 2011.