Limited Lookahead Supervisory Control with Buffering in Discrete Event Systems

Ehsan Ghaheri

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 2018

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Ehsan Ghaheri

Entitled: Limited Lookahead Supervisory Control with Buffering in Discrete Event Systems

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

Master of Applied Science (Electrical and Computer Engineering)

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

Signed by the final examining committee:

____ Chair
Dr. D. Qiu

____ External Examiner
Dr. Y. Zeng

____ Internal Examiner
Dr. A. Trabelsi

____ Supervisor
Dr. S. Hashtrudi Zad

Approved by: ______________________________
Dr. W.E. Lynch, Chair
Department of Electrical and Computer Engineering

August 21, 2018

____
Dr. Amir Asif, Dean,
Faculty of Engineering and Computer Science

Abstract

Limited Lookahead Supervisory Control with Buffering in Discrete Event Systems

Ehsan Ghaheri

The Supervisory Control Theory (SCT) of Discrete Event Systems (DES) provides systematic approaches for designing control command sequences for plants that can be modeled as DES. The design is done "offline" (before supervisor becomes operational) and is based on the plant and design specification DES models. These models are typically large, resulting in DES supervisors that require large computer memory - often unavailable in embedded mobile systems such as space vehicles. An alternative is to use the Limited Lookahead Policies (LLP) in which only models of individual plant components and specifications are stored (which take far less memory). The supervisory control command sequences are then calculated "online" during plant operation. In this way, "online" memory requirement can be reduced at the expense of higher "online" computational operations.

In this thesis, the implementation issues of LLP supervisors are studied. The design of LLP supervisors is based on assumptions some of which may not hold in practice. Notably it is assumed that after every event, the supervisory control command can be calculated and applied before the next event occurs. This assumption usually does not hold. To address this issue, a novel technique is proposed in which supervisory control commands are calculated in advance (and online) for a predefined window of events in the future and buffered. When the window starts, the commands would be ready after each event. This eliminates the delay due to online calculations and reduces the delay in responding to new events to levels close to those of standard supervisors (designed "offline").

In an effort to assess the proposed methodology and better understand the implementation issues of SCT, a two degree-of-freedom solar tracker with two servo motors is selected as the plant. Previously, a standard supervisor had been designed for this solar tracker to guide the tracker and perform a sweep to find a sufficiently bright direction to charge the battery and other parts of the system (from its Photo Voltaic cell).

The design of the standard supervisor and its software implementation is improved and polished in this thesis. Next the LLP with buffering is implemented. Several experimental results confirm that the plant under the supervision of LLP supervisor with buffering can match the behavior of the plant under the supervision of standard supervisor.

Acknowledgments

I would like to sincerely appreciate my supervisor Dr. Shahin Hashtrudi Zad for his support and advice throughout my research that always gave me the self-confidence to accomplish my goals.

Furthermore, I would like to thank my family, especially my wife, who have encouraged me in this path.

I dedicate this thesis to my son, Armin.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Supervisory Control Theory (SCT) of Discrete Event Systems (DES) provides systematic approaches for designing control command sequences for plants that can be modeled as DES. The design is done "offline" (before supervisor becomes operational) and is based on the plant and design specification DES models. These models are typically large, resulting in DES supervisors that require large computer memory -often unavailable in embedded mobile systems such as space vehicles. An alternative is to use the Limited Lookahead Policies (LLP) in which only models of individual plant components and specifications are stored (which take far less memory). The supervisory control command sequences are then calculated "online" during plant operation. In this way, "online" memory requirement can be reduced at the expense of higher "online" computational operations.

In this thesis, we study the implementation issues of LLP supervisors. The design of LLP supervisors are based on assumptions some of which may not hold in practice. Notably it is assumed that after every event, the supervisory control command can be calculated and applied before the next event occurs. This assumptions usually does not hold. To address this issue, we propose a novel technique which we call Limited Lookahead Policy with Buffering. We show that using this method may eliminate the delay due to online calculations, and reduces the delay in responding to new events to levels close to those of conventional supervisors (designed "offline"). We demonstrate our methodology by implementing it on a two degree-of-freedom solar tracker.

We start this chapter by reviewing discrete event models in Section 1.1, followed by the Supervisory Control Theory in Section 1.2. Then in Section 1.3, LLP will be outlined. Section 1.4 surveys recent results in the literature. Then the thesis contributions and outline will be briefed in Section 1.5 and 1.6.

1.1 Discrete Event Systems

Discrete Event Systems (DES), by definition, have a discrete set of states and their evolution is described in terms of transitions among states called events. Thus a sequence of events from one state to another state of the plant describes a state trajectory.

Example 1-1: Consider a processing factory shown in Figure 1.1-1. This plant includes a reactor, a pressure switch and a release valve. As long as the factory produces the final product, the plant is in normal operation (State: Normal in Figure 1.1-2). If the plant receives an emergency shutdown command activated by operator (Event: Emergency Push-button pressed), the plant will be steered to a shutdown state through some intermediate states (Intermediate 1 and 2) by receiving or activating relevant events: first opening a release valve (Event: Open release valve) to depressurize the reactor, a transition occurs from intermediate 1 state to the next state (State: Intermediate 2); then receiving low pressure signal as the next event (Event: Pressure low).

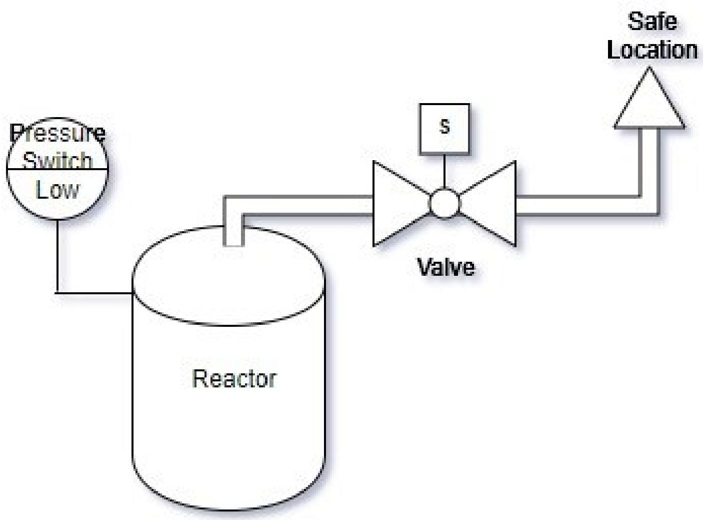


Figure 1.1-1: A simple processing factory.

The behavior of the plant is abstracted by a simple discrete-event model consisting of four separate states and three distinct transients.

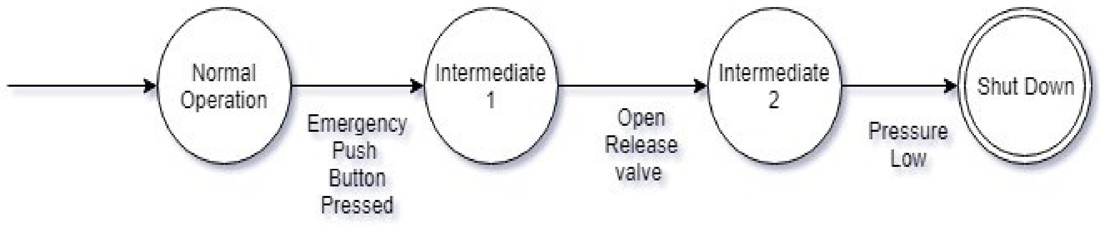


Figure 1.1-2: Discrete Event model of plant in example 1.1.

1.2 Supervisory Control

The role of Supervisory Control in DES is to prevent (disable) some events from happening in the plant to ensure its safe operation (i.e. to meet safety specifications) while guaranteeing the reachability of a designated set of states know as marked states (Figure 1.2-1).

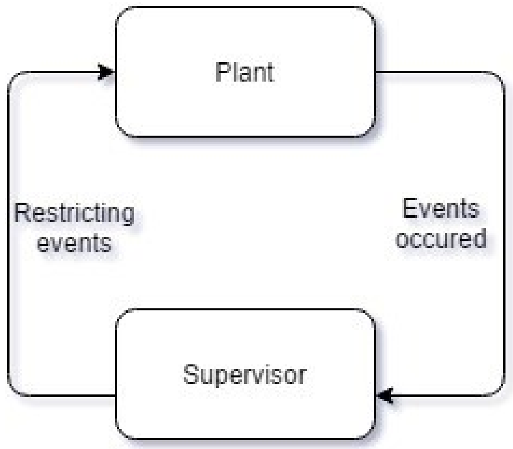


Figure 1.2-1: Closed loop Supervisory control block diagram.

In the previous example, the safety requirement calls for the decrease of excessive pressure of the reactor in an emergency condition to a safe level while reaching the shutdown state.

One can observe that the occurrence of some events such as "reading low pressure" is inevitable in some states and cannot be prevented directly by a supervisor. These events belong to the uncontrollable event set denoted as Σ_{uc}.

On the other hand, some events such as "opening the release valve" are issued by the supervisory control system and could be enabled or disabled (prevented). These events are referred to as controllable events. The set of controllable events is Σ_c.

As mentioned before, some states are marked. These states are important for instance completion of a task. In Example 1.1, the shut down state is marked (double circled in Figure 1.1-2) and the supervisor must be designed to guarantee that there is at least one trajectory from the initial state to a marked state.

1.3 Limited Lookahead Policy

In plants with an enormous amount of components (each with some states and transitions), the computation and the onboard storage of the conventional supervisory control becomes impractical if not impossible because of the state explosion phenomenon. For this reason, to decrease the onboard computer memory required for supervision, Limited Lookahead Policy (LLP) may be used. In an LLP approach to supervision, only models of individual plant components and specifications are stored (which take far less memory). The supervisory control command sequences are then calculated "online" during plant operation by considering future plant behavior over a limited horizon from its current state (Effectively, a sub-plant is to be supervised based on the design specifications of the system.) In this way, "online" memory requirement can be reduced at the expense of higher "online" computational operations.

Another case in which LLP may be preferred to conventional supervisor design is in plants in which the behavior of components varies from time to time and computing the supervisory control action is not possible when the entire plant model is not available at the time of supervisor design.

Therefore, LLP provides a synthesis method to not only mitigate the state space explosion problem but also to deal the unavailability of the complete plant model at the time of computation [1].

This thesis is on the implementation of LLP. We start by reviewing the research results that are relevant to our work.

1.4 Literature Review

Since there has been a tremendous interest in supervision and control in autonomous systems, a brief overview of studies in this field with emphasis on space applications is presented in Section 1.4.1. A literature review for SCT and LLP are provided in Sections 1.4.2 and 1.4.3 and some of implementation issues of SCT are explained in Section 1.4.4.

1.4.1 Autonomous Systems

Autonomous systems in which decision making is done without any human involvement [2] have been deployed over the past two decades especially in space explorations. The first spacecraft to test such systems was Deep Space 1 [3,4,6]. The necessity to make decisions in a time frame less than the communication latency between ground station and spacecraft makes space applications a great opportunity for the deployment of autonomous systems.

Applying computational intelligence has been studied by many especially for space exploration. The combination of Information Technology and Artificial Intelligence to guarantee stable autonomous operation while considering resource limitations was studied in [3]. This paper follows the concept of virtual presence in space and outlines the development of a system for autonomous operations whose successful operation in space (without human supervision) was confirmed later.

Model-based generation of computer code for supervisory control and decision making based on appropriate synthesis approach has been examined in [4] and [6]. In particular, the requirements for on-board decision making needed in case of unpredictable failures during various mission phases has been managed following a Lookahead Policy. A controller is provided in the software Remote Agent and Livingstone [4] which relies on discrete models of spacecraft components.

In [5] a Finite State Machine (FSM) model is used to build a virtual environment to test spacecraft protection system (since a rigorous verification of potential faults before and after launch is not plausible). [6] addresses the issue of insufficient modularity and the property of robustness, especially in space applications, and outlines a model-based programming framework in which models of plant components are used for code generation. [6] proposes a "Deductive Controller" (Figure 1.4-1) to determine the current state of the plant by observing data from sensors and performing mode estimation by computing the likelihood of the current state as belief state. Based on this information, control commands are issued to system components to follow a suitable sequence towards the configuration goals (by mode reconfiguration).

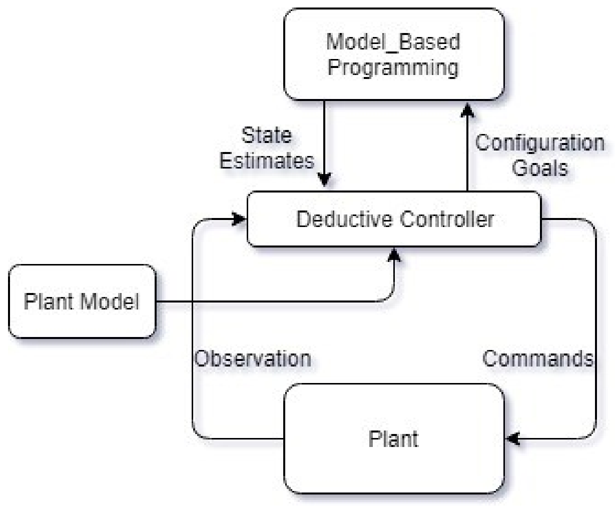


Figure 1.4-1: Architecture for the deductive controller in [6].

In order to define a formal method to check software used in space applications and to overcome testing complications, [7] presents a new method based on automaton models. The approach is applied in the development of functional levels of robotic systems facilitates software verification.

1.4.2 Supervisory Control

The theory of supervisory control (SCT) of discrete event systems was introduced by Ramadge and Wonham in [8], [9]. In this framework, the plant behavior is assumed to be described in terms of states and events of an automaton. The states change upon the occurrence of events. The events are partitioned into controllable and uncontrollable. Moreover, some events are observable by the supervisory system while others are unobservable. This also leads to two disjoint set of observable and unobservable events [10].

The role of a supervisor is to restrict the occurrence of controllable events in order to not only ensure system safety (as stated in system specifications) but also to guarantee accessibility to marked states.

In the case of full event observation (which is explored in this thesis), it is shown in [8] that an optimal (minimally restrictive) supervisor exists and is characterized by a supremal controllable sublanguage of the legal (safe) marked behavior of the plant.

Many studies have been done on using SCT in the real world. Some problems have been encountered in the implementation of this theory to control industrial systems. There is a conceptual difference between a controller and a supervisor, as shown in (Figure 1.4-2). A controller receives signals from the plant and sends unique commands to actuators (e.g. valves, motors) while a supervisor receives events from the plant and prevents (by disabling) controllable events which may lead to violation of design specifications.

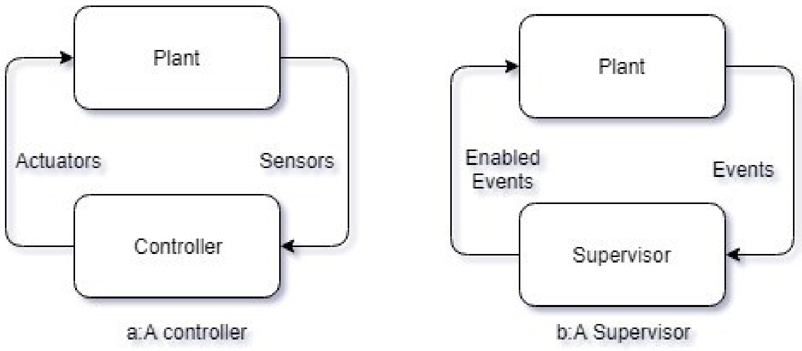


Figure 1.4-2: A controller and a Supervisor

Other solutions have been proposed for the issue of choice. In [13], costs are to all supervised paths to marked states and then the path with minimum cost is found to settle the issue of choice. The procedure is only suitable for models with acyclic graphs.

SCT presents a formal method for designing control and many researchers have explored the use of SCT as a general technique for industrial control systems. In [14] an automated small-scale assembly line is selected for SCT implementation. In this work, the selection of one controllable event from a set of eligible controllable events is done using an ad hoc manner.

In [15], a hybrid Compositional Interchange Format (CIF) model (in which interoperability of different systems is possible) is used to not only deploy untimed automata of DES but also use variables, differential equations, and conditions to overcome the time consuming and complicating design of some high-tech part of an MRI (Magnetic Resonance Imaging) scanner. Next the designed model in CIF is converted to PLC (Programmable Logic Controller) language for the purpose of controlling the associated part of the MRI system. It should be noted that there were some problems in this transformation which made the resulting code unstable.

Many researchers have favored the idea of model-based programming to obtain a uniform platform for all control systems. For this approach, SCT is an appealing candidate. In [16] a baggage transport system in an airport is chosen as the test bed. The objective of this project is to implement the resulting controller code in a real-world controller (e.g. PLC). In [17] a method is proposed to coordinate equipment operation in a flexible manufacturing system. In this paper, to use SCT in industrial/manufacturing applications, procedures are introduced to convert the designed supervisor DES model to PLC code.

The control system architecture in [18] is used to implement SFC (Sequential Function Chart) language in PLC. As one can see in Figure 1.4-3, the modular supervisor is in the top of control system hierarchy in which states are updated according to events received from product system and some events become disabled too. The product system which includes a complete model of the plant executes the received commands from the supervisor and also changes the states to keep them in sync with the physical system by receiving responses from operational procedure accordingly. The operational procedure which is an interface between real signals from hardware to higher control level interprets these signals as events.

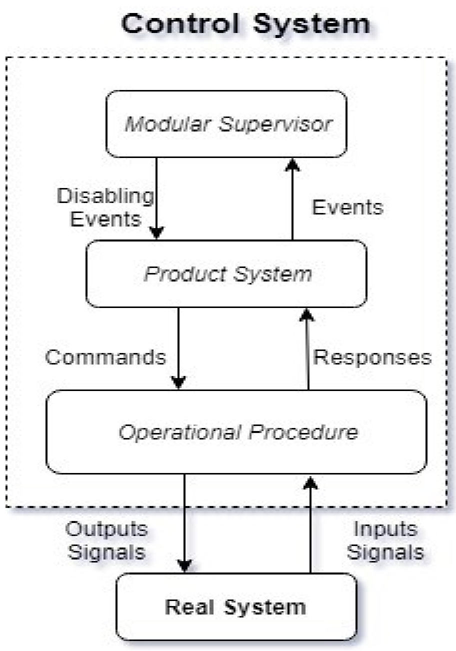


Figure 1.4-3: Control System Architecture

In [19], one of the reasons for unsuccessful use of SCT in industry is outlined as the high effort to model the system which results in high cost of design phase in factory automation applications. The solution containing a controller and a supervisor as depicted in Figure 1.4-4 is proposed. In this framework, supervisor prevents the sending of unsafe operator commands when interlocks are bypassed, and also avoids unsafe controller command (which can be issued in complex interlock systems). The focus here is on safety property since it is argued that the states which are marked do not necessarily have to be reached. Therefore, nonblocking is not a concern. The events are classified as (1) sensor events from input signals (uncontrollable), (2) operator events from the controller output signals (controllable), and (3) forcible events to prevent the occurrence of some events in certain states.

