Extension and Implementation of Look-ahead Supervisory Control with Buffering

Faiz Ur Rehman

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 2020

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Faiz Ur Rehman**

Entitled: **Extension and Implementation of Look-ahead Supervisory Control with Buffering**

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. K. Skonieczny

_____ External Examiner
Dr. W.F. Xie (MIAE)

_____ Examiner
Dr. K. Skonieczny

_____ Supervisor
Dr. S. Hashtrudi Zad

Approved by _____
Yousef Shayan, Chair
Department of Electrical and Computer Engineering

_____ 2020

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

Abstract

Extension and Implementation of Look-ahead Supervisory Control with Buffering

Faiz Ur Rehman

The Supervisory Control Theory of Discrete Event Systems (DES) provides procedures to design supervisors to control plants modeled as DES. The computed supervisor issues control commands to ensure design specifications, such as safety constraints, are met. In the supervisor design process, the plant and design specification models are used to obtain the supervisor in the form of a DES. In this approach, the control commands are effectively pre-calculated before implementation; thus the approach is known as Offline Supervisor Design. The challenge associated with this method is that it requires large onboard memory for supervisor (due to typically large DES models involved). Such large memory is not available on embedded systems.

In order to make the implementation of supervisory control feasible for embedded systems, an approach is proposed in the literature where at any given time, supervisory commands are computed on-the-fly based on models for plant and specifications covering a small window into the future (i.e., lookahead window). This method needs a significantly smaller onboard memory. As a result, however, frequent control command computation is needed. This could pose a implementation challenge since control commands must be computed after every new event in the plant. Sometimes two (or more) consecutive events could occur in rapid succession in the plant and there may not be enough time to compute control commands. To mitigate this problem, an approach has been proposed called

Lookahead Supervision with Buffering in which commands are computed and buffered in advance for a window.

This thesis makes contributions to the underlying theory of lookahead policy with buffering. Specifically it proposes a method to use the timed model of the plant to compute the timing information of event sequences. This timing information is used in choosing the buffer size and was previously obtained experimentally. The thesis also develops a method for computing plant and specification models over the loakahead window that is suited for computer coding.

The thesis also implements the lookahead supervision with command buffering. To study the feasibility of implementation and the complexity of proposed controller in detail, a two-degree-of-freedom solar tracker equipped is used as plant. The goal is to generate supervisory commands for maneuvering the solar tracker to find a bright light source for charging battery. For implementation, all supervisory control algorithms are written in C language for faster computation time. Look-ahead policy with command buffering is designed and implemented. In several tests, the supervisor successfully calculates on-line the control commands in a timely fashion and maneuvers the solar tracker to the bright source while respecting design specifications. The experimental results show that the timing information calculated with the proposed method based on timed model match the actual plant behavior. Furthermore, the experiments demonstrate that the length of command buffer (as design parameter) can be used to achieve a compromise between onboard memory requirement and computational power.

Acknowledgments

In the name of ALLAH Almighty, the Most Merciful and the Most Beneficent. All praises and thanks be to Him.

First, I would like to express my sincere gratitude to my supervisor Dr. Shahin Hashtrudi Zad, for his invaluable guidance throughout my research. This work would not have been possible without his continuous support.

Second, I would like to thank my loving parents, sisters, and brother for their love, prayers, support, and encouragement. I especially dedicate this thesis to my father, who has always been an inspiration to me.

Third, I would like to thank the Embedded team and the management of Neptronic for their support and flexibility that helped me throughout my work. Especially, my co-orkers who kept me motivated.

Finally, I would like to thank my friends, who were always there to encourage me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Discrete Event Systems (DES) are systems whose state evolve following the occurrence of events (which can be sporadic and depend on the physical state). Supervisory Control Theory (SCT) provides an approach to generate control commands as a sequence of events for any plant that is modeled as DES. In SCT, the plant's behavior is restricted by the supervisor which disables/enables events based on design specifications. This is a model-based approach that has significant advantages such as being easy to understand and implement, robust against programming errors and reducing the complexity of models with techniques such as a divide-and-conquer.

Normally the supervisor is computed before implementation; this is known as "offline" control computation. However, the number of states in the supervisor typically increases tremendously, requiring large memory. This leads to a feasibility issue for embedded implementations. An alternative approach, Limited Look-ahead Policy (LLP), is proposed in literature where the supervisory commands are computed on-the-fly, i.e. "online", after each event. Using this methodology imposes, time constraints for supervisor computation as commands must be computed during plant operation.

To overcome the implementation timing restriction, a new version of LLP with Buffering has been recently proposed. In this thesis, we intend to implement the LLP with Buffering

on a solar tracker plant, and analyze the performance of the algorithm in terms of computational complexity. In the process, we aim to extend the underlying theory of LLP with Buffering.

In this chapter, we will review SCT for DES followed by the work done in SCT implementation and LLP. Finally thesis objectives and outline will be discussed.

1.1 Supervisory Control Theory for DES

Discrete Event Systems (DES), as the name suggests, represent any system with discrete state set. The state of such system changes with the occurrence of events leading to transition from one state to another. Events depend on the dynamical behavior of the underlying physics. An automaton is one type of DES model.

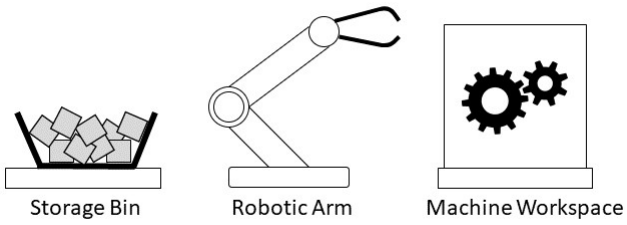


Figure 1.1: Small Factory

Let us discuss an example of a small factory modeled as DES. The setup consists of three components, storage bin, robotic arm and machining workspace. Assume the bin will never runs out of work-pieces and as system starts the machine is in idle state. The process is initiated with the press of the start button (Event: Start). The arm will wait (remains in the same state) until the machine is in idle state. The arm will place a new work-piece in the work-space (Event: New Feed), followed by work operation (Event: Start Task). And once the work is done, the system goes back to the idle state (Event: Task Completed). The

DES model is shown in Fig. 1.2.

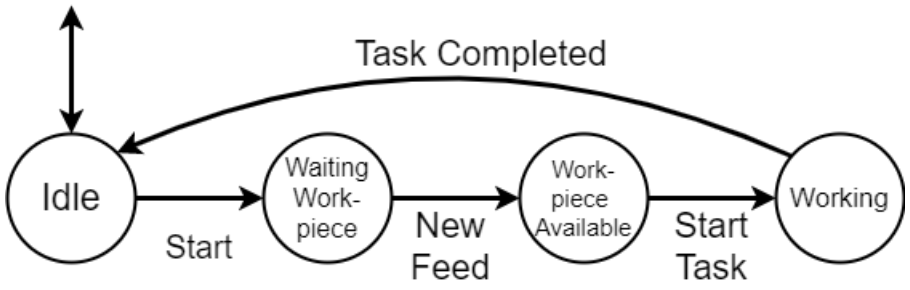


Figure 1.2: DES model of Small Factory.

In supervisory control theory, the designed controller is referred to as supervisor and is responsible for event disablement (in case where a certain event sequence does not meet the design specifications) to ensure safe operation. The trajectory of events enabled must lead the system to some state in which task is considered complete. These states are called marked states.

Events in DES are divided into two categories

(1) Uncontrollable Events

(2) Controllable Events

Inevitable events and those which are not meant to be controlled by supervisor (e.g. emergency shutdown issued by an operator) are modeled as uncontrollable event. In Fig 1.2, "Task Completed" is uncontrollable. Such events must not be disabled by the supervisor. While events like "Start" and "New Feed" from Fig 1.2 are controllable and can be enabled/disabled by supervisor. "Start Task" is uncontrollable event that can be triggered by proximity sensor (in case when work-piece is available).

Marked states signify the completion of any task and are represented by arrow directing away from the state (e.g. "Idle" state in Fig. 1.2).

The plant under supervision (i.e. closed-loop system) is shown in Fig. 1.3

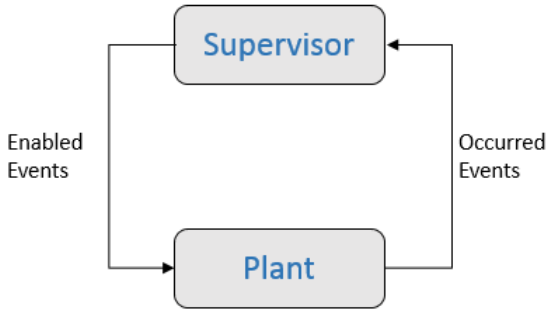


Figure 1.3: Plant under supervision (closed-loop system).

1.2 Literature Review

SCT has been very appealing for controller design due to its model-based approach (which makes it less prone to coding errors). Consequently, it is the focus of research in the field of autonomous systems especially for space applications. A lot of successful missions have been conducted in the past. In this section, we will discuss the study of SCT with respect to space applications. Followed by the discussion of the implementation of SCT in the literature, the limitations of conventional SCT and a review of limited look-ahead policy approaches. In last section, we will review the timed discrete event systems.

1.2.1 Autonomous Systems and Space Applications

Systems that can handle the decision making without human intervention are said to be Autonomous Systems [2]. Space exploration has always been expensive in terms of operations and manufacturing. Integrating the autonomous system with flight software significantly reduces the load of the task from the operating crew. Another reason for having autonomy in spacecraft is the ability to make decisions in less time than communication latency.

Deep Space 1 was the first spacecraft with artificial intelligence. In [3], an autonomous architecture was developed and implemented on the onboard computer to have a virtual presence in space. The hardware system and computational intelligence couples together forming a Remote Agent. It performs autonomous operations while respecting resource constraints. The developed architecture was integrated with DS1 flight software for testing and displayed successful autonomy for onboard operations.

A model-based approach reduces the cost of development due to re-usability. It helps the autonomous system to generate the model of the complete system in run-time. In [4] uses this approach to synthesize the complete model (from discrete models of components) for the decision-making process in case of failure of the anomaly. This architecture, Livingstone, was integrated with other components developed in [3] and implemented in DS1.

Similar model-based approaches have been used in [5] for fault diagnosis and in [4] to develop modular and robust controller. In [5], Virtual Finite State Machine (VFSM) is used to define control logic as a state machine in a virtual environment. It helps to perform extensive testing of the spacecraft's fault diagnosis system.

Generally, the development of embedded programs are application-oriented and error-prone. Especially in case of space applications where the size of program can increase and also robustness is crucial. In [4], a model-based programming approach is used to develop the code, that is robust and modular as it can reconfigure a discrete model of components to generate control commands. In the developed architecture, control goals are defined by the Control Sequencer, which set the goals while monitoring the estimated states (also known as configurable goals). These goals are provided to Deductive Controller; it estimates the plant's state using the sensor data and plant model to generate a control command for the actual plant. In [6], an automaton based approach is proposed and implemented for the development and verification of the sophisticated functional level robots, e.g., mars rovers, spacecraft and UGV etc.

1.2.2 Supervisory Control Theory

The Supervisory Control Theory (SCT) of DES was introduced in [7] and [8], and is known as Ramadge-Wonham framework. In this framework, the events are categorized into controllable and uncontrollable events. Events that cannot be directly controlled (or not meant to be controlled) are considered uncontrollable.

The supervisor restricts plant behavior. The supervisor observes the plant and disables the undesirable events (to ensure user-defined safety behavior), but it never disables any uncontrollable event. There may be the possibility that more than one controllable event is enabled from a state. This problem is known as "Issue of choice". In [9], the methodology is proposed to assign a cost to all paths from the current state to a marked state set. It minimizes the cost resulting in one controllable event but it's application is limited to acyclic graphs only. In [10], the random selection of controllable events is made as a solution to the problem of choice, however no formal proof is available to address problems generally. Much work has been done in real-time implementation of SCT, especially in manufacturing plants. The significant advantage of SCT applications is the automatic synthesis of the controller using the methodology proposed in [8]. This approach is used to design the supervisor for an automated assembly line in [11]. Moreover, the problem of choice has been tackled in an ad-hoc manner.

In [1], a high-level supervisor is generated to control multiple continuous-time robots. In this work, the supervisor generates the set of enabled events, and the task planner chooses a specific event to be executed. Later, the selected event is sent to the low-level controller, which translates the event to control command for the robot (voltage or PWM etc.). The event detector monitors the occurred uncontrollable events based on sensor data and relays it to the supervisor (Fig. 1.4).

In [12], a manually designed controller followed by automatic verification is compared with an automatically generated controller and the feasibility of the industrial control system is

proved. However, such approach may result in large supervisor size, causing implementation problems.

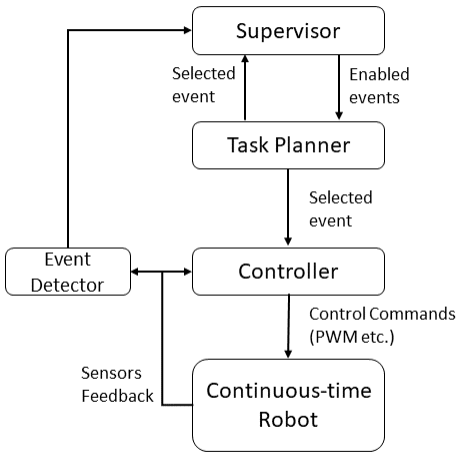


Figure 1.4: Real-time Control Architecture [1]

A modular and local modular supervision strategy are used in [13] and [14] respectively, to avoid the large size of the supervisor. The supervisors were designed for automated manufacturing cells consisting of a robot, a vision-system for feedback, a conveyor belt and two machines. Two PLCs were used for implementation and the resulting behavior of supervisors.

In [15], Compositional Interchangeable Format (CIF) is used for modeling and implementation of SCT. CIF has the advantage of building hybrid models of a system containing variables, conditions and differential equations other than untimed automata. SCT is implemented in visualization-based simulation for MRI scanner. The same tool is also used in [16] to control the airport baggage conveyor system. Initially, simulation is done with soft PLC and computer (as emulator) and later tested successfully on the real-time system at Veghel Airport.

Application of supervisors on PLC leads to several problems like avalanche effect, communication delay and inexact synchronization [17]. In [18], a heuristic approach is adopted to minimize these issues and implementing a non-blocking supervisor on PLC for an electrical power system. Similar problems have also been studied in [19], and to address them, local modular SCT is implemented using a micro-controller. Supervisors are stored as a vector called Memory safe. However, in [20], conventional SCT is implemented on micro-controller on a 2-DOF solar plant and the monolithic supervisor is stored in the form of State-Transition Table.

A lot of research has also been done for application in robotics, aiming to reduce complexity. As conventional SCT requires significant memory resources, a computer-based SCT approach is utilized where the supervisor is implemented on desktop PC and connected remotely to the plant. Such approach is adopted in [21] for wheeled robot. There, three-level control architecture is proposed to achieve position control. The supervisory level is the highest level responsible for making major decisions like motion control and obstacle avoidance. The intermediate control is the second level and manages map building, position updates, kinematics and generating secure control action for the robot to avoid collisions. These tasks are implemented on PC which is connected to robot remotely, for lowest-level control of motor actuation.

In [22], it is observed that with the increase in plant model details (or complexity), supervisor size increases rapidly; therefore decomposing complete system to sub-systems and building it again when required (for supervisor synthesis) is a viable solution to tackle memory requirement for storing supervisor.

A major limitation of conventional SCT is the large memory requirement for supervisors (which is not readily available in embedded systems, spacecraft or robotics platforms). To handle this limitation, an innovative approach is suggested in [23] where the supervisory command for next immediate action is computed by exploring a limited part of the plant

model, hence known as Limited Look-ahead Policy (LLP). The plant model is stored in the form of its components and a complete model is only built just before the synthesis of the supervisor. As the supervisor in computed on-the-fly, it is called "On-line Supervisor", while the conventional supervisor is known as "offline supervisor".

In LLP, a significantly smaller amount of memory is required as there is no need to store a complete supervisor. However, this comes at the cost of a high number of supervisory computations. As the next control action is required to be available before implementation and repeated for each iteration, not only computation frequency (for online supervisor) increases, but it also imposes strict time constraints on the system. While computing supervisory commands online, there is uncertainty in the plant's behavior as only a limited part of the plant model is explored. Therefore one needs to make sure of the optimality of computed supervisory command. Two attitudes suggested in [23] for the unexplored region of plant model are the conservative and optimistic attitudes. The choice directly affects the process of computing the supervisory command. The minimally restrictive offline supervisor is used as the benchmark for optimality of online command, i.e., if the online supervisory command is the same as the minimally restrictive offline supervisor, then it is said to be optimal. To compute the optimal supervisor command, the plant model is needed to be explored for enough depth. This parameter is known as the look-ahead window. Let the size of this window be represented by N_w. Using a large window size, complete plant model is explored which is same as offline supervisor. It would require large memory and not useful. Therefore, the window size is needed to be minimal to fully benefit from LLP computation and large enough to compute the optimal supervisory command.

In [24], a forward calculation algorithm is proposed which can cause the termination of expansion even before reaching the look-ahead window boundary. Therefore it may be less costly (in terms of supervisor computation). A similar approach is developed in [25] to reduce the computations using state information. In this algorithm, a cost is assigned to

each state: infinity if a state can lead to blocking or illegal state; otherwise zero cost is assigned. During the plant model exploration, if the algorithm comes across a state with already known cost, it stops expansion. Consequently, a state is not explored repeatedly and computational operations are reduced.

In LLP, after each event occurs in the plant, a supervisory control problem needs to be solved to find the next supervisory command. In many cases, the time between two consecutive events could be too short for such SCT calculations. This problem is addressed in [26] by introducing LLP with Buffering. Instead of computing one supervisory command, few extra commands are computed and buffered for future use. In this way, at any given moment there is control command readily available for implementation.

1.2.3 Timed Discrete Event Systems

In untimed DES, transitions do not possess any temporal information. But in real-world, all the systems evolve in timely fashion. Even though the transitions in SCT occurs instantly, the execution of each event takes time. In [27] and [28], the concept of time-based transition models is proposed and called as Timed Discrete Event System (TDES). The time is measured by the tick of a clock and each event has associated time bounds in terms of tick. The size of tick defines the resolution of timing information. For example, if tick size is small, then the model will be very detailed but the number of states will be very large. Therefore one needs to carefully tune the value of tick. Events are categorized in two groups based on time bounds: prospective (finite upper bound) and remote events (infinite upper bound). An event is only executed when it is enabled and a minimum time bound condition is fulfilled. Generally, tick event can never be preempted but to avoid illegal behavior, the idea of forcible events is put forth and only these events can preempt tick. In [29], a methodology is proposed to do failure diagnosis in TDES and a manufacturing cell is studied as an example. According to the study, state-space models can be very huge

because of system complexity and it could lead to computational complications. Few approaches are suggested to reduce complexity (e.g. choosing appropriate tick size, adopting modular design).

In order to reduce the supervisor state size in timed model, an algorithm is developed in [30]. However, in this thesis we will not implement supervisor for TDES but will use TDES models to provide a model-based approach for the calculation of execution time of event sequences.

1.3 Thesis Objectives and Contributions

In Lookahead Supervision with Buffering, commands are computed and buffered in advance for a window. This ensures that all the appropriate control commands are ready when needed. This thesis aims to make contributions to (1) the underlying theory of lookahead policy with buffering and (2) to its implementation, as described below.

- The thesis proposes a method to use the timed model of the plant to compute the timing information of event sequences. This timing information is used in choosing the buffer size in LLP with buffering. The timing information was previously obtained experimentally. A theoretical approach can analyze a wider range of system trajectories, resulting in more accurate timing information.

- An essential step in LLP is the construction of a partial model of the plant and specification. To facilitate the coding of this construction, the thesis develops a new modular and incremental algorithm for this construction. Furthermore, a library of supervisory control and LLP with the buffering algorithm is coded in C language for fast execution.

- The thesis implements the lookahead supervision with command buffering on a two-degree-of-freedom solar tracker. In several tests, the supervisor successfully calculates on-line the control commands in a timely fashion and maneuvers the solar tracker to a

bright source while respecting design specifications. The experimental results show that the timing information calculated with the proposed method based on the timed model matches the actual plant behavior.

- Furthermore, the experiments demonstrate that the length of command buffer (as design parameter) can be used to achieve a compromise between onboard memory requirement and computational power.

1.4 Thesis Outline

The outline of this thesis is as follows. In Chapter 2, we will discuss some background material of timed and untimed DES, followed by a detailed review of SCT. We will study the shortcomings of conventional methods of SCT and review the Limited Look-ahead Policy (LLP) and LLP with Buffering from the perspective of computational complexity, in Chapter 3. In Chapter 4, we will present the development of computer code (in C) for supervisory control and LLP with Buffering and compare their performance with MATLAB and MATLAB generated C functions. In Chapter 5, we will discuss the architecture of the solar tracker and model it as DES. Chapter 6 presents the model-based approach to compute the execution time of event sequences using the TDES model of the plant. Furthermore, in Chapter 7, we will present the implementation of regular LLP and LLP with Buffering on the solar tracker. Moreover, we will discuss the feasibility of the implementation of LLP with Buffering on systems with low resources (memory and processor). In Chapter 8 we will discuss the conclusion and provide some suggestions for future work.

Chapter 2

Background

In this chapter, we will discuss the preliminaries required to work with Discrete Event Systems (DES), operations that can be performed on any DES model and detailed review of Timed Discrete Event System (TDES). Later, few supervisory control problems and strategies are discussed to lay the ground-work for this thesis.

2.1 Discrete Event System

A Discrete Event System is a system with a discrete set of states. The occurrence of events causes transitions in DES from one state to another. Hence, in order to study such systems, discrete mathematics [31] is required. All events involved in DES form the alphabet of a language on which various operations can be performed. In the following section, some definitions and preliminary explanations of languages and automata are presented.

DES can be modeled as untimed and also as a timed automaton. In timed-DES (TDES), all events have defined time bounds, physical set of rules. TDES helps to perform temporal analysis on any system under supervision. Both timed and untimed models are discussed in this chapter.

2.1.1 Languages and Preliminaries

Events in DES form a set of symbols called an alphabet (Σ). Any sequence of events is called string, trace or word. Σ^+ is set of all finite events. ϵ is used to denote the trace with no event. Σ^* includes all events and empty sequence.

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

Length of any sequence is denoted by $|s|$. If s=ϵ, then $|s| = 0$.

Operations on Languages

Here we discuss some of the operation, that can be performed on languages. Let L and M be two languages over Σ,

$$L, M \subseteq \Sigma^*$$

Let string $l = xyz \in \Sigma^*$. Then x and z are prefix and suffix of l respectively.
$\overline{L}$ denotes the set of prefixes of sequences of L.

$$\overline{L} = \{x \in \Sigma^* | \exists\, y \in \Sigma^* (xy \in L)\}$$

If $L = \overline{L}$, L is called prefix-closed (or closed).
The **union** of two languages (L, M) is the set of strings belonging to L or M.

$$L \cup M = L + M = \{x | x \in L \ \ or\, x \in M\}$$

The **intersection** of two languages (L, M) contains all the common strings of L and M.

$$L \cap M = \{x | x \in L\, and\, x \in M\}$$

L^{co} denotes the **complement** of language L and contains all strings of Σ^* except those of L.

$$L^{co} = \Sigma^* - L = \{x \in \Sigma^* | x \notin L\}$$

LM is said to be the **concatenation** of languages L and M and contains strings formed from catenating one sequences from L by a sequence from M.

$$LM = \{xy \,| x \in L \ and \ y \in M\}$$

The **Kleene-Closure** of a language L is denoted by L^* and formed by concatenation of all (finite number of) strings of L, including empty string $\{\epsilon\}$. This property is idempotent i.e. $(L^*)^* = L^*$

$$L^* = \{\epsilon\} \cup L \cup LL \cup LLL...$$

The **truncation** of L until N contains all strings of L with maximum length of N (N is a non-negative integer).

$$L|_N = \{x \in L| \ \ |x| \leq N\}$$

The **post-language** of L after sequence x, is denoted by L/x and contains extensions of x in L.

$$L/x = \{y \in \Sigma^* | xy \in L\}$$

2.1.2 Automata and Operations

An automaton is a way to represent a language according to a defined set of rules. A deterministic automaton is a five tuple,

$$G = (X, \Sigma, \eta, x_\circ, X_m)$$

where

X is finite state set

Σ is finite event set of G

η is partial transition function $X \times \Sigma \to X$

$x_\circ$ is initial state ($x_\circ \in X$)

X_m is set of marked states ($X_m \subseteq X$)

Marked states are significant as they may for example represent completion of task.

The **language generated** by automaton G is defined as

$$L(G) = \{s \in \Sigma^* |\ \eta(x_\circ, s)!\}$$

$\eta(x_\circ, s)!$ means starting from x_o, the sequence of events s can take place in G.

The **marked language** of G is defined as

$$L_m(G) = \{s \in \Sigma^* |\ \eta(x_\circ, s)! \text{ and } \eta(x_\circ, s) \in X_m\}$$

The makred language is a subset of language generated ($L_m(G) \subseteq L(G)$) as it contains all traces which lead to some marked state in G.

L(G) is always prefixed closed and called the closed behavior of G . While $L_m(G)$ is not always prefix-closed as any prefix of path(s) may not lead to marked state.

Consider an automaton G with $\Sigma = \{a, b\}$ and $X_m = \{3\}$ as shown in Figure 2.1.

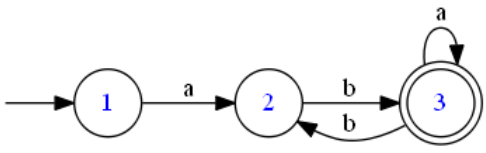


Figure 2.1: Automaton G

Then, L(G) = $\{\epsilon$, a, ab, aba, , abaa, ... $\}$ and $L_m(G)$ = {ab, aba, abaa, ...}.

Next we review some operations on automata.

A state $x \in X$ is **reachable** if for some $t \in L(G)$, $\eta(x_o, t) \in X$.

A state $x \in X$ is **coreachable** if for some $t \in \Sigma^*$, $\eta(x, t) \in X_m$.

An automaton G is **non-blocking** if every reachable state is coreachable.

Trim

Trim operation is used to remove all states of G which are not reachable or not coreachable. Let the trimmed automaton be represented by G_t.

Product

The product operation on two automatons (G_a, G_b) provides an automaton with only common behavior of operands in terms of states, events and marked behavior.

Let $G_a = \{X_a, \Sigma_a, \eta_a, x_{oa}, X_{ma}\}$ and $G_b = \{X_b, \Sigma_b, \eta_b, x_{ob}, X_{mb}\}$. Then define

$G_p = \{X_p, \Sigma_p, \eta_p, x_{op}, X_{mp}\}$

$$X_p = X_a \cap X_b$$

$$\Sigma_p = \Sigma_a \cap \Sigma_b$$

$$x_{op} = (x_{oa}, x_{ob})$$

$$X_{mp} = X_{ma} \cap X_{mb}$$

$$\eta_p((x_a, x_b), \sigma) = \begin{cases} \eta_a(x_a, \sigma), \eta_b(x_b, \sigma) & if \quad \eta_a(x_a, \sigma)! \ and \ \eta_b(x_b, \sigma)! \\ undefined & otherwise \end{cases}$$

In G_p transition to new state will only occur when same the event can happen from source states in G_a and G_b. Therefore transitions are synchronized. The $G_a \times G_b$ is the reachable part of G_p. The generated and marked languages of resultant automaton are as

follows:

$$L(G_a \times G_b) = L(G_a) \cap L(G_b)$$
$$L(G_{ma} \times G_{mb}) = L(G_{ma}) \cap L(G_{mb})$$

The **self-loop** of an event is the transition from each state to itself (x, σ, x) such that $\sigma \in \Sigma_{SL}$ and there is no transition of the event in the operand automaton i.e. $\eta(x, \sigma)$ does not exists.

$$G_{SL} = selfloop(G, \Sigma_{SL})$$

The **sync** operation on two automatons results into a new automaton with the behaviors of both operand automatons. Let us consider two automatons G_a and G_a then sync of them is represented as:

$$G_s = G_a || G_b$$
$$\Sigma_s = \Sigma_a \cup \Sigma_b$$

The transitions in sync from any state of G_a and G_b happen according to the following rule:

$$\eta_s((x_a, x_b), \sigma) = \begin{cases} \eta_a(x_a, \sigma), x_b & if \quad \eta_a(x_a, \sigma)! \\ x_a, \eta_b(x_b, \sigma) & if \quad \eta_b(x_b, \sigma)! \\ undefined & otherwise \end{cases}$$

The generated and marked languages are as follows:

$$L(G_a||G_b) = L(G_a)||L(G_b)$$
$$L(G_{ma}||G_{mb}) = L(G_{ma})||L(G_{mb})$$

One can perform the sync operation on two automatons by using product and selfloop operations. Let G_1 and G_2 are two automatons over languages Σ_1 and Σ_2 respectively. Then

$$G_{SL1} = selfloop(G_1, \Sigma_2 - \Sigma_1)$$
$$G_{SL2} = selfloop(G_2, \Sigma_1 - \Sigma_2)$$
$$G_s = product(G_{SL1}, G_{SL2})$$

Complement

Complement of any automaton G = $(X, \Sigma, \eta, x_\circ, X_m)$ is denoted by G_{co}. It generates and marks the following languages,

$$L(G_{co}) = \Sigma^*$$
$$L_m(G_{co}) = \Sigma^* - L_m(G)$$

2.2 Timed Discrete Event Systems (TDES)

So far we have discussed automata without timing information in their structure. Timing information provides another dimension to any DES model. This not only helps to analyze implementation feasibility of DES but also augments untimed DES model to do timed simulation for Supervisory Control Theory (SCT). Consequently, one gains the timed perspective of controller execution in plant. One issue with timed models is that they are

significantly more complex compared with untimed models. In this section, the modeling of TDES [32] is discussed in detail.

Similar to automaton, a TDES is also a five tuple,

$$G_{act} = (A, \Sigma_{act}, \delta_{act}, a_o, A_m)$$

A is activity set of timed automaton.

Σ_{act} is events labels (finite) .

δ_{act} is activity function, $\delta_{act} : A \times \Sigma \rightarrow A$.

a_o is initial activity.

A_m Marked activities, $A_m \subseteq A$

Σ_{act} is a finite event set [27]. Activity transition function is a partial function: $\delta_{act} : A \times \Sigma_{act} \rightarrow \Sigma_{act}$. In untimed system, state transition happens at same instant when events is enabled due to absence of timing information while in TDES, activities occur over a duration of time, while event still occurs instantaneously. In order to augment the timing behavior to activities, two additional parameters, lower time bound ($l_\sigma \in \mathbb{N}$) and upper time bound ($u_\sigma \in \mathbb{N} \cup \{\infty\}$) are introduced for each activity (σ). This triple ($\sigma, l_\sigma, u_\sigma$) is known as timed event. It is formally defined as,

$$\Sigma_{tim} = \{(\sigma, l_\sigma, u_\sigma) \mid \sigma \in \Sigma_{act}\}$$

Lower time bound (l_σ) represents delay due to implementation (of control or computation) or communication in the system, while the upper bound (u_σ) signifies the maximum permissible delay (defined by specification or physics of real plant) and l_σ is always less than or equal to u_σ.

We have two distinct types of timed-events ($\Sigma_{act} = \Sigma_{spe} \cup \Sigma_{rem}$),

1: Σ_{spe} : the set of prospective events

2: Σ_{rem} : the set of remote events

Type of event is defined according to its time bounds. Events with finite upper and lower bounds, $(0 \leq u_\sigma < \infty)$ and $(0 \leq l_\sigma \leq u_\sigma)$ are known as ***prospective events***. If the upper bound is infinite $(u_\sigma = \infty)$ and lower bound is finite $(0 \leq l_\sigma < u_\sigma)$ then such events are called as ***remote event***.

In TDES, time is represented by *tick*. It is the measure of single time unit elapsed in a global clock. The clock is never paused or stopped at any state of timed automaton. Time is measured with respect to global clock using $tickcount : \mathbb{R}^+ \rightarrow \mathbb{N}$ function, where,

$$tickcount(t) = n, \quad n \leq t < n + 1$$

Tick is also an event to simulate the time elapse for any event in timed model. Therefore complete event-set for any TDES is σ,

$$\Sigma = \Sigma_{act} \cup \{tick\}$$

State set (Q) for TDES is defined as,

$$Q = A \times \Pi\{T_\sigma \mid \sigma \in \Sigma_{act}\}$$

Here T_σ is known as timer for event σ and defined as,

$$T_\sigma = \begin{cases} [0, \mu_\sigma] & if \quad \sigma \in \Sigma_{spe} \\ [0, l_\sigma] & if \quad \sigma \in \Sigma_{rem} \end{cases}$$

Each state is (q) has an activity (a) and timer assigned to each activity within its range of time bounds. Initially timers are set to default value ($t_{o\sigma}$) which is equal to μ_σ and l_σ for prospective and remote events respectively. If σ is defined in activity a, and a tick occurs, the timer t_σ is decremented.

State transition function defines the transitions based on events and defined as

$$\delta : Q \times \Sigma \to Q$$

From any state q, transition is only possible if there is an event (σ) enabled. Let, transition be defined as $\delta(q, \sigma) = q'$, where q and q' are,

$$q = (a, \{t_\alpha | \alpha \in \Sigma_{act}\})$$
$$q' = (a', \{t'_\alpha | \alpha \in \Sigma_{act}\})$$

An event σ is said to be enabled at q if $\delta_{act}(a, \sigma)!$ and it is said to be eligible if $\delta(q, \sigma)!$, as per time bound conditions. Events which are enabled but not eligible are known as *pending* events and only eligible ones can occur.

There are only three possible scenarios for σ to occur. Firstly, that event is tick and at q there is no deadline of any prospective event; secondly it is a prospective event (i.e. activity is defined and timer t_σ is within range of its bounds); lastly σ is remote event (i.e. activity is defined and timer is equals to 0).

σ = tick, $\quad \delta(a, \alpha)! \quad$ and $\quad t_\alpha > 0$, where $(\forall \alpha \in \Sigma_{spe})$

$\sigma \in \Sigma_{spe}, \quad \delta(a, \sigma)! \quad$ and $\quad 0 \leq t_\alpha \leq (\mu_\sigma - l_\sigma)$

$\sigma \in \Sigma_{rem}, \quad \delta(a, \sigma)! \quad$ and $\quad t_\alpha = 0$

Tick event occurs when no prospective event is imminent. With the occurrence of each tick, timer (t_σ) of events is altered, but activities do not change. When an event occurs, its timer

is reset to default value immediately. If it is enabled, then t_σ decreases by one with each tick. Ultimately σ will occur or its timer reaches 0 or it is disabled due to the occurrence of another event. In case if an event occurs, t_σ will reset or it waits to be enabled again from another state.

Any prospective event must not be delayed more than $(\mu_\sigma - l_\sigma)$ ticks from the moment when it gets enabled, and it can never become eligible before l_σ ticks. When the timer reaches 0, it must occur unless (only) preempted by another eligible event. While remote event (σ) can occur any time provided that l_σ ticks for t_σ has elapsed. Its timer will reset (if it does not occur from an enabled state) if another transition occurs.

So far, we have considered the event from source state (q) reference and explained how timers in q are affected. Now let us discuss the transition from the perspective of the target state (q'). Assume at this moment transition with σ has occurred. There are two possibilities either activity is same (i.e. $\sigma = tick$) or it is altered when $\sigma \in \Sigma_{act}$.

In the case of tick event $a' = a$, and only timer values are updated. If an event (σ) is prospective and activity transition is defined, then the timer will be decremented by one otherwise resets to μ_σ (default value). In case of a remote event, if the transition is defined and also its timer value is non zero (l_α is not reached), then the timer is decremented by one. Nevertheless, if the lower bound condition is already met, then the value will remain 0 until either it is executed or disabled. If the transition is not defined from that state, then the timer value will set to the default value (l_α).

The second case is when the executed event is non-tick ($\sigma \in \Sigma_{act}$) and consequently, activity has changed. If α is the executed event ($\alpha = \sigma$), then timers are updated to respective default values for both prospective and remote events. In case when the executed event is not α, timer value remains the same only if activity α from a' is defined; otherwise, it is set to the default value. These unchanged timer values signify the fact that non-tick events are instantaneous.

Formally we can express the above rules as follows.

(1) $\sigma = tick$. Then $a' = a$

$$t'_\alpha = \begin{cases} \begin{cases} t_\alpha - 1 & \text{if} \quad \delta_{act}(a,\alpha)!\ and\ t_\alpha > 0 \\ \mu_\alpha & \text{if not} \quad \delta_{act}(a,\alpha)! \end{cases} & \text{if} \quad \alpha \in \Sigma_{spe} \\ \begin{cases} t_\alpha - 1 & \text{if} \quad \delta_{act}(a,\alpha)!\ and\ t_\alpha > 0 \\ 0 & \text{if} \quad \delta_{act}(a,\alpha)!\ and\ t_\alpha = 0 \\ l_\alpha & \text{if not} \quad \delta_{act}(a,\alpha)! \end{cases} & \text{if} \quad \alpha \in \Sigma_{rem} \end{cases}$$

(2) $\sigma \in \Sigma_{act}, \quad a' = \delta_{act}(a,\sigma)$

$$t'_\alpha = \begin{cases} \begin{cases} \mu_\alpha & \text{if} \quad \sigma \in \Sigma_{spe} \\ l_\alpha & \text{if} \quad \sigma \in \Sigma_{rem} \end{cases} & \text{if} \quad \alpha = \sigma \\ \begin{cases} t_\alpha & \text{if} \quad \delta_{act}(a,\alpha)! \\ \mu_\alpha & \text{if not} \quad \delta_{act}(a,\alpha)!\ and\ \sigma \in \Sigma_{spe} \\ l_\alpha & \text{if not} \quad \delta_{act}(a,\alpha)!\ and\ \sigma \in \Sigma_{rem} \end{cases} & \text{if} \quad \alpha \neq \sigma \end{cases}$$

Let us consider an example. Let $G_{act} = (A, \Sigma_{act}, \delta_{act}, a_o, A_m)$ with,

$A = A_m = 1$

$\Sigma_{act} = \{\alpha, \beta\}$

Events are $(\alpha, 1, 1), (\beta, 1, 2)$ and ATG is given as below.

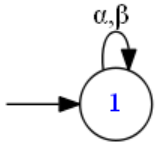


Figure 2.2: G_{act}

States of timed automaton are given as,

$$
\begin{aligned}
Q &= \{1\} \times T_\alpha \times T_\beta \\
&= \{1\} \times [0,1] \times [0,2] \\
&= \{(1,[1,2]),\ (1,[0,2]),\ (1,[1,1]),\ (1,[0,2]),\ (1,[0,0]),\ (1,[1,0])\} \\
|Q| &= 6
\end{aligned}
$$

Fig. 2.3 shows the transitions among state set Q and is called the Timed Transition Graph (TTG). Note that for ease of depiction, the states have been renamed as follows:

(1,[1,2]) 1

(1,[0,2]) 2

(1,[1,1]) 3

(1,[0,2]) 4

(1,[0,0]) 5

(1,[1,0]) 6

From Figure 2.3, we can see that at states 1 and 3 both events (α, β) are pending while at 2 and 5, they get eligible. Such states where event timer reaches 0, *tick* is preempted by the respective event and timer value is set to default value.

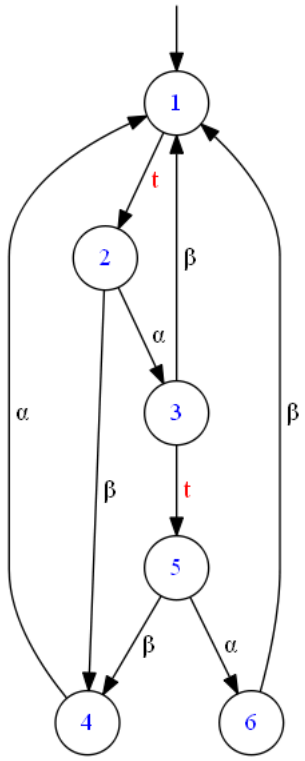


Figure 2.3: TTG of G_{act}

There is a possibility that an activity can preempt tick infinitely in TTG, and is known as activity-loop. For some $q \in Q$ and $s \in \Sigma_{act}^{+}$

$$\delta(q, s) = q$$

According to this scenario tick event will never occur due to preemption (time stops) which is physically impossible. Therefore we assume that all TTG must be activity-loop-free (alf).

2.3 Supervisory Control Theory

A supervisor controls any plant by monitoring the events generated by the plant and restricts its behavior according to the specification (also known as legal behavior). By assumption, the supervisor can only manipulate controllable events while uncontrollable can not be disabled. All events that are not intended to be disabled by the supervisor are modeled as an uncontrollable event, e.g., emergency plant shutdown, manual over-ride switch, and fault event.

Various strategies are available to implement the supervisory controller, which are explained in detail here.

Conventional Supervisory Control

Consider a plant modeled as DES denoted by G and a specification H. Suppose the events of this system are $\Sigma = \Sigma_c \cup \Sigma_{uc}$, where Σ_c and Σ_{uc} are controllable and uncontrollable events of the plant.

A supervisor (S), limits the plant behavior to legal behavior (2.4). Formally S is a function from language generated by plant to the power set of events.

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

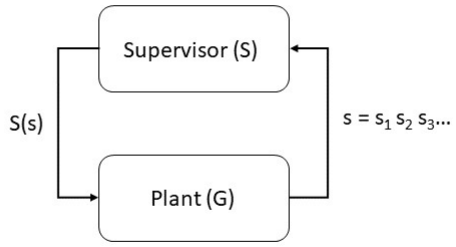


Figure 2.4: Closed Loop plant

Define a function, $\Gamma : x \rightarrow Pwr(\Sigma)$ to provide the active event-set feasible at any state. Supervisor (S) will disable events from active event-set only. However, it must not disable any active uncontrollable event because it has no control over Σ_{uc}. Such supervisors are called an admissible supervisor. In the above figure, s denotes the sequence generated by the plant and current state. The events permitted by the supervisor after disablement are represented as K(s). Enabled event-set from any current state of the plant is defined as,

$$S(s) \cap \Gamma(\eta(x, s)!) \tag{1}$$

According to admissibility condition,

$$\Sigma_{uc} \cap \Gamma(\eta(x, s)!) \subseteq S(s)$$

Only such events are executed which are enabled by supervisor, even if such event is present in current event-set of plant. Let K/G denote the plant under supervision of K.

Generated and marked language by S/G:

The language generated by plant under supervision is denoted as L(S/G). $L(S/G)$ is defined as follows:

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

(2) If $s \in L(S/G)$ and $s\sigma \in L(G)$ and $\sigma \in S(s)$, then $s\sigma \in L(S/G)$

Hence, according to definition $L(S/G)$ is prefix closed language and $L(S/G) \subseteq L(G)$. The marked behavior of controlled plant is defined as

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

Problem: Supervisory Control Problem

Consider a DES G with uncontrollable event set Σ_{uc} and legal marked behavior E with $E \neq \phi$ and $E \subseteq L_m(G)$. Find a supervisor such that

(1) $L_m(G) \subseteq E$ (safety property)

(2) $\overline{L_m}(S/G) = L(S/G)$ (Non blocking property)

Theorem 2.3.1 (discussed in the following) provides the set of following Supervisory Control Problem in terms of controllable and $L_m(G) - closed$ sub-languages. Before we present the theorem we review the definitions of controllable languages and $L_m(G) - closure$ property.

Definition 2.3.1 *A language K is controllable (with respect to a DES G) if*

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

The set of controllable sub-languages of a given language K is denoted $\mathcal{C}(K)$.
For any K, $\mathcal{C}(K)$ is nonempty and has a supremal element denoted by Sup$\mathcal{C}(K)$.

Definition 2.3.2 *A language $K \subseteq L_{m(G)}$ is $L_m(G) - closed$ if*

$$K = \overline{K} \cap L_m(G)$$

It can be shown that if K is $L_m(G) - closed$, Sup$\mathcal{C}(K)$ is also $L_m(G) - closed$.

Theorem 2.3.1 *Consider the Supervisory Control Problem. For every nonempty sub-language $K \subseteq E$ that is controllable and $L_m(G) - closed$, there is a solution S to the Supervisory Control Problem such that $L_m(S/G) = K$ and vice versa.*

It follows from the theorem that if E is $L_m(G) - closed$, Sup$\mathcal{C}(K)$ offers the minimally restrictive solution.

Limited Lookahead Policy based Supervisory Control

The supervisor can be built offline. However, as the plant size increases, the number of states will increase exponentially. This leads to high memory consumption during implementation. In order to mitigate the memory issue, a new methodology is introduced in [23]. It provides a formal approach to compute supervisory control on-the-fly i.e., to compute the supervisor in run-time. This method is called Limited Look-ahead Policy (LLP).

In LLP based supervisory control, the DES model is expanded as a tree till N-levels from current state and control action is generated based on the behavior of the expanded model. Parameter N can vary depending on multiple factors such as processing power, memory availability or plant depth. Once the N-level tree is expanded, traces in the expansion are assigned an attitude to define their behavior (legal or illegal). Two attitudes considered in[23] are: 1) "Optimistic" in which traces are assumed legal and 2) "Conservative" in which pending traces are assumed illegal. Afterward, control action is computed in a few steps as explained below.

Consider an automaton G over alphabet $\Sigma = \Sigma_{uc} \cup \Sigma_c$. Suppose a trace *s* has been executed during online control implementation and current state is *x*. The next steps are taken to compute Supervisory Control Command denoted by $\gamma^N(s)$

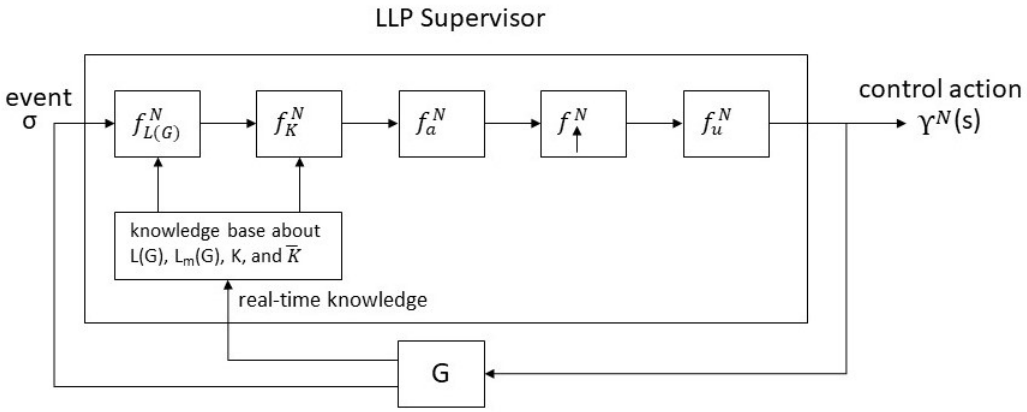


Figure 2.5: Block diagram of LLP supervisor

Step 1: From state *x*, G is expanded to N-levels beyond trace *s* to explore limited behavior into the future (block $f^N_{L(G)}$).

$$f^N_{L(G)}(s) = L(G)/s|_N, L_m(G)/s|_N$$

Step 2: Block f^N_K identifies and removes the illegal traces in N-level tree from result of first step.

$$f^N_K \circ f^N_{L(G)}(s) = (\overline{K}/s|_N, K/s|_N)$$

Step 3: Remaining traces in N-level tree are known as *pending traces*. These traces may have an uncontrollable event which can either lead the system to illegal zone or create blocking. As supervisor has to deal with this ambiguity two attitudes are introduced.

- Optimistic attitude: All pending traces are considered marked and legal
- Conservative attitude: All pending traces are considered illegal.

Block f_a^N adopts one of the attitude regarding pending traces.

$$f_a^N \circ f_K^N \circ f_{L(G)}^N(s) = \begin{cases} \text{optimistic:} & K/s|_N \\ \text{conservative:} & K/s|_{N-1} \end{cases}$$

Step 4: The block $f_\uparrow^N$ computes the supremal controllable sub-language of result of f_a^N w.r.t. Σ_{uc}.

$$f^N(s) = f_\uparrow^N \circ f_a^N \circ f_K^N \circ f_{L(G)}^N(s) = [f_a^N \circ f_K^N \circ f_{L(G)}^N(s)]^{\uparrow/s|_N}$$

Step 5: The final step is to compute the control action by limiting the result of $f_\uparrow^N$ to level one only. Afterwards the set of uncontrollable events after trace s is added to it in order to avoid the disablement of uncontrollable event(s).

$$\gamma^N(s) = f_u^N \circ f_\uparrow^N = \overline{f^N|_1} \cup \overline{\Sigma_{uc} \cap \Sigma_{L(G)}(s)}$$

$\Sigma_{L(G)}(s)$ is defined as the active set after trace s: $\Sigma_{L(G)}(s) = \{\sigma \in \Sigma : s\sigma \in L(G)\}$.

For the implementation of above discussed algorithm the following two notions have to be examined, *validity* and *Run-Time Error*.

Definition 2.3.3 ***Validity:*** *Any control policy, γ^N of LLP supervisor is said to be valid if*

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

This notion will make sure that result of computed control policy is same as offline supervisor and no control action will lead the plant to illegal region due to an uncontrollable event.

Definition 2.3.4 ***Run-Time Error (RTE):*** *During LLP computation if $f^N(s) \neq \phi$ for any trace, $s \in L(G, \gamma^N)$, then it is said to be RTE. In special case if $s = \epsilon$ then, RTE is known as starting-error (SE).*

Proposition 2.3.1 *For any valid control action γ^N the following statements are true.*

(i) $(\forall s \in L(G, \gamma^N))\ \ \gamma^N(s) = \overline{K^{\uparrow}}/s_1$

(ii) $K^{\uparrow} \neq \phi$ *and* $(\forall s \in \overline{K^{\uparrow}})\ \ \gamma^N(s) = \overline{K^{\uparrow}}/s_1$

We know at in function block f_a^N one of the two attitudes are adopted regarding the pending traces to compute control action. An important question is that is there a minimum value for N that can guarantee validity and RTE.

Optimistic Attitude

In this policy all pending traces are assumed to be legal and marked to provide freedom (which can have some events, in controllable action, leading to illegal zone or blocking state). Therefore one would have to look far enough into the model, choose large N, to have valid behavior. Two cases are considered.

Case 1: $K = \overline{K}$

Denote the longest trace of uncontrollable events in language L by $N_u(L)$.

$$N_u(L) = \begin{cases} max|s| \text{ such that } s \in \Sigma_{uc}^* \text{ and } (\exists u, v \in \Sigma^*) usv \in L & if\ exists \\ undefined & if\ not \end{cases}$$

Theorem 2.3.2 *For $K = \overline{K}$, if $N \geq N_u(K) + 2$ or $N \geq N_u(L(G)) + 1$, then computed control action is valid. $L(G, \gamma_{opt}^N = K^{\uparrow})$*

Case 2: $K \subseteq \overline{K}$

In case if K is not closed, another constraint for non-blocking is required to be respected while finding minimum bound. Therefore one has to explore (more than the case of closed-K) at-least till the boundary for illegal region, known as *frontier*.

Definition 2.3.5 *Let K_{mc} be the language that contains all traces lead to marked states with controllable events only.*

$$K_{mc} = \{s \in K \textit{ such that } (\forall \sigma \in \Sigma_{uc})\ s\sigma \notin L(G)\}$$

Definition 2.3.6 *Let $K_{f\bar{c}}$ be the language that contains traces leading to illegal zone, $L(G) - \overline{K}$, due to uncontrollable event.*

$$K_{f\bar{c}} = ((L(G) - \overline{K})/\Sigma_{uc}) \cap \overline{K}$$

In optimistic policy, supervisor will continue until it foresees traces leading to illegal zone from marked states with only controllable events (σ_c), which will cause RTE by disabling σ_c. Therefore minimum bound must be large enough that supervisor can see traces leading to $K_{f\bar{c}}$ from K_{mc} in N-level expanded.

$$N_{mcf\bar{c}} = \begin{cases} max\{|t| : \ \ (\exists s \in K_{mc} \cup \epsilon)\ \ [st \in K_{f\bar{c}} \wedge (\forall \epsilon < v < t) sv \notin K_{f\bar{c}} \cup K_{mc}]\} & if exists \\ undefined & otherwise \end{cases}$$

Theorem 2.3.3 *Let $K^{\uparrow} \neq \phi$, if $N \geq N_{mcf\bar{c}} + 1$ then $L(G, \gamma_{opt}) = \overline{K^{\uparrow}}$*

Conservative Attitude

In this policy the worst case scenario is assumed i.e. all pending traces are illegal or may lead to blocking. Consequently, close-loop behavior is more restrictive than conventional supervisor, hence $L(G, \gamma_{con}^{N}) \subseteq \overline{K^{\uparrow}}$. The the following two cases of K are required to be analyzed for computing N_{min}.