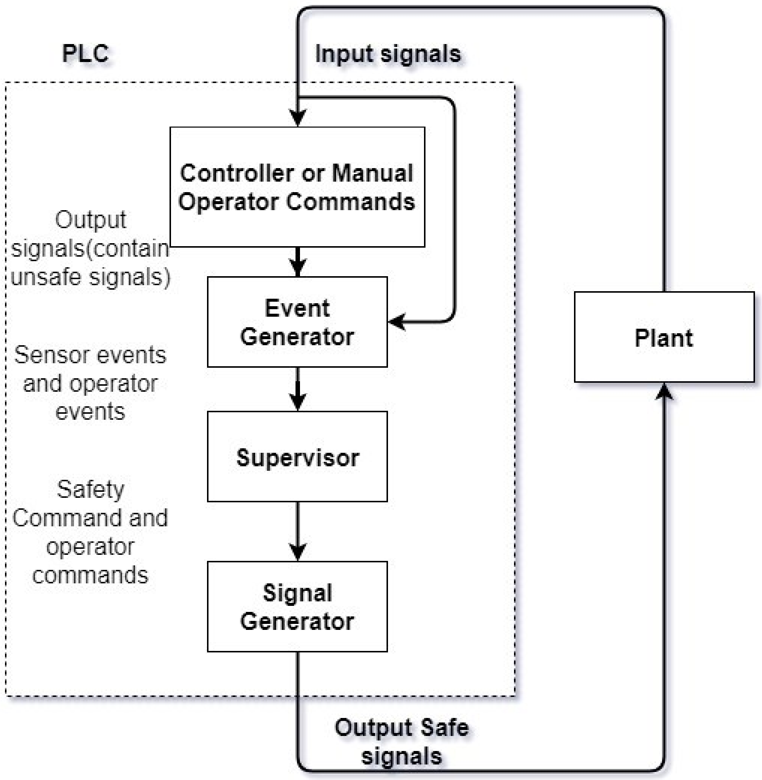

PLC
Input signals
Controller or Manual Operator Commands
Output signals(contain unsafe signals)
Event Generator
Sensor events and operator events
Supervisor
Safety Command and operator commands
Signal Generator
Plant
Output Safe signals

Figure 1.4-4: Supervisor with a controller framework

1.4.3 Limited Lookahead Policy

Limited Lookahead Policy (LLP) was introduced in [1] to tackle some of the issues of the conventional SCT. In the conventional design, the design is done "offline" (before supervisor becomes operational) and is based on the plant and design specification DES models. These models (e.g. in [20]) are typically large resulting in DES supervisors that require large computer memory - often unavailable in embedded, mobile systems such as space vehicles. An alternative is to use the Limited Lookahead Policies (LLP) in which only models of individual plant components and specifications are stored (which take far less memory). The supervisory control command sequences are then calculated "online" during plant operation. In this way, "online" memory requirement can be reduced at the expense of higher "online" computational operations. Another benefit of LLP is in those cases in which the behavior of the plant is not completely known at the design stage (e.g. in [21], [22]).

In a nutshell, only a portion of the plant ($\mathrm{N_w}$ events into the future from current state, known as LLP window) is considered for calculating control commands rather than the entire plant model. Then after any LLP computation (online) by choosing one of those enabled events by LLP (σ in Figure 1.4-5) the control action will be taken. Two attitudes as “optimistic” and “conservative” are considered to address uncertainty of the plant behavior outside of the window under investigation. The attitude will affect the resultant supervisor’s size. In cases where the time-varying plant cannot be defined a priori due to lack of enough information, a class of dynamic DES is defined in [24] to optimize online control. The growth in the size of lookahead tree is in form of exponential or polynomial [23]. In [24] a method is introduced to estimate the state space of the lookahead tree which grows exponentially, based on the window size, $\mathrm{N_w}$ selected by designer. One of the key aspects of LLP is the minimum window length to guarantee results (control commands) identical to those of a minimally restrictive offline supervisor. Although LLP requires less computer memory for the states of the plant which results in less, the computation time (online complexity) increases which is the main concern in LLP since the result of the supervisor must be available online after the execution of each event. Therefore, the size of the window size ($\mathrm{N_w}$) and online constraints of the system such as response time to events will determine the efficiency of LLP in comparison to a conventional supervisor.

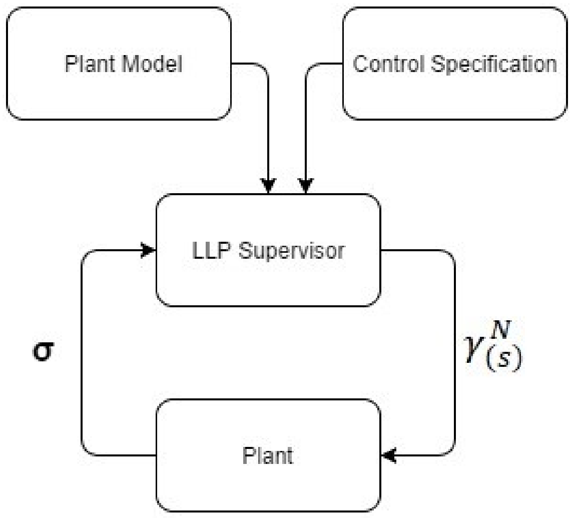


Figure 1.4-5: Limited Lookahead Supervisory Control

In [25] a recursive computational method, based on backward dynamic programming, is shown which uses previous window results for current window computation in order to reduce computation time. In [26] a forward calculation method is used to make control decision unambiguous which may need less than N-level tree size; therefore the size of lookahead window can be variable (Variable Lookahead Policy, VLP). Since the computation may terminate before the boundary of the N-level tree, VLP is a more efficient than the LLP method.

In the previously mentioned works, there is an assumption of the knowledge of events in the future steps and their conformance with respect to the legal behavior; but supervisor does not use the information about the state of the system, and because of that, the lookahead window is represented by an N-level tree. In [27], a supervisor with state information was studied which adds the state of the system for the computation of Variable Lookahead Policy (VLP-S). Using this method makes the computations simpler because in practice, the total number of the states in the lookahead window is less than the event-based LLP. Furthermore, repeated state and loops are no longer expanded in the N-level window. Moreover, in the case of plant with uncontrollable events loops, in event-based LLP, the minimum length of the window to have an optimal supervisor is infinite whereas in state-based LLP, the window size is bounded by the number of states of the plant times the number of states of the specification model.

1.4.4 Implementation problems of SCT

Moving from SCT framework (which is asynchronous and event-based) to practice (with synchronous environment (cyclic execution of program) and signals) usually has some problems which need to be addressed. [28] considers Programmable Logic Controller (PLC) as a platform to implement SCT and examines major problems of implementation. Some of these issues are not encountered in microcontroller-based systems (e.g. avalanche). In this thesis, our focus is on the implementation of SCT on embedded systems and therefore we discuss the problem of inexact synchronization, simultaneity and choice which may occurred in any systems.

1.4.4.1 Inexact synchronization

In any type of SCT implementation, it is assumed that the state of the plant is tracked (implicitly or explicitly) by the supervisor. However, the tracking may not be done completely due to inaccurate modeling of the plant or undetected uncontrollable events. Hence, inexact synchronization problem happens when the plant model inside the controller may be out of phase with physical plant because of undetected uncontrollable event. For instance, let e_uc be an uncontrollable event (input signal) and e_c be a controllable event (output signal) in a plant depicted in Figure 1.4-6. Suppose in state 1, the supervisor enables e_c but before e_c issues, e_uc happens. Therefore, the supervisor sends e_c to the plant, while the plant changes state from 1 to 3 and then by receiving e_c command to state 4. The supervisor may mistakenly assume the plant has moved to state 2.

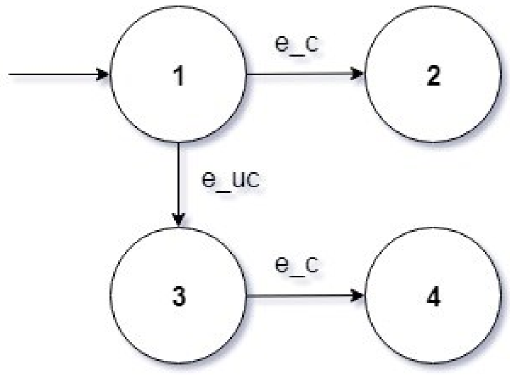


Figure 1.4-6: Inexact synchronization problem

1.4.4.2 Simultaneity

Control system typically polls their inputs (from the plant) to detect events. This may lead to two issues referred to as simultaneity problem. First issue is when the order of detected events is not the same as the order of their. The second problems happens when an input signal (event) appears between two instances of polling and goes undetected.

For example, consider the sampling time (red bars) and changes of input signals "a" and "b" as it is shown in Figure 1.4-7. The system cannot detect the first occurrence of the event "a" and therefore, detects a "b" after the second polling. Later "a" and "b" occurs after the third and fourth polling but the order ("ab" or "ba") is unknown.

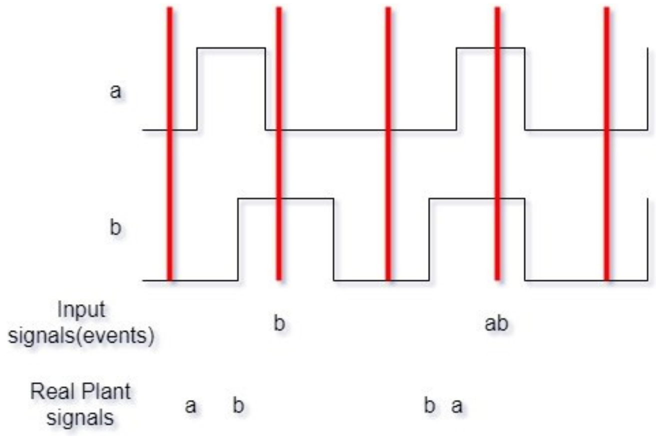


Figure 1.4-7: Simultaneity problem

1.4.4.3 Choice

Although previous problems are associated with constraints in the hardware such as sampling period, the problem of choice is related to the implementation of the theory of supervisory control. Suppose that at a given state of plant more than one controllable event is in the enabled events set of the supervisor and no uncontrollable is enabled or occurs at the state. Then a controllable events has to be issued to the plant. As far as the supervisor is concerned, any of the two evens may occur. But the system requires a rule to decide which of the controllable events has to be activated. This is known as the choice problem.

The choice problem becomes problematic when it causes a blocking in a nonblocking system. For example, the system in Figure 1.4-8 is nonblocking, but always choosing α between α and β in state 3 results in system getting trapped in the loop between states 2 and 3 and never reaching 4.

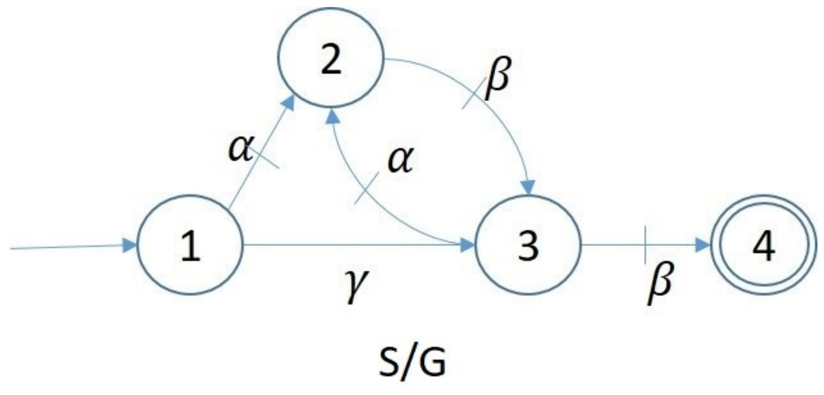


Figure 1.4-8: The problem of choice in system under supervision.

In [12] some the issues of implementation such as communication and determinism (also known as choice) are discussed. The first problem is related to receiving controllable and uncontrollable events at the same time by the supervisor. The second problem is associated with choosing among more than one controllable event in a set of enabled events. Giving higher priority to uncontrollable events at the same time of their detection (compared with controllable events) is their solution for the first problem; and they propose an algorithm to check nonblocking of the system under supervision (to deal with the problems resulting from the issue of choice).

In [29] a method is tested to maintain nonblocking property of the supervised system. The method is based on random selection of two controllable events. However, the proposed method does not have solid formal proof to address this problem generally.

In [19] the authors proposed to use a separate deterministic controller over the supervisor to generate output signals to prevent the choice problem. However, by introducing forcible events this problem arises whenever one of the several forcible events in a state must be picked up arbitrarily.

1.5 Thesis Contributions

In this thesis, the implementation issues of LLP supervisors are studied. The design of LLP supervisors is based on assumptions some of which may not hold in practice. Notably it is assumed that after every event, the supervisory control command can be calculated and applied before the next event occurs. This assumption usually does not hold. To address this issue, a novel technique, referred to as LLP with buffering, is proposed in which supervisory control commands are calculated in advance (and online) for a window of events in the future and buffered. Once the window starts, the commands would be ready after each event. This eliminates the delay due to online calculations, and reduces the delay in responding to new events to levels close to those of standard supervisors (designed "offline").

In an effort to assess the proposed methodology and better understand the implementation issues of SCT, a two degree-of-freedom solar tracker with two servo motors is selected as the plant. Previously, a standard supervisor had been designed for this solar tracker to guide the tracker and perform a sweep to find a sufficiently bright direction to charge the battery and other parts of the system (from its Photo Voltaic cell).

The design of the standard supervisor and its software implementation is improved and polished in this thesis. Next the LLP with buffering is implemented. Several experimental results confirm that the plant under the supervision of LLP supervisor with buffering can match the behavior of the plant under the supervision of standard supervisor.

To our knowledge, this is the first implementation of LLP on a real-world system.

In summary, the main contributions of this thesis are as follows:

1. The introduction of Limited Lookahead Policy with Buffering and the corresponding design process to eliminate the delay caused by online LLP calculations of control commands.

2. Improving and polishing the implementation of the conventional SCT.

3. Implementing LLP with Buffering policy on the solar tracker and performing experimental evaluations.

1.6 Thesis Outline

The outline of the thesis is as follows. In Chapter 1, a brief introduction of supervisory control was provided and the related research was reviewed. Background information on supervisory control and the notation used throughout the thesis is discussed in Chapter 2. Details of the solar tracker system and the offline method of SCT design and implementation along improvements are explained in Chapter 3. The novel method of Limited Lookahead Policy with Buffering and its implementation are set out in detail in Chapter 4. Chapter 5 presents the implementation results of LLP with Buffering and compares them with the theoretical analysis of Chapter 4. Further the results of LLP with Buffering is also compared with those of the conventional supervision of Chapter 3. Chapter 6 discusses our conclusions and suggests directions for future research.

Chapter 2

Background

2.1 Discrete Events Systems

Discrete Event System [41], [31] presentation of any real-world system is based on discrete mathematics. The states in DES change as a result of occurrence of events. Thus, the entire behavior of the DES system can be shown as sequences of these events. In order to facilitate the discussion, the following definitions of automata and languages are used.

2.1.1 Languages

An **alphabet** is a finite set of symbols which is denoted by Σ**.** Each symbol represents an event. A sequence of these events is called a **trace**, **string** or **word**. The ϵ symbol is for an empty string. A set of strings over an alphabet Σ is called a **language**. The language of all finite sequences except the empty string is defined as:

$$\Sigma^{+} = \{\sigma_1\sigma_2 \dots \sigma_k \mid k > 0, \sigma_i \in \Sigma\}$$

After adding the empty string, Σ^* is obtained:

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

A language is a set of strings, and thus for any two languages L_1 and L_2, set operations can be applied ($e.g.$ $L_1 \cap L_2$, the intersection of L_1 and L_2, $L_1 \cup L_2$, the union of L_1 and L_2, L_1 - L_2 the difference of L_1 and L_2, and L_1^{co} the complement of L_1).

Moreover, for a string s=tuv with t, u, v $\in \Sigma^*$, t is called a **prefix** of s, u is called a **substring** of s, and v is called a **suffix** of s. The notation s/t is to denote the suffix of s after its prefix t, i.e., uv.

2.1.2 Operations on Languages

Let L_1, L_2 be two languages and L_1 and $L_2 \subseteq \Sigma^*$.

Definition 2.1: Concatenation

$$L_1L_2 = \{s \in \Sigma^* \mid \exists\, s_1 \in L_1, \exists\, s_2 \in L_2 \; such\ that\ s = L_1L_2\}$$

It means that any string in concatenation of L_1 and L_2 (L_1L_2) is a concatenation of a string in L_1 with a string in L_2 .

Definition 2.2: Prefix-closure. For $L \subseteq \Sigma^*$,

$$\bar{L} = \{s \in \Sigma^* \mid \exists t \in \Sigma^* \text{ such that } st \in L\}$$

The prefix-closure of L, denoted by $\bar{L}$, is all prefixes of every string in L.

L is a prefix-closured language if $L = \bar{L}$.

Definition 2.3: Kleene-closure. For $L \subseteq \Sigma^*$

$$L^* = \{\varepsilon\} \cup L \cup L\,L \cup L\,L\,L \cup \ldots$$

The Kleene-closure of L, denoted by L^*, is formed by concatenation of any finite number of strings of L and also includes the empty string.

Definition 2.4: Post-language. For $L \subseteq \Sigma^*$, $s \in \bar{L}$

$$L/s = \{t \in \Sigma^* \mid st \in L\}$$

The post-language of L after s which is denoted by L/s, is all suffixes of string s in L.

Definition 2.5: Truncation. For $L \subseteq \Sigma^*$, $N \in \mathbb{N}$,

$$L|_N = \{t \in L \mid |t| \leq N\}$$

The truncation of L to $N \in \mathbb{N}$ which is denoted by $L|_N$ contains all strings of L with a length of at most N.

2.1.3 Automata

An automaton is a tool to present languages according to specific rules. A deterministic automaton is a five-tuple

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

where

X is the set of states,

Σ is the finite set of events,

$\eta : X \times \Sigma \rightarrow X$ is a partial transition function,

x_0 is the initial state and

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

An automaton (generator or state machine) is considered to be deterministic if the destination of the transition with a particular event from any state is always a single state.[31]

Definition 2.6: Language generated by G

Language generated by G is denoted by $L(G)$ and is defined as

$$L(G) = \{s \in \Sigma^* \mid \eta(x_0, s)!\}$$

$\eta(x_0, s)!$ means there exists a trajectory in the automaton from the initial state following the string s.

In other words, the language generated by G includes all strings from the initial state which lead to same state of G (includes x_0). In the above definition, an extension of transition function η to sequences ($\eta : X \times \Sigma^* \rightarrow X$) is used. By definition, L(G) is a prefix-closed language because it contains all string in any path from the initial state to other states.

Definition 2.7: Language marked by G

Language marked by G is denoted by $L_m(G)$ and defined as

$$L_m(G) = \{s \in L(G) \mid \eta(x_0, s) \in X_m\}$$

The marked language of G includes all strings that take a state from the initial state to some marked state. Obviously $L_m(G)$ is a sublanguage of $L(G)$.

2.1.4 Operations on Automata

Definition 2.8: Reachable part (Accessible part)

Let $G = (X, \Sigma, \eta, x_0, X_m)$. The reachable states of G is the state set which can be reached from the initial state by at least one string $s \in L(G)$. In other words, all $x \in X$ are reachable if there is a string $s \in L(G)$ such that $\eta(x_0, s) = x$. X_r is set of all reachable states (The reachable part of G is the subautomaton of G that contains the state in X_r only). An automaton is called reachable if $X_r = X$.

Definition 2.9: Coreachable part (Coaccessible part)

Let $G = (X, \Sigma, \eta, x_0, X_m)$. The coreachable states of G is the state set which have access to marked states through some string. In other words, a state $x \in X$ is coreachable if there is a string $s \in \Sigma^*$ such that $\eta(x, s) \in X_m$ (The coreachable part of G is the subautomaton of G that contains only the coreachable states of G).

Definition 2.10: Nonblocking

An automaton G is nonblocking if for any reachable state, there is a string to a marked state. One can easily see that automaton G is nonblocking if and only if $L(G) = \overline{L_m(G)}$.

Definition 2.11: Trim

The trim operation on an automaton removes all states which are not reachable or not coreachable. The state set of trimmed automaton is denoted by X_{tr} ($X_{tr} = X_r \cap X_{cr}$).

Definition 2.12: Complement

Let $G = (X, \Sigma, \eta, x_0, X_m)$. $L(G)$ and $L_m(G)$, the closed and marked languages of G. The complement of G denoted by G^{co} is defined as a automaton that generates Σ^* and marks $L_m(G^{co})$.

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

Definition 2.13: Product

Let $G_1 = (X_1, \Sigma_1, \eta_1, x_{o_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, \eta_2, x_{02}, X_{m_2})$. The product of G_1 and G_2 denoted by $G_1 \times G_2$ is the reachable part of the following automaton

$$(X_1 \times X_2, \Sigma_1 \cap \Sigma_2, \eta, (x_{o_1}, x_{02}), X_{m_1} \times X_{m_2})$$

in which $\eta((x_1, x_2), \sigma) = \begin{cases} (\eta_1(x_1, \sigma), \eta_2(x_2, \sigma)) & if\ \eta_1(x_1,\sigma)!\ \ and\ \ \eta_2(x_2,\sigma)! \\ not\ defined & otherwise \end{cases}$

In the product of two automata, an event can occur in a state if and only if it happens in both automata in their respective state. It means that both automata should be synchronized at the current state to proceed in their product.

Therefore, the language of the resulting product is the intersection of the languages of G_1 and G_2.

$\mathrm{L}(G_1 \text{ x } G_2) = L(G_1) \cap L(G_2)$

$\mathrm{L_m}(G_1 \text{ x } G_2) = \mathrm{L_m}(G_1) \cap \mathrm{L_m}(G_2)$

Definition 2.14: Parallel Composition (Synchronous Product)

To build a model for a system composed of several components, the synchronization product (parallel composition) of the component automata can be used.