Case 1: $K = \overline{K}$

Unlike optimistic policy, conservative attitude have no relation with RTE but minimum bounds for validity can be computed by deriving closed language of $L(G, \gamma_{con}^{N})$, in case of no SE.

Theorem 2.3.4 *For closed K, if there is no SE in close-loop behavior,* $L(G, \gamma_{con}^{N})$ *and* $K \cap \Sigma_{uc}^{N-1} = \phi$ *then,*

$$L(G, \gamma_{con}^{N}) = (K - (K/\Sigma_{uc}^{N-1})\Sigma^{*})^{\uparrow}$$

The following corollary is inferred from above theorem.

Corollary 2.3.1 *For closed K, if there is no SE in* $L(G, \gamma_{con}^{N})$ *and if* $N \geq N_u(K) + 2$*, then* $L(G, \gamma_{con}^{N}) = K^{\uparrow}$.

According to Corollary 2.3.1, complete K is required but if its not available once can use this minimum bound constraint, $N \geq N_u(L) + 2$, as $K \subseteq L$. Also from Theorem 2.3.2, $N \geq N_u(L) + 1$ which is smaller than the above result. Hence in case of closed-K, one can use $N \geq N_u(L) + 2$, valid for both attitudes.

Case 2: $K \subseteq \overline{K}$

As all pending traces are immediately invalidated by supervisor, it is required to check the marking and legality of upcoming traces in N-tree till marked state with only controllable events. This condition is necessary so that supervisor can disable controllable event prior to execution of illegal trace due to any uncontrollable event.

Definition 2.3.7 *N_{mcmc} is window size from current or initial state, with only controllable events, to the state with same property.*

$$N_{mcmc} = \begin{cases} max\{|t| : \ (\exists s \in K_{mc} \cup \{\epsilon\}) \ [st \in K_{mc} \wedge (\forall \epsilon < p < t) \ sp \notin K_{mc}]\} & if exists \\ undefined & otherwise \end{cases}$$

Theorem 2.3.5 *Assuming $\overline{K} = \overline{K_{mc}}$ and there is no SE in $L(G, \gamma^N_{con})$, if $N \geq N_{mcmc} + 1$ then $L(G, \gamma^N_{con}) = \overline{K^{\uparrow}}$*

If the supremal controllable sub-language K is not empty and $\overline{K} = \overline{K_{mc}}$, all marked states of supervisor have only controllable events, then $N_{mcmc} \geq N_{mcf\bar{c}}$ can be proven. As a result minimum bound from Theorem2.3.5 will also compute valid supervisor by adopting optimistic policy.

State based Limited Look-ahead Policy

In [23], the plant model has explored N events into the future from the current state and the expanded model is used to compute the supervisory command, but no information of model state is used during computation, therefore, we call this as event-based LLP. This computation gets complicated if there is a loop uncontrollable event because it will make the minimum look-ahead window infinity (unbounded). Furthermore, it is vital to explore plant at least until minimum window to guarantee the validity of computed supervisory

command.

In [25], information of the model's state is used, which bounds the minimum window length, as the maximum number of states can never be larger than the total number of states of the model.

Definition 2.3.8 *Let the set of marked states with only controllable events be denoted by* X_{mc}*. Then*

$$X_{mc} = \{x \in X_m \textit{ such that } \Sigma_{G(x)} \subseteq \Sigma_c\}$$

Definition 2.3.9 *Let* $\omega(x, s)$ *denotes the state set from* x*, following the sequence* s *such that none of the forth-coming states belongs to* X_{mc} *and illegal states.*

$$\omega(x,s) = \begin{cases} \{\delta(x,t) : t \leq s\} & if \ \ (\forall t \leq s)(\delta(x,t) \notin X_{mc} \wedge \delta(x,t) \in X_H) \\ \phi & otherwise \end{cases}$$

In above equation, X_H is the set of legal states (in specification automaton H). If H is not the sub-automaton of G then one need to transform it such that $L_m(H) = K$, using procedure provided in [33]. Once the H is transformed then minimum bound for the validity of supervisor is defined as N_B.

$$N_B = \max_{x \in X, s \in L(G)/[x]} |\omega(x,s)|$$

Chapter 3

Research Objectives

We discussed the LLP algorithm for SCT implementation (in section 2.3). In LLP, only one valid control action for the next step is computed after the occurrence of each event in a plant. As a result, computation time is constrained, especially in embedded applications. Computing new commands are very computationally expensive in terms of processing and also sometimes there may not be enough time to compute the next command due to the occurrence of back-to-back events. To uproot this problem, an innovative idea is suggested in [26]; instead of computing only one command, supervisory commands are generated for multiple events which are buffered and can be used readily once required. Next computation is only triggered when buffer level reach to lower bound (user-defined). This strategy is called as LLP with buffering. As a result, all the supervisory command can be calculated in timely fashion.

In this chapter, first of all, we plan to discuss the theoretical framework of LLP with buffering in detail. Later, we will formulate the research objectives, of this thesis, based on the current knowledge of LLP with buffering. Afterwards in chapters 4 to 6, we will present a model-based approach to calculate the execution time of event sequences and develop C-code for the efficient implementation of LLP with buffering.

3.1 Review of Limited Look-ahead Policy with Buffering

In LLP [23], the proposed methodology computes the supervisor on-the-fly (as discussed thoroughly in section 2.3). In real-time implementations, computing supervisory commands in each iteration may not be feasible in some situations due to different constraints. For example, if multiple events occur consecutively, there may not be enough time to compute the next event. Waiting for the next control command to be computed may cause undesired behavior or unnecessary delay.

To uproot this problem, an innovative strategy is proposed in [26] to compute the supervisor. In this methodology, in addition to the next control command, the commands for the following few events are computed and buffered. This strategy is called Limited Look-ahead Policy with Buffering. Similar to LLP, the plant model is expanded for some window in future called limited look-ahead window of the length N_w but instead of expanding it for only the minimum length N_{min}, the plant is explored for an extra Δ events (Fig. 3.1).

This section contains a discussion for the selection of buffer sizes and conditions for the validity of the computed supervisor using LLP with buffering.

3.1.1 Minimum Look-ahead Window Size

During LLP computations, as the plant model is expanded to a certain depth called the look-ahead window. Therefore the main selection criterion of selecting minimum window length of generating supervisory command such that it is minimally restrictive. Look-ahead window size N_{min} depends on several factors, one of them is the length of uncontrollable strings in the plant.

Solar tracker plant (used in this thesis) has no state without any uncontrollable event and therefore, infinite controllable loops are available throughout the plant automaton. Due to this reason, the length of the maximum string of uncontrollable events ($N_u(L(G))$), discussed in section 2.3, becomes infinity. Consequently, methods used in [23] cannot be

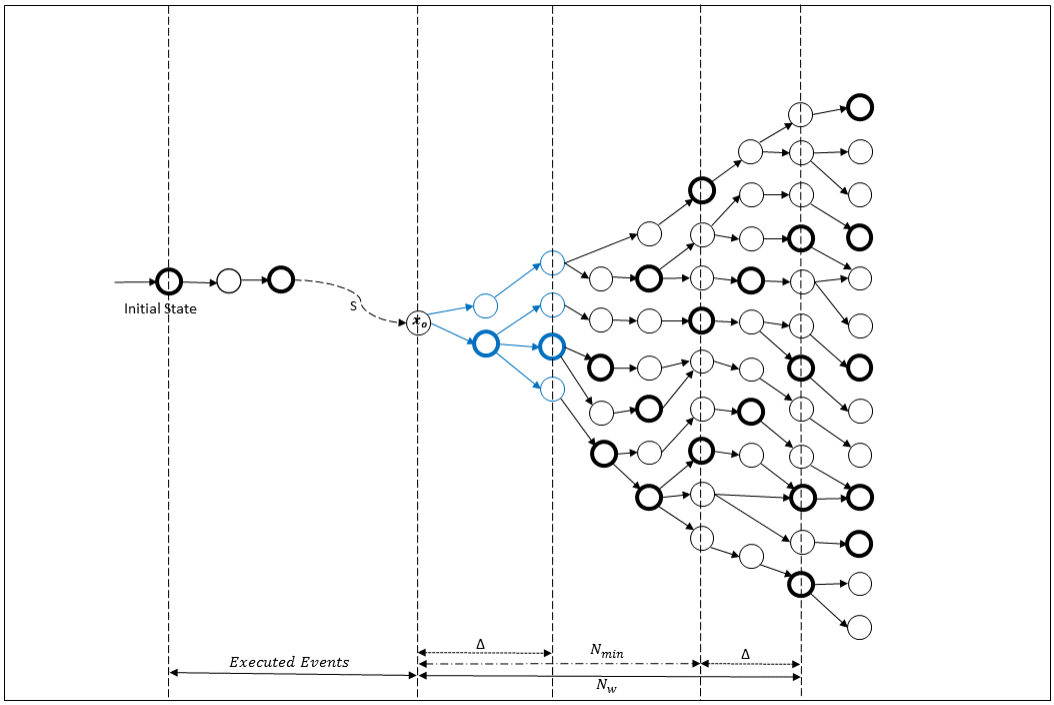


Figure 3.1: LLP with expanded window

used to compute minimum bound for solar tracker.

In section 2.3, we studied [25] in which a state-based approach is proposed to implement LLP. In this approach, a finite upper bound for N_{min} exists (and denoted by N_B). In [26], using the brute force method, N_{min} was found to be 6.

3.1.2 Supervisor Validity of LLP with Buffering

In Chapter 2, we defined an LLP supervisor to be valid if it is minimally restrictive (optimal). For an LLP supervisor to be valid, the size of the expansion window must be sufficiently large. We denote the minimum window size for LLP supervisor validity by N_{min} (Fig. 3.1). As mentioned in, 2.3, in state-based LLP, N_{min} is always bounded. In fact, $N_{min} \leq N$ where N is the number of states of the plant (if specifications are in the form of a subset of plant states).

In LLP with buffering, we wish to calculate the set of enabled events for present and up to

Δ events ($\Delta \geq 1$) in the future. In [26] it is shown that the minimum window size, in this case, will $N_{min} + \Delta$, and the solution of supervisory control problem for the plant model expanded by $N_w = N_{min} + \Delta$ events provides the control policy up to Δ events in the future.

Performing computation using a window of N_{min} guarantees the validity and optimality of the behavior of the supervisor for an immediate subsequent event. In the LLP with buffering, expansion size is increased by Δ to have control commands of Δ events beyond the current state and buffered for later use. In order to discuss the validity of buffered events, let us consider the following theorem.

Theorem 3.1.1 *[34] If $s \in \overline{K^{\uparrow}}$, then*

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

Here, $K^{\uparrow}$ is the $SupC(K)$ with respect to language L(G), generated by automaton G and Σ_{uc} (uncontrollable events). It states that post-language of supremal controllable sub-language ($K^{\uparrow}$) of any string (s) is equivalent to the supremal controllable sub-language of post-language of the string.

Let us consider an example in which a small portion of expanded plant tree is shown in Fig 3.2,

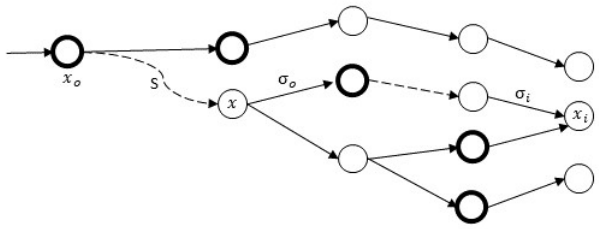


Figure 3.2: Sub-expanded Tree

Let s be a trace in plant G from initial state $x_\circ$ to current state x permitted by the optimal supervisor (s in Fig 3.2). Then according to **Theorem 3.1.1**, any event σ_i after i steps allowed by LLP supervisor, also belongs to supremal controllable sub-language of post-language of s.

Plant Depth

Plant Depth (PD) is defined as the minimum size of the look-ahead window size for which the entire plant model must be explored for the LLP window. PD depends on the plant model and specifications. One can use this parameter to measure the efficiency of LLP, e.g., if online computations are done with a window of N_w close to PD, then complete plant model is explored and the behavior of LLP is close to conventional supervisory (thus requiring long computation time and large memory). In this case, LLP is not efficient.
In other words, one needs to keep the ratio of the look-ahead window to plant depth as small as possible (to explore a small portion of the plant) and fully utilize the advantages of LLP computation. Therefore

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

are the measures of usefulness of LLP in a particular problem.

Plant depth of solar tracker is 11 and using minimum look-ahead window size to ensure validity of online LLP supervisor,

$$\frac{N_w}{PD} = \frac{6}{11} = 0.54$$

3.1.3 Choosing Buffer Size

Once events are buffered, one does not have to worry about rapidly executed events as no computation is required. During this free time, CPU can be used to compute the supervisory command for the next window, but there is time constraint as computation must be finished before the execution of previously buffered events. Otherwise, there would be a delay which might have undesirable consequences. To find a suitable buffer size, two functions are defined.

Definition 3.1.1 *Minimum Execution Duration* $T_{min} : \mathbb{N} \rightarrow \mathbb{R}$
$T_{min}(n)$ *is a function which returns the minimum execution time of any sequence with n number of events.*

$$T_{min} = min\{\ T(s^{'}) \mid \exists s,\ ss^{'} \in L(G) \text{ and } |s^{'}| = n\}$$

where $T(s^{'})$ is the execution time of sequence $s^{'}$.

Definition 3.1.2 *Maximum Computation Delay* $C_{max} : \mathbb{N} \rightarrow \mathbb{R}$
$C_{max}(N_w)$ *provides the maximum time consumed during LLP computation over the window size of* N_w.

Due to hard time constraints, all computations must be done in a very timely manner, and control commands (events) must be readily available for the implementation, causing no unnecessary delay. The timeline of LLP with buffering for events is shown in Fig. 3.3.

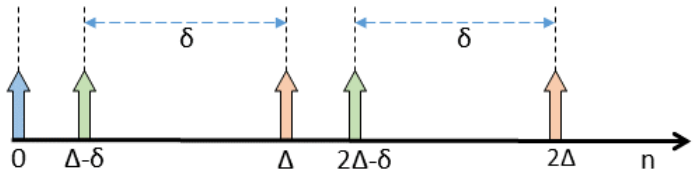


Figure 3.3: Timeline of LLP with buffering

The following explains the sequence of operations.

Step 1 As initial step (n=0), supervisory commands are computed for a window of $N_w = N_{min} + \Delta$. At this point we have Δ number of valid events in buffer.

Step 2 Let δ denotes the count of events left in buffer when new Δ supervisory commands are computed. Once $\Delta - \delta$ events pass, LLP computations of supervisor is done for a window of $N_w = N_{min} + \Delta + \delta$, generating events for interval $\Delta - \delta$ to 2Δ. The computations must be done in δ events (before n=Δ). Thus

$$C_{max}(N_w) \leq T_{min}(\delta) \quad and \quad \delta < \Delta$$

It signifies that LLP computation must be done prior to the execution of δ events (remaining in the buffer).

Step 3 After n=Δ, the newly computed supervisory commands between n=Δ up to n=2Δ will be used. New LLP computation begins at $2\Delta - \delta$, similar to step-2 followed by third step and so on.

In [26], LLP with buffering was implemented successfully in MATLAB on 2-DOF solar tracker (to be discussed in Chapter 5) for trajectory control.

In order to optimize the computation time of supervisor generation, MATLAB code was converted to C-code using $MATLAB^{®}\ Coder^{™}$. Consequently, no optimization is done to reduce memory usage, e.g., same data type (double) are used in C-code (MATLAB

generated) as that in MATLAB and a large number of variables are declared to perform the same task. The default data type of MATLAB is double (consumes 8bytes). In [26], the maximum number of transitions of the plant is 16800, which can be represented by an unsigned integer (consumes 2bytes), which immediately cuts down the memory usage by four times. Furthermore, machine-generated code can be optimized in terms of code execution (which can be achieved by the compiler) while hand-written code is optimized in terms of process. For example, memory to store automaton can be allocated dynamically i.e., allocate memory according to the requirement in run-time.

Improvement in computation time of supervisor is observed (later discussed in section 4.4) but still algorithms can be further improved by using modified search algorithm and few other customized features.

In [26] execution time of events is computed by extensive experiments. Therefore one needs to perform a real-time experiment to obtain function T_{min}.

3.2 Thesis Objectives

In this thesis, we will translate LLP supervisory control procedures to C-language for optimization and prepare them for implementation on embedded systems. C-language is a low-level language compared to MATLAB and therefore has more advantages in terms of memory and time complexity. Nevertheless, these benefits come at the cost of the programming complexities as one haS to manually manage memory at the hardware level. The aim is to reduce the execution time of the algorithm by customizing them and especially, to improve memory management during run-time (Details in section 4.2).

Previously, values for T_{min} function were obtained experimentally. In this thesis, we present a model-based approach to compute the T_{min} function via theoretical analysis using *Timed Discrete Event Systems (TDES)*. This methodology will eliminate the need for experimentation and be used with any plant to study the timed closed-loop system.

To study the implementation of LLP with command buffering, two degrees of freedom (2 DOF) solar-tracker will be used as a plant. It is equipped with a micro-controller for data acquisition and low-level control (motor control). Closed-loop timed solar tracker will be compared with experimental results for validation. Online supervisory will be implemented to maneuver the plant to find the brightest light spot for battery charging. This step will be used to explore the efficiency of LLP with buffering. This would help with a better understanding of the benefits and limitations of LLP with buffering.

Chapter 4

Development of Discrete Event Control Kit for Embedded Systems

In this chapter, we will discuss the development of DECK procedures in C-language with all required alterations. A model-based methodology is proposed to generate and analyze timed plants under the supervisor using the developed code and TTCT software. At the end of the chapter, we will perform the comparison of manually developed C-code, and MATLAB converted C-code.

A major purpose of this thesis is to translate LLP supervisory control procedures to C-language for embedded systems applications (i.e., computing supervisory command using minimal resources). Being a low-level language has critical advantages over other languages, as it provides more control over code and hardware (RAM, flash memory, etc.).

Let us consider an example of the spacecraft Perseverance (a MARS rover), which is scheduled to launch in 2020. The rover is equipped with single board radiation-hardened computer RAD750, possessing the processing power of up-to 200MHz with RAM of 256MB while desktop used for this thesis has 8GB RAM with quad Core-i5-2400 @3.10GHz processor. DES model of a solar plant used in this thesis (equipped with far fewer components as compared to the rover), has 16800 transitions and 1584 states, which requires 102KB

and 60KB for a supervisor to store on RAM and take 2.5 seconds for computing supervisor in MATLAB. Let us say, if the plant model has 100,000 states and 1,000,000 transitions, then 5.91MB will be required. One can observe the exponential trend of memory requirement. Hence, it is not feasible to perform the computations for supervisory controls using onboard resources. To tackle this problem, DECK procedures are written from scratch in C language to optimize LLP algorithms with respect to embedded applications.

A limited look-ahead policy with buffering was suggested and implemented successfully using the MATLAB generated C code in [26]. In this strategy, a complete plant is not required to generate a supervisor. Consequently, time and memory complexities are decreased. In order to do further optimizations, only main procedures for LLP with buffering are develop in C along-with a few other assisting functions, discussed in detail in section 4.2.

4.1 Structures and Data Types for DECK Procedure

In this section, we will define structures that are required during the development of functions. All required structures and data types are defined in header file "structs.h".

An unsigned integer *deck_int* is defined and used throughout the algorithms because the number of states and transitions will never be negative.

$typedef\ uint16\_t$ ***deck_int***;

Size of *deck_int* can be changed to 8/16/32/64 bit unsigned integer, in "structs.h", according to the number of states or transitions of the automaton. Following structures are defined to manipulate DES models LLP algorithms.

Array

In C language, array of any data type needs special attention as C does not have any parameter to keep the track of its size. Array is handled using pointer which contains the

address of a first element only. Consequently, when a pointer is passed as argument, we also need to send the size of the array, to avoid the illegal memory access (memory beyond the actual array size). Rather then using new variable every-time, an array structure (with two elements) is defined as follows.

typedef struct array{

$deck\_int \;\; size$;

$deck\_int \;\; element[max\_Array\_Size]$;}

max_Array_Size is the maximum number of elements. It can be customized, for this thesis it is fixed to 3000.

Automaton

Automaton structure is used to store DES model. It is defined as:

typedef struct automaton{

$deck\_int \;\; N$; (states)

$deck\_int \;\; TLs$; (transition list size)

$deck\_int \;\; Xms$; (marked states size)

$deck\_int \;\; *TL$; (pointer to transition list)

$deck\_int \;\; *Xm$;} (pointer to marked states)

Automaton's transition list and marked states are stored in 2D and 1D arrays, respectively. TL and Xm are the pointers containing the address of transitions and marked state arrays. To keep track of the array's size, separate variables (TLs and Xms) are used. Automaton's number of states is stored in a separate variable (N).

Memory for transition list and marked states is allocated dynamically to avoid wastage of

memory. When an automaton undergoes through an operation like trim, product, or reachable, it is size can change. For example, the resultant of trimming an automaton can reduce the number of transitions and states due to the removal of co-reachable states. Let us consider sync operation with the example of two automatons, A and B. Suppose A has ten transitions while B has 20. If we sync A and B, in the worst-case scenario where

$$\Sigma_A \cap \Sigma_B = \emptyset$$

the resultant automaton will have 200 states and if

$$\Sigma_A \cap \Sigma_B \neq \emptyset$$

then resultant automaton will have fewer transitions than 200. Thus the unused allocated memory will be wasted.

Sometimes, memory required to store an automaton exceeds the stack memory of the function, causing the stack overflow. To solve this issue and also to reduce memory complexity, *dynamic memory allocation* is used. In this method, memory is allocated (in run-time) on the heap memory instead of the stack. Following function is developed for the dynamic memory allocation,

automaton G = aut_alloc (G, tls, xms);

A downside of dynamic memory allocation is a problem of memory leakage. In a program, during the run-time, if memory is allocated dynamically and not freed after usage, it will remain allocated. Moreover, recursive calls to that function will continue allocating more memory, and eventually, the system will run out of RAM. This process is known as memory leakage. Hence, each dynamically allocated memory block must be freed manually after usage. For this purpose, the following function is developed.

init_automaton (&G);

Timed_Ev

Timed_Ev structure is defined to work with timed event (discussed in section 2.2). It consists of three components, event code, lower bound and upper bound of the event.

typedef struct Timied_Ev{

$deck\_int \;\; Event;$

$deck\_int \;\; LowerBound;$

$deck\_int \;\; UpperBound;\}$

Timed_Ev_array

Array of timed events is statically defined with size of 30 elements. However, it can be customized before executing program, according to the required number of events.

typedef struct Timed_Ev_array{

$deck\_int \;\; Size;$

$Timed\_Ev \;\; element[Max\_Timed\_Ev\_Array\_Size];\}$

4.2 DECK Procedures

All DECK [35] procedures are implemented in source file "deck_c.c" along-with the few customized supportive functions to ease the design of main algorithms and visualization. Procedures to compute a DES supervisor are,

- reach
- reachable
- trim
- product

- supcon

Generally, *Breadth First Search* graph traversal strategy is used to explores the transitions in an automaton.

Breadth First Search (BFS)

BFS is a common graph traversing algorithm [36]. It initializes from source state and explore all branches before moving to next node. Transitions from each node is explored only once and newly discovered nodes are explored in the same order as they are discovered.

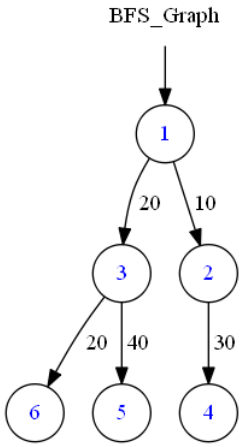


Figure 4.1: BFS Example

Consider Fig. 4.1. Let node 1 (layer 0) be the source state. From 1, BFS will explore node 3 then 2 in layer 1 and add them to the queue. Node 3 is the next in the queue to be explored, here nodes 6 and 5 are discovered and added to the queue, next to the node 2. Now, node 2 will be explored and process will continue until queue of unexplored states is empty or all the nodes are explored. The final order of the graph exploration using BFS is as follows.

$$1 \rightarrow 3 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 4$$

Binary Search Algorithm (BSA)

The Binary Search Algorithm is a very efficient way to find elements in a sorted 1D array of distinct elements. It uses the "divide and rule" concept. Suppose, there is an array (arr) of n elements and we have to find x in this array. Then

- Find teh mid point of array and compare it with x. If $x == arr[\frac{n-0}{2}]$, then return.
- If $x < arr[\frac{n-0}{2}]$ search in left side, i.e. from left (0) to mid ($\frac{n-0}{2}$).
- If $x > arr[\frac{n-0}{2}]$ search in right side, i.e. from mid ($\frac{n-0}{2}$) to end (n).