Let $G_1 = (X_1, \Sigma_1, \eta_1, x_{o_1}, X_{m_1})$ and $G_2 = (X_2, \Sigma_2, \eta_2, x_{02}, X_{m_2})$. The synchronous product of G_1 and G_2, denoted by $G_1 \parallel G_2$, is the reachable part of

$$(X_1 \times X_2, \Sigma_1 \cup \Sigma_2, \eta, (x_{o_1}, x_{02}), X_{m_1} \times X_{m_2})$$

where $\eta((x_1, x_2), \sigma) =$

$$\begin{cases} (\eta_1(x_1,\sigma), \eta_2(x_2,\sigma)) & if\ \sigma \in (\Sigma_1 \cap \Sigma_2)\ and\ \eta_1(x_1,\sigma)!\ and\ \ \eta_2(x_2,\sigma)! \\ ((\eta_1(x_1,\sigma), x_2) & if\ \ \sigma \in (\Sigma_1 - \Sigma_2)\ \ and\ \ \ \eta_1(x_1,\sigma)! \\ (x_1, \eta_2(x_2,\sigma)) & if\ \ \sigma \in (\Sigma_2 - \Sigma_1)\ \ and\ \ \ \eta_2(x_2,\sigma)! \\ undefined & otherwise \end{cases}$$

It is clear that if $\Sigma_1 = \Sigma_2$ then the product and synchronous product of G_1 and G_2 are equivalent up to renaming of states.

2.2 Supervisory Control

The role of supervisory control is to control a system (assumed to be modeled as automaton) in such a way that the system's behavior meets a set of specifications. Control is done by limiting events at some states. Therefore, S, as a supervisor, observes the events at each state (as active events) and then prevents the occurrence of any events which violate the design specification.

In this context, the events are partitioned to uncontrollable and controllable. The supervisor has the ability to restrict only the controllable events. In general, some events may not be observable. However, in this thesis, we only study the case of full event observation.

2.2.1 Basic Supervisory Control

Consider a system with automaton model of $\mathrm{G} = (X, \Sigma, \eta, x_0, X_m)$. Let $\mathrm{L(G)}$ be the closed language and $\mathrm{L_m(G)}$ be the marked language of G. The supervisor S is a function from $\mathrm{L(G)}$ to the power set of Σ:

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

in which $\Sigma = \Sigma_c \cup \Sigma_{uc}$ and Σ_c is the set of controllable events and Σ_{uc} is the set of uncontrollable events ($\Sigma_c \cap \Sigma_{uc} = \emptyset$).

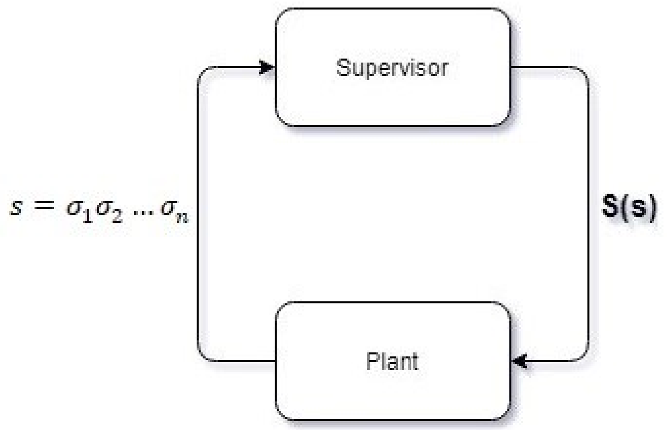


Figure 2.2-1: Control feedback of supervisor

As it is shown in Figure 2.2-1 the supervisor receives the sequence of events (string s), and then allows some events S(s) to happen in the plant. To precisely present supervisor, $\Gamma: \mathrm{X} \rightarrow 2^{\Sigma}$ is defined to provide the active events at each plant. For any $\sigma \in \Sigma, \sigma \in \Gamma(x)$ if and only if $\eta(x, \sigma)!$. Therefore for the string $s \in L(G)$, the set of events that remain enabled will be $(S(s) \cap \Gamma(\eta(x_0, s))$.

The supervisor has no control over uncontrollable events and then $S(s)$ includes all uncontrollable events in the active set of events. In other words,

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

If the above condition is satisfied, the supervisor is called admissible. Let S/G denoted the plant G under the supervision of S.

Definition 2.15: Languages generated and marked by S/G

The language generated by G under the supervision of S denoted by $L(S/G)$ is defined recursively as follows:

1. $\varepsilon \in L(S/G)$
2. $s \in L(S/G)$ and $s\sigma \in L(G)$ and $\sigma \in S(s) \Rightarrow s\sigma \in L(S/G)$

Obviously, $L(S/G)$ is a closed language and $L(S/G) \subseteq L(G)$.Then the language marked by of S/G is defined as: $L_m(S/G) = L(S/G) \cap L_m(G)$.

Definition 2.16: Nonblocking Supervisor

Supervisor S controlling G is a nonblocking supervisor if and only if S/G is nonblocking ($\overline{L_m(S/G)} = L(S/G)$).

As mentioned earlier, the supervisor restricts the behavior of plant to a safe (legal) sublanguage of $L(G)$, denoted by L_a. The specification is considered as a legal closed behavior of the DES and let L_{am} be the legal marked behavior ($\overline{L_{am}} = L_a$). Therefore, the language generated by applying the supervisor must be a subset of this specification.

$$L(S/G) \subseteq L_a$$

In the case when nonblocking is an issue then S/G must satisfy the following:

$$\begin{cases} L_m(S/G) \subseteq L_{am} \\ \overline{L}_m(S/G) = L(S/G) \end{cases}$$

Definition 2.17: Controllability [31]

Let K and $M = \overline{M}$ be languages over event set Σ and $\Sigma_{\mathrm{uc}} \subseteq \Sigma$. K is said to be controllable with respect to M and Σ_{uc} if

$$\overline{K}\Sigma_{\mathrm{uc}} \cap M \subseteq \overline{K} \; .$$

Therefore $K \subseteq \Sigma^*$ is controllable with respect to $\mathrm{L(G)}$ and Σ_{uc} if

$$\forall \mathrm{s} \in \overline{K} \text{ and } \sigma \in \Sigma_{\mathrm{uc}} \text{ and } \mathrm{s}\sigma \in \mathrm{L(G)} \Rightarrow \mathrm{s}\sigma \in \overline{K}$$

It is noticed that K is not necessarily a subset of $\mathrm{L(G)}$ and K is controllable if and only if $\overline{K}$ is controllable.

Theorem 2.1: Controllability Theorem [31]

Consider an automaton $\mathrm{G} = (X, \Sigma, \eta, \mathrm{x}_0, X_m)$ and let $\Sigma_{\mathrm{uc}} \subseteq \Sigma$ be the uncontrollable event set. For $K \subseteq \mathrm{L(G)}$ $(K \neq \emptyset)$ there exists a supervisor S such that $\mathrm{L\,(S/G)} = \overline{K}$ if and only if K is controllable with respect to $\mathrm{L(G)}$ and Σ_{uc}.

Up to now, the supervisor is defined as a control action over a string which is not a convenient way of deploying a supervisor. A realization of the supervisor by building an automaton which represents the supervisor can be done as follows.

Consider Theorem 2.1. Let R be an automaton that marks the language $\overline{K}$ with ($\mathrm{R} = (\mathrm{Y}, \Sigma, \delta, \mathrm{y}_0, Y_m)$). All states are marked so $\mathrm{L(R)} = \mathrm{L_m(R)} = \overline{\mathrm{K}}$. Then just by forming the product of R and G, the automaton of a closed-loop system of $\mathrm{S/G}$ is built.

Let $\mathrm{C}(K)$ be the class of controllable sublanguages of K: $\mathrm{C}(K) = \{L \subseteq K \mid \overline{L}\Sigma_{\mathrm{uc}} \cap M \subseteq \overline{L}\}$. There exists supremal controllable sublanguage of K, denoted by SupC (K).

$$\mathrm{SupC\ (K)} = \bigcup_{\mathrm{L} \in \mathrm{C(K)}} L$$

One of the most important properties of $\mathrm{SupC(.)}$ is that if K is a closed language then $\mathrm{SupC(K)}$ has to be a closed language too.

BSCP: Basic Supervisory Control Problem

Consider automaton $\mathrm{G} = (X, \Sigma, \eta, \mathrm{x}_0, X_m)$ and legal language of $\mathrm{L_a} = \overline{\mathrm{L_a}} \subseteq L(G)$. A supervisor S with following properties is required:

1. $\mathrm{L}\,(S/\mathrm{G}) \subseteq \mathrm{L_a}$
2. If there is another supervisor which $\mathrm{L}\,(S_{other}/\mathrm{G}) \subseteq \mathrm{L_a}$ then $\mathrm{L}\,(S_{other}/\mathrm{G}) \subseteq \mathrm{L}\,(S/\mathrm{G})$.

While the first condition is related to the safety of the supervisor, the second one asks the supervisor to cover all possible solutions, that is, the supervisor needs to be optimal or minimally restrictive. It follows from Theorem 2.1 that S exists and $\mathrm{L}\,(S/\mathrm{G}) = \mathrm{SupC}(\mathrm{L_a})$.

Theorem 2.2: Nonblocking Controllability Theorem [31]

Let $K \subseteq \mathrm{L_m}(\mathrm{G})$ $(K \neq \emptyset)$. There exists a nonblocking supervisor S such that $\mathrm{L}\,(\mathrm{S}/\mathrm{G}) = \overline{K}$ and $\mathrm{L_m}\,(\mathrm{S}/\mathrm{G}) = K$ if and only if the following two conditions are met:

1. K is controllable with respect to $\mathrm{L(G)}$ and Σ_{uc} ,
2. K is $\mathrm{L_m}(\mathrm{G})$-closed that means $K = \overline{K} \cap \mathrm{L_m}(\mathrm{G})$.

BSCP-NB: Basic Supervisory Control Problem- Non-Blocking case

Consider an automaton $\mathrm{G} = (X, \Sigma, \eta, \mathrm{x}_0, X_m)$ and the legal language of $\mathrm{L_{am}} \subseteq \mathrm{L_m}(\mathrm{G})$. Suppose $\mathrm{L_{am}}$ is $\mathrm{L_m}(\mathrm{G})$-closed. A supervisor S with following properties is required:

1. $\mathrm{L_m}\,(\mathrm{S}/\mathrm{G}) \subseteq \mathrm{L_{am}}$
2. S/G is nonblocking
3. $\mathrm{L_m}\,(\mathrm{S}/\mathrm{G})$ is the largest it can be.

By using Theorem 2.2, the solution exists and is characterized by $\mathrm{L_m}\,(S/\mathrm{G}) = \mathrm{SupC}(\mathrm{L_{am}})$ and $\mathrm{L}\,(S/\mathrm{G}) = \overline{\mathrm{Supc}(\mathrm{L_{am}})}$. The solution is minimally restrictive.

In [32] an algorithm to build an automaton for marking $\mathrm{Supc}(\mathrm{L_{am}})$ based on the plant and the specification automata is introduced.

2.2.2 Limited Lookahead Supervisory Control

In the previous section, the objective was to build an offline supervisor (conventional supervisor), hence the resulting supervisor is stored in the computer system and during operation at every state obtained the enabled events are from a lookup table (to be explained in Section 3.6.2).

Contrary to the conventional supervisory control in which all calculations have to be done before controlling the system starts, in LLP the computation of $S(s)$ is done on-the-fly and during operation [1]. The main reason LLP is introduced is that a conventional supervisor may be too large for storing in computer memory. In LLP the control action is calculated for N-step ahead projection after the current state. This N-step window shown by an N-level tree is shown in Figure 2.2-2.

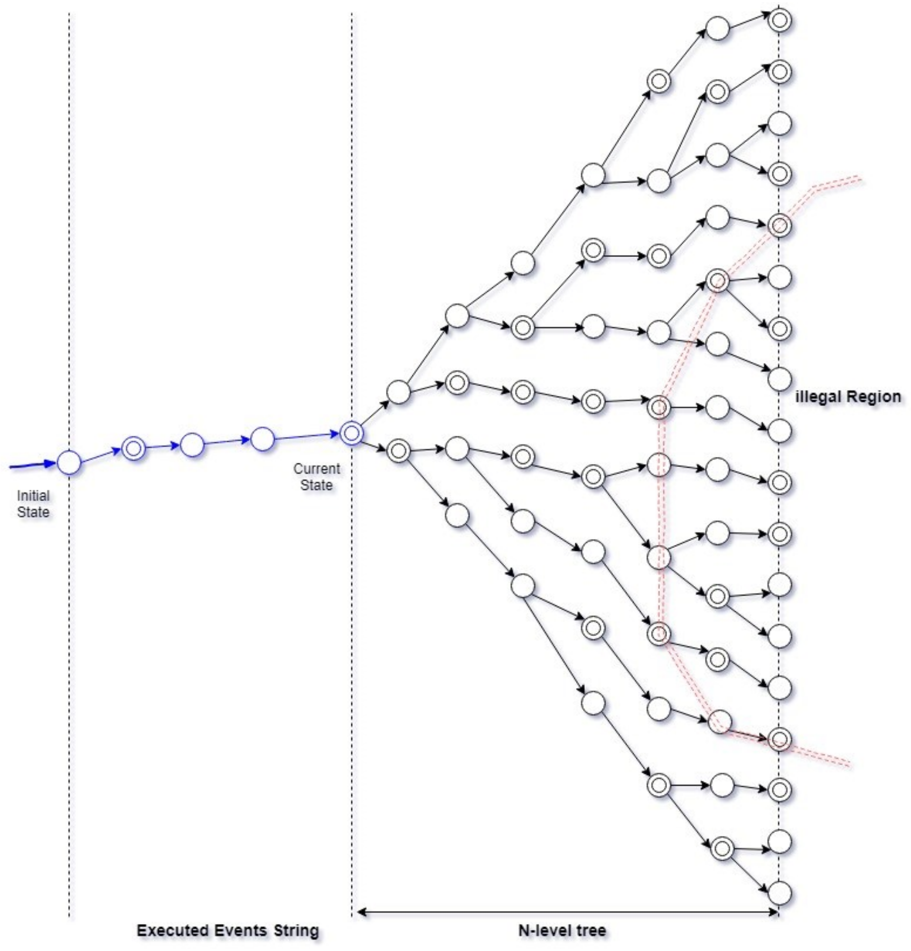


Figure 2.2-2: Limited Lookahead N-level tree

Because of the unexplored states further away of Nth level, the behavior of this level has to be assumed to be legal or illegal with respect to the specification. Therefore, one of two attitudes of optimistic and conservative is chosen before computation of LLP supervisor starts. Pending traces are the traces of length N that are not in the illegal zone ($\overline{K}/s|_N - \overline{K}\ /\ s|_{N-1}$).

Let G be DES over event set Σ and $\Sigma = \Sigma_c \cup \Sigma_{uc}$. We suppose that G is nonblocking: $L(G) = \overline{L_m(G)}$.

Let $K \subseteq L_m(G)$ be a nonempty and $L_m(G)$-closed language which presents the legal behavior of the system ($K \neq \emptyset, K = \overline{K} \cap L_m(G)$).

Like offline supervisor, in LLP, the supervisor S, should control the system under supervision to meet the legal specification and be nonblocking ($L_m(S/G) \subseteq K$ and $L(S/G) = \overline{L_m(S/G)}$).

The online control function is defined in five steps as follows:

1. The first step is to make a subsystem as is shown in Figure 2.2-3 for the N-level tree. In function of block $f^N_{L(G)}$, the post languages of L(G), $L_m(G)$ are taken from the DES G after the current string s:

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

2. Then the illegal part of step 1 should be removed (function f^K_N):

$$f^K_N \text{ o } f^N_{L(G)}(s) = (\overline{K}/s|_N\, , K/s|_N)$$

3. In this step, the result of prior step needs to be adopted according to one of the attitudes which is decided for the supervisor. In optimistic attitude, all pending traces are assumed to be legal and marked and in the conservative attitude pending traces are considered as illegal. The corresponding function is denoted by f^N_a (K assumed to be closed):

$$f^N_a \text{ o } f^K_N \text{ o } f^N_{L(G)}(s) = \begin{cases} conservative & K/s|_{N-1} \\ optimistic & K/s|_N \end{cases}$$

4. Now the supremal controllable sublanguage of the result of step 3 (f^N_a) as the specification for the plant given by $f^N_{L(G)}(s)$ and Σ_{uc} as uncontrollable events is calculated which is denoted by $f^N_\uparrow$.

$$\mathrm{f}^{\mathrm{N}}(s) = \mathrm{f}_{\uparrow}^{\mathrm{N}} \text{ o } \mathrm{f}_{\mathrm{a}}^{\mathrm{N}} \text{ o } \mathrm{f}_{\mathrm{N}}^{\mathrm{K}} \text{ o } \mathrm{f}_{\mathrm{L(G)}}^{\mathrm{N}}(s) = [\ \mathrm{f}_{\mathrm{a}}^{\mathrm{N}} \text{ o } \mathrm{f}_{\mathrm{N}}^{\mathrm{K}} \text{ o } \mathrm{f}_{\mathrm{L(G)}}^{\mathrm{N}}(s)]^{\uparrow/s|_N}$$
$$= \mathrm{SupC}\left(\ \mathrm{f}_{\mathrm{a}}^{\mathrm{N}} \text{ o } \mathrm{f}_{\mathrm{N}}^{\mathrm{K}} \text{ o } \mathrm{f}_{\mathrm{L(G)}}^{\mathrm{N}}(s)\right)$$

(For $L \subseteq \mathrm{L(G)}$, the supremal controllable sublanguage of L with respect to $\mathrm{L(G)}/s|_N$ is denoted by $(\mathrm{L})^{\uparrow/s|_N}$).

5. Finally, the control action is generated by the union of the active events from the former step and the uncontrollable events of the current active events set of the plant ($\Sigma_{L(G)}(s)$ step 1). This function is denoted by $\mathrm{f}_{\mathrm{u}}^{\mathrm{N}}$:

$$\gamma^{\mathrm{N}}(s) = \mathrm{f}_{\mathrm{u}}^{\mathrm{N}}\ o\ \mathrm{f}^{\mathrm{N}}(s) = \overline{f^N(s)}|_1 \cup \overline{\Sigma_{uc} \cap \Sigma_{L(G)}(s)}$$

All the steps of LLP calculation are depicted in Figure 2.2-3. Let the closed behavior of the controlled plant be denoted by $\mathrm{L}(\mathrm{G}, \gamma^{\mathrm{N}})$ with γ^{N} being the control policy ($\gamma^{\mathrm{N}}: \mathrm{L(G)} \rightarrow 2^{\Sigma \cup \{\varepsilon\}}$).

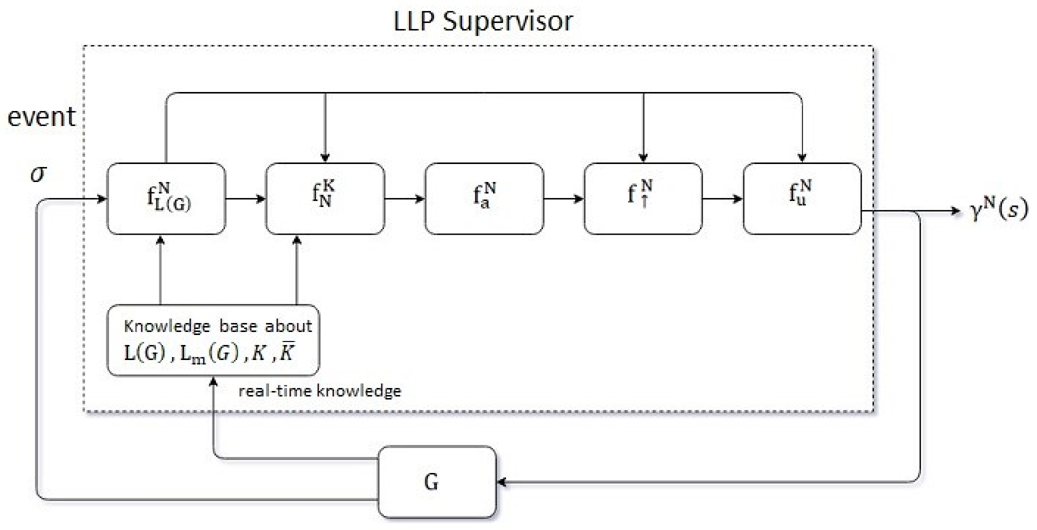

LLP Supervisor
event
σ
$\mathrm{f}_{\mathrm{L(G)}}^{\mathrm{N}}$
$\mathrm{f}_{\mathrm{N}}^{\mathrm{K}}$
$\mathrm{f}_{\mathrm{a}}^{\mathrm{N}}$
$\mathrm{f}_{\uparrow}^{\mathrm{N}}$
$\mathrm{f}_{\mathrm{u}}^{\mathrm{N}}$
$\gamma^{\mathrm{N}}(s)$
Knowledge base about
$\mathrm{L(G)}, \mathrm{L_m}(G), K, \bar{K}$
real-time knowledge
G

Figure 2.2-3: LLP supervisor block diagrams

To compare the result of LLP with conventional supervisor, we define the notion of validity and run-time error. Although due to some constraints LLP calculation is performed on a limited window of the plant, our objective is to make it complete the LLP supervisor have the same performance as the conventional supervisor. The result of a "valid" LLP supervisor is the same as the conventional supervisor. However, because of the limited horizon for LLP, sometimes the

$\mathrm{f}^{\mathrm{N}}(s)$ (supremal controllable sublanguage) becomes empty and the continuation of LLP is no longer possible which is called a "run-time error".

Definition 2.18: Validity [1]

An LLP supervisor with the control policy γ^{N} (γ^{N} hereafter as a supervisor) is called valid if $\mathrm{L}(\mathrm{G}, \gamma^{\mathrm{N}}) = \overline{\mathrm{SupC}(K)}$.

This means online and offline supervisors have the same control action for valid supervisor.

Definition 2.19: Run-time error (RTE) and starting error [1]

For a string $s \in \mathrm{L}(\mathrm{G}, \gamma^{\mathrm{N}})$ if $\mathrm{f}^{\mathrm{N}}(s) = \emptyset$, a Run-Time Error (RTE) has happened for string s.

If $s = \{\varepsilon\}$ and $\mathrm{f}^{\mathrm{N}}(s) = \emptyset$, an Starting Error (SE) has occurred.

The LLP supervisor should always have nonempty result for $\mathrm{f}^{\mathrm{N}}(s)$ in order to avoid being trapped in blocking or illegal region by an uncontrollable event.

Proposition 2.1 [1]:

The validity of supervisor γ^{N} is equivalent to the following statements:

1. For all strings of $s \in \mathrm{L}(\mathrm{G}, \gamma^{\mathrm{N}})$, $\gamma^{\mathrm{N}}(s) = \overline{SupC(K)}|_1$
2. $SupC(K) \neq \emptyset$ and ($\forall s \in \overline{SupC(K)}$) $\gamma^{\mathrm{N}}(s) = \overline{SupC(K)}|_1$

It can be seen from step 3 of LLP supervisory building blocks that one of two attitudes should be chosen and therefore different consequences are imposed on the resulting supervisor. While in the optimistic attitude by assuming all pending traces as marked, by going further illegal behavior or blocking may be encountered. In the conservative case which assumed pending traces as illegal, only marked legal strings may be found in the next steps.

Theorem 2.3 [1]: $\mathrm{L}(\mathrm{G}, \gamma_{\mathrm{opt}}^{\mathrm{N+1}}) \subseteq \mathrm{L}(\mathrm{G}, \gamma_{\mathrm{opt}}^{\mathrm{N}})$

Theorem 2.4 [1]: $\mathrm{L}(\mathrm{G}, \gamma_{\mathrm{cons}}^{\mathrm{N}}) \subseteq \mathrm{L}(\mathrm{G}, \gamma_{\mathrm{cons}}^{\mathrm{N+1}})$

Furthermore, since in the optimistic case, maximum freedom is given to the supervisor, it is expected that the language of plant under control would be larger than offline result, as it is shown in the next theorem.

Theorem 2.5 [1]: $\overline{SupC(K)} \subseteq \mathrm{L}(\mathrm{G}, \gamma_{\mathrm{opt}}^{\mathrm{N}})$

In the conservative case, because of considering worst-case-scenario, the language of plant under control is smaller than the offline result, which results in the following theorem.

Theorem 2.6 [1]: $SupC(K) \neq \emptyset$ if and only if $\mathrm{L}(\mathrm{G}, \gamma_{cons}^{\mathrm{N}}) \subseteq \overline{SupC(K)}$

The window size N plays a crucial role in the LLP supervisory control. Apart from external issues which affect the selection of N , it is desired to choose a value for N to reach the same results as the offline supervisor to guarantee the validity of LLP supervisor. To obtain the optimal N in terms of validity, two possibilities for the specification as a legal behavior are considered.

1. $K = \overline{K}$

Lemma 2.1 [1]:

Let $K = \overline{K}$. If there is no RTE in $\mathrm{L}(\mathrm{G}, \gamma_{\mathrm{opt}}^{\mathrm{N}})$, then $\gamma_{\mathrm{opt}}^{\mathrm{N}} = SupC(K)$.

The longest substring of the uncontrollable events in language L is defined as:

$$\mathrm{N_u}(\mathrm{L}) = \begin{cases} \max\{|s| : s \in \Sigma_{uc}^{*} \; and \; (\exists u, v \in \Sigma^{*}) usv \in \mathrm{L} & \text{if it exists.} \\ undefined & otherwise \end{cases}$$

Lemma 2.2:

Let $K = \overline{K}$. If $\mathrm{N} \geq \mathrm{N_u}(K) + 2$ or $\mathrm{N} \geq \mathrm{N_u}(\mathrm{L}(\mathrm{G})) + 1$, then there is no RTE in $\mathrm{L}(\mathrm{G}, \gamma_{\mathrm{opt}}^{\mathrm{N}})$.

From lemma 2.1 and 2.2 the following theorem is concluded.

Theorem 2.7:

Let $K = \overline{K}$. If $\mathrm{N} \geq \mathrm{N_u}(K) + 2$ or $\mathrm{N} \geq \mathrm{N_u}(\mathrm{L}(\mathrm{G})) + 1$, then $\mathrm{L}(\mathrm{G}, \gamma_{\mathrm{opt}}^{\mathrm{N}}) = SupC(K)$.

In conservative case there is no relation between RTE and validity of the supervisor, but by forming closed language for $\mathrm{L}(\mathrm{G}, \gamma_{cons}^{\mathrm{N}})$, the condition for validity can be extracted as follows.

Theorem 2.8:

Let $K = \overline{K}$. If there is no SE in $\mathrm{L}(\mathrm{G}, \gamma_{cons}^{\mathrm{N}})$ and $K \cap \Sigma_{uc}^{N-1} = \emptyset$ then

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

Then from above theorem, the following corollary can be inferred.

Corollary 2.1:

Let $K = \overline{K}$. If there is no SE in $\mathrm{L}(\mathrm{G}, \gamma_{cons}^{\mathrm{N}})$ and if $\mathrm{N} \geq \mathrm{N_u}(K) + 2$, then

$$\mathrm{L}(\mathrm{G}, \gamma_{cons}^{\mathrm{N}}) = SupC(K).$$

Obviously, $\mathrm{N_u}(\mathrm{L}(\mathrm{G})) \geq \mathrm{N_u}(K)$. Therefore $\mathrm{N_u}(\mathrm{L}(\mathrm{G})) + 2$ is sufficient for validity. Moreover, if a language is closed, its supremal controllable sublanguage is closed too.

2. $K \subseteq \overline{K}$

In this case, nonblocking has to be verified and so larger N is needed to make an LLP supervisor valid. To define minimum length of N, the following terms are defined.

Definition 2.20:

$$K_{mc} = \{ s \in K \mid \forall \sigma \in \Sigma_{uc}, s\sigma \notin \mathrm{L}(\mathrm{G})\}$$

K_{mc} denotes all marked strings of K which have just controllable events in their active events set.

Definition 2.21:

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

$K_{f\bar{c}}$ denotes all traces which bridge from legal zone to the illegal zone by just the execution of uncontrollable events.

Since in the optimistic attitude, all pending traces are marked, then the legality and marking of traces beyond the boundary have to be examined.

Definition 2.22:

$$\mathrm{N}_{mcf\bar{c}} = \begin{cases} \max\{|t| : \exists s \in K_{mc} \cup \{\varepsilon\}\ (st \in K_{f\bar{c}}\ and\ (\forall\ \varepsilon < v < t) sv \notin K_{f\bar{c}} \cup K_{mc} & \text{if it exists.} \\ undefined & otherwise \end{cases}$$

$\mathrm{N}_{mcf\bar{c}}(\mathrm{L})$ is the maximum string length which begins from the initial or legal marked state with just controllable events and leads to the illegal zone and it has no prefix in the legal marked with just controllable events or illegal zone. Therefore neither nonblocking nor safety can be guaranteed by this length and the window size must be larger than it.

Theorem 2.9:

Let $SupC(K) \neq \emptyset$. If $\mathrm{N} > \mathrm{N}_{mcf\bar{c}} + 1$, then $\mathrm{L}(\mathrm{G}, \gamma_{\mathrm{opt}}^{\mathrm{N}}) = \overline{SupC(K)}$.

In the conservative attitude since all pending traces are supposed to be illegal, the marking of the traces after the boundary needs to be checked.

Definition 2.23:

$$\mathrm{N}_{mcmc} = \begin{cases} \max\{|t| : \exists s \in K_{mc} \cup \{\varepsilon\}\ (\mathrm{st} \in K_{mc}\ and\ (\forall\ \varepsilon < v < t) sv \notin K_{mc} & \text{if it exists.} \\ undefined & otherwise \end{cases}$$

N_{mcmc} is the longest string from the initial or marked legal state with just controllable events which leads to another marked legal state with just controllable events without any prefix with the same properties.

Theorem 2.10 [1]:

Suppose $\overline{K} = \overline{K_{mc}}$ and there is no SE in $\mathrm{L}(\mathrm{G}, \gamma^{\mathrm{N}}_{cons})$. If $\mathrm{N} \geq \mathrm{N}_{mcmc} + 1$, then $\mathrm{L}(\mathrm{G}, \gamma^{\mathrm{N}}_{cons}) = \overline{SupC(K)}$.

The assumption is that no legal marked state has an uncontrollable event in its active events set. It can be shown that when $\overline{K} = \overline{K_{mc}}$ and $SupC(K) \neq \emptyset$, $\mathrm{N}_{mcmc} \geq \mathrm{N}_{mcf\bar{c}}$. Therefore, the $\mathrm{N} \geq \mathrm{N}_{mcmc} + 1$ is a general sufficient condition in both optimistic and conservative attitudes which means the longest string between the two marked legal states with just controllable events determine the minimum window size for validity.

2.2.3 State-Based Limited Lookahead Supervisory Control [27]

In the previous section, it was assumed that the N steps ahead of current state as N successive events are known for the LLP supervisor (we call this event-based LLP). In that approach, no information about plant state is used; and it leads to form an *N*-level tree to see the system after the current state. To simplify the supervisor computation and to find the optimum window size when there is an uncontrollable loop inside the system (which makes minimum window size for validity unbounded), the state information is added to the supervisory synthesis. This is called state-based LLP. As in most cases the model of plant and legal behavior exist in form of automaton, this information is already ready for extraction.

Definition 2.24:

The set of strings in $\mathrm{L}(\mathrm{G})$ from the initial state x_0 leading to the state x is denoted by $[x]$.

Definition 2.25:

$\mathrm{X_{mc}}$ is the set of marked states that only have controllable events.

$\mathrm{X_{mc}} = \{\, x \in \mathrm{X_m} \mid \Sigma_{\mathrm{G}(x)} \subseteq \Sigma_c \}$

($\Sigma_{\mathrm{G}(x)}$ is active events set at state x).

Definition 2.26:

For an automaton $\mathrm{G} = (X, \Sigma, \eta, \mathrm{x_0}, X_m)$

$$\omega(x, \mathrm{s}) = \begin{cases} \{\eta(x,t) \mid t \leq \mathrm{s}\} & if\ (\forall\, t \leq \mathrm{s})\ \eta(x,t) \notin \mathrm{X_{mc}}\ and\ \eta(x,t) \in X_H \\ \emptyset & otherwise \end{cases}$$

H is a subautomaton of G which marks the specification K ($L_m(H) = K$). If H is not a subautomaton of G , it should be transformed according to the procedure in [33] .

$\omega(x, \mathrm{s})$ is the set of states which are reached from x and they are accessible through a sequence of states none of which belongs to the marked controllable or illegal states.

To define the bound for the validity of supervisor, $\mathrm{N_B}$ is defined as follows:

Definition 2.27:

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

Since there is no sign of any attitudes in the above definition, it means it is valid for optimistic and conservative attitudes.

2.3 Discrete Event Control Kit (DECK)

Discrete Event Control Kit (DECK) is a toolbox written in the programming language of MATLAB [34] for the analysis and design of supervisory control systems based on discrete-event models.

In this section, the functions in DECK which will be used in the next chapter for supervisory computation are explained.

2.3.1 Automaton

A DES model is defined as the class “Automaton” in DECK.

G=automaton(N,TL,Xm)

Inputs:

N Number of states,

TL Transition list,

Xm Marked states (row vector),

Outputs:

G Output automaton,

For example, the automaton G in Figure 2.3-1 has the following arguments.

G=automaton(N,TL,Xm),

N=5,

TL= [1 α 2; 2 β 3; 3 α 2; 3 γ 1; 3 β 4; 4 γ 5],

Xm=[3 4],

(The events must be labeled with numbers. For simplicity, we use the original Greek labels.)

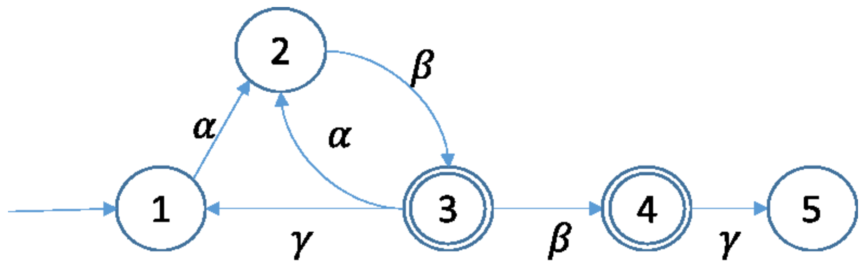


Figure 2.3-1: Automaton G

2.3.2 Reach

This function finds the reachable states through the transitions list from the source state set.

Xr=reach(TL,S)

Inputs:

TL Transition list,

S Source states (vector),

Outputs:

Xr States reachable from S (row vector),

Consider the automaton G in Figure 2.3-1, Xr=reach(G.TL,[1]) generates the reachable state set as Xr=[1 2 3 4 5].

2.3.3 Reachable

This function returns the reachable part of an automaton as a subautomaton (see ***Definition 2.8)***.

[Gr,Xr]=reachable(G)

Inputs:

G Input automaton,

Outputs:

Gr Reachable subautomaton,

Xr Reachable states of G (row vector),

As all states of automaton G in Figure 2.3-1 are reachable, Gr=G and Xr=[1 2 3 4 5].

2.3.4 Trim

Trim returns the reachable and coreachable (***Definition 2.9***) part of an automaton (***Definition 2.11)***.

[Gt,Xrc]=trim(G)

Inputs:

G Input automaton,

Outputs:

Gt Trim subautomaton,

Xrc States of G that are reachable and coreachable (row vector).

For example, since state 5 of automaton G in Figure 2.3-1 is not coreachable, Xrc=[1 2 3 4] and Gt=automaton(4, [1 α 2; 2 β 3; 3 α 2; 3 γ 1; 3 β 4],[3 4]).

2.3.5 Product

This function generates the product of automata (***Definition 2.13).***

[G,States]=product(G1,...,Gn)

Inputs:

Gi Input automaton i (i=1, ..., n),

Outputs:

G Output automaton,

States State set of output automaton,

For example, the product of automaton G in Figure 2.3-1 and H in Figure 2.3-2 results in the automaton P in Figure 2.3-3.

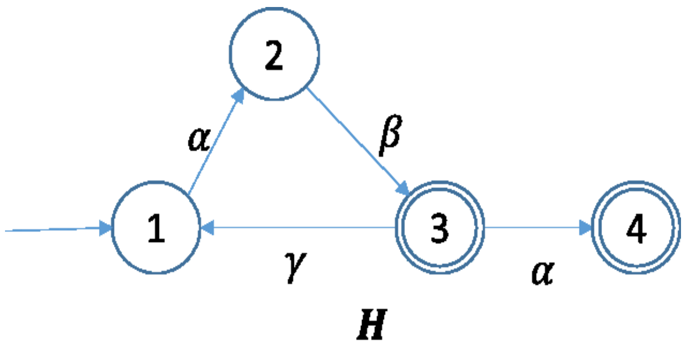


Figure 2.3-2: Automaton H

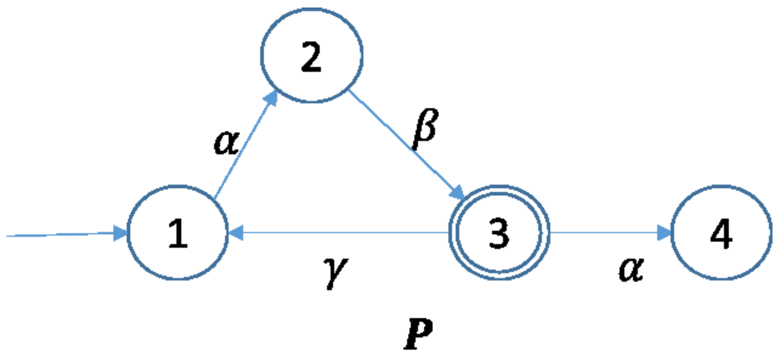


Figure 2.3-3 : Automaton P

H=automaton(4, [1 α 2; 2 β 3; 3 γ 1; 3 α 4],[3 4]),

[P,States]=product(G,H),

P.N=4,

P.TL=[1 α 2; 2 β 3; 3 γ 1; 3 α 4],

P.Xm=[3],

States=[1 1;2 2;3 3;2 4],

As can be seen from automaton P, since event α from state 3 of automaton G reach state 2 which is not marked, then state 4 (state 2 of G and 4 of H) of automaton P is not marked.

2.3.6 Sync

Sync, generates the synchronous product of automata (***Definition 2.14)*** with the following format.

G=sync(G1,...,Gn),

[G,States]=sync(G1,...,Gn),

Inputs:

Gi Input automaton i (i=1, ..., n),

Outputs:

G Output automaton,

States State set of output automaton,

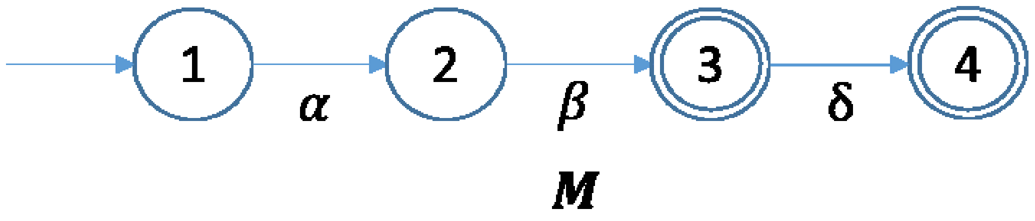


Figure 2.3-4: Automaton M

For example, automaton N (Figure 2.3-5) is the result of synchronous product of automaton P in Figure 2.3-3 and automaton M in Figure 2.3-4, has the following properties:

M=automaton(4, [1 α 2; 2 β 3; 3 δ 4],[3 4]),

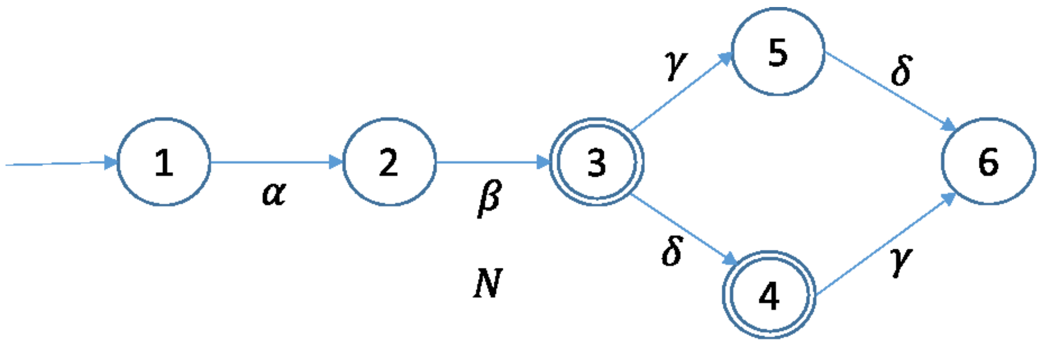


Figure 2.3-5: Automaton N

[N,States]=sync(P,M),

N.N=6,

N.TL=[1 α 2; 2 β 3; 3 δ 4; 3 γ 5; 5 δ 6; 4 γ 6],

N.Xm=[3 4],

States=[1 1;2 2;3 3;3 4;1 3;1 4],

It can be observed that event α is a common event between automaton P and M; therefore this event cannot occur when P and M are both in their state 3 (M cannot execute α in state 3). But γ and δ are not common events and can occur in P and M respectively.