This process will continue until the element x is found, assuming x belongs to the array. It can be noticed that after each iteration the search horizon is halved. Therefore the time complexity of BSA is O(log n) i.e. if the size of an array increases exponentially, the time to find the element x will increase linearly.
Let us discuss the example of finding element 8 in the following array.

$$arr = [1, 2, 5, 6, 7, 8, 10]$$

The mid point of array is at index 4 so, $arr[4] = 6$, which is less than 8 therefore the left side is discarded. Current search index ranges from 4 to 7. Mid point is at 5 i.e. $arr[5] = 7 < 8$. So now the algorithm will leave right side of new portion and search in indexes from 5 to 7. Mid of this portion is at 6 and $arr[6] = 8$, the element is found. We can see that instead of 7 iterations element was found in 3 only.
To use this algorithm in this thesis two customizations are done,

(1) Adjust it for array with repeating elements

(2) Adopt it for 2D array

In this thesis, transition lists are stored in 1D arrays mapped from a 2D array. For example, transitions G.TL,

$$G.TL = \begin{bmatrix} 1 & 10 & 3 \\ 1 & 20 & 4 \\ 2 & 10 & 1 \end{bmatrix}$$

are stored as,

$$G.TL = [1, 10, 3, 1, 20, 4, 2, 10, 1]$$

Therefore as 1D array it seems unsorted but as 2D it is sorted in first column and also repetitive, with respect to source state (two transitions from state 1). After changes this BSA was ready to use in DECK procedure.

Reach

This procedure computes the reachable states in an automaton from an array of source states.

$reach(G, Sr, s)$;

Inputs:

G is the automaton to be explored.

Sr is pointer to array of source states.

s is array size.

Output:

It does not return anything but stores reachable states in global array Xr.

Global variables are useful, when the data is required to share among multiple functions frequently. Therefore, instead of using the variables as an argument (with limited scope) to the function, they are defined for the global scope.

Reachable

This function returns the reachable sub-automaton of an input automaton.

$Gr = reachable(G);$

Inputs:

G automaton.

Output:

Gr is the reachable automaton.

Trim

This function returns sub-automaton with transitions to or from only those states which are reachable and co-reachable. It also stores the co-reachable states in global array Xrcr.

$Gt = trim(G)$

Inputs:

G automaton.

Output:

Gt is trimmed automaton.

Product

This functions take two automatons as input. The output automaton is the product of both input automatons. To implement a limited look-ahead policy, we need to take the product up until a certain depth of automatons. Therefore general Deck product algorithm is modified by introducing limitations in depth during automaton traversing. This is controlled by a signed integer (nw) as an argument.

In the previous version of this procedure, the memory of input automatons remained allocated. In order to optimize memory, another feature is added in the function which frees the memory of input automatons, if required, using the boolean argument (memFree).

All states explored during the execution of product are stored in $prod\_states[max\_TLr][3]$, global 2D array separately for troubleshooting. These states can be displayed using flag "Inter_Prod_State". The maximum size of this array can be changed; for thesis $max\_TLr = 30000$.

$$Gp = product\_LLP(nw, G, H, memFree); \begin{cases} default\ product & nw < 0 \\ LLP\ product & nw \geq 0 \\ keep\ input\ memory & memFree = false \\ free\ input\ memory & memFree = true \end{cases}$$

In the case of large transition lists, it can be very time consuming to explore transitions through a transition matrix due to a large number of iterations for finding the required state. Hence customized BSA algorithm is used as explained in section 4.2.

Inputs:

G is first input.

H is second input.

nw is depth control parameter.

memFree is a flag to free memory.

Output:

Gp is the resultant automaton.

Supcon

Supcon algorithm [37] builds a minimally restrictive supervisor of a plant(G), specification(H) and uncontrollable events (Euc).

$K = supcon(G, H, Euc);$

Inputs:

G is plant automaton.

H is specification automaton.

Euc is array of uncontrollable events.

Output:

K is supervisor automaton

Selfloop_Ev

This function adds self loops of events to all states of an input automaton.

$Gs = selfloop\_Ev(G, Ev, Evs);$

Inputs:

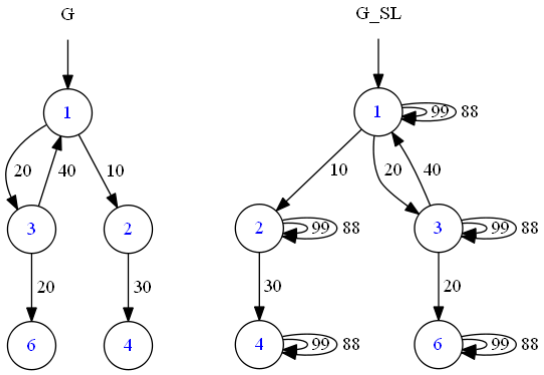
G is plant automaton.

Ev is array of events.

Evs is array size.

Output:

Gf is self-looped automaton.



(a) G (b) Gs

Figure 4.2: Self-looped Automaton G

In Fig 4.2 self-loops of events 88 and 99 are added using selfloop.

Sync

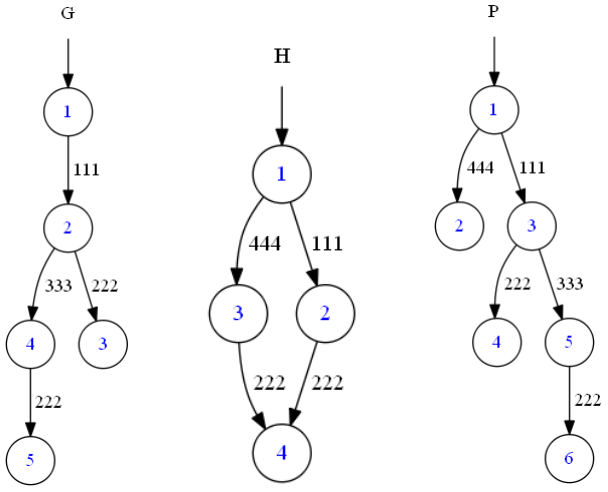
Sync is used for the synchronous product of automatons. It is typically used to build plant model from components and interactions. This function is not written in C because sync can be performed by executing already available procedures, explained here. For to automatons G and H, the following three steps yields the synchronous product of G and H.

- Self-loop G with events $\Sigma_H - \Sigma_G$
- Self-loop H with events $\Sigma_G - \Sigma_H$
- Product of self-looped G and H

Let us consider and example in Fig. 4.3. The event sets are,

$\Sigma_G = \{111, 222, 333\}$
$\Sigma_H = \{111, 222, 444\}$

According to the procedure firstly ,G is self-looped with event $\Sigma_H - \Sigma_G = \{444\}$ and H is self-looped with $\Sigma_G - \Sigma_H = \{333\}$. The resulting automatons are not shown here for brevity. Then product function is used for the resultant automatons and final automaton P in Fig 4.3(c) which is the sync of G and H. For simplicity all states are consider marked in this example.



(a) G (b) H (c) sync of G,H

Figure 4.3: Sync Procedure

It is not necessary to write a separate function for the sync procedure.

4.3 Automaton I/O Functions

Other functions have been written to integrate DECK-C with TTCT software and MATLAB in addition to the mentioned DECK procedures. These functions explained in detail in the following section.

Draw_automaton

In order to represent automaton graphically, an open-source software Graphviz [38] is used. It is visualization software that represents structure information in the form of a network or graph. This software can be used as a library or standalone software in different operating systems (Windows, Linux and Mac etc.).

Using *Draw_automaton*, automaton is written in text file according to Graphviz file format. Once the file is created, gvedit.exe application is executed and graphical automaton is drawn by opening the generated file and executing *run* command.

$draw\_automaton(G, filename, ShowName)$;

Inputs:

G is plant automaton.

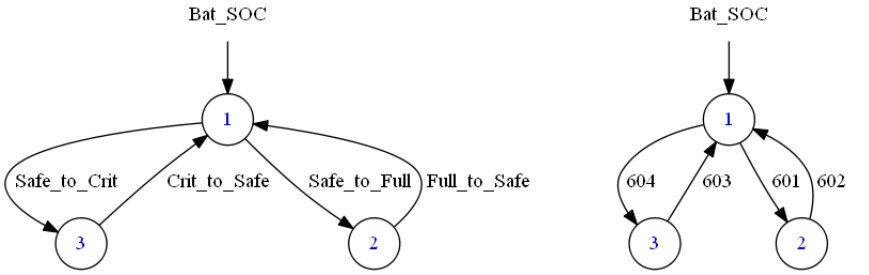
filename is a character string.

ShowName is a flag to display event name instead of code.

Output:

Gvedit readable ASCII file named "*filename*".

In order to display a label for each event instead of a numeric code third argument can be set to true (otherwise false). The labels of event codes must be defined in $EvCode\_to\_EvName(event)$, defined in "deck_c.c", prior to using this function. It takes event code as input and stores event name in global variable *EvName*. Fig. 4.4 is an example by Graphviz.



(a) Automaton with name

(b) Automaton with code

Figure 4.4: Graphical representation of automatons

Aut_to_file

The purpose of aut_to_file is to write automaton information on a file. One can also use this function to manually write an automaton. For this, empty automaton will be given as input to this function and the parameters of automatons will be updated using simple text editor.

$aut\_to\_file(filename, G);$

Inputs:

filename is character string.

G is plant automaton.

Output:

Automaton file named "*filename*".

For this thesis, a simple text file is used, but it's format can be changed in the function.

File_to_aut

This function reads automaton and stores it in program memory. One can also use this function to import automatons from other software e.g., MATLAB, provided that the file data format is in accordance with the readable data format.

$G = file\_to\_aut(filename);$

Inputs:

filename is character string.

Output:

G is automaton.

Aut_to_TTCT_AAS

TTCT software only reads the files of format $.AAS$, tAut_to_TTCT_AAS writes an automaton in this specific format. The purpose of this function is to export any un-timed automaton, with its events time bounds, from DECK-C to TTCT.

$aut\_to\_TTCT\_AAS(filename, G, timed\_Ev\_array);$

Inputs:

filename is character string.

G is automaton.

timed_Ev_array is array of timed events.

Output:

$filename.AAS$

Before using this function, it is important to convert the events and time bounds of input automaton according to the TTCT format. The following are the rules used in TTCT for automatons,

(1) Event code range is 1 to 999

(2) Odd event code represents controllable events

(3) Even event code represents uncontrollable events

(4) Lower and upper timed bound range is 0 to 1000

(5) 1000 time bound is interpreted as *infinity*.

Once changes are done in automaton, the above function can be used to generate TTCT compatible file.

TTCT_PDS_to_aut

In TTCT, once a timed automaton is computed, it is written in file a .PDS file. TTCT_PDS_to_aut can be used to read timed automaton and store it as automaton in program's memory, event code zero (0) is fixed for tick. The filename of a timed automaton file generated by TTCT software is an input to this function.

$G = TTCT\_to\_PDS\_to\_aut(filename);$

Inputs:

filename is character string.

Output:

G is timed automaton.

4.4 Time Complexity Comparison

In this section, a comparison of DECK procedures, written in different languages is discussed. Any function to be analyzed is compared based on execution time of computation for three sizes of automatons on the same machine. Programs are written in MATLAB, MATLAB generated C code and C-language (from scratch).
The computer used for experiments has following specifications,

- Intel $Core^{TM}$ i5-2400
- Max Clock speed: 3.10 GHz
- RAM: 8 GB

MATLAB Coder tool is used to convert MATLAB code to C language, and after conversion, the generated program, in the form of MEX file, is either executed in MATLAB as a function or exported to C compiler to be executed as standard C code. However, as the code is converted by machine, it is not as optimized as the manually written C-code. This can be validated with the results of the following analysis.
Execution time values in the following tables are the average of 10 to 200 tests.

Reach

Reach function takes an automaton, traverse its graph using BFS, and stores reachable states in an array.

Machine: *Intel* $Core^{TM}$ *i5-2400*					
Automaton Size		**Code Language**			**Time (ms)**
States	**Transitions**	**MATLAB**	**MATLAB generated C**	**Manual C-Code**	
42	101	0.35	0.085	0.0047	
323	1495	5.061	5.513	0.468	

Table 4.1: Comparison of execution times

Reachable

This procedure generates the reachable sub-automaton. In Table 4.2, automatons of three different sizes are used as input arguments. It can be seen that when transitions are low, MEX format is twice fast as MATLAB. In contrast, when the model size increases to thousands, the performance of mex functions degrades and becomes almost identical to MATLAB. However, there is a significant difference when manually written C code is used; even when the automaton's transitions increase significantly, C code is still quite efficient: fourteen times faster than MATLAB function.

Machine: *Intel* $Core^{TM}$ *i5-2400*					
Automaton Size		**Code Language**			**Time (ms)**
States	**Transitions**	**MATLAB**	**MATLAB generated C**	**Manual C-Code**	
42	101	0.500	0.102	0.0062	
323	1495	7.028	6.114	0.535	
969	6809	101.75	97.97	7.394	

Table 4.2: Comparison of execution times for Reachable.

Product

Here is the comparison of product procedures with various automaton sizes. This is one of the expensive algorithms in the DECK as both automatons need to be traversed concurrently using the BFS search. Moreover, an increase in the number of transitions from the same state causes a significant change in the loop iterations, as the algorithm has to look for source state in the transition list in each iteration. Modified BSA is used to minimize the number of iterations during the exploration of the product.

Machine: *Intel $Core^{TM}$ i5-2400*					
Automaton Size (G,H)		**Code Language**			**Time (ms)**
States	**Transitions**	**MATLAB**	**MATLAB generated C**	**Manual C-Code**	
134,9	471,50	3.29	1.18	0.059	
323,130	1495,974	13.84	4.677	0.206	
969,355	6809,3362	132.06	55.78	3.875	

Table 4.3: Comparison of execution time of Product.

In Table 4.3, the first two columns contains the number of states and transitions of the input automatons (G and H) of product function, separated by comma. It can be seen that MATLAB generated C code is on average three times faster than MATALB, while C code written from scratch is at-least 14 times faster then MATLAB generated C code. It is interesting to see that as the automaton size increases, the time difference between MATLAB and C code decreases but even when input automatons have transitions of the order of thousands, C code is 34 times faster.

To sum up, manually written C-code has a significant advantage over the other programs. This efficiency is due to run-time memory management of dynamic allocations, the use of optimized data types, and a modified search algorithm instead of a brute-force approach.

Chapter 5

Experimental Setup and Supervisory Design Setup

In this chapter, we will discuss the hardware architecture of the plant. In section 2, the solar tracker is modeled as untimed DES, followed by the supervisor synthesis based on the provided specifications.

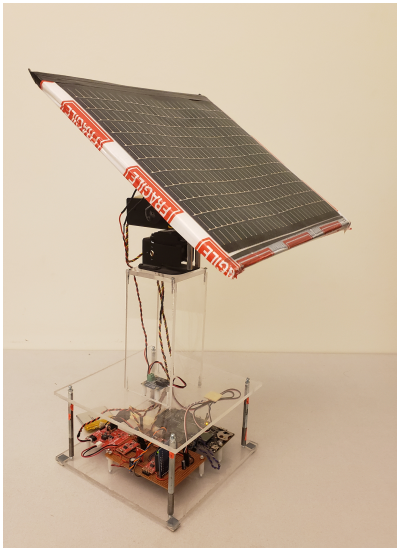


Figure 5.1: Solar tracker as Remote station

5.1 System Hardware

The primary purpose of this thesis is to improve the implementation of DES-based supervisory controls. Here first, the hardware setup is briefly explained.
Complete setup is comprised of two main systems:

(1) Remote Station and

(2) Ground Station

Fig. 5.2 depicts the schematic diagram.

5.1.1 Remote Station

Generally, any system controlled from a remote location using wireless communication is known as *Remote Station*.
In this thesis, a solar tracker plant [20] is selected as a remote station. It is equipped with a PV panel, two servo motors for maneuvering providing two degrees of freedom in motion (2 DOF), a micro-controller, a battery, and a communication module.
The solar tracker is meant to find the brightest light source in its surrounding to charge the batteries with a PV panel. It follows the user-defined trajectory (modeled as specification and will be discussed later in this chapter) to find the light source.

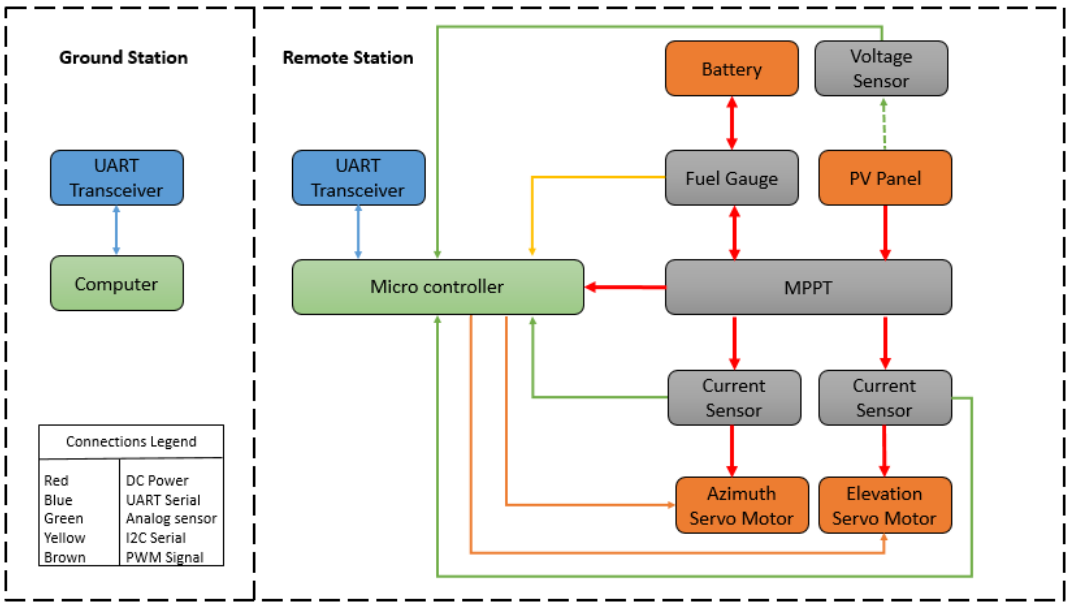


Figure 5.2: System Hardware Architecture

PV panel

The PV panel converts light to electrical energy, which is used to charge the battery. For this thesis Flexible Solar Cell is used. It is in the form of a flexible sheet; this avoids damage. The maximum power that can be generated is 3.08 watts at 15.4 Volts and 200mA. To harvest maximal energy, the Maximum Power Point Tracker (MPPT) is used to regulate the voltage level to generate full power and store in the battery. MPPT uses a DC-DC converter for the generation of approximately 3W continuously and keeps the battery charging volt at the optimal level. For this purpose SparkFun Sunny Buddy is used, which is capable of seamless power flow due to the parallel connection of solar panel and single-cell lithium-ion polymer (LiPo) battery. It can work with up to an input of 20 V, which is higher than the maximum solar panel output voltage (15.4V).

Precision Voltage Sensor is used to keep track of the solar panel voltage. It can measure $\pm$30V, and output is in the range of 0 to 5V.

Servo Motors

Two servo motors are used to control motion in both azimuth and elevation directions of the solar panel. Both motors are capable of moving clockwise (CW) and counterclockwise(CCW).

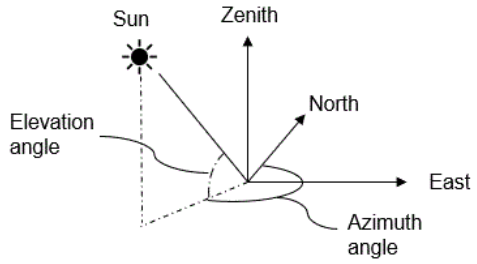


Figure 5.3: Azimuth and Elevation Angle

Azimuth and elevation coordinates are used to measure a celestial body's position in the sky from a particular point of reference. As shown in figure 5.3, zenith is the vertical axis at the location of the observer, azimuth is the clockwise angle between the projected object on observer plane and north, while elevation is the vertical angle between observer plane and object. In this setup, the origin point is the midpoint of the solar plane, and the sun will be the source of light.

HS-645MG and HS-805MG are the motors for azimuth and elevation motion with the ranges of 0° to 180° and 0° to 90° respectively. The latter has higher torque than the former because it has to move the panel against gravity. For this thesis, the initial conditions of angle set to be as 90° and 45° for azimuth and elevation motor, respectively. Then maneuver constraints are defined with respect to the physical constraints of motors, as:

- Azimuth motor range: $-90^\circ \leq \theta_{AZ} \leq 90^\circ$
- Elevation motor range: $-45^\circ \leq \theta_{EL} \leq 45^\circ$

These motors have high speed, specifically taking 0.20 seconds and to 0.18 seconds to rotate 60° by azimuth and elevation motors. Hence it is necessary to limit there speed for safe operation. The following two safety constraints are imposed on both motors to avoid physical damages:

- Single step of rotation = 2°
- 2 sec wait time between two consecutive steps

These limitations will keep the maneuvers slow and safe. Motors are getting power from the battery while the control signal (PWM) comes from a micro-controller. Pulse Width Modulation (PWM) is a square wave whose on-time signifies the time for which the armature of the servo will be energized, and the internal feedback controller will keep the position locked. In order to rotate the motor PWM signal with different on-time is generated.

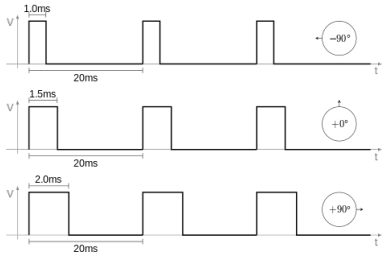


Figure 5.4: Pulse Width Modulation (PWM)

The health of motors is monitored by measuring current consumed continuously during operation. If the drawn current crosses the limit (stall current of motors 2500mA and 6000mA for azimuth and elevation motor), it means the motor is obstructed. SparkFun Current Sensor is interfaced with micro-controller for motor health monitoring.

Micro-Controller

The micro-controller is the brain of the remote station. It is responsible for all processing and communications. EFM32^{TM} Leopard Gecko micro-controller has 48MHz, 32bit

ARM Cortex M3 processor, 256KB flash memory, 32KB RAM and 12bit ADC. It is very suitable for low power applications due to its flexible energy modes.

Battery

A Lithium Polymer (Li-Po) is used as the primary energy source for the system. It has a nominal voltage of 3.7V and 2000 mAH current capacity. The PV panel charges this battery during operation. A fuel sensor is attached to measure the state of the battery charge. This sensor is interfaced with the micro-controller via the IIC communication protocol.

5.1.2 Ground Station

The ground station is responsible for computing and transmitting high-level supervisory commands based on the sensors' measurements received from the solar tracker.
Here, the ground station is a windows-based computer with a wireless transceiver for communication with the remote station. Hardware specifications of the computer are:

- Processor: Inter(R) Core(TM) i5-2400 CPU@ 3.10 GHz
- RAM: 8 GB

Communication module

For wireless communications, a pair of Digi XBee S1 802.15.4 modules are used. It is serial asynchronous communication, UART(Universal Asynchronous Receiver-Transmitter). The speed for this protocol can be customized; however, in this thesis, the maximum baud rate of 115,200 bps is used for high-speed communication, and packets are also encoded for transmission.

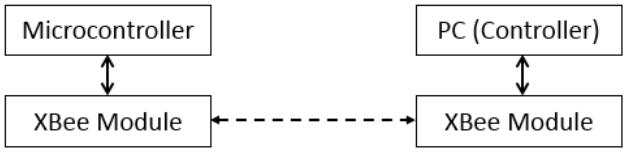


Figure 5.5: Serial Communication

5.2 Untimed DES Model and Supervisor Design

Initially, to start DES modeling, all basic components of plants are modeled (as discrete event systems in the form of the automaton); next, the component interactions are modeled. This step is very significant as it defines the dependency of plant components; even one interaction can lead to the plant's unwanted behavior. Afterward, specifications are defined; these are the set of behaviors expected from the plant. The last step is to define uncontrollable events, which the supervisor can never disable them. Sometimes, those events that are not wanted to be controlled by a supervisor are also modeled as uncontrollable e.g., an emergency shutdown of the plant. For the sake of simplicity, all states are marked.

5.2.1 Components

PV Panel

PV panel generates power based on the intensity and angle of the incident of light. Maximum energy is generated when the light source is normal to the surface of the PV panel. If the generated power is not enough, it will not be able to charge the battery. Therefore the followings are states that are selected for modeling:

1: Dark

2: Dim

3: Bright

Events are defined based on the panel voltage changes, as shown in Table 5.1. As the light source is an environmental factor that cannot be controlled, all related events are uncontrollable. In the table, e.g., ≥ 6 means voltage has passed the threshold 6V.

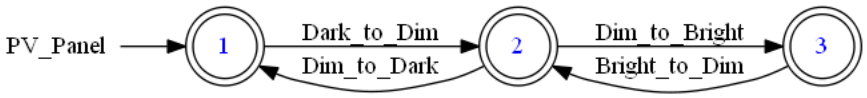


Figure 5.6: PV automaton

Event	Analog Voltage (V)	Controllability
Dark_to_Dim	≥ 6	Uncontrollable
Dim_to_Bright	≥ 16	Uncontrollable
Bright_to_Dim	≤ 15	Uncontrollable
Dim_to_Dark	≤ 5	Uncontrollable

Table 5.1: PV panel events based on voltage

PV panel voltages can be noisy, which may lead to undesirable event triggering. To prevent this, the thresholds of increasing and decreasing the voltages have been chosen to differ by 1V. Consequently, the event will only trigger when there is a difference of at least 1V. Moreover, PV can only charge the battery when it generates sufficient power, i.e. when it is in a bright state.

Battery State of Charge

Battery's charge is measured in percentage using a fuel gauge sensor interfaced with the micro-controller. The battery is modeled with the following three states.

1: Safe

2: Full

3: Critical

Each state is related to the percentage of battery charge (SOC). Hysteresis of 5% is included when defining the SOC limit for events to avoid undesired event triggers due to noisy data acquisition. The reason is that when servo motor is actuated, high current is drawn, which leads to a sudden drop in battery's SOC.