We will discuss the **supcon** function to generate supervisor in Section 3.6.2.

Chapter 3

Two Degree-of-Freedom Solar Tracker

The system for implementation of supervisory control in this thesis is a two degree-of-freedom solar tracker (Figure 3-1) controlled with a microcontroller. In the next sections after presenting the system hardware and software, steps toward offline design (conventional supervisory control) along with the corresponding implementation are explained.

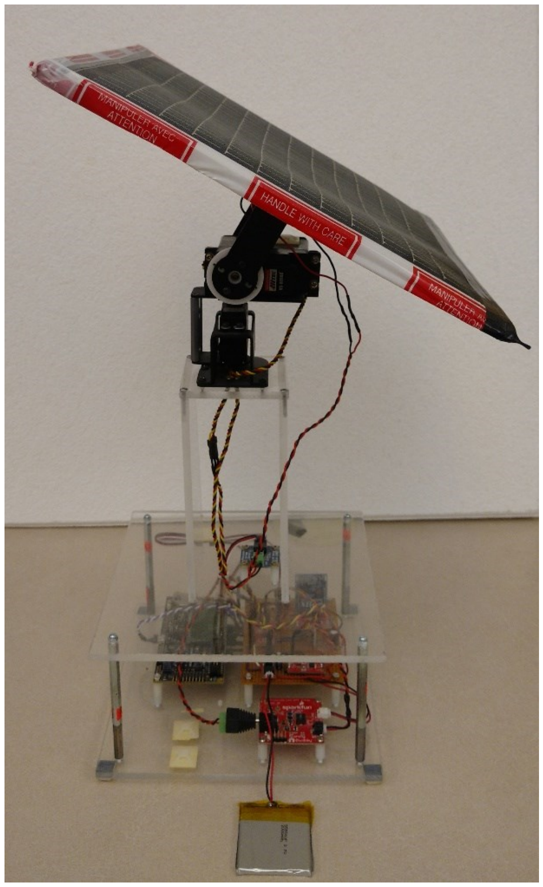


Figure 3-1: Two degree-of-freedom solar tracker

3.1 Schematic Diagram

The schematic diagram of the solar tracker is depicted in Figure 3.1-1. It consists of a Lithium-ion Polymer battery, a photovoltaic (PV) cell, two servomotors for azimuth and elevation directions, and an EFM32™ Leopard Gecko, 32-bit Microcontroller as the processor. The objective of this system is to find the direction in which the brightness is higher than a predefined threshold by searching in azimuth and elevation directions while charging the battery. It has a serial communication port (through a wireless RF module) which sends and receives signals from a PC mimicking a ground station. The tracker can be assumed to be a satellite subsystem in charge of supplying solar energy.

The reason for the selection of this type of microcontroller is that this microcontroller series is ideal for battery operated and low-energy consumption applications. This microcontroller (EFM32LG990F256) is a 32-bit ARM Cortex-M3 processor running at up to 48 MHz and has 256kB flash memory and 32kB RAM which is quite enough for conventional supervisory control implementation. The two degree-of-freedom tracker was developed in [30]. In this thesis, we use the same hardware; however, the software and the design and implementation of supervisory control are novel and are among the contributions of this thesis.

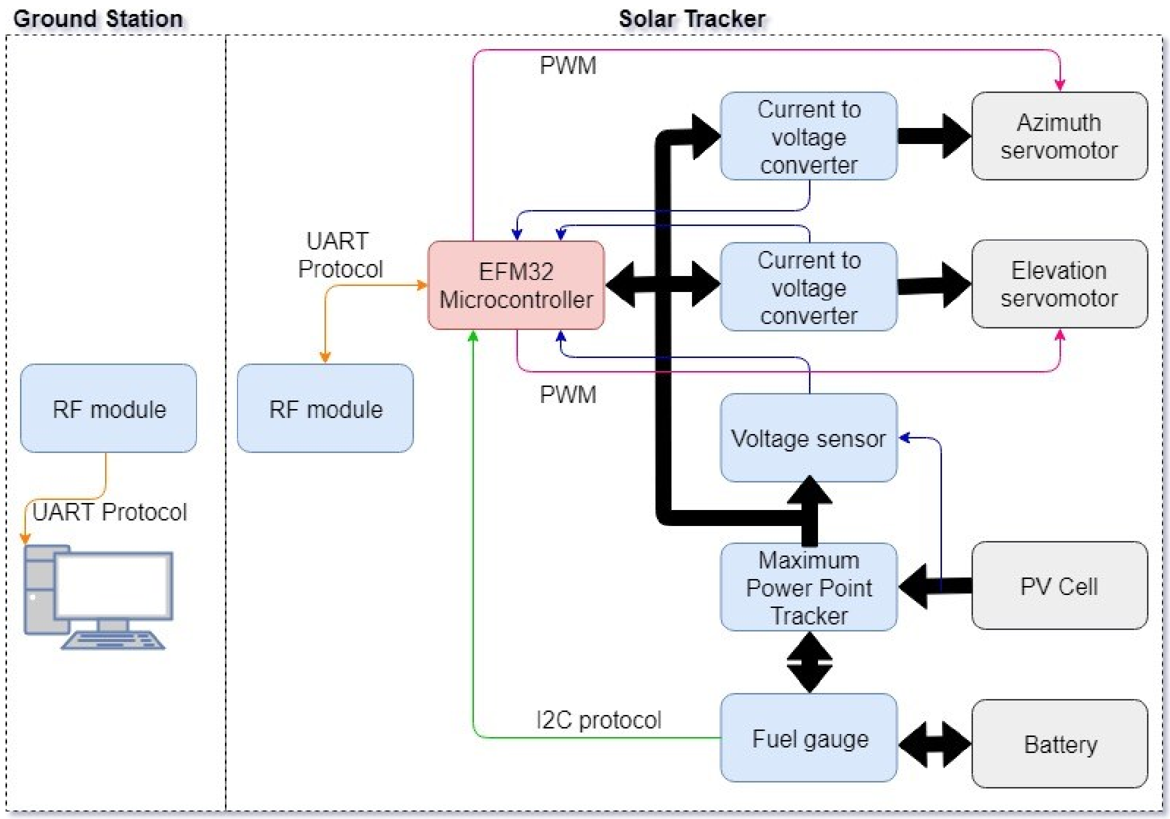


Figure 3.1-1: Solar Tracker schematic diagram

3.2 System Hardware

Since the thesis objective is to implement supervisory control, for brevity, the details of hardware specifications are not discussed. More details can be found in [30], [42]. The data sheets of the hardware components are collected in Appendix E. In the following sections, the main components are explained in brief.

3.2.1 Battery

The battery supplies the entire system energy and is being charged from the PV cell. The battery type is LiPo with a nominal voltage of 3.7 volts and a capacity of 2200 mAh supply. The battery is connected to a fuel gauge which is a Sparkfun LiPo Fuel Gauge to show the state of charge (SOC) and voltage level using the I2C protocol to send these data in percentage and volt respectively to the microcontroller.

3.2.2 PV Cell

The PV cell absorbs sunlight and converts it to electricity to charge the battery and supply the system. The solar cell is a PT15-300 from Flex Solar Cell which can supply at most 3.08 Watts whenever it is exposed to full sunlight.

Since the current of the PV cell decreases when output voltage increases and at a certain point the current drops suddenly, to take the most power of the PV cell, a Maximum Power Point Tracker (MPPT) is used as a DC to DC converter to keep wattage of the PV cell at the top. A SunnyBuddy MPPT from Sparkfun is used in this system for this purpose. As it is shown in Figure 3.1-1, the MPPT is connected to the battery through the fuel gauge to charge the battery. Moreover, to measure the voltage level of the PV cell a Phidgets 1135 precision voltage sensor is used to convert the DC voltage of the PV cell to the range of 0.5-4.5 volts which is in the voltage range of analog to digital (ADC) ports of the microcontroller.

3.2.3 Servomotors

There are two servomotors in the azimuth and elevation directions for adjusting the orientation of the PV cell with respect to the sunbeam. In the spherical coordinate system, the elevation angle is defined as the angle between the object in the space and the observer horizon and azimuth angle is the angle between the north to the elevation vector which is perpendicularly projected to the observer plane reference (Figure 3.2-1).

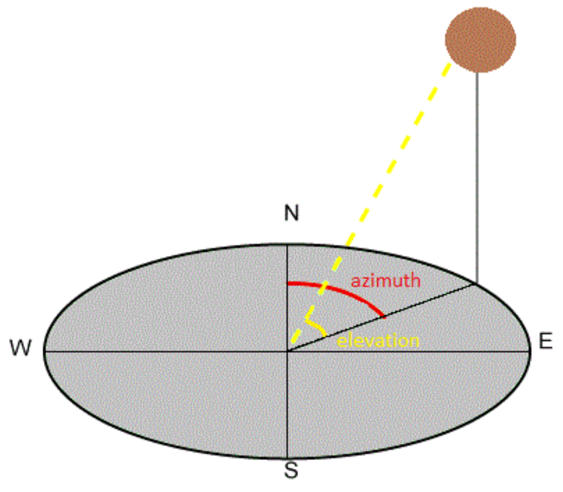


Figure 3.2-1: Azimuth and elevation with respect to the object

In the solar tracker, the motor which changes the direction of PV cell parallel to the fixture is named azimuth and the other servomotor which moves in the plane perpendicular to the fixture is elevation servomotor (Figure 3.2-2).

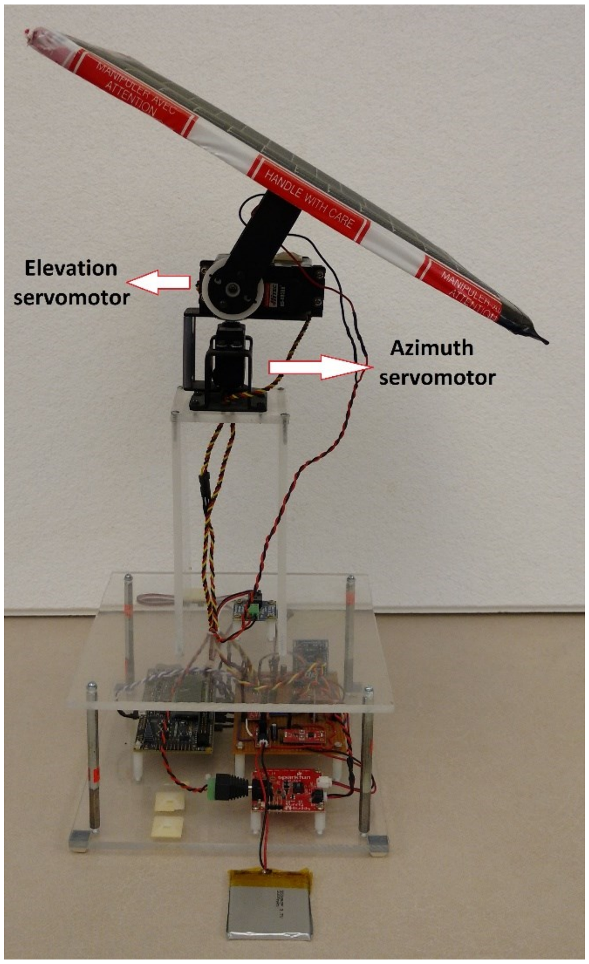


Figure 3.2-2: Azimuth and elevation servomotors of solar tracker

Each servomotor can move clockwise (CW) and counterclockwise (CCW). The azimuth servomotor range of rotation is from 0 degree to 180 degrees and the start position is chosen as 90 degrees, while the elevation servomotor range is limited to 90 degrees, between +45 degrees (fully CCW, as starting position) and -45 degrees.

The azimuth and elevation servomotors are HS-645MG and HS-805BB from Hitec. The rotational speed of these servomotors is very fast (rotating 60 degrees in 140 ms and 200 ms for elevation and azimuth servomotor respectively). Each servomotor movement is limited to 2-degree steps in order to control the rotation of PV cell safely and avoid any damages since the angular speed of selected servomotors are well beyond the required rates for the solar tracker. The servomotors have internal position feedback control; the microcontroller sends the request of an angle and then servomotor system moves the armature to the target position. For changing the position, the Pulse Width Modulation (PWM) command from the microcontroller carries the required position to each servomotor.

The servomotors can be obstructed during maneuvers and to detect it, the current of servomotors are sampled by SparkFun Low Current Sensor Breakout - ACS712 and converted to voltage for ADC channels of the microcontroller. As stall current for azimuth and elevation servomotors is 2500mA and 6000mA respectively, the effect of being stuck can be seen by the voltage drop at the ADC port and then detected by the microcontroller [30], [42].

3.2.4 RF module

To communicate events between the ground station (PC running MATLAB) and the microcontroller, a pair of DIGI XBEE S1 802.15.4 MODULES are used. The communication protocol is UART from the microcontroller to the RF module and then to the PC. The maximum speed of wireless module is 115,200bps and the data are packed in packets for transmission (see Appendix C).

3.3 System Software

Programming the microcontroller is done in C language in Integrated Development Environment (IDE) (Silicon Lab is the manufacturer of EFM32 and the developer of this IDE) and the ground station platform is programmed by MATLAB. The solar tracker and ground station were configured to implement conventional supervisory control in [30] (Graphic User Interface in Python is used for the ground station). In the next chapter, a new algorithm for online supervisory control is discussed and implemented. In this chapter, first a modified software implementation of conventional SCT is developed which implements SCT supervisor more closely and accurately in the sense that the implementation better matches the theory.

The supervisor in Figure 1.2-1 in offline design (i.e. conventional approach) performs in the following steps:

1. The occurrence of events at specific time period is evaluated inside each component software module.
2. According to the detected events, the next state of the supervisor is computed.
3. If there is a controllable event among enabled events, it is activated.
4. Once again, the occurrence of events is evaluated inside each component software module and then the next cycle starts from step 1. This is to see if a new event has occurred while supervisory control was done in step 2 and 3.

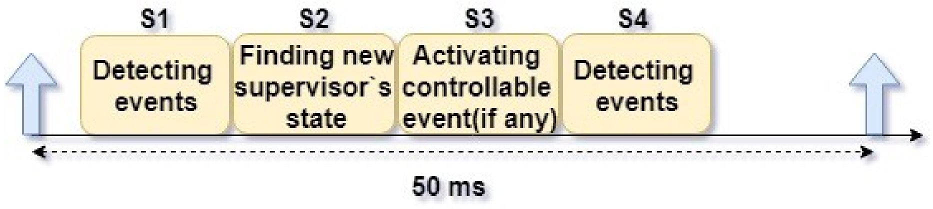

S1
Detecting events
S2
Finding new supervisor`s state
S3
Activating controllable event(if any)
S4
Detecting events
50 ms

Figure 3.3-1: Offline supervisory implementation timeline

As it is illustrated in Figure 3.3-1, the control code is run every 50ms; however, the uncontrollable events may happen at any time causing some issues. Suppose multiple uncontrollable events happen in the real plant after S4 and before S1 or even an uncontrollable event happens after S1 and before S3, in both cases the wrong control decision can be taken by the system due to missing the right order of the events (causing simultaneity and inexact synchronization problems respectively). To address the mentioned issues, event detection is done twice per any timer interruption in every 50ms (i.e. step 4).

3.4 Conventional Supervisor Implementation

In Section 3.5, we will discuss plant and specification modeling and the design of supervisor automaton will be discussed.

Once the automaton of the supervisor is calculated, it is used in the solar tracker control system to implement supervisory control. In this section, we discuss some of relevant implementation issues.

3.4.1 Supervisor structure Implementation

In this section, the procedure for the implementation of conventional supervisor is discussed.

It is desired to make the behavior of implemented supervisor as close as possible to that of theory. This implementation as shown in Figure 3.4-1, is a symmetric feedback loop in which the supervisor in general, is not a passive controller to just prevent some controllable events from happening (asymmetric feedback loop), but it sends controllable events to the plant and plant sends uncontrollable events in a symmetric feedback loop [35].

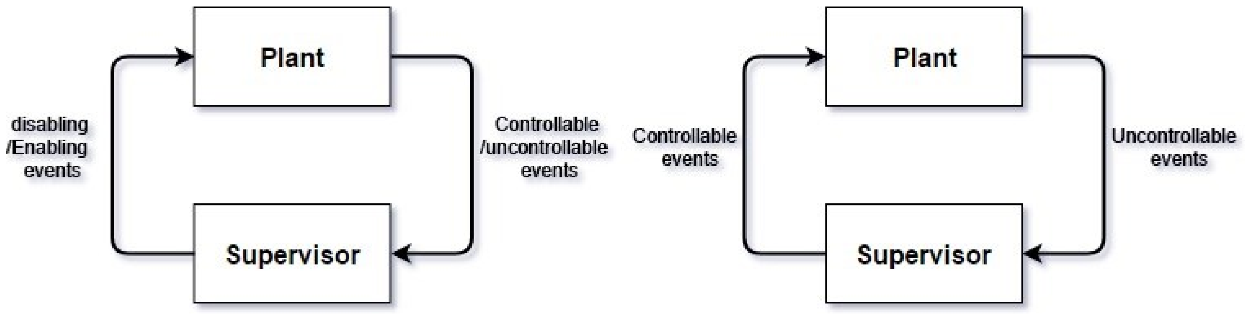


Figure 3.4-1: Asymmetric feedback loop (left) and symmetric feedback loop (right)

Moreover, in SCT there is no preference among events in active events set, in practice though due to the cyclic execution of program, and hardware limitations, some rules are needed to be set in order to process multiple active events as will be explained later.

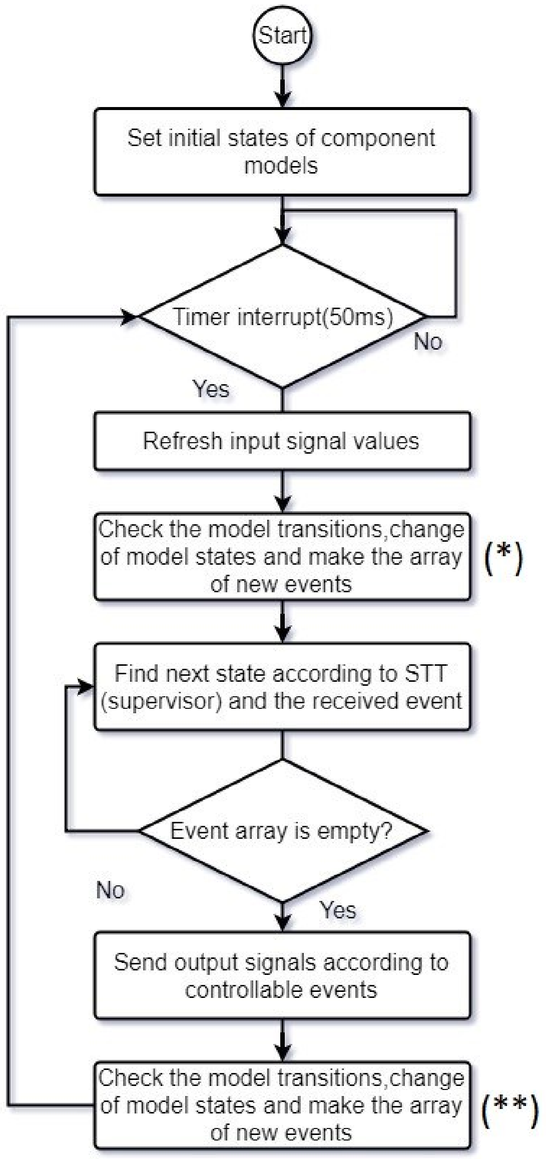

Start
Set initial states of component models
Timer interrupt(50ms)
No
Yes
Refresh input signal values
Check the model transitions,change of model states and make the array of new events
(*)
Find next state according to STT (supervisor) and the received event
Event array is empty?
No
Yes
Send output signals according to controllable events
Check the model transitions,change of model states and make the array of new events
(**)

Figure 3.4-2: Implementation flowchart for conventional supervisor in Solar Tracker.

In Figure 3.4-2, the implementation flowchart is shown. This flowchart has some differences compared to the flowchart in [30] in which all controllable events are being disabled at the beginning of each cycle and according to the active events set the possibility of them are being checked and upon validation of the first event the associated transition is being triggered.

In Figure 3.4-2, if more than one event is detected (*), these are checked in order of their priority. The priority list is formed as follows. As a rule of thumb, uncontrollable events have higher priority compared to controllable events since they may appear at any time and the system should respond to their occurrence first.

Since the active event set may contain several uncontrollable events, they also need to be arranged in terms of evaluation (step **). One way of arrangement is claimed that the uncontrollable events which cannot occur if other events preempt them should be evaluated first [42].

For each component that generates uncontrollable event (input) a software subroutine is written to track the component and detect events. For instance, a subroutine reads PV cell output and checks the occurrence of PV cell events. The models of all components are stored in the microcontroller. The detected events are sent to the supervisor. This cycle repeated every 50ms by a timer interrupt.

The priority which is considered for this implementation is based on the component-based priority. The components with internal events have higher priority compared to these which have interface events because their events can be generated immediately after reaching a new state. Since the rate of event change in the servomotors motion is higher than PV cell and battery (SOC is sent every 10 sec.) and the master controller, these models have higher priority among mentioned models.

Table 3.4-1: Priority of models in events detection

Priority Level	**Component Model**
(high) 1	Movement time interval
2	Azimuth servomotor position
3	Elevation servomotor position
4	Azimuth servomotor motion
5	Elevation servomotor motion
6	PV cell
7	Battery
(low) 8	Master Controller

Based on Table 3.4-1, a maximum of eight events can happen in one event check function, one per the component. Since the servomotors cannot move at the same time and events of master controller occur rarely, the maximum number of events in one function execution is never reached. This number plays a crucial role in the next state computation of supervisor because the supervisor has to check next state according to all of the received events from the plant. Therefore, the step in software flowchart which is for finding the next state has to be repeated as long as the events array is not empty.

The interaction of models to each other is considered as events condition in each model which may prevent occurrence of events because of unsuitable states in other models, however, if the plant is modeled precisely, they are not needed but to have a synchronous model with the real plant they have to be used in events generation functions.

The main difference between [30] and this method is that in the former method at most one event is considered as an occurred events but here the number of occurred events can be as much as the number of components model which are recorded in an array. Since the checking of input signals and their associated uncontrollable events is being done periodically, following all of the occurred events right after they are detected is more beneficial and makes the supervised system behave closer to theoritical behavior.

In [30], analog values (in the solar tracker all input signals are analog) are read periodically every 250ms and then in the main program loop the occurrence of each event is checked with a very fast cycle time which is almost below 1ms. Eventually, the associated interface events may change every 250ms while other internal events change within the main loop cycle time and it makes an inconsistent event check procedure. But in this thesis, the sampling time of interface events is reduced to 50ms to have fresh data in shorter time and then all interface and internal events are checked together within their models in order to have uniform event check. Furthermore, by taking samples more frequently every 50ms, the order of events is better detected.

Moreover, the voltage and current measurement in the solar tracker have considerable fluctuations (which should be addressed in the future in hardware parts). But to mitigate the undesired effect of this variation which causes unnecessary events, moving average of signals is applied which decreases false transitions dramatically.

As mentioned earlier, there are three main problems in SCT implementation. Next we explain the measures we have taken to mitigate them.

1. **Simultaneity**: One can notice in Figure 3.4-2 that the event detection check is done twice per a cycle, one time before checking the next state (*) and another time after that (**). The reason of having this second event evaluation is to mitigate the problem of

simultaneity in which the order of events becomes ambiguous or wrong. For example, after the internal controllable event of polling range AZ_POLL_RANGE, the next internal uncontrollable event AZ_RANGE_OK can be generated right way without any need to refresh input signals. In general, to dedicate a specific time for other CPU tasks, occupying entire CPU time to track events is not practical in larger systems. Because of servomotors move in every two seconds (according to specification discussed in the next section) and then the change in PV cell voltage and battery SOC, the solar tracker system characteristics in terms of total number of events and generated events per any cycle show that 50ms scan time is quite enough to track events and leaves enough time for other CPU tasks.

2. **Inexact synchronization**: Although the pace of change in the events is not high, scanning every 50ms is enough to see any interface uncontrollable events before sending controllable events which in turn alleviates the **inexact synchronization** problem. The added monitoring function to check the occurrence of uncontrollable events right before activation of the controllable event shows no record of inexact synchronization
3. **Choice**: The **choice** problem is not present in this system since there is no more than one controllable event in each state of the supervisor.

After implementing the supervisor, according to the flowchart in Figure 3.4-2, the system is tested to seek a bright direction as per specific steps. To have a clear view of what events happen after starting the sequence by sending Full_Sweep command to the microcontroller, one of the very first loops of events is depicted in Figure 3.4-3. As it can be seen from state 1 after receiving Full_Sweep, the supervisor state changes to state 3 and then after sending controllable event AZ_POLL_RANGE to evaluate the current position of the servomotor, the response of it which is an uncontrollable event is received as AZ_RANGE_OK. Then the controllable event for rotating the azimuth servomotor (AZ_CCW_MOVE) is followed by wait_2sec as an uncontrollable event and at the end of the loop, the feedback of normal current of servomotor appears as AZ_CCW_OK (uncontrollable event).

This is very important for further discussions in the next sections to understand cyclic trace of events in this system. The gap between states number is due to the occurrence of other events (e.g. Dark_to_Dim) and the sequence of events may be conducted to other loops with different state numbers, but until the end of azimuth CCW motion this loop keeps essential mentioned events. After reaching maximum CCW point, the transition from state 9 to another loop will be followed.

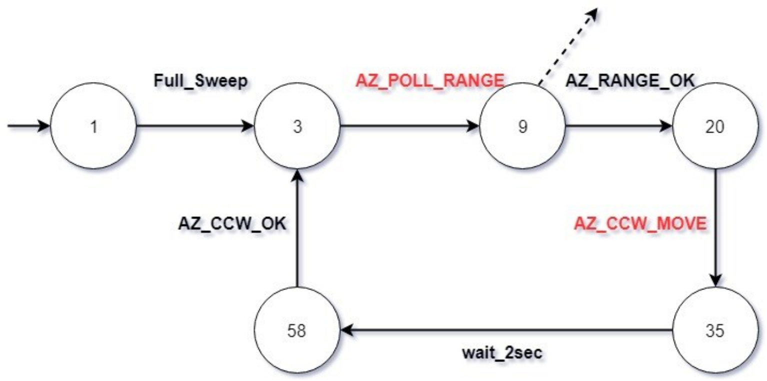


Figure 3.4-3: Part of the supervisor automaton.

In Table 3.4-2, the occurrence time of events are shown to provide a better understanding of the system behavior. Here t=0 is the start of test. The CPU clock is used to time stamp events. This is the most accurate internal method to record events inside the microcontroller.

Table 3.4-2: Events and their timeline in the Full Sweep test.

Event	**Time of occurrence (t)**	**Time between consecutive events**
Full_Sweep	23.390 sec.	-
AZ_POLL_RANGE	23.392 sec.	2 ms
AZ_RANGE_OK	23.440 sec.	48 ms
AZ_CCW_MOVE	23.443 sec.	3 ms
wait_2sec	25.441 sec.	1998 ms
AZ_CCW_OK	25.441 sec.	<1 ms

The list of customized files which are prepared to implement conventional and LLP supervisory control are in Appendix B.

As it is mentioned in Appendix B, the files which are built to read fuel gauge data, move the servomotors and read data from memory directly remained as in [30], but the other files were changed in order to achieve the thesis objectives (e.g. to overcome the undesired fluctuations in signal levels, a moving average algorithm is added in the thesis_adc.c).

The resulting supervisor which is obtained in the previous section is in the form of automaton. This automaton has a State Transition Table (STT). This STT has the complete supervisor information and by storing this table in the computer system, the supervisor states and transitions can be followed. As mentioned earlier, one of the offline drawbacks of conventional supervisor is state space explosion which requires enormous memory for a large plant. To store computed supervisor in the flash memory of the system, the STT is stored with the following structure:

```
struct state_elements{
uint16_t len;
const uint16_t (*stt)[2];
};
```

For each state, the number of outgoing transitions from that state (len) is stored and then all transitions from that state are stored with event number and destination state in the format of 2 dimensions array (stt). All numbers are unsigned integers (16 bits) because the largest number is below 65536 and above 256.

In the following section, the DES model of main hardware components (gray blocks in Figure 3.1-1), design specification and supervisor design will be explained.

3.5 System Discrete-Event Model

In the following sections the modeling each major component which has a role in supervisory control is discussed. Next the interactions of these components are described.

As mentioned earlier, all sensor signals (e.g. SOC, servomotors current and PV cell voltage) which are not under the direct control of the system, generate uncontrollable events and all actuator signals (e.g. servomotors move) make controllable events. Another partition of events used in later sections is the interface and internal events. In the software architecture [36], the events between physical environment and computer (in our case the solar tracker system) are defined as “Interface events” like the voltage of PV cell (Interface in) or PWM signal to the servomotors (Interface out) and the events which are generated and consumed solely within the software are defined as "Internal events" (e.g. reading the servomotor position).

3.5.1 Battery

The three states of the battery are Critical, safe and full and are defined based on the state of charge (SOC) of the battery as shown in Figure 3.5-1. These state change as SOC increases or decreases. The changes are modeled with four uncontrollable events, marking the crossing of specific thresholds. There is a hysteresis of 5% for each stage which prevents changes of model state with fluctuations of SOC due to measurement noise. The reason for the separation of critical and safe states is that the movement of servomotors must be prevented when the battery is in critical condition (since there is not enough energy for successful movement). But in the other two states the servomotors can move without any problem; therefore, three distinct states are considered.

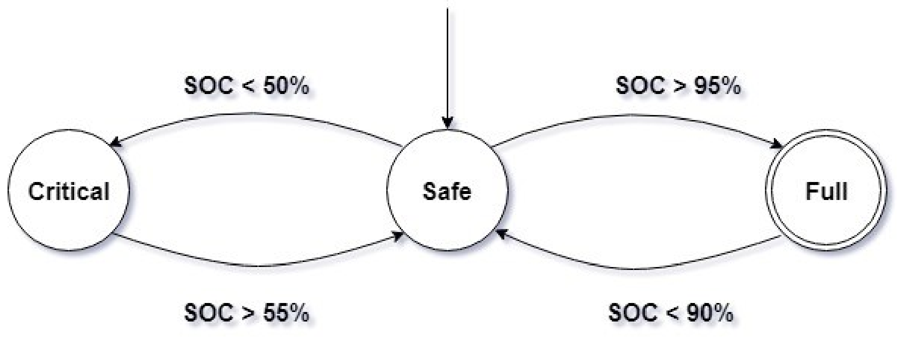


Figure 3.5-1: Battery model

The list of relevant events is in Table 3.5-1.

Table 3.5-1: Battery events list

Origin State	**Event Name**	**Destination State**	**Interface/Internal**
Safe	Safe_to_Crit	Critical	Interface In
Safe	Safe_to_Full	Full	Interface In
Critical	Crit_to_Safe	Safe	Interface In
Full	Full_to_Safe	Safe	Interface In

3.5.2 PV Cell

As the intensity of sunlight varies with the position of the PV cell, the output voltage generated by the PV cell changes. The maximum energy can be captured when the PV cell is perpendicular to the sunlight beam. Therefore, three different states are defined and the change of PV cell output voltage is modeled with four uncontrollable events as shown in Figure 3.5-2. The objective of the solar tracker is to find a sufficient bright spot in one maneuver; hence one might think that two states would be enough for this purpose but as the battery cannot charge in the dark state, three different states are considered.

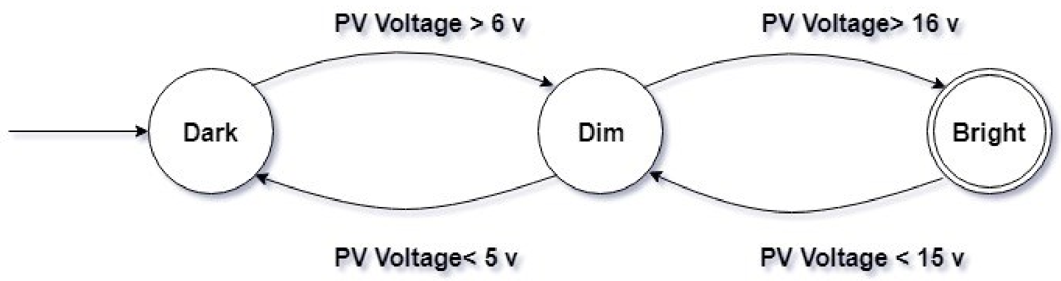


Figure 3.5-2: PV cell model

The list of relevant events is in Table 3.5-2.

Table 3.5-2: PV cell events list

Origin State	Event Name	Destination State	Interface/Internal
Dark	Dark_to_Dim	Dim	Interface In
Dim	Dim_to_Bright	Bright	Interface In
Bright	Bright_to_Dim	Dim	Interface In
Dim	Dim_to_Dark	Dark	Interface In

3.5.3 Servomotors

The operation of each servomotor is described with two DES models. The first model (motion model) is for safe movement in which the command is sent by the microcontroller and successful movement is evaluated in terms of the current which is drawn by the servomotor. The second model (range model) is designed to control the position of servomotor to limit the movement in a predefined range.

3.5.3.1 Motion Models

The goal of this model is to show commands to the servomotors. When a command is sent to the servomotor, it energizes the armature and then the specific current is consumed which shows the successful movement. Therefore, by measuring the servomotor current, the system verifies whether the movement is done or not.

If the PV cell encounters an obstacle, the servomotors should not proceed anymore and the system needs to take another decision to avoid damages to the PV cell and servomotors. Hence, the current of servomotors is read by the microcontroller and if any excessive current is detected, it will be interpreted as an obstacle and then the next command toward this direction is prevented (This will be discussed in specifications). In this implementation, for the elevation servomotor, a fault state as a sign of abnormal situation is considered, but the azimuth servomotor is assumed to be free of any obstacle or fault for simplicity.

As it is shown in Figure 3.5-3, two directions of Counter Clockwise (CCW) and Clockwise (CW) are triggered by their commands which are controllable events (the controllable events are shown in red). If the current is less than a specific value, the system backs to the idle state (initial state) through the uncontrollable events (As in [30] it is assumed that the azimuth motor works all the time and then the minimum current of 100mA is not checked).

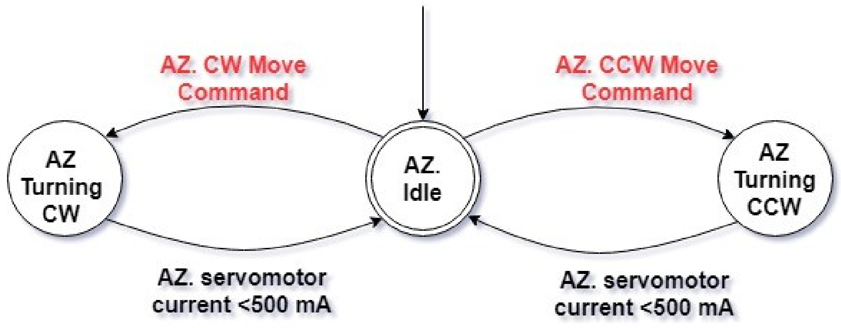


Figure 3.5-3: Azimuth servomotor, motion model

The list of relevant events is given in Table 3.5-3.

Table 3.5-3: Azimuth servomotor, motion model events list

Origin State	Event Name	Destination State	Interface/Internal
AZ. Idle	AZ_CW_MOVE	AZ. Turning CW	Interface Out
AZ. Idle	AZ_CCW_MOVE	AZ. Turning CCW	Interface Out
AZ. Turning CW	AZ_CW_OK	AZ. Idle	Interface In
AZ. Turning CCW	AZ_CCW_OK	AZ. Idle	Interface In

The elevation servomotor has a fault state. In any abnormal current case, the state changes to the fault state as depicted in Figure 3.5-4. The events have similar meanings to those in the azimuth model.

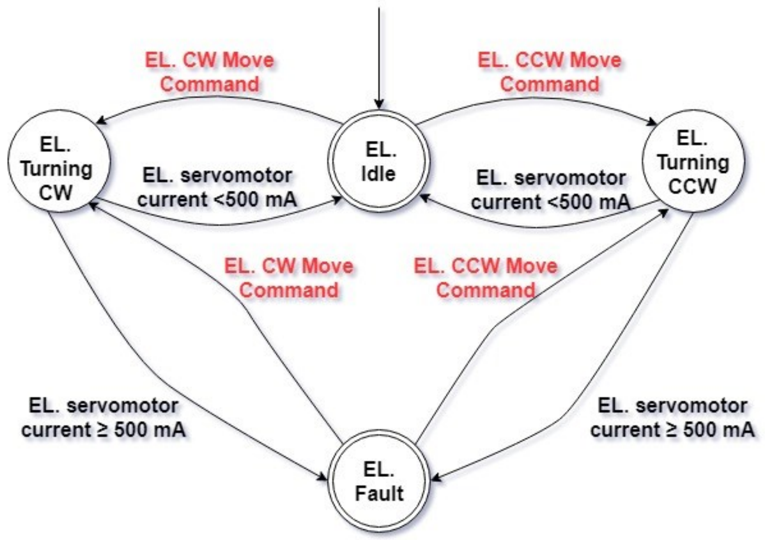


Figure 3.5-4: Elevation servomotor, motion model

Table 3.5-4: Elevation servomotor, motion model events list

Origin State	Event Name	Destination State	Interface/Internal
EL. Idle	EL_CW_MOVE	EL. Turning CW	Interface Out
EL. Idle	EL_CCW_MOVE	EL. Turning CCW	Interface Out
EL. Turning CW	EL_CW_OK	EL. Idle	Interface In
EL. Turning CCW	EL_CCW_OK	EL. Idle	Interface In
EL. Turning CW	EL_FAIL_MOVE	EL. Fault	Interface In
EL. Turning CCW	EL_FAIL_MOVE	EL. Fault	Interface In
EL. Fault	EL_CW_MOVE	EL. Turning CW	Interface Out
EL. Fault	EL_CCW_MOVE	EL. Turning CCW	Interface Out

3.5.3.2 Movement time interval Model

Since the servomotors can rotate very fast which can damage the PV cell, in addition to limiting any movement to 2 degrees, a two-second delay is added to the measurement of servomotors current after a command. This creates a two-second interval between any two movements. This model is shown in Figure 3.5-5.

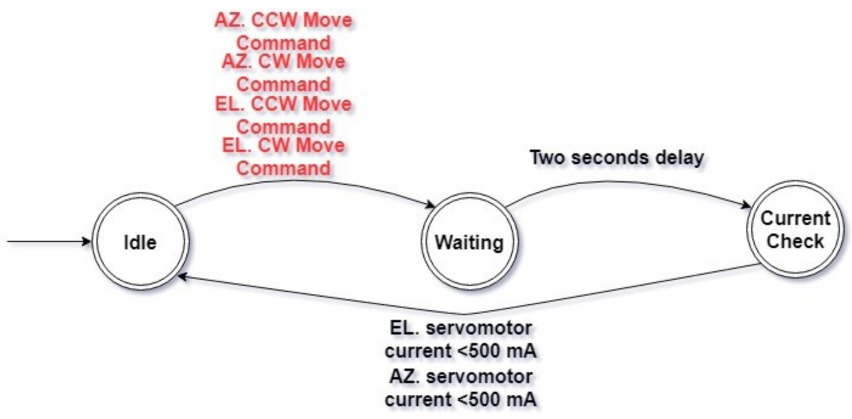


Figure 3.5-5: Movement time interval model

Table 3.5-5: Movement interval events list

Origin State	Event Name	Destination State	Interface/Internal
Idle	EL_CW_MOVE	Waiting	Interface Out
Idle	EL_CCW_MOVE	Waiting	Interface Out
Idle	AZ_CW_MOVE	Waiting	Interface Out
Idle	AZ_CCW_MOVE	Waiting	Interface Out
Waiting	wait_2sec	Current check	Internal
Current check	EL_CW_OK	Idle	Interface In
Current check	EL_CCW_OK	Idle	Interface In
Current check	AZ_CW_OK	Idle	Interface In
Current check	AZ_CCW_OK	Idle	Interface In

3.5.3.3 Position Models

The current position of each servomotor stored in the RAM of the microcontroller and then, after each successful two-degree movement, it is increased after a CW movement or decreased after a CCW movement. Therefore, the current angular position of servomotors are available and when they reach the boundary of movement and the corresponding states are reached further movement in the same direction is forbidden. As shown in Figure 3.5-6 and Figure 3.5-7, for azimuth and elevation servomotors respectively, for any movement, the current angle is polled and then if it is in acceptable range, then a return to the initial state event is generated and if it is not in the range, then the corresponding state depending on the direction will be reached.