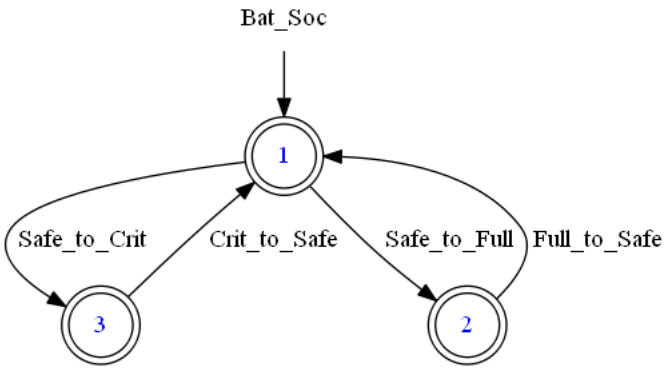


Figure 5.7: Battery SOC automaton

The charging state of the battery cannot be controlled; therefore, all relevant events uncontrollable. SOC thresholds for events are defined in Table 5.2. As mentioned before,

Event	Battery's SOC	Controllability
Crit_to_Safe	$\geq 55\%$	Uncontrollable
Safe_to_Full	$\geq 95\%$	Uncontrollable
Full_to_Safe	$\leq 90\%$	Uncontrollable
Safe_to_Crit	$\leq 50\%$	Uncontrollable

Table 5.2: Battery SOC events and percentage thresholds

servo motor motion requires significant current to keep the armature energized, which is not possible in a critical state due to insufficient charge. However, servo motor motion is possible in the other two states.

Azimuth Motor Motion

Azimuth motor can rotate in either direction clockwise(CW) or counter-clockwise(CCW) (provided that it is in within range, discussed in 5.2.1)). The motion of the azimuth motor has three distinct states:

1: Idle

2: Turning CW

3: Turning CCW

When the micro-controller issues command to move, the position of the servo changes according to the specified direction in a step of 2°. The motor is considered to remain in the state of motion until AZ_X_OK ($X = CW, CCW$). After each motor step, there is a short delay of 2sec for safety. Once the movement is done, the state of the model changes back to idle. A current sensor is used for feedback To measure the completion of motion. If the motor current decreases to less than 500mA, it signifies a successful motion.
Motor movement commands are controllable events and sent by the supervisor while the events defining successful motions are uncontrollable.

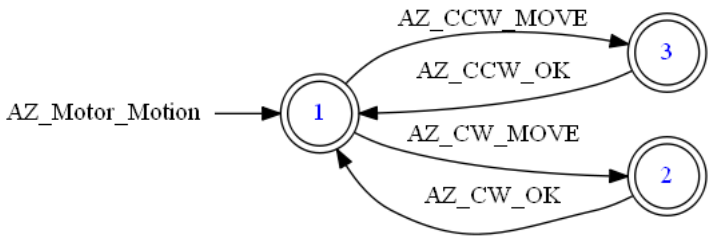


Figure 5.8: Azimuth Motor Motion automaton

Azimuth motion is considered as fault free for simplicity.

Events	Current Threshold	Controllability
AZ_CW_MOVE	N/A	Controllable
AZ_CCW_MOVE	N/A	Controllable
AZ_CW_OK	$\leq 500mA$	Uncontrollable
AZ_CCW_OK	$\leq 500mA$	Uncontrollable

Table 5.3: Azimuth motor events and current thresholds

Elevation Motor Motion

Elevation motion has the same movement description as azimuth motion except that it is not assumed to be fault-free. If the elevation motor failure occurs due to electrical problems or physical hindrance, the model transits to a faulty state.

Failure of the elevation motor will be detected by checking the current. If the motor's average current remains above 500mA, it is considered stuck, and EL_FAIL_MOVE event is triggered. This failure is assumed to be permanent (it cannot be removed). The elevation model has the same states as the azimuth motor with an additional faulty state:

1: Idle

2: Turning CCW

3: Turning CW

4: Faulty State

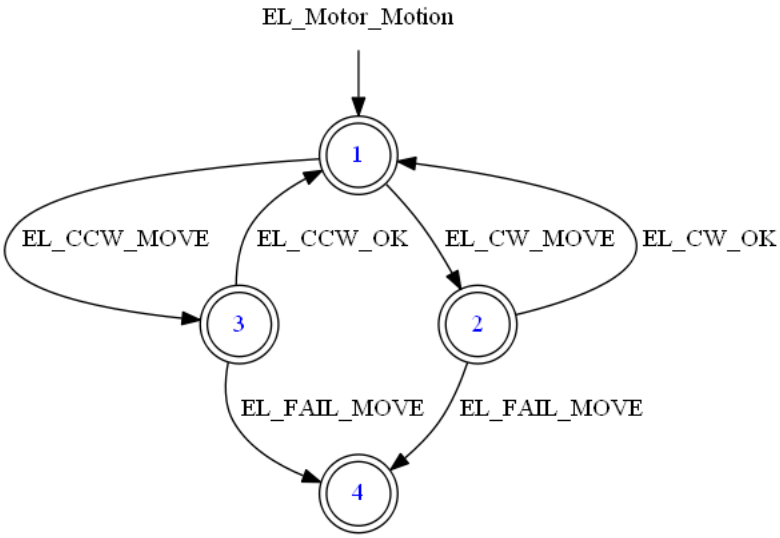


Figure 5.9: Elevaton Motor Motion automaton

All the events related to the current threshold are shown in Table 5.4.

Event	Current Threshold	Controlability
EL_CW_MOVE	N/A	Controllable
EL_CCW_MOVE	N/A	Controllable
EL_CW_OK	$\leq 500mA$	Uncontrollable
EL_CCW_OK	$\leq 500mA$	Uncontrollable
EL_FAIL_MOVE	$\geq 500mA$	Uncontrollable

Table 5.4: Elevation motor events and current thresholds

Elevation motion follows the same safety specification as azimuth motor i.e., 2° unit rotation step and waits for 2 seconds after each step.

Azimuth Motor Range

In this thesis, azimuth motion is restricted to a certain angle range of $0^\circ \leq \theta \leq 180^\circ$. Initial position for azimuth motor is $\theta = 90^\circ$. From here, it can move 90° in either direction

CW or CCW. The position of the servo is stored in a micro-controller.

As illustrated in Figure 5.10, azimuth servo angle is polled by event AZ_POLL_RANGE. If the position has reached maximum boundary AZ_MAX_CW or AZ_MAX_CCW is triggered; otherwise, AZ_RANGE_OK, followed by azimuth motor movement and process, continues. The states are:

1: Idle

2: AZ Polling Range

3: Maximum CCW

4: Maximum CW

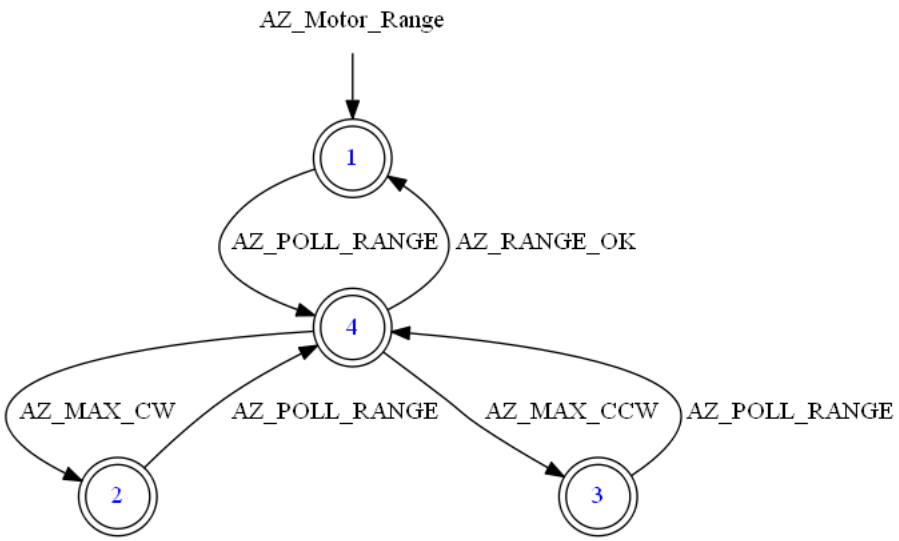


Figure 5.10: Azimuth Motor Range automaton

The controllability and range of relevant events are provided in Table 5.5

Events	Angle Range	Controllability
AZ_MAX_CCW	$\theta = 0°$	Uncontrollable
AZ_MAX_CW	$\theta = 180°$	Uncontrollable
AZ_RANGE_OK	$0° < \theta < 180°$	Uncontrollable
AZ_POLL_RANGE	N/A	Controllable

Table 5.5: Azimuth motor range events and angle thresholds

Elevation Motor Range

The range of elevation servo angle is $0° \leq \phi \leq 90°$; at the initial position ϕ is set to $90°$. The mechanism of this model is identical to the azimuth motor shown Figure 5.11, and the corresponding events are shown in Table 5.6. The states of this model are:

1: Idle

2: EL Polling Range

3: Maximum CCW

4: Maximum CW

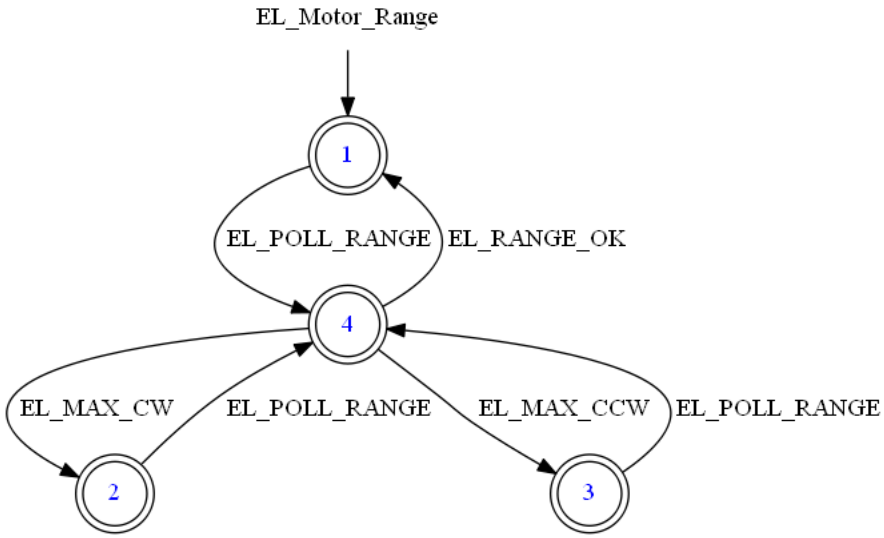


Figure 5.11: Elevation Motor Range automaton

Events	Angle Range	Controllability
EL_MAX_CCW	$\phi = 0°$	Uncontrollable
EL_MAX_CW	$\phi = 90°$	Uncontrollable
EL_RANGE_OK	$0° < \phi < 90°$	Uncontrollable
EL_POLL_RANGE	N/A	Controllable

Table 5.6: Elevation motor range events and angle range

Master Controller

In Supervisory Control Theory, the supervisor does not generate new events; it can only enable or disable them. Therefore, events that are not part of the plant components are modeled by a hypothetical one state component. For this thesis, the master controller (MC) is such a hypothetical component shown in Figure 5.12. The controllability of the events is shown in Table 5.7. Note that the three controllable events are signals that are generated by the supervisory control system.

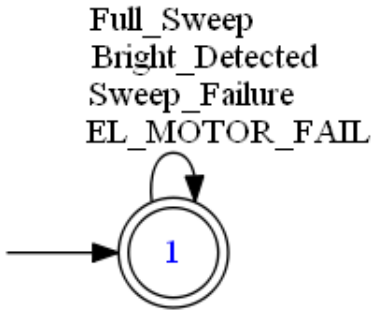


Figure 5.12: Master Controller Automaton

Events	Controllability
Full_Sweep	Uncontrollable
Sweep_Failure	Controllable
EL_Motor_Fail	Controllable
Bright_Detected	Controllable

Table 5.7: Master Controller Events

Delay for Servo motion

Servo motors can rotate with very high speed. Hence a delay of two seconds is added after each step rotation to impose safe and smooth motion, which is considered to be the same for both motors. Therefore, whenever a move command is issued for either motor in either direction, automatons reach the state of "wait of two seconds". Once the time duration is complete, the current of the motor is used to check for failures. Figure 5.13 represents the model of this component. In this model, event "wait_2_sec" is uncontrollable.

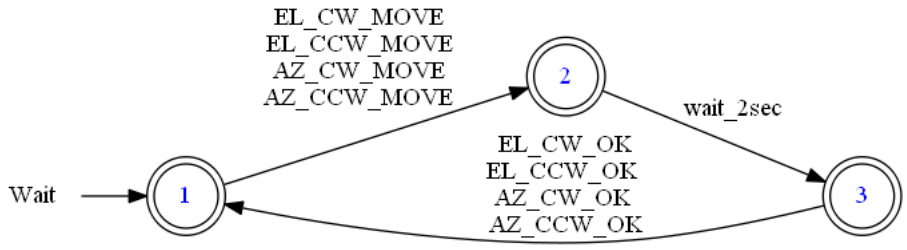


Figure 5.13: Wait Automaton

5.2.2 Interactions

The crucial interaction models must be obtained to complete the plant model, as they define the physics or working of the plant. In interactions, the dependency of components is also modeled e.g., the motion of servo motor depends on the state of the battery. If there is sufficient charge (safe or full state of battery) only then motor motion is possible; hence the model must prevent the servo rotating if the battery is in a critical state. Therefore, it is imperative to capture the physical attributes of the plant. All interactions of the solar tracker are defined in this section.

Motor Motion as function of Battery

In a solar tracker, motor motions are only possible if the battery is in a SAFE or FULL state ($SOC \geq 50\%$). This condition will allow successful servo rotation. Consequently, events that represent motion completion are only allowed at SAFE and FULL states, as shown in Figure 5.14.

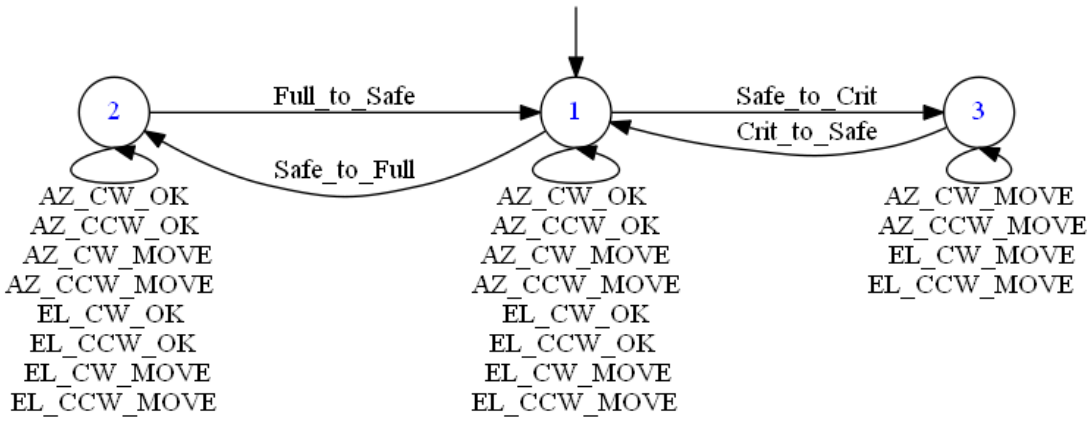


Figure 5.14: Servo motion function of Battery SOC

Battery as function of PV Panel Illumination

The PV panel charges the battery, and the charging process is only possible when the PV panel generates enough power. According to PV specifications, when PV is in Dim or Bright state, sufficient power is generated, hence battery SOC can increase, and consequently, the events Crit_to_Safe and Safe_to_Full can occur. Conversely, battery SOC can reduce at any given instance due to servo motor motion; therefore, events (related to decrease in battery SOC) Full_to_Safe and Safe_to_Crit can occur in all states of PV panel.

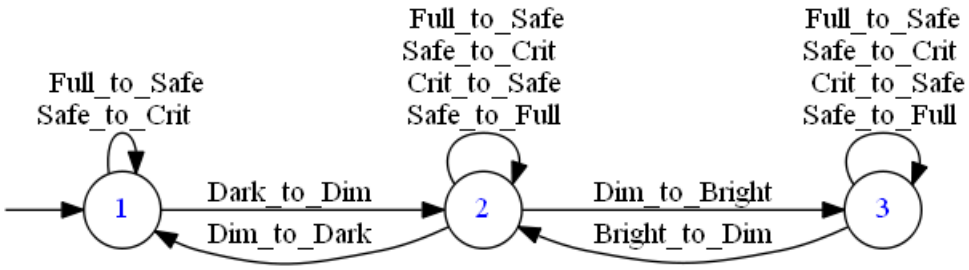


Figure 5.15: Battery SOC function of PV panel

Battery as function of Motor Motion

As discussed above, motor motion is only possible if the battery is in Safe or Full state, and as the motion happens, battery SOC decreases every time. Only PV panel charges the battery, and according to PV specification, a maximum of 200mA current can be generated (while no-load current for azimuth and elevation servos are 450mA and 520mA). Hence discharging current is always higher than charging. Therefore, the state of battery SOC always decreases during motion, and only Full_to_Safe and Safe_to_Crit events are possible during the states of motion. While all the events can occur when both motors are idle, or azimuth motor is idle, and elevation motor failed.

To build this interaction, the synchronous product of both motor motions (Figure 5.8 and Figure 5.9) is formed and suitable self-loops of events are added (according to the above discussion) on resultant automaton, as shown in Figure 5.16.

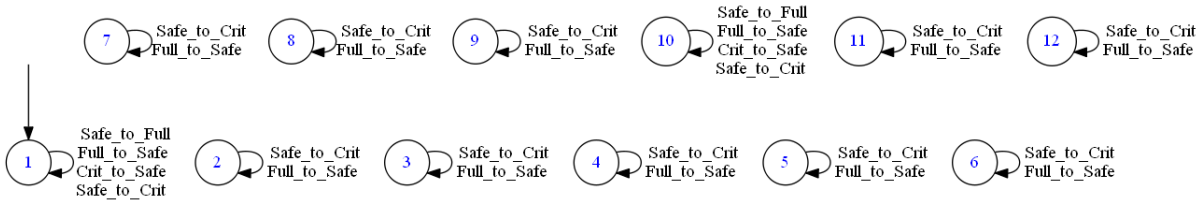


Figure 5.16: Battery SOC function of Motor Motion

As we have all the required models, all components' simple synchronous product is formed to build plant automaton. The generated model has 1584 states and 16800 number of transitions.

5.2.3 Specifications

Specifications are the set of rules, defines by the user, to be imposed on the plant. The following are the specifications considered for this thesis.

Motor motion as function of motor range

Servo motors work in the preset range defined by the manufacturer. Therefore, to ensure the safety of both servos, elevation, and azimuth ranges are limited to $90°$ and $180°$, respectively. If the elevation motor reaches to maximum limits of CCW ($0°$) and CW ($90°$), it must not be allowed to move further in the same direction as shown in Figure 5.17. Similarly, the motion of the azimuth servo is restricted. For brevity, the specification model of azimuth servo is not shown.

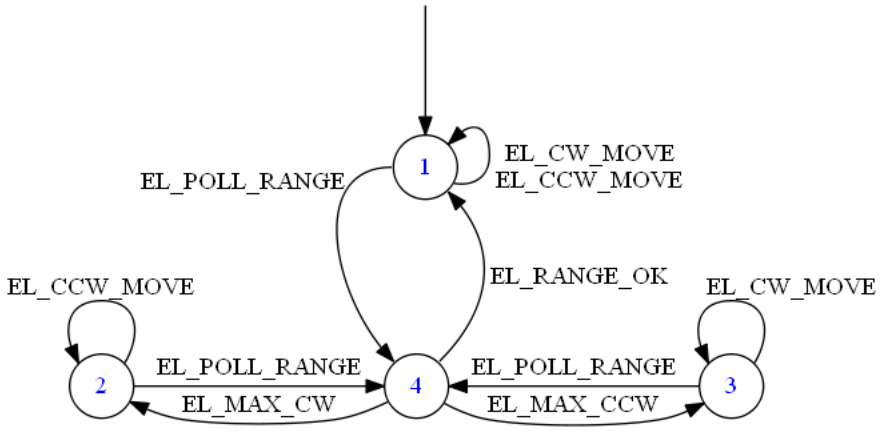


Figure 5.17: Specification of Elevation motor motion as function of range

Motor range as function of motor motion

This specification states that motor motion is only allowed when motors are idle (state 1 in Figure 5.8 and Figure 5.9). Once the event for successful motion (XX_YY_OK) occurs, and the motor is in idle state, it can be polled to get the position and rotate according to the next command, as shown in Figure 5.18. For brevity, the azimuth motor specification is not included.

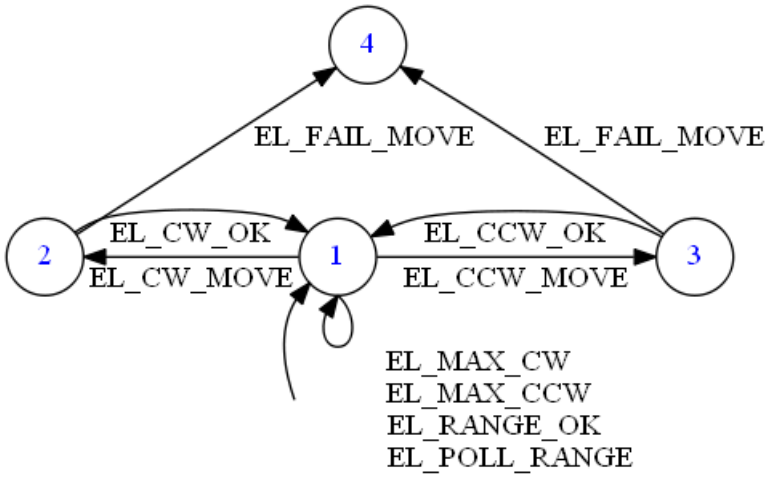


Figure 5.18: Specification of Elevation motor range as function of motion

Maneuver Specifications

The specifications defined so far are crucial concerning the operation safety. In the following specification, the motion trajectory will be defined to find a bright spot. A thorough exploration of the hemisphere may be required to search for a sufficient light source. Therefore sequences of motions are executed in a particular order, leaving no portion unexplored. This scenario is similar to the industrial case where the plant is required to follow a specific set of steps for a particular task, such as emergency, shutdown, or startup sequences. In this thesis, it is named as "Full Sweep".

Full sweep is initiated once the remote station receives the command for full sweep (CMD_Full_Sweep). In the beginning, azimuth and elevation motors are set to initial positions ($\theta = 90^\circ$ and $\phi = 45^\circ$, respectively). Once the sweep begins, the motor will start rotating in commanded direction. Only one motor is allowed to rotate at the moment to keep the battery voltage at a sufficient level for motion. Once either motor has reached its limit (CW or CCW), it begins to rotate in the reverse direction. Meanwhile, if any bright spot is detected, the sweep maneuver is terminated by sending the "Bright_Detected" signal to the ground station. Otherwise motor continues until it reaches the end of maneuver and

sends “Sweep_Failure”. Sweep motion is also terminated in case of elevation motor failure, thus generating “El_Motor_Fail”.

Full sweep motion consists of small sweep sequences as follows:

(1) Rotate azimuth motor till maximum CCW.

(2) Rotate azimuth motor till maximum CW.

(3) Sweep elevation motor to maximum CCW.

(4) Sweep elevation motor to maximum CW.

This specification has 58 states and 209 transitions and is shown in Appendix B.
Now all specifications are modeled; the last step is to include the self-loops of all irrelevant events. Hence, the product of all specs is built to get the complete specification model. The size of the final resultant model is given below:

- 416 states
- 4216 transitions.

5.2.4 Supervisor Synthesis

The supervisor of the plant is generated by using Discrete Event Control Kit (DECK)[35] developed in the MATLAB environment. In this thesis DECK procedure is written in C language in Code::Blocks.
We already have plant model G and specification model H and a list of uncontrollable events Σuc. Thus using the supcon algorithm, the supremal sub-language of the product of marked language of plant and specification $L_m(G) \cap L_m(H)$ is generated with respect to $L(G)$ and Σuc [37].
The generated supervisor is in the form of an automaton with,

- 2061 states
- 9527 transitions.

This is offline supervisor for the complete plant model.

Chapter 6

Calculation of Sequence Duration using Timed DES

In this chapter, we will first discuss the timed modeling of the solar tracker. Next, we will propose a method for the built timed model of the plant under supervision to calculate the execution time of the event sequences. Moreover, in the last section, we will compare the built timed plant under supervision with the event sequence generated during the real-time implementation of an untimed solar tracker.

6.1 Modeling of Solar Tracker as Timed DES

At the beginning of this section, all the components of the solar tracker are modeled as TDES. As discussed earlier, time is measured in terms of tick in TDES. Therefore, we will discuss the effect of tick size on the timed model. Afterward, a methodology is proposed to develop a timed plant under supervision.

The timed model of automatons is built using Timed Toy Control Theory (TTCT) [39]. All models with time bounds are written to file in DECK_C and exported to TTCT. Then TDES of the plant is imported to DECK_C and multiplied by the supervisor (possessing

time information). Resultant automaton is timed plant under supervision, which is used to analyze the execution time of events theoretically.

6.1.1 Modeling Time Bounds of Events

As discussed in section 2.2, Σ_{act} is the set of activities, equipped with lower time bound ($l_\sigma \in \mathbb{N}$) and upper bound ($\mu_\sigma \in \mathbb{N} \cup \infty$). These bounds signify the delay that can be caused due to various factors like communication or computation and represent the maximum permissible time limit. An event can be affected by different factors e.g., Dark _to_Dim event depends on the light intensity of an environment or orientation of PV panel with respect to the light source while AZ_CW_OK merely depends on the computation time of microcontroller. We have thirty (30) events in total for this project. Each one is required to be modeled as accurately as the physical system to obtain TDES of the plant. All controllable events are prospective events. Therefore, the associated upper bound is infinity.

Note: ***All time bounds are modeled in seconds.***

Controllable Events

As mentioned earlier, all controllable events are prospective events. Therefore they have an upper bound of infinity. Lower time-bound of these events is set to sampling time (50ms) of the system. Anything that happens before 50ms is not detectable as change is only observed at every 50ms. Communication delay is 15ms which is well under the sampling time.

Events	Lower bound	Upper bound
Bright_Detected	0.05	∞
Sweep_Failure	0.05	∞
EL_Motor_Fail	0.05	∞
AZ_Poll_Range	0.05	∞
EL_Poll_Range	0.05	∞
AZ_Move_CW	0.05	∞
AZ_Move_CCW	0.05	∞
EL_Move_CW	0.05	∞
EL_Move_CW	0.05	∞

Table 6.1: Time bounds of Controllable events

Battery SOC related Events

In this thesis, 2000mAH LiPo of 3.7V is used, and it is charged by PV panel via the MPPT module. The current charging rating of the MPPT module and servo motor's current consumption is used to model the time bounds of events according to the physical behavior. Time bounds of all events are shown in Table 6.2.