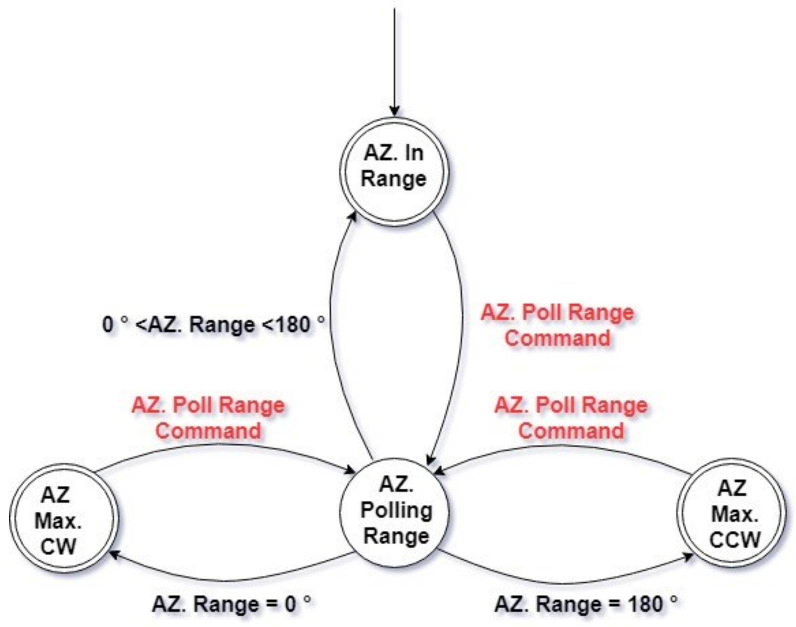


Figure 3.5-6: Azimuth servomotor position model

Table 3.5-6: Azimuth servomotor position events list

Origin State	Event Name	Destination State	Interface/Internal
AZ. In Range	AZ_POLL_RANGE	AZ. Polling Range	Internal
AZ. Polling Range	AZ_RANGE_OK	AZ. In Range	Internal
AZ. Polling Range	AZ_MAX_CCW	AZ. Max. CCW	Internal
AZ. Polling Range	AZ_MAX_CW	AZ. Max. CW	Internal
AZ. Max. CCW	AZ_POLL_RANGE	AZ. In Range	Internal
AZ. Max. CW	AZ_POLL_RANGE	AZ. In Range	Internal

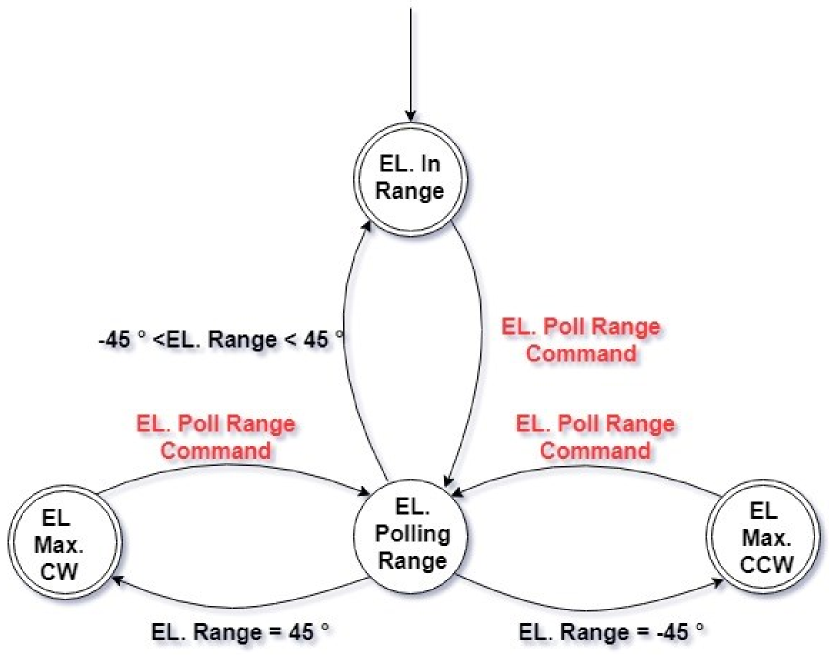


Figure 3.5-7: Elevation servomotor position model

Table 3.5-7: Elevation servomotor position events list

Origin State	Event Name	Destination State	Interface/Internal
EL. In Range	EL_POLL_RANGE	EL. Polling Range	Internal
EL. Polling Range	EL_RANGE_OK	EL. In Range	Internal
EL. Polling Range	EL_MAX_CCW	EL. Max. CCW	Internal
EL. Polling Range	EL_MAX_CW	EL. Max. CW	Internal
EL. Max. CCW	EL_POLL_RANGE	EL. In Range	Internal
EL. Max. CW	EL_POLL_RANGE	EL. In Range	Internal

3.5.4 Master Controller

Some of the events in the solar tracker system are not generated by the previous components. The "start full sweep" command comes from the PC and is uncontrollable (by the supervisor). The events "Elevation motor fault", "Sweep Failure" and "Bright (direction) detected" are generated by the supervisor itself. To model the generation of these events, it is convenient to assume they are generated by a hypothetical component, Master Controller (Figure 3.5-8).

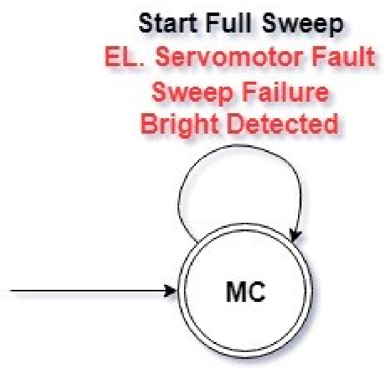


Figure 3.5-8: Master Controller (MC) model

Table 3.5-8: Master controller events list

Origin State	Event Name	Destination State	Interface/Internal
MC	Full_Sweep	MC	Internal
MC	Bright_Detected	MC	Internal
MC	Sweep_Failure	MC	Internal
MC	EL_MOTOR_FAIL	MC	Internal

3.5.5 System Interactions

To build a complete model, we have to model the interactions of the system components. For example, consider the SOC of the battery when the PV cell is in darkness. It is clear that going to the higher level for SOC is impossible. This behavior (restriction) should be captured by the model. Modeling interactions needs a rigorous attention to the physical characteristics of components, otherwise the constructed plant will not follow the real plant. In the following sections, the interactions of different components are modeled.

3.5.5.1 Battery and PV cell

As mentioned before, in the dark state, the maximum output voltage of the PV cell is 6 volts which cannot charge the battery SOC according to the battery specification. Therefore, in the dark state just the reduction of SOC can occur and in other states both reduction (because of servomotors consumption) or increase of SOC (because of PV cell supply) are possible. The battery SOC events as a function of the state of the PV cell is shown in Figure 3.5-9.

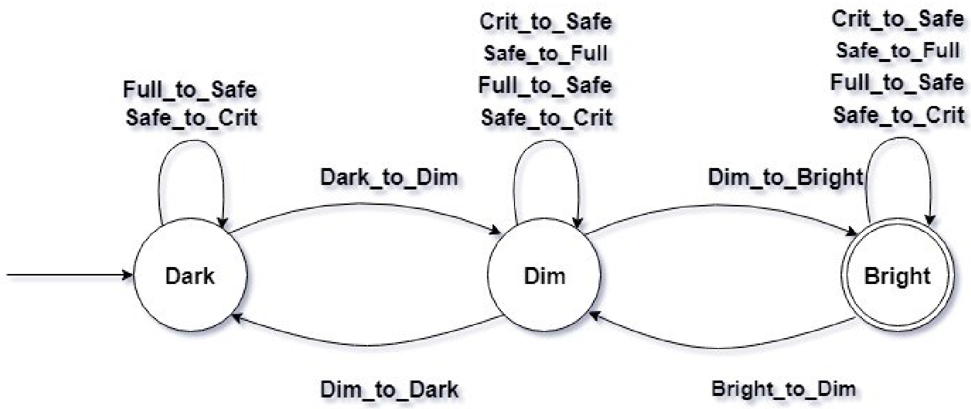


Figure 3.5-9: PV cell interaction with battery SOC

3.5.5.2 Battery and Servomotors

According to the battery discharge curve and the current consumption of servomotors, it is essential that the SOC of battery be above 50% (in order to energize the servomotors sufficiently for a move). Thus, in the critical state of battery SOC, the events which guarantee a successful movement cannot happen as shown in Figure 3.5-10.

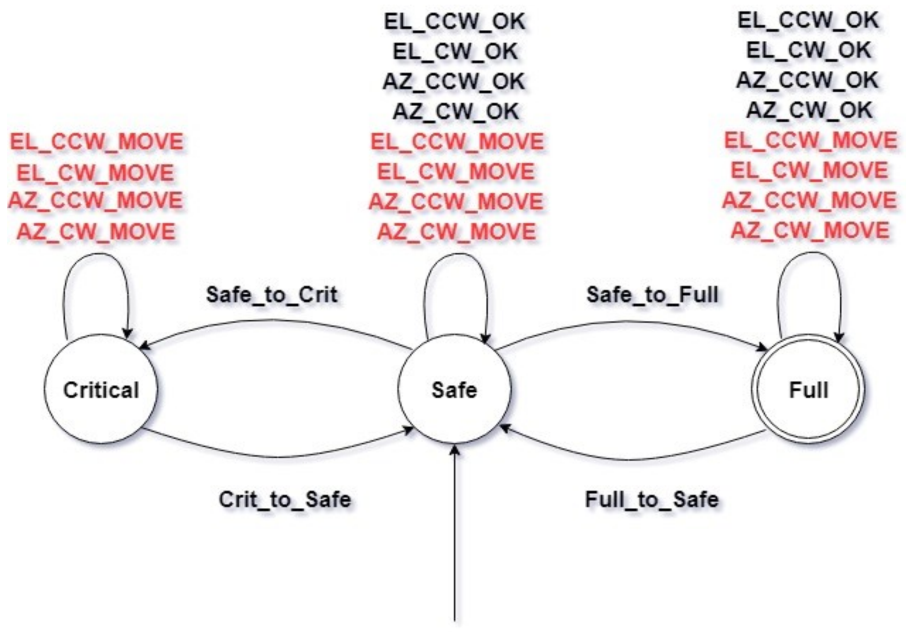


Figure 3.5-10: Battery SOC and servomotors interaction

According to the data sheets of the servomotors [42], the azimuth and elevation servomotors consume 350 mA and 700 mA respectively which are more than the maximum current the PV cell supplies (200 mA). Therefore, during any servomotor movement, the battery is discharged and reaching a higher level of SOC is not possible. To model this behavior, the synchronous product of two servomotors models (shown in Figure 3.5-3 and Figure 3.5-4) is formed and decrease in SOC events are added as self-loop in the states in which movement occurs and increase and decrease of the SOC self-loops are added to the rest of states (Figure 3.5-11). For clarity, the transitions of the synchronous are not shown in the figure; only the self-loops are shown.

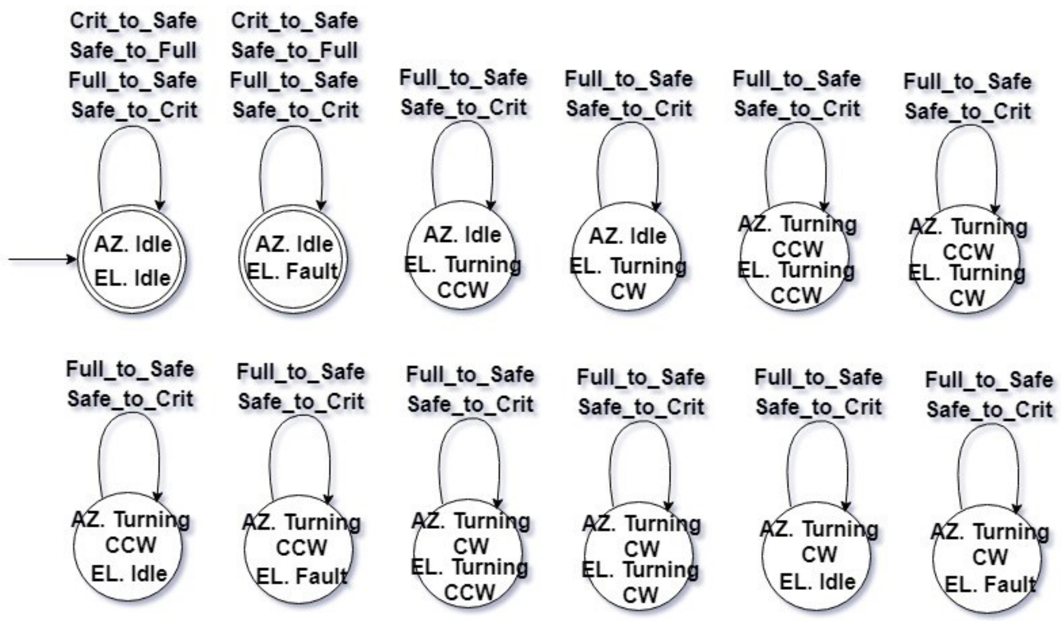


Figure 3.5-11: Servomotors and battery SOC interaction

At this point, the model of the plant can be constructed by the synchronous product of the DES models of the components and their interactions. The resulting plant automaton has 1584 states and 16800 transitions.

3.6 Supervisor Design

In this section, the design specifications are explained and the supervisor is designed.

3.6.1 Specification

The solar tracker system has three groups of design specification. A DES model for each group is introduced. A model for the overall specification is obtained using the automaton product operation.

3.6.1.1 Servomotor Motion Range specifications

The rotation should be limited to 180 degrees for the azimuth and 90 degrees for the elevation servomotor. Therefore, whenever in the position models (Figure 3.5-6 and Figure 3.5-7) the states of maximum CW or CCW are reached, further movement in the same direction should be prevented. Thus, in the mentioned states just the reverse movement is allowed as it is shown in Figure 3.6-1 for the azimuth servomotor. Similarly, the elevation servomotor should not rotate beyond its limit (the model specification is not shown for brevity).

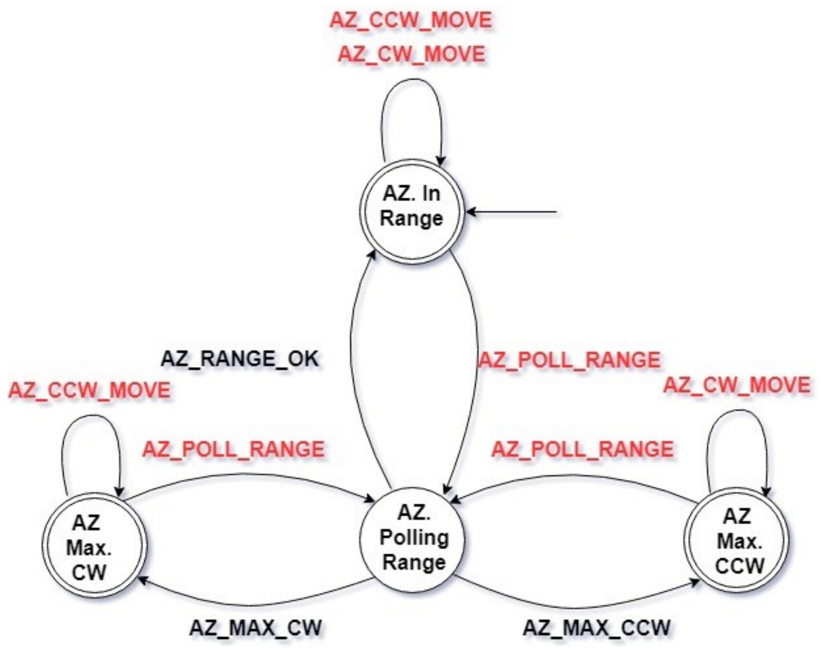


Figure 3.6-1: Azimuth servomotor rotation specification

3.6.1.2 Servomotor Polling Range specifications

Since the current position of servomotor should be read after of a successful movement (XX_CW_OK, XX_CCW_OK events), the polling of this value should be done when the state of servomotor motion model (Figure 3.5-3and Figure 3.5-4) is in the idle state. This specification is depicted in Figure 3.6-2. For the elevation servomotor, the same rule is applied (For brevity the model is not shown).

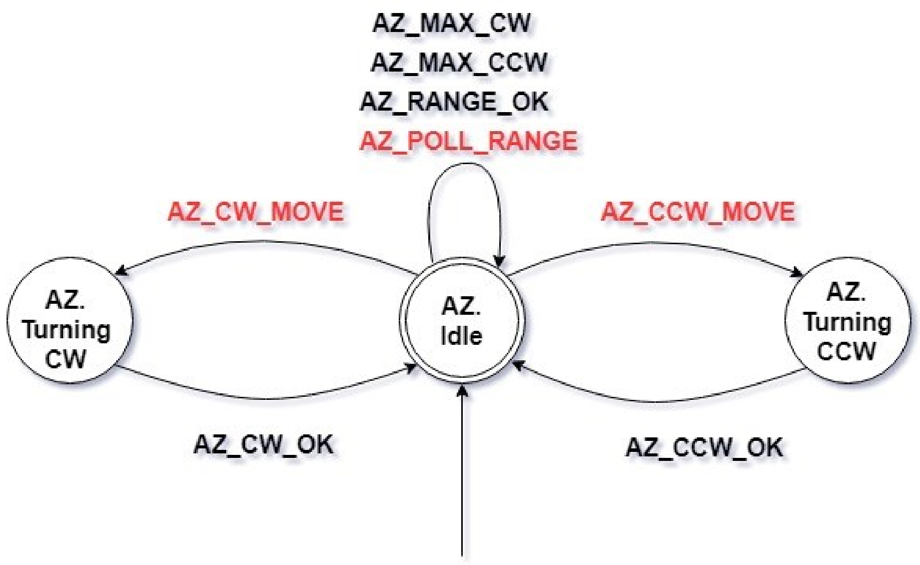


Figure 3.6-2: Azimuth servomotor polling specification

3.6.1.3 Servomotor Movement Trajectory specifications

The former specifications are essential for a safe operation of the solar tracker. In addition, a procedure to indicate the sequence of steps toward finding a bright spot is required. It is common in industry to define a procedure to conduct the operation through steps for a variety of purposes. For example, in most petrochemical plants, startup phase and shutdown phase must be done by taking sequential steps to normal operation and safe state respectively. In this system, the goal is to sweep a hemisphere to find the spot with the sufficient sunbeam intensity. As each servomotor rotates in a plane, combining them, allows a complete hemisphere sweep. Assuming the solar tracker is located in a dark zone, a full sweep is done in the following order.

The azimuth servomotor starts rotation from 90 degrees in CCW direction by receiving "Full_Sweep" event from the MC; then in the maximum CCW position, it turns in the reverse direction and rotates to CW maximum position. Afterward, the elevation servomotor starts to rotate from its position (maximum CCW) to maximum CW for 90 degrees and then azimuth servomotor continues sweep in CCW direction for 180 degrees. In the end, if no bright spot is discovered, the Sweep_Failure event is generated; otherwise, the Bright_Detected as an indication of finding a bright direction is generated. Furthermore, EL_MOTOR_FAIL as an alarm for elevation servomotor fault is issued to MC if failure is detected.

As a general restriction, at any moment just one servomotor can move to maintain the battery voltage at sufficient level for the system. This specification`s model has 58 states and 209 transitions (see Appendix A). One can see various loops inside this specification in each stage representing search for a bright direction until either the end of movement in one direction and the direction is changed or a bright source is detected.

Thus far, the individual automaton for each specification is generated. Next, the events which are not relevant for each specification model are added as self-loop to all states. Finally, to find the overall specification model, the product of all specifications should be taken. The resulting specification model has 416 states and 4216 transitions.