According to the datasheet of MPPT charging current is 450mA. So the total time to charge is calculated as,

$$Charging/discharging\_time = \frac{battery\_capacity}{charging/discharging\_current} \tag{2}$$

$$\begin{aligned} discharge\_time &= \frac{2000}{450} \\ &= 4.44hours \\ &= 267minutes \end{aligned}$$

Now from Table 5.2, the minimum time for Safe_to_Full to occur is when battery SOC changes from 94% to 95%. And maximum time this event can take is when SOC changes from 55% to 95%. Similarly, time for Crit_to_Safe is calculated.

The battery is always discharging when motors are in motion and the maximum current consumed by motor is 520mA and using equation 2, calculated discharging time from 100% to 0% is 267 minutes. Using this information, time bounds for decreasing SOC are calculated.

Events	Lower bound	Upper bound
Safe_to_Full	960	7200
Full_to_Safe	1380	6240
Crit_to_Safe	960	2400
Safe_to_Crit	720	6240

Table 6.2: Time bounds of Bat_SOC events

PV panel related Events

As this project is built in a constrained lab environment, the light source is fixed at one point to simplify the experiment. Time bounds for relevant events depending on the relative motion of light source and PV panel; therefore, upper bounds can be infinity. Consequently, events are modeled according to the lab (constrained) environment. Threshold voltages for event occurrence are provided in Table 5.1.

The minimum time bound for all relevant events is chosen as 2.1 seconds because unit step motion (2° motion followed by 2 sec wait time plus delay in communication and implementation of 100ms) can trigger these events while maximum time bounds depend on the time taken by PV to change orientation to achieve target voltage (for an event). Due to the ambient light of the lab, the PV panel generates voltage relevant to the dim state.

Therefore events related to dim states have the longest upper bounds. Time bounds of all relevant events are shown in Table 6.3.

Events	Lower bound	Upper bound
Dark_to_Dim	2.1	60
Dim_to_Bright	2.1	150
Bright_to_Dim	2.1	150
Dim_to_Dark	2.1	60

Table 6.3: Time bounds of PV_Panel events

Rest of the Events

Events other than those discussed in section 6.1.1 and 6.1.1 are simple to model as they are not environment-dependent. Instead, they only depend on the interval factor of computation in the micro-controller. As discussed earlier, the sampling time is 50ms, which is the time interval available to measure the change in the plant. Hence minimum time-bound is set to sampling time while the upper bound is fixed at 0.5 sec. This number is chosen by factoring in computation time, communication delay and some required flexibility.
Event "wait_2sec'' is to implement a safety delay of 2 sec. Hence lower and upper bounds are 2 and 2.1sec. The upper limit is set a bit higher to provide some room for flexibility.

Events	Lower bound	Upper bound
AZ_CW_OK	0.05	1
AZ_CCW_OK	0.05	1
AZ_MaxX_CW	0.05	1
AZ_Max_CCW	0.05	1
AZ_Range_OK	0.05	1
EL_CW_OK	0.05	1
EL_CCW_OK	0.05	1
EL_Max_CW	0.05	1
EL_Max_CCW	0.05	1
EL_Range_OK	0.05	1
EL_Fail_Move	0.05	1
Full_Sweep	0.05	1
Wait_2sec	2	2.1

Table 6.4: Time bounds of PV_Panel events

Now the time bounds of all events are modeled. The next step is to choose the Tick. Selecting a reasonable Stick size is very important because it affects the size of the timed automaton model and adjusts the details of timing information of events. This is discussed in detail in the following section.

6.1.2 Selection of Tick Size

In TDES models, time is measured in terms of global ticks, as discussed in section 2.2. The size of the tick can range from milliseconds to hours, depending on the nature of the event. Using a smaller tick size is a detailed TDES; consequently, the size of TDES

increases drastically. In contrast, if the larger tick size is selected, the timed model's size will be smaller at the cost of fewer details with respect to timing information. Hence there is a trade-off between the model size and timing information in TDES, as shown symbolically in Figure 6.1.

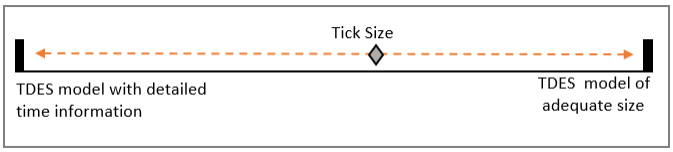


Figure 6.1: Tick size trade-off for TDES

In order to build TDES for this project, different tick sizes were selected. Figure 6.2 provides a brief insight to this relation. It can be seen that when tick size is 0.5sec, number of transitions is 2904819 while that of states is 363192. As tick size increases to 1 second, transition and state numbers decrease by $\approx 80\%$.

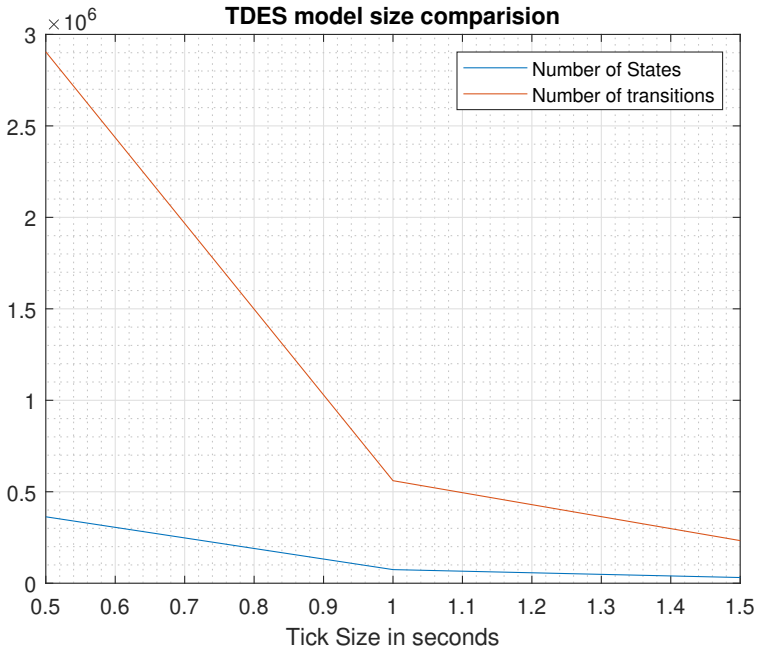


Figure 6.2: Built TDES model size comparison

Initially we began with the tick size of 50ms. But due to the significant large number

of states, computer ran out of memory. The tick size was gradually increased up to 2 sec. However, model size was still too large to be stored on available memory. Finally, we built the timed model of partial plant with tick size of 0.5 sec, for which enough memory of computer was available.

In TTCT manual[40], it is defined that infinity timed bound must be represented as 1000 ticks. Therefore the maximum number of ticks representing finite timed bound is 999. All the time bounds in ticks (according to selected tick size) are shown in Table 6.5.
For some events related to the battery, the lower bound (in ticks) surpasses the defined limit. For example, the minimum time-bound of Safe_to_Crit is 720sec (i.e. $1440ticks > 999$). The lower bound is chosen as 999 in such events. This makes the TDES model a bit more conservative. In these cases, the upper bounds are infinity because if upper tick bound were also set to 999, then it would imply that event must occur at 999^{th} tick, which is inaccurate. Hence upper ticks bounds are set to infinity even though they are finite. This will avoid hard time constraint for such events.

Events	Tick bounds		Events	Tick bounds	
	Lower	Upper		Lower	Upper
Safe_to_Full	999	1000	AZ_POLL_RANGE	1	1000
Full_to_Safe	999	1000	EL_POLL_RANGE	1	1000
Crit_to_Safe	999	1000	EL_CCW_OK	1	2
Safe_to_Crit	999	1000	EL_CW_OK	1	2
Dark_to_Dim	5	120	EL_CCW_MOVE	1	1000
Dim_to_Bright	5	300	EL_CW_MOVE	1	1000
Dim_to_Dark	5	300	EL_FAIL_MOVE	1	2
Bright_to_Dim	5	120	EL_MAX_CW	1	2
AZ_CCW_OK	1	2	EL_MAX_CCW	1	2
AZ_CW_OK	1	2	EL_RANGE_OK	1	2
AZ_CCW_MOVE	1	1000	Wait_2sec	4	5
AZ_CW_MOVE	1	1000	Full_Sweep	1	5
AZ_MAX_CW	1	2	Bright_Detect	1	1000
AZ_MAX_CCW	1	2	Sweep_Failure	1	1000
AZ_RANGE_OK	1	2	EL_MOTOR_FAIL	1	1000

Table 6.5: Time bounds of all solar tracker events in Ticks

For events, whose where minimum bound is less than 50ms, the lower bound in ticks is

taken as one tick (0.5 sec) because this is the minimum time required according to selected tick size and sampling time of the plant. Now all the events are modeled next step is to build TDES using this information.

6.1.3 TDES Model of Plant Under Supervision

The untimed model of the solar tracker has 1584 states and 16800 transitions with 30 unique events. Figure 6.3 shows the process of building a TDES plant under the supervisor. Template for *.AAS* file is provided in Appendix A. The constructed TDES model is imported in Deck_C in text format (.PDS) for other operations.

First, all untimed component models and timed events are modeled in Deck_C, then rewritten in a compatible format to export. All models are then converted to an activity transition graph (ATG) in TTCT. Meanwhile, the untimed supervisor computed (in Chapter 5) is self-looped with a Tick event on each state (in Deck_C). This step incorporates the time information in the un-timed supervisor and builds a timed model for the plant under supervision. In TTCT, *Timed Graph* function [40] is used to convert ATG to timed transition graph (TTG). Once TDES of all components is ready, a complete TDES model of plant can be computed using sync operation. The resulting model is not in a readable format (.TDS); therefore it is converted (within TTCT) to text format (.PDS) to be imported in Deck_C; the template is provided in Appendix A.

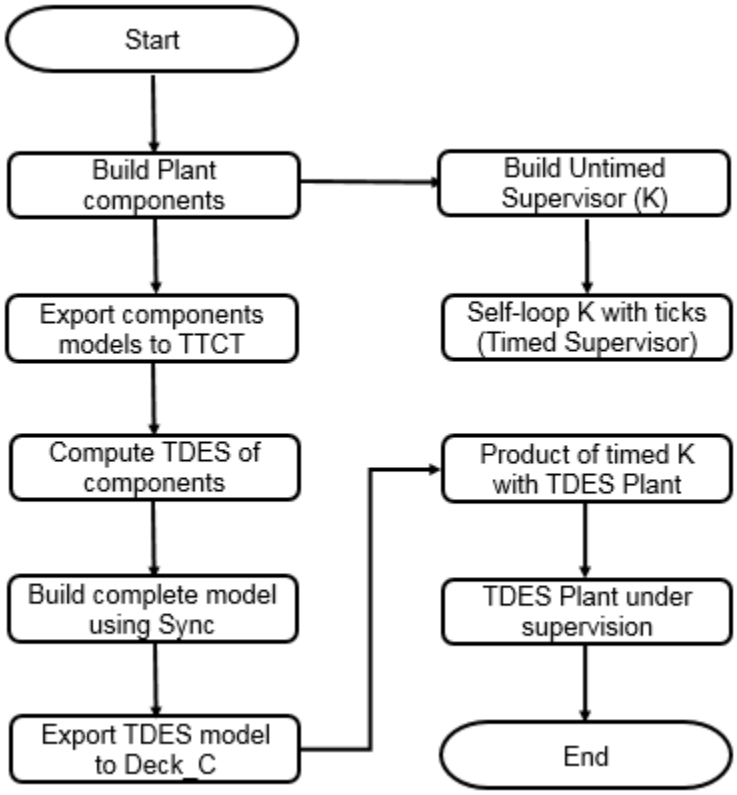


Figure 6.3: Building TDES plant under supervision

The resulting system under supervision is computed by the TDES plant product and timed supervisor in Deck_C. This model is then further used to explore the timing behavior of event sequences.

Due to the limited processing power of the ground station (desktop computer), it was impossible to build the TDES for the complete plant. Also, the TTCT software is only compatible with Windows OS; therefore, Linux based fast servers could not be used. Furthermore, while TTG for a complete plant was being built using available desktop PC, there was an error of illegal memory access after 4 hours of computation indicating RAM's insufficiency. Hence the following new strategy was adopted to build a timed model. Rather than building TDES of the whole plant, a subset of the complete model was used.

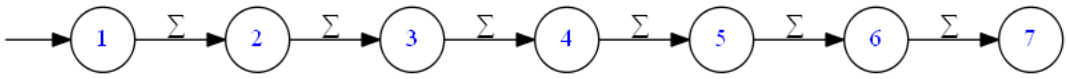


Figure 6.4: G_6

Instead of exporting all untimed components to TTCT, the complete plant model is computed in Deck_C. Then it is explored to limited depth to get a sub-automaton, which is exported to TTCT to build TDES. The solar tracker plant is explored to a depth of 6 events from the initial state. To get the desired sub-automaton of plant, a new automaton (G_6) with seven states was built, as shown in Figure 6.4. Transitions of all events (Σ) from one state to another were added. Taking the product of this automaton with complete plant models results in a sub-automaton of the plant ($G_{subPlant}$) with the depth of 6 events.

$$G_{subPlant} = G_{plant} \times G_6 \tag{3}$$

The resulting automaton is exported to TTCT to build the timed model and the meantime untimed-supervisor is computed in Deck_C and self-looped with *tick*. Subsequently, the product of both automatons (timed-plant and supervisor with ticks) is used to build the closed-loop timed-plant. The size of the computed models is shown in Table 6.6.

-	Untimed Model	Timed Model	TDES Under Supervision
States	2095	363192	12547
Transitions	12476	2904819	42509

Table 6.6: Size comparison of automatons (Depth: 6 events form initial state)

The above process can be used to build the timed model with a depth 6 from various starting states.

6.2 Analysis of TDES under Supervision

To explore the TDES under supervision, a new function is written in C language. Event string used to explore the model is generated during the real-time implementation of LLP with buffering. TDES is built for a depth of 6. Therefore the system is simulated for only six non-tick events. Figure 6.5, is the result of simulation.

Using Table 6.5, we can verify the occurrence of events in Figure 6.5. For example, once Full_Sweep is enabled, it will become eligible when the lower bound condition is met; therefore this event is executed after 1 tick. Similarly, Wait_2Sec must not execute until 4 ticks (2 sec) have passed from the enabled state. After the execution of AZ_CCW_Move, wait event is enabled but not eligible and therefore *tick* is executed once time condition is met. A string of events executed in the experiment is as follows;

(1) Full_Sweep

(2) AZ_Poll_Range

(3) AZ_Range_OK

(4) AZ_CC_Move

(5) Wait_2sec

Hence, we can say that the theoretical exploration of timed solar tracker is in-accordance with real-time results.

So far, only one sequence of TDES is explored. Of course there are several scenarios in which TDES can be explored. In the above method, the system is simulated by selecting the eligible events manually, but this methodology is not enough if one wants to explore all possible trajectories during execution. Therefore, a proper strategy is required to extensively analyze the timed model of a closed-loop system (GK_T).

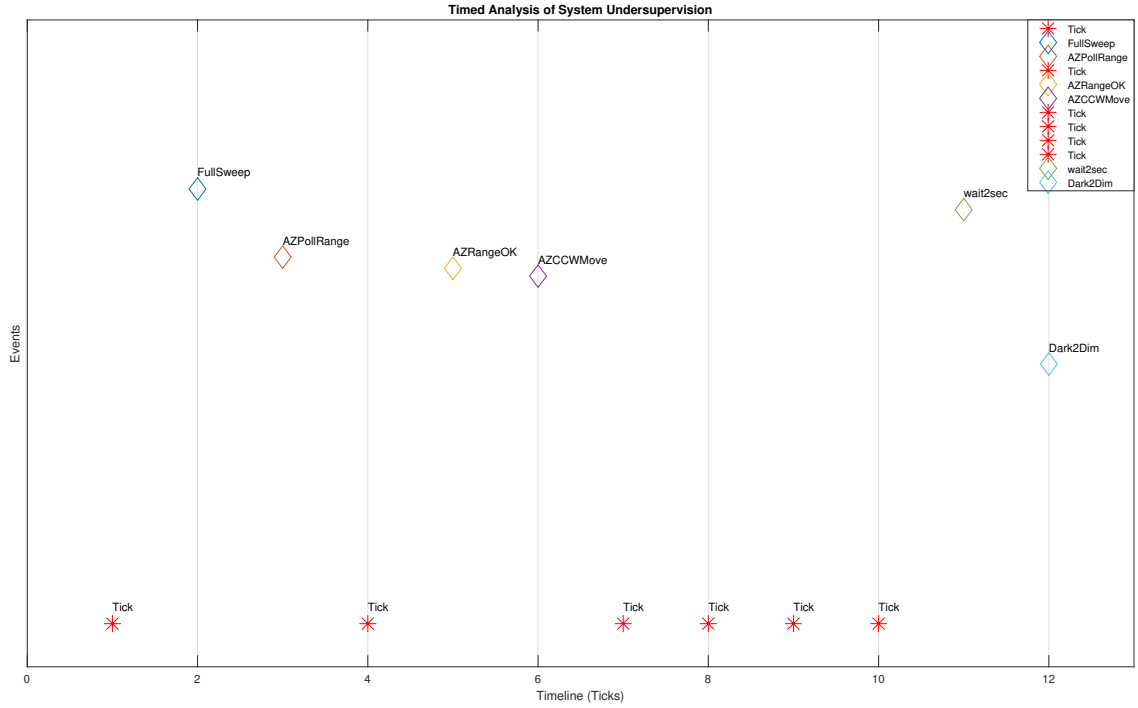


Figure 6.5: Events occurrence in TDES

So far, we have only explored six events in a timed plant under supervision (GK_T). While exploring, it was observed that beyond specific state, no *non-tick* events could occur. This state is named as boundary state, which signifies the depth of GK_T. One can compute the system's depth by iterating through each state until no further non-tick events are available.

Once the depth is known, an automaton, G_τ as shown in Fig 6.6, is built with the same depth. In G_τ, state transition occurs only due to tick (τ), while in the case of non-tick events state remains unchanged. However, there is also a self-loop of τ in the last state only, as we know beyond this state only tick is available.

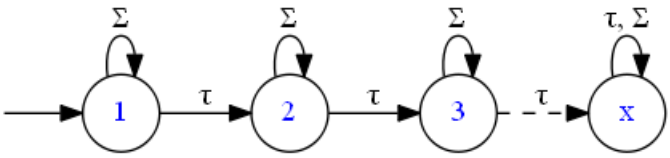


Figure 6.6: G_τ

Next we build G_R as, $G_R = sync(GK_T, G_\tau)$.

We can use the G_R automaton to extract the timing information of each sequence. In both DECK (MATLAB) and Deck_C, all states of sync operation can be saved in an array as pair (x, y), where x and y are states of the first and second argument of sync function respectively. Hence, each state of G_R is in the form (x, η_τ), where x is the state of timed model GK_T and η_τ is the number of ticks that it takes to reach x. This bu exploring all states of G_R we can extract timing information of each sequence.

Let another automaton G_{Ev}, with same depth as G_τ but with transitions of all events (Σ) and self-loops of ticks(τ),as shown in Fig. 6.7.

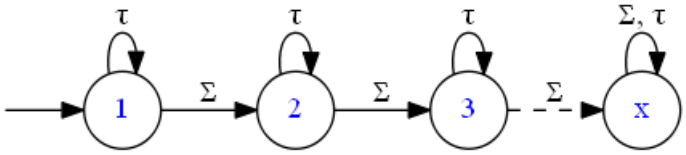


Figure 6.7: G_{Ev}

If we perform sync operation on G_R and G_{Ev} as,

$G_{R'} = sync(G_R, G_{Ev})$

The output state set of $G_{R'}$ automaton can be represented as ((x,n_τ), n_e), where,

x: State of TDES model

n_τ: Ticks to reach x^{th} state

n_e: Number of events to reach x^{th} state

Performing the above operations in the same procedure, we can have complete information of number of events and ticks to reach a state of GK_T.

Let us consider the few transitions of sub-automaton of solar tracker TDES under supervision (GK_τ) built with the proposed methodology, as shown in Fig. 6.8 and the event information of the states are provided in Table 6.7 .

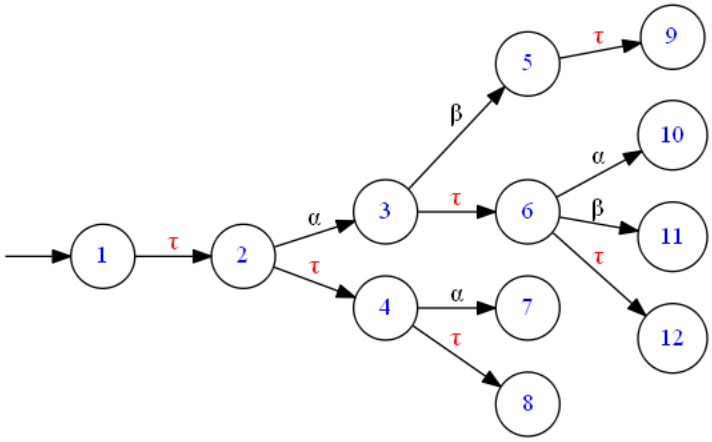


Figure 6.8: Sub-automaton of TDES under supervision

x	η_τ	η_e
1	0	0
2	1	0
3	1	1
4	2	0
5	1	2
6	2	1
7	2	1
8	3	0
9	2	2
10	2	2

Table 6.7: Timining information of GK_τ Events

Let us consider the state five from above table. We can see that state 5 requires one *tick* event (represented by 0) and two non-tick events to reach from the initial state. And from Fig. 6.8, the event sequence from state 1 to 5 is:$\{\tau, \alpha, \beta\}$. One can also compute the transitions to reach any state of TDES under supervision by adding η_τ and η_e of the relevant state, i.e., is 3 in case of the provided example.

In the next chapter, we will use the procedure described above to design a lookahead policy with buffering for the solar tracker.

Chapter 7

Control Implementation and Analysis

In this chapter, we discuss a detailed implementation of LLP with and without buffering on the plant in real-time and provide an analysis. The computation time of supervisory commands is measured experimentally, then compared with the execution time of events to validate the timed plant under supervision generated by the proposed methodology in Chapter 6. A formulation is proposed to compute time per event during LLP with buffering for performance analysis that helps for selecting buffer size while fully utilizing LLP's benefits with command buffering.

Solar tracker and computer are the two main hardware components of this project. Therefore communication between them is very crucial for the proper implementation of the controller.

In a solar tracker, all sensors are scanned periodically to measure the dynamics of the plant. The sampling time of 50ms is enough to check the event occurrence and measure feedback from sensors. All uncontrollable events of the plant are tracked within a micro-controller while both uncontrollable and uncontrollable events are transmitted to PC. Only controllable events are sent back to the solar tracker from PC in order to reduce the effect of communication delay. The detailed flow chart of the control algorithm is shown in Fig 7.1.

In this thesis, both algorithms, LLP and LLP with buffering, have been implemented to compare the computation time. During the execution of the controller, all necessary information (e.g., automaton sizes and supervisor computation time etc.) is stored in a text file for analysis.

The software architecture of the ground station consists of two main parts:

(1) Communication link

(2) Controller Loop

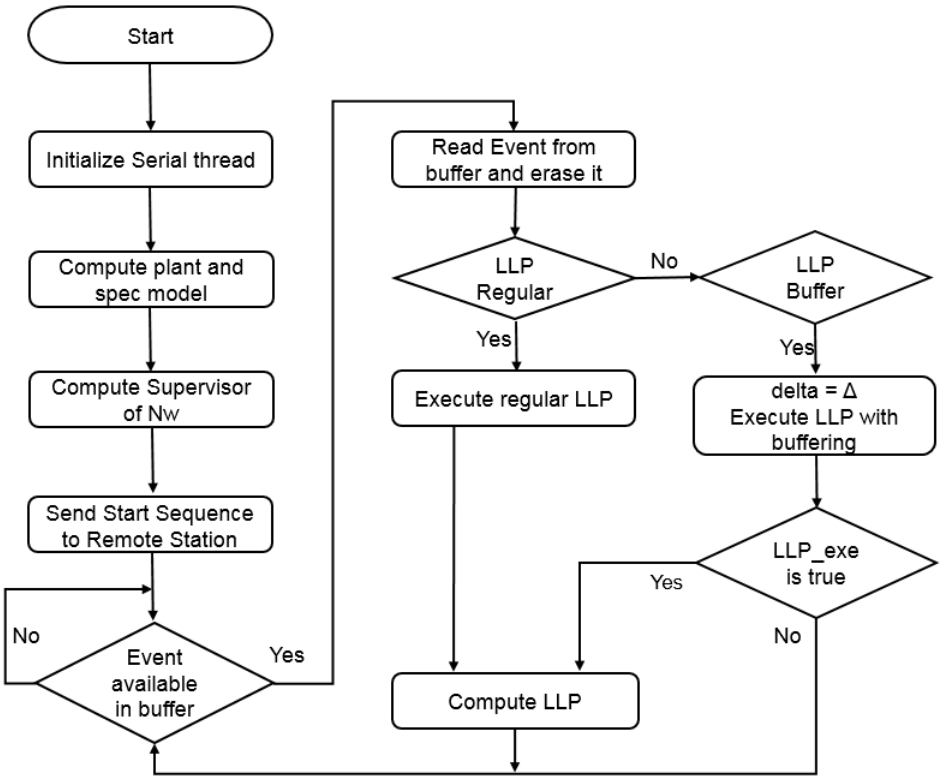


Figure 7.1: Main Control Loop

We start by presenting the communication link, especially it is software implementation and timing characteristics. After that, we turn our attention to the main topic of control logic and its implementation.

7.1 Communication Link

Data sent by the remote station is sporadic because event occurrence depends on the plant's real-time dynamics; consequently, communication with the ground is also sporadic. As C language follows the procedural programming paradigm, a single program is not capable of doing multiple tasks simultaneously (communication and computation). If serial data arrives during the supervisor's computation, the control loop will surely miss that data. And also, sometimes multiples (uncontrollable) events happen back to back (e.g., when servo motor is polled (AZ_Poll_Range), AZ_CCW_OK will occur within milliseconds and transmitted to the ground station immediately). In this case, the system will receive an event while updating the supervisor according to the previous event (AZ_Poll_Range), thus missing the subsequent event.

In order to tackle this problem ***multi-threaded programming*** [41] is implemented. In multi-threading, the program runs multiple tasks concurrently, not in parallel, as they are not being executed simultaneously. Also, one task is not blocked by another because tasks are switched to each other when in the waiting state. This makes the program to handle two tasks at once, and it is called concurrency.

Once a separate thread is created for serial communication, the buffer is required to store the received packets. If data is not stored, it will be rewritten and lost. To avoid this situation First In First Out (FIFO) circular buffer is designed. It is a fixed-size buffer implemented using a 2D array. Once the maximum slots are occupied, new incoming data is stored to the initial position, as shown in Fig 7.2.

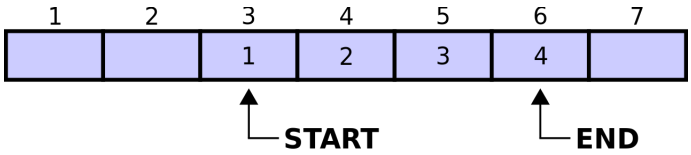


Figure 7.2: Circular Buffer

All serial tasks are handled by a communication link in a separate thread initiated at the

beginning of the main program. Mainly two tasks are performed by this thread:

- Receive/Transmit data
- Store frame in memory

Each frame is received/transmitted via Universal Asynchronous Receiver/Transmitter (UART) [42], single byte each time. The validity of each packet is checked before processing. If data is valid, it is stored in memory immediately. Otherwise, the program waits for the next packet. To avoid overwriting of memory, data is always stored at an empty frame location in the FIFO buffer. When the controller has read the data (from the main thread), it is deleted to make room for upcoming packets.

There is no instance when both tasks (reception/transmission) are required to be done simultaneously. Because the program only starts reading once, the interrupt of data reception goes high. Fig 7.3 represents the detailed process of serial communication.

Communication delay is measured using CPU clock cycles, 15msec. Only initiation commands (e.g., Reset motor, Full Sweep etc.) and controllable events are sent to remote stations from PC to minimize the effect of communication delay.

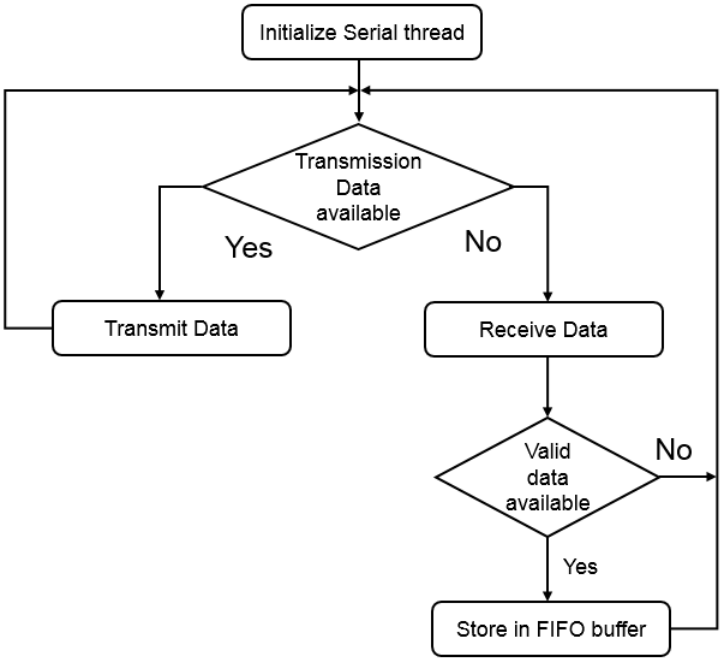


Figure 7.3: Serial Communication Flowchart

The data packet may contain encoded commands or sensor readings. Each frame consists of ten characters in the following arrangement:

!**@xxxxx&

! and & represents the start and end of the frame respectively while @ is acting as a separator between identifier and data. These symbols are also used for packet validation. ** is the header identifier and XXXXX is formatted data e.g, the identifier for battery state of charge is “05” and “xxxxx” will contain the value. A list of all packets is shown in Appendix C.

7.2 Execution of Controller

Both types of controllers (regular LLP and LLP with buffering) are implemented in the main thread. First of all, minimum plant depth [26] (already discussed in Chapter 3) is needed to be determined to implement these controllers. Recall that N_{min} is the minimum depth of plant expansion, which confirms the computed supervisor’s validity. Secondly,

the look-ahead window(N_w) is defined to explore the plant. In the case of the solar tracker, $N_{min} = 6$. To compute a valid supervisor, N_w must be a minimum of 6.

$$N_w \geq N_{min} \tag{4}$$

In the case of LLP with buffering N_w is a sum of three parameters, N_{min}, Δ and δ. Δ is the size of the buffer, while δ is the time (in events) required to calculate the Δ events in the buffer. If δ events are left in the buffer from Δ, then the new supervisor is computed over a window of N_w.

$$N_w = N_{min} + \Delta + \delta \tag{5}$$

For this thesis, as we will see later, δ is chosen to be 2, while Δ is changed over the range of values to measure computation time.

To compute the supervisor at each step of a state-based LLP with a look-ahead window of size N_w, first state starting from the current state, the plant's model must be built up to N_w events into the future. Let G denotes the plant model with the current plant state chosen as the initial state of model G. Then define $G|_{N_w}$, with $N_w \geq 0$, as the sub-automaton of G consisting of states that can be reached with a sequence of length N_w or less.

Example: Suppose G is as given below

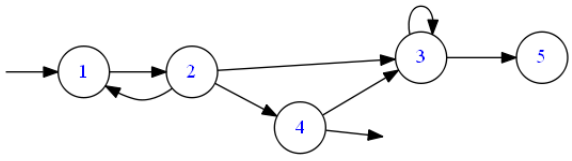
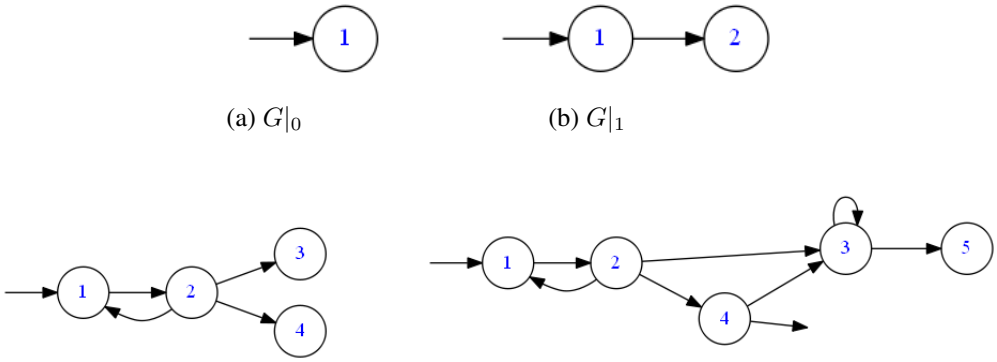


Figure 7.4: G

Then $G|_0$, $G|_1$, $G|_2$ and $G|_3$ are shown below.



(a) $G|_0$ (b) $G|_1$

(c) $G|_2$ (d) $G|_3$

Figure 7.5: Expansion of G

Note that in $G|_3$, G is completely expanded hence $G|_3 = G$.

In LLP calculations, $G|_{N_w}$ is obtained using the synchronous product of components and interactions. For this, the first component and interactions are self-looped appropriately, and then the product will be formed. Finally, the product of the resulting automaton and spec automatons will be computed. Hence the overall computation (after self-loops) is in the form of

$$G = G_1 \times G_2 \times \cdots \times G_n$$

A computer code can compute the product using a breadth-first search limited to N_w steps to find $G|_{N_w}$. This would require a product procedure with n input arguments. If the plant model or specs change, n and as a result the product procedure would have to change. An easier way is to code a product procedure with two input arguments (say, G_i and G_j to find $G_i \times G_j$). Since we would like to avoid building the large model $G = G_1 \times G_2 \times \cdots \times G_n$ (and thus obtain G_{N_w}), it would be interesting to see if we could build $G|_{nw} = (G_1 \times G_2 \times \cdots \times G_n)|_{N_w}$ using $G_1|_{N_w} \times G_2|_{N_w} \times \cdots \times G_n|_{N_w}$. The following theorem provides the answer.

Theorem 7.2.1 *Suppose for automata G_1 and G_2, $\Sigma_1 = \Sigma_2$. Then for every integer $N \geq 0$*

$$(G_1 \times G_2)|_N = (G_1|_N \times G_2|_N)|_N$$

Note: In the theorem it is assumed that the labels of states $G_i|_N$ is the same as the corresponding states in G_i. Also the states of $G_1 \times G_2$ is $(x_1 \times x_2)$ with x_i the states of G_i.

To prove the theorem, we use the following lemma. The proof of the lemma is easy and omitted for brevity.

Lemma 7.2.1 *Suppose automaton G_1 is a sub-automaton of G_2 ($G_1 \subseteq G_2$). Then for integer $N \geq 0$,*

$$G_1|_N \subseteq G_2|_N$$

Proof of Theorem 7.2.1

First we show

$$(G_1|_N \times G_2|_N) \subseteq (G_1 \times G_2)|_N \tag{6}$$

It follows from definition that $G_1|_N \subseteq G_1$ and $G_2|_N \subseteq G_2$. Thus $G_1|_N \times G_2|_N \subseteq G_1 \times G_2$ and finally by Lemma 7.2.1:

$$(G_1|_N \times G_2|_N)|_N \subseteq (G_1 \times G_2)|_N$$

Next we show

$$(G_1 \times G_2)|_N \subseteq (G_1|_N \times G_2|_N)|_N \tag{7}$$

Suppose (x_1, x_2) is a state of $(G_1 \times G_2)|_N$. Then there exists a sequence $s \in L(G_1) \cap L(G_2)$ with length $|s| \leq N$ such that in $(G_1 \times G_2)|_N$ (and thus in $G_1 \times G_2$)

$$(x_{1_o}, x_{2_o}) \xrightarrow{s} (x_1, x_2)$$

Here x_{1_o} and x_{2_o} are initial states of G_1 and G_2. Therefore in $G_1|_N$, $x_{1_o} \xrightarrow{s} x_1$ and in $G_2|_N$, $x_{2_o} \xrightarrow{s} x_2$. Hence in $G_1|_N \times G_2|_N$

$$(x_{1_o}, x_{2_o}) \xrightarrow{s} (x_1, x_2)$$

This implies that (x_1, x_2) is a state of $(G_1|_N \times G_2|_N)|_N$. Thus every state of $(G_1 \times G_2)|_N$ is a state of $(G_1|_N \times G_2|_N)|_N$. A similar argument can be used to show that every sequence $s \in L((G_1 \times G_2)|_N)$ also belongs to $L((G_1|_N \times G_2|_N)|_N)$. This shows equation (7). The theorem follows from (6) and (7).

Remarks: Note that $G_1|_N \times G_2|_N$ and $(G_1|_N \times G_2|_N)|_N$ are not necessarily equal. Here is an example.

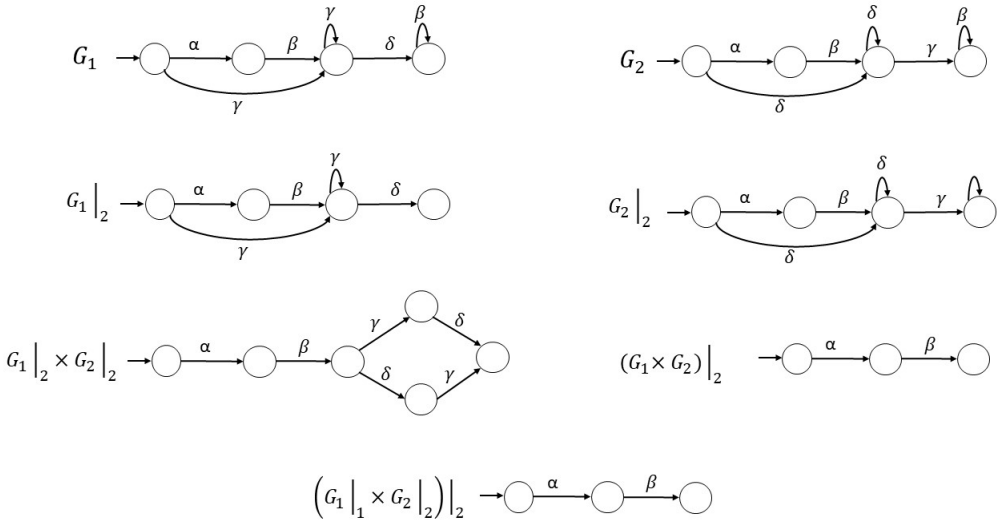


Figure 7.6: Example for Theorem 7.2.1

7.2.1 Regular LLP

In regular LLP, unlike offline supervisor computation, partial plant and specification models are computed. All components are synced up till N_w window for the plant, and all specifications (with $N_w \geq N_{min}$) are built up until the same depth using product operation. After models are ready, the valid supervisor is computed for a limited window (according to eq(4)) using the supcon procedure. New events enabled by the supervisor are used as a control command. This process iterates and in each iteration, partial plant and spec models are computed. Although building models for small windows decrease the size of resulting automata, computation in each iteration makes it expensive in terms of time.

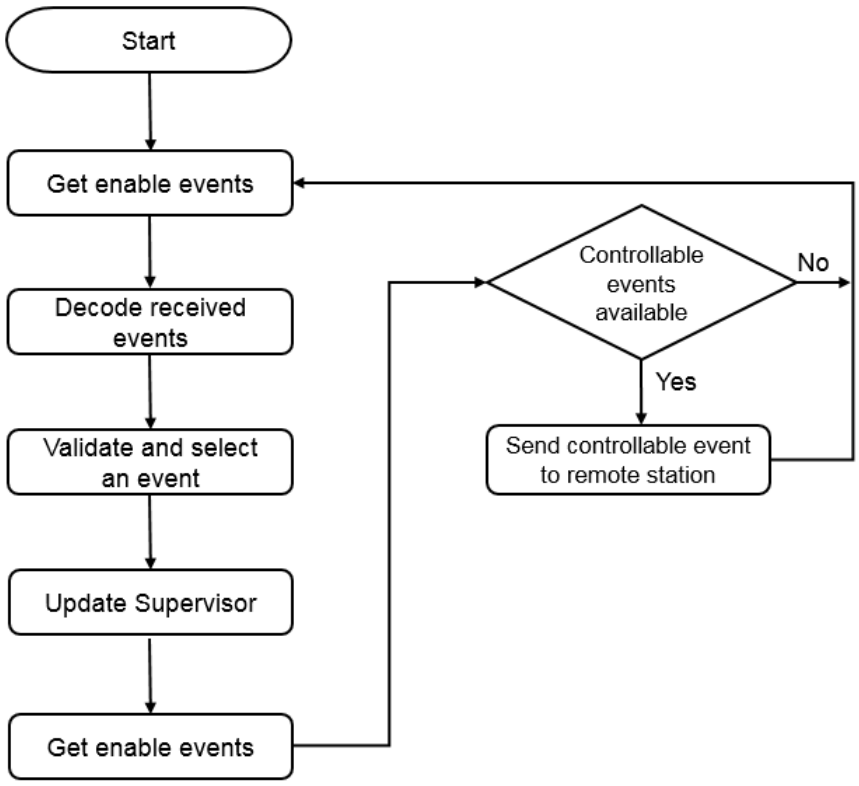


Figure 7.7: Regular LLP Flowchart

When an event occurs, it is compared with already enabled events in the control algorithm of regular LLP. Then the supervisor is updated based on this event and new enabled

events are prepared. Controllable events are sent to micro-controller if available, and the loop continues. The flowchart of the regular limited look-ahead algorithm is shown in the above figure.

We can imagine that using this method for computing supervisor is not feasible for real-time applications, as it requires high processing speed. Consequently, it is not possible for an embedded system application where limited computing resources are available on-board. To tackle this problem, a new methodology was suggested to compute the supervisor for a few extra events further into the future instead of only for the next immediate action, i.e., LLP with buffering. In this method, the supervisor is computed for a larger window, with added depth providing future buffered supervisory commands.

7.2.2 LLP with Buffering

In LLP with buffering, Δ and δ are defined before control loop initiation. To get the look-ahead window size N_w, we need to define the size of all parameters of equation (5). For the solar tracker $N_{min} = 6$. δ is the time (in terms of the number of events) required for the calculation of supervisory commands over the window of N_w. Experimental, as well as theoretical calculations, show that $\delta = 2$ is enough. Lower values 0 and 1 may not leave a suitable margin. Buffered length Δ is varied over the range of 3 to 30. The minimum $\Delta = 3$ is because $\delta < \Delta$.

As the control loop begins, after initiating the serial communication thread, plant and specification models are built over the selected window size. Once the supervisor is computed, the Full Sweep command is sent to the remote station. Full sweep maneuver consists of complete azimuth and elevation sweeps in both directions clockwise (CW) and counter-clockwise (CCW), with a defined range of 35° to 145° for elevation motor and 90° to 270° for azimuth motor.

The flowchart of LLP with buffering is shown in Fig 7.8. When the command is sent, the

micro-controller updates the supervisor's state and sends enabled events to PC. The number of executed events is tracked in LLP with buffering. If the event counter counts down to δ, the flag is raised to start LLP computation, as shown in Fig. 3.3. Once the event counter reaches zero, it is reset to Δ, and the new supervisor replaces the previous one. If there is a problem with validation, i.e., the event enabled contradicts the new events allowed by the supervisor, the control loop will not proceed, and the program will exit.

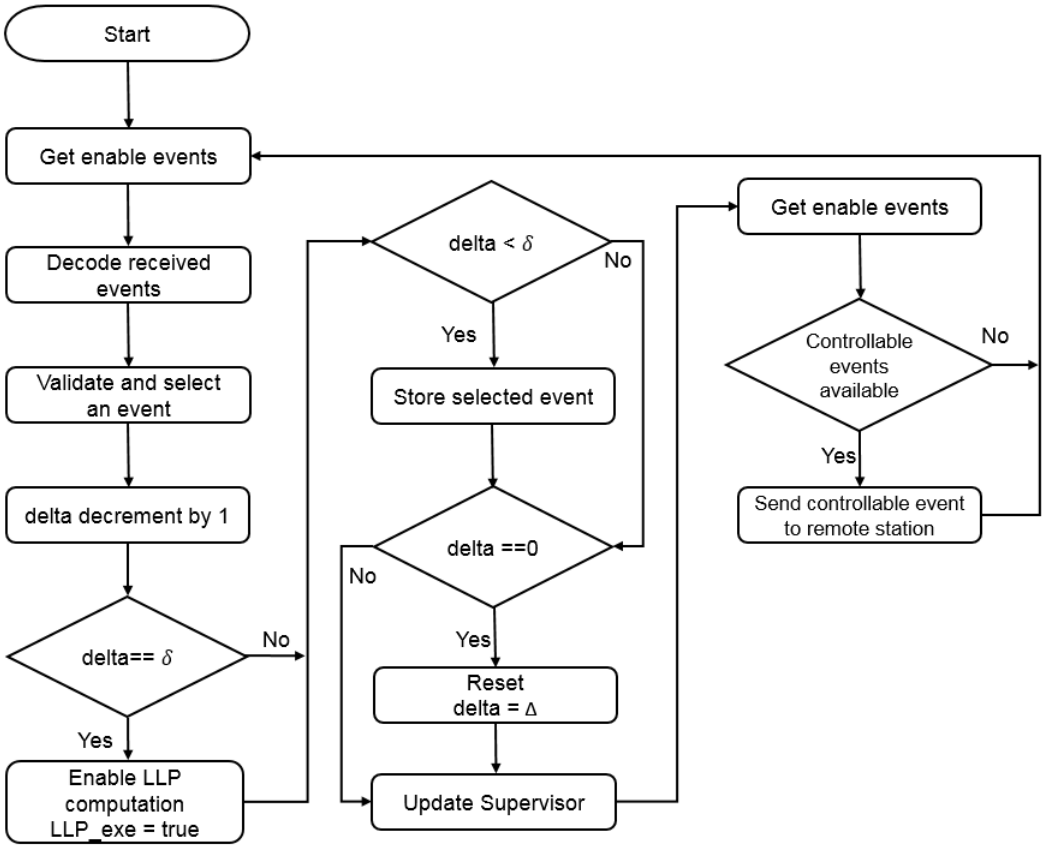


Figure 7.8: LLP with Buffering Flowchart

If two controllable events are enabled, it means there is a problem of choice. In the solar tracker, there is no such issue.

Unlike regular LLP in this algorithm, the supervisor is computed for several extra events as

a buffer, hence saving processing power for other essential tasks. Generated data (computation time, size of plant spec and supervisor models, and selected events) during computations are written on separate text files for later analysis.

7.2.3 Computation Time Analysis of LLP

In order to measure the computation time of the supervisor, CPU clock cycles are used. To measure the accuracy of measuring the computation time, the small program was created, in which the program was forced to sleep for a known amount of time (e.g., 1 sec) and then sleep duration was measured in clocks cycles. The error of $\pm$16ms was observed. The following two main functions associated with LLP with buffering are already discussed in section 3.1.3:

(1) Maximum Computation Time C_{max}

(2) Minimum Execution Duration T_{min}

$C_{max}(N_w)$ is the maximum computation time needed to find supervisory commands over the look-ahead window of N_w and is determined by experimentation only. While $T_{min}(n)$ is the minimum execution duration of n events. It can be calculated by extensive experimentation or by building the timed model of the plant under supervision as suggested in chapter 6.

Let $\eta = \Delta + \delta$. To calculate the computation time for LLP, it is implemented over the range of $0 \leq \eta \leq 30$. When $\eta = 0$, there is no event in the buffer; therefore, the supervisor is valid for only one event. This is the regular LLP.

Plant and specification depths are 12 and 25 respectively. While using $N_{min} = 6$, the expanded plant model is identical to complete model when $\eta = 6$ and complete product of plant and spec model for $\eta = 19$. Hence using buffer size larger than 19 will not affect

η	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
δ	0	0	0	1	1	2	2	2	2	2	2	2	2	2	2
Δ	0	1	2	2	3	3	4	5	6	7	8	9	10	11	12
C_{max}	62	78	110	156	187	187	188	250	265	281	281	328	312	312	328
15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
312	312	328	312	328	330	328	312	312	328	312	312	312	312	328	312

Table 7.1: C_{max} (msec) for LLP with buffering

supervisor size and computation time. Let us name this look-ahead window size as *Saturation Point (SP)*. Using the above values SP for solar tracker plant is computed as,

$$SP = N_{min} + 19 = 25 \tag{8}$$

Figure 7.9 depicts the minimum, average and maximum computation times. For this plot Full_Sweep maneuver is executed to explore the system model from each possible state.

Red and black plots represent the maximum and minimum computation time bounds for complete maneuver. It can be seen that the maximum time taken is 330ms at $\eta = 20$. It can be verified from Table 7.1 that after $\eta = 11$ computation time is almost the same, time difference (of 16ms) is due to accuracy factor of machine explained in the beginning of this section.

Note that for $\eta = 0$, δ and Δ are zero, signifying that the supervisor is generated for only one subsequent transition (case of regular LLP). For $\eta = 1$ and 2, Δ is non-zero, but δ is zero, which means that the LLP computations are executed in each iteration with one and two buffered events (respectively). However, from $\eta \geq 3$, Δ number of events are buffered and the LLP computations are triggered when δ number of events is reached.

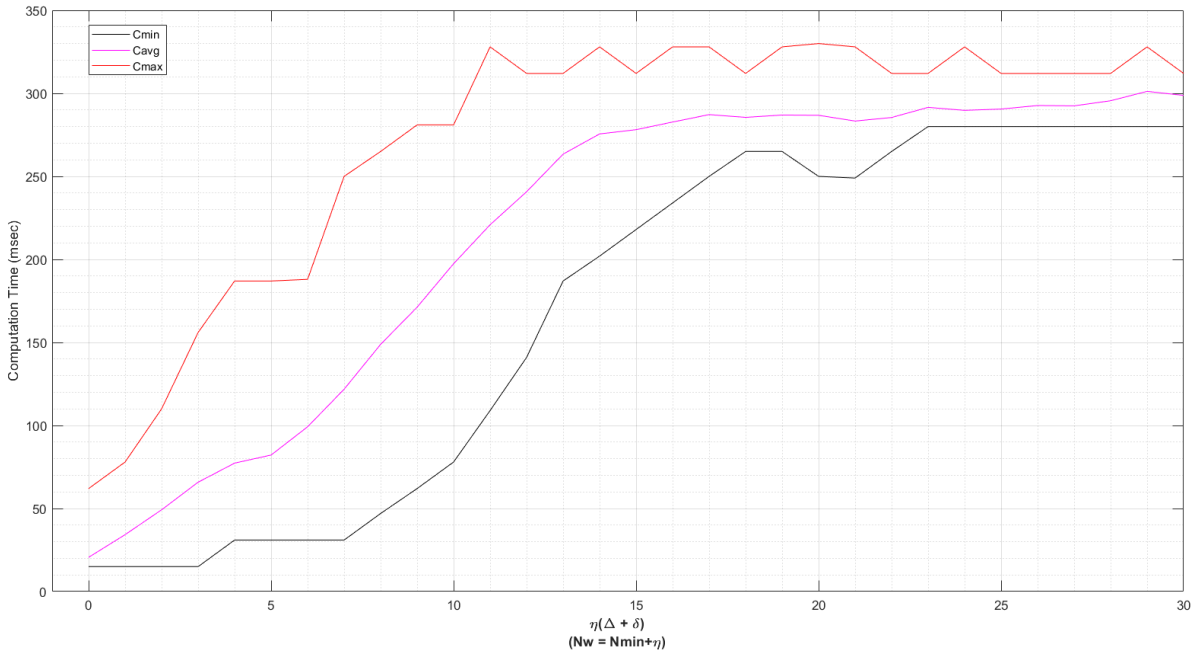


Figure 7.9: Computation time for LLP

In this thesis, execution time of events is calculated by building the timed model of the plant using TTCT. For the successful implementation (without unnecessary delay) of LLP, the computation of the supervisor must be done before the execution of the commands left in buffer. Therefore

$$C_{max}(N_w) \leq T_{min}(\delta) \tag{9}$$

Fig 7.10 depicts the execution time (in milliseconds) up to six events plotted in blue color. The least count is of 0.5 second, due the selected tick (1 Tick = 0.5sec) size as mentioned in Chap. 6. After building the timed plant under supervision, the resultant automaton was explored using the trajectory followed by the real-time solar tracker. Therefore it only explores one branch of the tree. One can explore all the scenarios, using the method explained in Chap. 6.

Since $N_{min} = 6$, $\delta = 2$ and if we choose $\Delta = 18$ ($\eta = 20$), then

$C_{max}(26) = 330ms$

and

$T_{min}(2) = 500ms$

Thus $C_{max} \leq T_{min}(\delta)$.

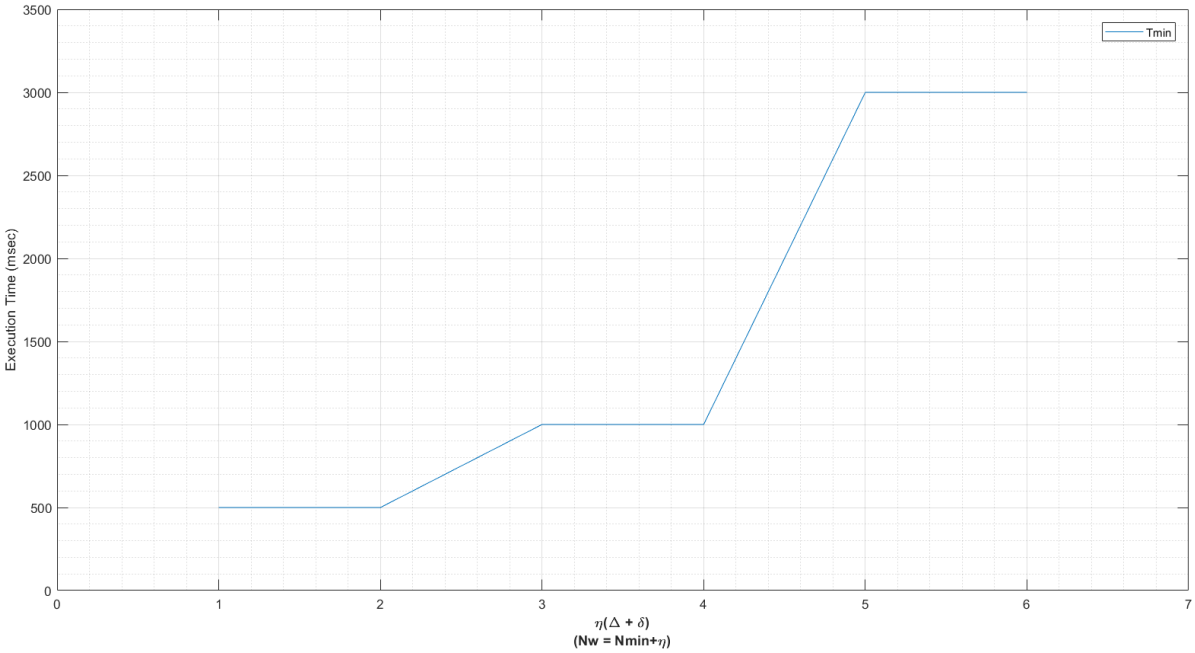


Figure 7.10: $T_{min}(\delta)$

Therefore, we have 170ms for communication and other tasks (if required), which is more than enough. Also that it can be seen in Fig. 7.9 that computation time for all windows is well under execution time of $T_{min} = 500ms$. Figure 6.5 shows the occurrence of events with respect to the Ticks, in the same order as explored using timed plant under supervision.

7.2.4 Computation Time per Event

Let $C_{max}(N_w)$ is maximum computation time for $\Delta + 1$ events. Therefore computation time for a single event (C_1) can be calculated by,

$$C_1 = \frac{C_{max}(N_w)}{\Delta + 1} = \frac{C_{max}(N_{min} + \Delta + \delta)}{\Delta + 1} \tag{10}$$

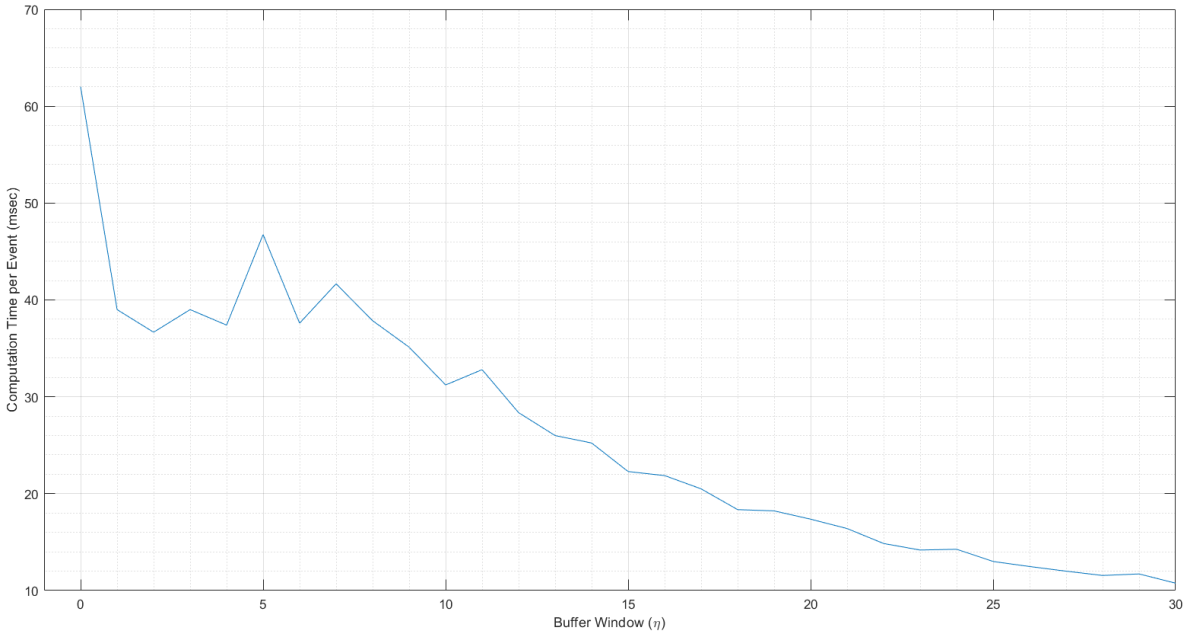


Figure 7.11: Computation time per event during LLP with buffering

When supervisor is computed for window of $N_w = N_{min} + \Delta + \delta$, it has control command for one immediate action and Δ buffered events for future. Hence $C_{max}(N_w)$ is divided by $\Delta + 1$.

Fig. 7.11 depicts the plot of C_1 based on the data in Table 7.1. For example, for $\Delta = 18$, $\delta = 2$,

$$C_1 = \frac{330}{18 + 1} = 17.36ms \tag{11}$$

In case of regular LLP, $\eta = 0$, as no event is buffered, from Table 7.1,

$$C_1 = \frac{C_{max}(0)}{0 + 1} = \frac{62}{1} = 62ms \tag{12}$$

From Eq. 11 and 12, we can calculate that LLP with buffering consumes 3.57 times less

time than regular LLP for computation of single event, therefore three times more efficient at event buffer size of $\Delta = 18$, $\delta = 2$.

In Fig 7.11, the trend of computation time for a single event decreases along the buffer window axis. The reason behind it that after SP, the complete plant is explored and time to generate the supervisor remains the same while Δ remains increasing i.e., numerator and denominator of Eq. 10 respectively. Consequently, time to compute single event decreases but memory requirement increases at the same time since the whole plant is explored and resulting in larger supervisor size.

Note that when SP is reached, LLP calculates the offline supervisor. Once could take $\Delta = \infty$ which results in $C_1 = 0$.

It leads to a compromise of a selection of buffer size, such that one can fully utilize the benefits of LLP with buffering at the cost of minimal resources.

7.2.5 Selection of Buffer Size for LLP

As we know that a finite-state DES model has maximum depth known as Plant depth (PD). If η reaches to SP then during LLP computation complete plant and spec models are explored. Therefore it becomes offline supervisor and memory complexity of algorithm increases. While if we decrease η till N_{min} (for valid supervisor), control algorithm becomes regular LLP and computation will occur after each event (requiring more processing power), as shown in Fig 7.12,

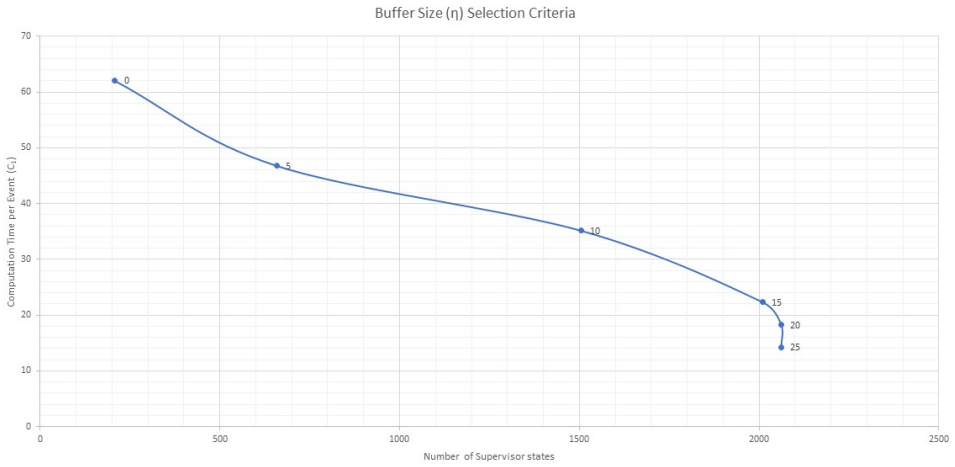


Figure 7.12: Trade off between memory and computation for buffer size

In Fig. 7.12, one can notice the increase in supervisor states as buffer size increases. The solar tracker saturates at depth of 25 therefore at $\eta = 20$, N_w is 26, hence from this point onwards, size of the supervisor remains the same. Table 7.2 provides the average automaton sizes used in LLP for various ηs. At $\eta = 0$ models are only explored till the depth of N_{min}. Consequently, plant size is approximately half of full size. As PD of solar tracker is 12 therefore at $\eta = 5$ (here $N_w = 6 + 5 = 11$) plant is almost completely explored, and size is constant till the end while that of spec keep on increasing till 20 and on-wards and complete models are explored, and therefore supervisor size is the same as of offline.

One fascinating observation is that the number of LLP computation (supervisor generation) during complete maneuver decreased significantly. Even while using small buffer size (e.g., $\eta = 5$), LLP computations are decreased by more than 50%. In real-time systems, control tasks are of high priority, and task scheduler needs to free the processor for processing low priority tasks to avoid starvation. Performing computation in high frequency might preempt other tasks. Hence by varying buffer size, one can also avoid all such scenarios by

decreasing the number of calls to compute the supervisor.

($N_{min} = 6$) η	Number of LLP Computations	Computation Time Per Event C_1 (msec)	Supervisor Size (States,Transitions)
0	1152	62	208, 763
5	472	46.75	659, 2825
10	178	35.12	1505, 6776
15	110	22.28	2010, 9261
20	79	18.22	2061, 9527
25	64	14.26	2061, 9527

Table 7.2: Model sizes and number of Computations w.r.t. η

Keeping in mind all of the above discussion, we need to carefully choose η to utilize LLP's advantages with buffering. Selecting the buffer size of $\eta = 10$ generates a supervisor with significantly less memory requirement i.e., only two-third of offline supervisor in contrast gain for computation time per event during buffering is reduced to 43.35%, from Table 7.1. One can notice that a substantial percentage of 85% reduces the frequency of supervisor computation.

As we know that in any system, a control algorithm is one of the highest priority tasks, and therefore, by minimizing control command generation time, the processor becomes available for other tasks. Furthermore, less processor consumption will allow the use of the low-speed processor, resulting in cutting-down power usage and ultimately decreasing the system's cost.

Chapter 8

Conclusion

In this chapter, we will finalize our discussion by summarizing the work done and the results achieved. Also, some suggestions for the future work in online-SCT are provided.

8.1 Summary

In this thesis, DECK procedures are translated in C language to improve the computation time and memory complexity of the supervisor computation. It has improved the computation time by at least 85% as compared to the MATLAB version. C language is selected since it provides more control over code and hardware (RAM). Consequently, one has more capability to optimize algorithm development. Also, C language is very suitable for application in embedded systems.

As a proof of concept for feasibility of online supervisory control computation, the solar tracker is used as a physical plant and Cortex-M3 based micro-controller is used for low-level control and communications.

Both supervisors, conventional and LLP with buffering, are successfully implemented in this work. Moreover, the analysis for supervisor computation time is done on the data received from the real-time implementation results. In the previous work, execution time

was computed by measuring the time between two events (experimentally); in this thesis, a model-based theoretical approach is proposed. The T_min function is computed by building the timed discrete events system of the solar tracker. The time bounds for all the events are modeled by experimentation and data-sheets of sensors. Due to a large number of states of the untimed plant model, the timed model of the complete plant is not generated, instead only sub-plant is developed. The timed model of solar tracker is explored using the event string generated by the real-time implementation. The analyses proved the feasibility of an online LLP strategy with buffering as the supervisor's computation time is well under the execution time. A method is also suggested to compute the execution time from each state by exploring TDES using a brute-force approach.

8.2 Future Work

For real-world applications, FPGAs are more appropriate, as one can modify it (configuring memory and processing speed) according to certain an application. Also, FPGAs are customizable for multi-core processing, thus very suitable for online-SCT implementations. For example, while computing supervisor in an online fashion one can build plant and specification in parallel which may decrease computation time by factor up-to 50%.
In case of single-core processors, adopting multi-threading in DECK_C procedures will enable algorithms to perform computations concurrently rather than performing sequentially (as performed in this thesis).
During the online supervisor computation with buffering, the size of buffer events is constant in this work, and it would be interesting to adopt the variable buffer size approach based on the execution time of events. Let us say at any time instance during LLP with buffering, few events are left in the event buffer, and the execution time of those events provides enough time margin for computation of subsequent supervisor, then one can increase the buffer size. Formally we can say that by getting the T_min function of remaining

events in the buffer, one can customize the current buffer size. This approach will help to reduce the computation time per event, as it is inversely proportional to buffer size.

Appendix A

TTCT File Templates

```
# TTCT AAS Template

AZ_Mtr_Motion

State size (State set will be (0,1....,size-1)):
# – Enter state size, in range 0 to 2000000, on line below.
3
Marker states:
# – Enter marker states, one per line.
# To mark all states, enter *.
*
Vocal states:
# – Enter vocal output states, one per line.
# If no vocal states, leave line blank.

Transitions:
# – Enter transition triple, one per line.
0 405 1
1 402 0
0 403 2
2 400 0
Event time bounds:
# – Enter time bound triple, one per line.
# Lower Time Bound in range 0..1000
# Upper Time Bound in range 0..1000 [Note: 1000 = Infinity].
400 0 2
402 0 2
```

403 0 1000
405 0 1000
Forcible events:
– Enter forcible events, one per line.
To mark all events, enter *.

TTCT PDS Template
Name # states: 7 state set: 0 ... 6 initial state: 0
marker states: 0 1 2 3 4 5 6
vocal states: none
forcible events: none
state and timer information:
0 : 0 [403, 0] [405, 0]

1 : 2 [400, 2]

2 : 1 [402, 2]

3 : 2 [400, 1]

4 : 1 [402, 1]

5 : 2 [400, 0]

6 : 1 [402, 0]
transitions: 13
transitions:
[0, 0, 0] [0,403, 1] [0,405, 2] [1, 0, 3]
[1,400, 0] [2, 0, 4] [2,402, 0] [3, 0, 5]
[3,400, 0] [4, 0, 6] [4,402, 0] [5,400, 0]
[6,402, 0]

Appendix B

Full Sweep Spec Model

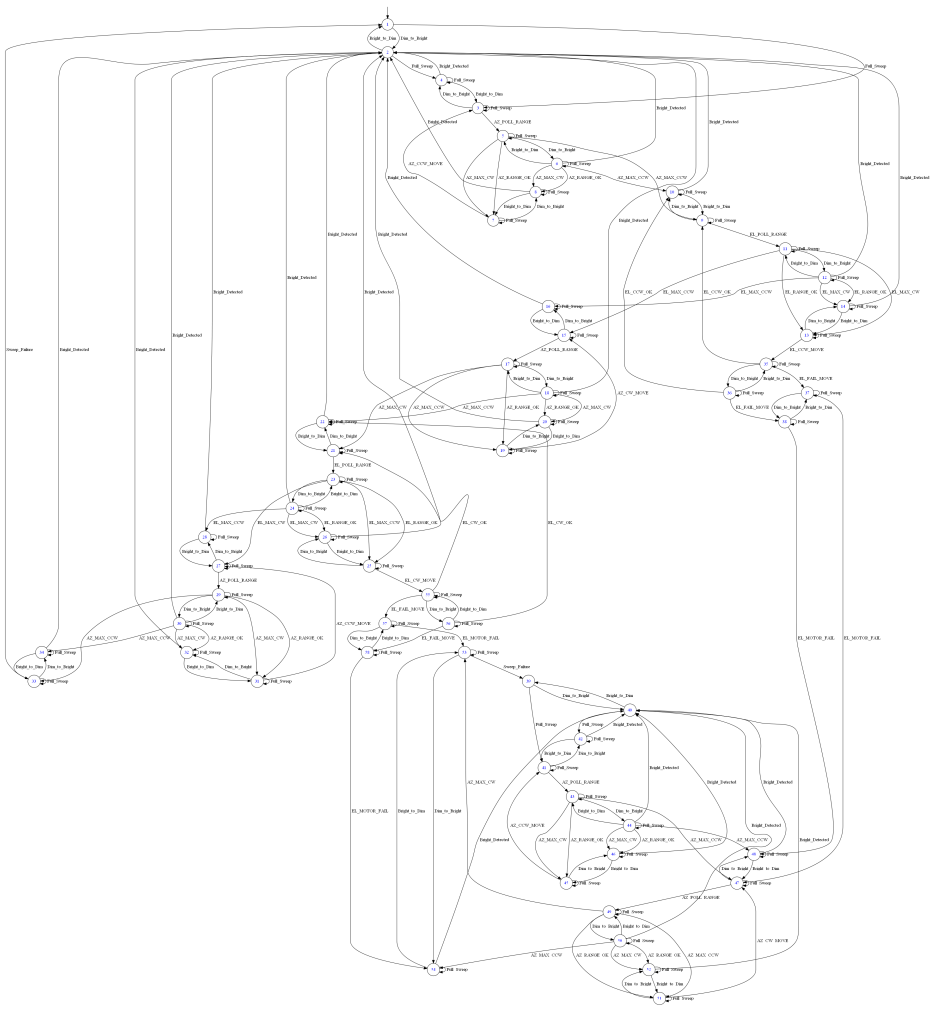


Figure B.1: Full Sweep Spec

Appendix C

Communication Packets

Header	**Data Packet**	Sent by
00	Start Command	GS
01	EL current	RS
02	AZ current	RS
03	PV panel voltage	RS
04	System Time	RS
05	Battery Voltage	RS
06	Battery SOC	RS
15	Full Sweep Command	GS
20	Bright Source Found	RS
21	Sweep Failure	RS
22	EL Motor Fail	RS
23	AZ Poll Range	GS
24	EL Poll Range	GS
25	AZ CW Move	GS
26	AZ CCW Move	GS
27	EL CW Move	GS
28	EL CCW Move	GS
29	Bright Detected	GS
30	Sweep Failure	GS
31	EL Motor Fail	GS
32	Online Command	GS
33	Reset Motors Command	GS

Table C.1: Communication Data Packets

Bibliography

[1] J. Goryca and R. C. Hill, "Formal synthesis of supervisory control software for multiple robot systems," in *2013 American Control Conference*, 2013, pp. 125–131.

[2] C. Frost, "Challenges and opportunities for autonomous systems in space," in *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2010 Symposium*, 2010.

[3] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams, "Remote agent: To boldly go where no ai system has gone before," *Artificial intelligence*, vol. 103, no. 1-2, pp. 5–47, 1998.

[4] 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.

[5] M. Pekala, G. Cancro, and J. Moore, "Verifying executable specifications of spacecraft autonomy," in *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space, Los Angeles*, 2008.

[6] S. Bensalem, L. de Silva, F. Ingrand, and R. Yan, "A verifiable and correct-by-construction controller for robot functional levels," *arXiv preprint arXiv:1309.0442*, 2013.

[7] 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.

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

[9] S. R. Mohanty, V. Chandra, and R. Kumar, "A computer implementable algorithm for the synthesis of an optimal controller for acyclic discrete event processes," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, vol. 1. IEEE, 1999, pp. 126–130.

[10] A. B. Leal, D. L. L. da Cruz, and M. d. S. Hounsell, "Supervisory control implementation into programmable logic controllers," in *2009 IEEE Conference on Emerging Technologies Factory Automation*, 2009, pp. 1–7.

[11] V. Chandra, Zhongdong Huang, and R. Kumar, "Automated control synthesis for an assembly line using discrete event system control theory," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 33, no. 2, pp. 284–289, 2003.

[12] E. Tronci, "Automatic synthesis of control software for an industrial automation control system," in *14th IEEE International Conference on Automated Software Engineering*, 1999, pp. 247–250.

[13] B. A. Brandin, "The real-time supervisory control of an experimental manufacturing cell," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 1, pp. 1–14, 1996.

[14] M. H. de Queiroz and J. E. R. Cury, "Synthesis and implementation of local modular supervisory control for a manufacturing cell," in *Sixth International Workshop on Discrete Event Systems, 2002. Proceedings.*, 2002, pp. 377–382.

[15] J. Geurts, "Supervisory control of mri subsystems," *Master's thesis, Eindhoven University of Technology*.

[16] R. Kamphuis, "Design and real-time implementation of a supervisory controller for baggage handling at veghel airport," *Master's thesis, Eindhoven University of Technology*, 2013.

[17] M. Fabian and A. Hellgren, "Plc-based implementation of supervisory control for discrete event systems," in *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*, vol. 3, 1998, pp. 3305–3310 vol.3.

[18] M. Noorbakshsh and A. Afzalian, "Design and plc based implementation of supervisory control for under-load tap-changing transformers," in *2007 International Conference on Control, Automation and Systems*, 2007, pp. 901–906.

[19] Y. K. Lopes, A. Leal, R. Rosso Jr, and E. Harbs, "Local modular supervisory implementation in microcontroller," 2012.

[20] K. Searle and S. Hashtrudi-Zad, "Microcontroller based supervisory control of a solar tracker," in *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, April 2017, pp. 1–6.

[21] M. O. Bayoume, M. A. El-Geliel, and S. F. Rezeka, "Supervisory position control for wheeled mobile robot," in *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, 2016, pp. 228–233.

[22] M. Cantarelli and J. Roussel, "Reactive control system design using the supervisory control theory: Evaluation of possibilities and limits," in *2008 9th International Workshop on Discrete Event Systems*, 2008, pp. 200–205.

[23] 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.

[24] S. Chung, S. Lafortune, and F. Lin, "Supervisory control using variable lookahead policies," in *1993 American Control Conference*, 1993, pp. 1203–1208.

[25] 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.

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

[27] 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, Feb 1994.

[28] B. A. Brandin and W. M. Wonham, "The supervisory control of timed discrete-event systems," in *[1992] Proceedings of the 31st IEEE Conference on Decision and Control*, 1992, pp. 3357–3362 vol.4.

[29] Yi-Liang Chen and G. Provan, "Modeling and diagnosis of timed discrete event systems-a factory automation example," in *Proceedings of the 1997 American Control Conference (Cat. No.97CH36041)*, vol. 1, 1997, pp. 31–36 vol.1.

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

[31] C. G. Cassandras and S. Lafortune, *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

[32] W. M. Wonham, *Supervisory Control of Discrete-Event Systems.* Springer, 2019.

[33] S. Lafortune and E. Chen, "The infimal closed controllable superlanguage and its application in supervisory control," *IEEE Transactions on Automatic Control*, vol. 35, no. 4, pp. 398–405, April 1990.

[34] S. Chung, "Addendum to'limited lookahead policies in supervisory control of discrete event systems,'proofs of technical results," *Tech. Rep. CGR-92-6, College of Engineering Control Group Reports*, 1992.

[35] S. Hashtrudi Zad, S. Zahirazami, and F. Boroomand, "Discrete event control kit (deck)," November 2013. [Online]. Available: https://users.encs.concordia.ca/~shz/deck/

[36] K. H. Rosen and K. Krithivasan, *Discrete mathematics and its applications: with combinatorics and graph theory*. Tata McGraw-Hill Education, 2012.

[37] W. Wonham and P. Ramadge, "On the supremal controllable sublanguage of a given language," *SIAM Journal on Control and Optimization*, vol. 25, no. 3, pp. 637–659, 1987. [Online]. Available: https://doi.org/10.1137/0325036

[38] "Graphviz - graph visualization software." [Online]. Available: https://www.graphviz.org

[39] W. M. Wonham, "Timed toy control theory (ttct)," July 2016. [Online]. Available: https://www.control.utoronto.ca/cgi-bin/dlttctdos.cgi

[40] C. Medar and A. Saadarpoor, "Timed toy control theory (ttct) user manual," July 2006. [Online]. Available: https://www.control.utoronto.ca/cgi-bin/dlttctdos.cgi

[41] J. M. Hart, *Windows System Programming*. Pearson Education Incorporated, 1900. [Online]. Available: https://books.google.co.in/books?id=vb6NJPlpP9sC

[42] N. Grattan and M. Brain, *Windows CE 3.0: application programming*. Prentice Hall Professional, 2001.