3.6.2 Supervisor

To design the supervisor using SCT, the toolbox Discrete Event Control Kit (DECK)) [37] which is developed in MATLAB [34] environment is used. The automata which are needed for making supervisor are prepared by using the functions of DECK. To compute the supervisor, first the components models are defined in DECK with the following format:

The automaton in DECK (G=automaton(N,TL,Xm)) is defined as a class with the following properties:

N: The number of states,

TL: Transition List,

Xm: marked states (row vector),

TL is a matrix with three columns. In each row, the first entry is a source state of each transition, the second element is the event related to the transition and the third element is the destination state. Thus the number of rows indicates the number of transitions.

There are plenty of functions in DECK which do specific DES operations. The procedure to design the supervisor is as follows.

First, the plant model is constructed by using the **sync** function of component models and their interactions ([G,States]=sync(G1,...,Gn)). Second, the general specification is made by using

product ([H,States]=product(H1,...,Hn)) of all specifications automata. The last step to obtain the supervisor from the plant and specification is to use the **supcon** function with the following format:

K =supcon(H,G,Euc).

H is the specification, G is the plant and Euc is the uncontrollable events set.

Supcon generates the supremal sublanguage of $L_m(H) \cap L_m(G)$ that is controllable with respect to $L(G)$ and Σ_{uc}. The result is returned in the trim automaton K which marks the supremal controllable sublanguage.

$$\begin{cases} L_m(K) = \mathrm{SupC}\big(L_m(H) \cap L_m(G)\big) \\ L(K) = \overline{\big(L_m(K)\big)} = \overline{\mathrm{SupC}\big(L_m(H) \cap L_m(G)\big)} \end{cases}$$

The resulting supervisor size is given below.

Table 3.6-1: The plant, specification and supervisor size

Automaton	Number of States	Number of Transitions	Number of Marked States
Plant (G)	1584	16800	27
Specification (H)	416	4216	18
Supervisor (K)	2061	9527	27

Chapter 4

Limited Lookahead Supervisory Control with Buffering

In supervisory control theory, the supervisor (i.e. control logic) is designed based on a DES model of the plant and a DES model of the design specifications. The design procedure is typically performed "offline" and the designed supervisor in the form of computer code is implemented "online" on, say, a microcontroller. This approach was used in the previous chapter.

An alternative approach, as discussed in Chapter 2, is to use the Limited Lookahead Policy in which the plant model and design specifications are stored in the control computer and are used to generate supervisory control commands "online". The main goal of this chapter is to explain the implementation of LLP. The solar tracker is used as the experimental setup. As we will see later, sometimes events are generated at a fast pace in the plant and there is not enough time to perform the LLP calculations between two consecutive events. In order to mitigate this issue, we propose a new method in which the supervisory commands are pre-calculated and "buffered" in advance.

In Sections 4.1 and 4.2 we will discuss code generation and implementation of LLP for the solar tracker as an example. Section 4.3 presents LLP with buffering.

4.1 Generating C Code for the Microcontroller

To store the information of the automaton we use the C **struct**. The required C code for LLP is obtained by converting DECK functions to C code using MATLAB coder. The DECK code for supervisory control has to be modified for use in LLP and be made compatible with MATLAB coder. G.n, G.TL and G.Xm are used as fields of **struct** G. This resembles the automaton objects in DECK. Although in MATLAB there is no need to define the size of TL and Xm explicitly, in C language, it is not possible to have more than one variable size array as an argument (Simply, TL and Xm can be seen as two and one-dimensional array, respectively). Therefore the **coder.varsize** is used to declare variable size array in functions in MATLAB. For example, consider the **reach** function, Xr=reach (TL, S), in which TL and S are the input arguments and Xr the reachable states from the states S as the output of the function in the form of row vector. Then to declare the size of Xr, the following line is added in the function:

```
coder.varsize('Xr',[1,10000],[0 1]).
```

This means that Xr is a one dimensional array and its size is bounded by 10,000 elements.

After modifying the required functions to make them ready for compilation (by right-click on any m file and "Check Code Generation Readiness", it should have a full score), the inputs of the functions need to be defined. For example, for the **reach** function, TL is defined as a matrix with bounded number of rows up to 10000 and exactly 3 columns with double elements (double(:10000x3)). Since MATLAB, for every variable, uses as double-precision floating-point values that are 8 bytes, it takes a large amount of memory to store and perform the calculations which is one of the disadvantages of direct C code generation from MATLAB coder.

The other required functions are similarly adapted and compiled using MATLAB coder. It should be noted that, there are several parameters in MATLAB coder to make source code adjust to specific purposes in terms of execution speed, memory usage and even the hardware platform which can be selected for ARM Cortex –M (e.g. for EFM32 Series). Finally, the generated source codes can be used as a C function in the microcontroller to do the supervisory calculations.

We used MATLAB coder for converting DECK functions to C code for the microcontroller EFM32 used in the solar tracker. But because of memory constraints and the slow execution, we decided to compile and run the code on the PC and use the microcontroller for interfacing with the hardware. Our reasons are explained in more detail in the following.

1. The functions of DECK have many subfunctions like **unique**, **ismember**, **intersect** and so forth that make the compilation very complex in terms of debugging and memory management. Despite the fact that the generated C codes consume large flash memory, but

256KB flash memory is enough to contain all generated codes. For instance, the **reach** function which is one of the smallest functions in DECK takes almost 14 KB of flash memory.

Since the default data type of MATLAB is a double which takes 64 bits and in spite of declaring input arguments in different data types which take less memory, the internal variables which are used by the compiler are all double and consequently the amount of RAM which is needed for computation surges up dramatically. For example to check a very simple automaton with the **reachable** function ([Gr,Xr]=reachable(G)), which contains the **reach** function as a subfunction, the total required RAM is almost 27KB (increased by 13KB) and the rest of memory for the most complex functions like **product** and **supcon** is not adequate.

2. The execution time is one the main concerns in online computation. Therefore, the generated codes must be optimized in such a way that they take the least possible time for execution in the microcontroller. To have an estimation of computation time, consider the result of computation time for a simple function like **reach** with 100 transitions and a state set S with one element in Table 4.1-1.

Table 4.1-1: Execution time for the ***reach*** *function with 100 transitions*

Platform and function	**Execution time**
Function in MATLAB (Intel(R) Core™ i5-6200U)	1.3 ms
Mex function in MATLAB(Intel(R) Core™ i5-6200U)	0.5 ms
C code in EFM32 microcontroller	8 ms

The execution time of MATLAB code is obtained after several runs to reach the minimum stable time. Then it is compared to a C code function (mex function) which runs in the MATLAB environment. It is clear that converting MATLAB code to the C code (mex file in MATLAB) speeds up the execution time as it is expected but the computation time for executing a very simple function like **reach** in the microcontroller is quite high considering the scan time of 50 ms.

Examining the profiler in MATLAB using "Run and Time" shows that the **reach** function is called 4 times during supervisory computation which means that for the assumed small size TL, it takes at least 32ms (without considering overheads). Therefore, by considering other functions and the solar tracker plant size, we conclude that it is not practical to deploy these generated functions for LLP algorithm in this microcontroller.

Ultimately, because of the mentioned problems and limitations and to accomplish this project in the limited timeframe, it was decided to move the execution of LLP from the microcontroller to the PC as part of the system. This type of implementation can still reveal many features and constraints of a complete LLP implementation in an embedded system and gives us the requirements for the hardware and software to implement LLP on any type of platform.

4.2 Generating Code for Implementation of LLP in MATLAB

Following the discussion of the previous section, by compiling the MATLAB code to the C code and executing it inside the MATLAB environment, execution time is improved adequately. Thus all functions in DECK whose execution consume large time, are converted to mex functions.

Figure 4.2-1 shows the schematic diagram of the solar tracker, the microcontroller as an interface and the PC which performs supervisory calculations.

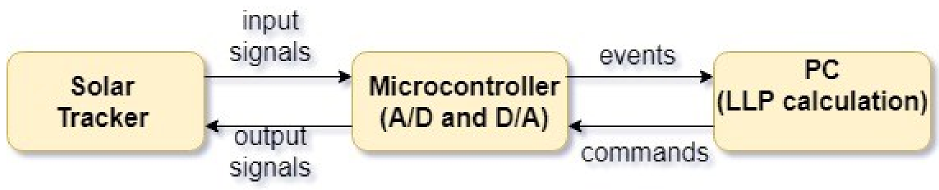
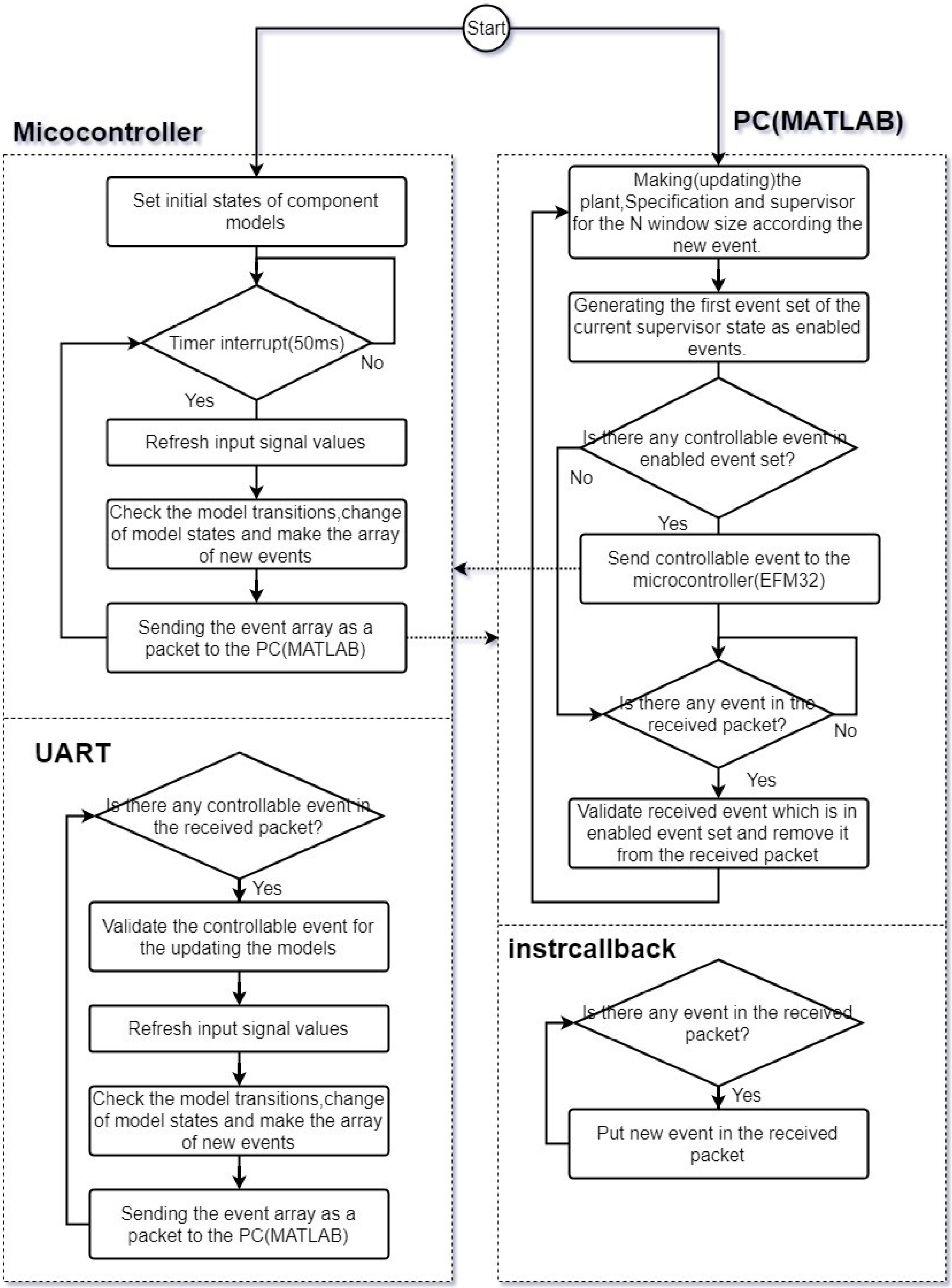


Figure 4.2-1: Schematic diagram of the solar tracker and control system

The flowchart for the implementation of LLP inside the microcontroller and PC is shown in Figure 4.2-2. Detecting events is the microcontroller's duty because it interacts with the solar tracker system and the latest status of the components is tracked by the microcontroller software. In every scan time, the newly detected events are packed in a packet and sent to the PC. The microcontroller has a communication serial port which sends and receives data from the ground station (PC) through a wireless module which has the ability to work at the baud rate of 115200bps as maximum speed. Although the EFM32 maximum UART speed is much higher than 115200bps but the wireless module as a bottleneck of this communication system limits this capability.

After scanning events by a timer interrupt and sending them to the PC, an **instrcallback** function inside the MATLAB receives the data and stores them in the receiving packet. The callback functions in MATLAB can be triggered by the occurrence of any type of event. In this application, for receiving serial data whenever they are ready in the serial port, **instrcallback** is called and it stores data. Meanwhile, on the PC side, the plant, specification and supervisor are calculated for a predefined window size of LLP of length N_w. Then by considering the first events of the current state of the supervisor, one event is sent to the microcontroller as supervisory command.



Start
Micocontroller
PC(MATLAB)
Set initial states of component models
Timer interrupt(50ms)
No
Yes
Refresh input signal values
Check the model transitions,change of model states and make the array of new events
Sending the event array as a packet to the PC(MATLAB)
UART
Is there any controllable event in the received packet?
Yes
Validate the controllable event for the updating the models
Refresh input signal values
Check the model transitions,change of model states and make the array of new events
Sending the event array as a packet to the PC(MATLAB)
Making(updating)the plant,Specification and supervisor for the N window size according the new event.
Generating the first event set of the current supervisor state as enabled events.
Is there any controllable event in enabled event set?
No
Yes
Send controllable event to the microcontroller(EFM32)
Is there any event in the received packet?
No
Yes
Validate received event which is in enabled event set and remove it from the received packet
instrcallback
Is there any event in the received packet?
Yes
Put new event in the received packet

Figure 4.2-2: Online LLP implementation flowchart

4.2.1 The Minimum Size of Lookahead Window

As mentioned in Section 2.2.2, the minimum size of lookahead window to guarantee that the resulting supervisor behavior is the same as a minimally restrictive conventional depends on several factors, in particular, the length of uncontrollable strings in the plant.

After we build the plant model of solar tracker, we observe several loops consisting of uncontrollable events only. For example, among 1584 states of the plant, there is a loop of uncontrollable events (Dark_to_Dim and Dim_to_Dark) as it is depicted in Figure 4.2-3 because changes in the PV cell voltage can happen any time.

Therefore, according to the ***Theorem 2.8***, $N_u(L(G))$ is infinite and therefore no minimum length for tree expansion of LLP can be determined to guarantee the validity of the supervisor.

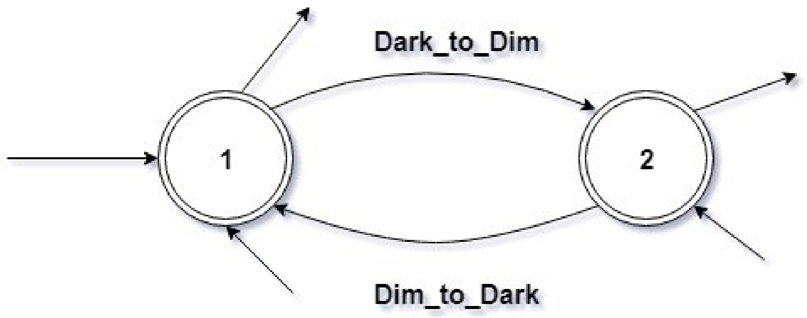


Figure 4.2-3: An uncontrollable loop in the plant

It is pointed out in Section 2.2.3 that one of the advantages of state-based supervisory in comparison to event-based is that in the state-based supervisory with finite states, the minimum window length to make LLP supervisor valid is always finite. Let us define this minimum bound by N_{min}. Hence, to find N_{min} for the solar tracker plant, assuming state-based algorithm is used, and according to Definition 2.27**,** N_B has to be calculated prior to LLP execution to provide the optimal supervisor.

In order to calculate N_B, first of all, the set of all marked controllable states should be extracted. The function Xmc_verify (Plant,Ec) is prepared in MATLAB to show the set of X_{mc} (Definition 2.25). After verifying the solar tracker plat, it turns out that there is no state with just controllable event in this plant. Secondly, the set of legal states (Definition 2.26) has to be investigated. It is worth noting that the generated supervisor using **supcon** in DECK, marks the supremal sublanguage of $L_m(H) \cap L_m(G)$ which is controllable with respect to $L(G)$ and uncontrollable events. Hence, the language of $L_m(H) \cap L_m(G)$ is the legal language which has to be examined to find N_B. However, this legal behavior should be a subautomaton of the plant in order to be used in the definition of N_B. This requires applying the procedures of [33].

In this thesis, instead we have used an exhaustive method for finding N_{min}. That is conventional supervisor and LLP supervisor (for various values expansion N_w) are compared in every single state and then if there is no difference between enabled events, it means that $N_w \geq N_{min}$. After running the code several times it was seen that for $N_w \geq 9$, no difference is observed between LLP and conventional results. Thus $N_{min} = 9$.

4.2.2 Plant Depth

The parameter N_w plays a very significant role in the LLP computation time due to the fact that for small N_w, the size of the expanded plant will be small resulting in lower LLP computation time. Therefore in the system with small time for event response, this number has to be in reasonable range. On the other hand, this number cannot be adjusted as a parameter, since it depends on system characteristics. To have a better understanding of this number, Plant Depth (PD) is defined as the lookahead window size in which the size of expanded plant model (in the state-based expansion) becomes equal to the plant model. The ratio of N_w to PD shows how much LLP computation is effective. If the ratio is near one, it means that LLP implementation value declines since the LLP supervisor almost equals to the conventional supervisor. Hence, LLP has no memory advantage and requires more computational time.

The PD size for the solar tracker plant is 11 which is so close to minimum lookahead window size (N_{min}=9).

Nonblocking and safety properties are two major concerns in the context of supervisory control which have to be met. As shown in Table 3.6-1, there are 27 marked states in the plant which should be reachable and since the resulting supervisor is trim and then nonblocking, reachability of them is guaranteed. To simplify the following discussion in this thesis, we only consider the safety property and effectively mark all states.

It can be shown that if the nonblocking is no longer an issue and all states consider to be marked, then through the exhaustive method, $\mathrm{N_{min}}$ becomes smaller and equals to 6 which is almost half of the PD ($\frac{\mathrm{N_{min}}}{PD} = 0.54$).

It should be noted that a better measure of the efficiency of LLP is to compare $\mathrm{N_{min}}$ with the depth of the product of plant and specification.

4.2.3 LLP Computation Time

Computation time is one of the most important issues in LLP because of online calculation and limited CPU resources to respond to the upcoming events. Using lookahead window size of $\mathrm{N_w} = \mathrm{N_{min}} = 6$, LLP computation times for a sample string of 259 events are found experimentally and shown in Table 4.2-1. For a more accurate measurement of the execution time in MATLAB, the priority of MATLAB in task manager of Windows is set to real-time priority which increases scheduling priority of the MATLAB among other tasks in Windows.

Table 4.2-1: LLP computation time

LLP computation time for Lookahead window Size $\mathrm{N_w}$ =6 (MATLAB(Intel(R) Core™ i5-6200U)		
Average	Minimum	Maximum
139 ms	42 ms	306 ms

The supervisory control calculation of LLP consists of three steps. The calculation of each step can be in a number of different orders, resulting in different execution time. We have chosen an order for each step so as to minimize the execution time. The details are explained in the following.

1. Plant expansion: The **sync** function is used to build a subautomaton of the plant with a depth of $\mathrm{N_w}$ events from the current state. The computational time and required memory used by **sync** function depends on the order of its input automaton (even though the end result is the same). It is well-known that automaton with significant common events should be synced together and therefore be close to each other on the list of arguments of **sync**. Therefore, a specific function based on product is provided in MATLAB to take every two automata and the window size $\mathrm{N_w}$, and make the synchronous product of them after adding

the required self-loops. By ordering every two automata according to the number of events in common (the order of automata is illustrated in Figure 4.2-4) the least computation time of this phase is achieved.

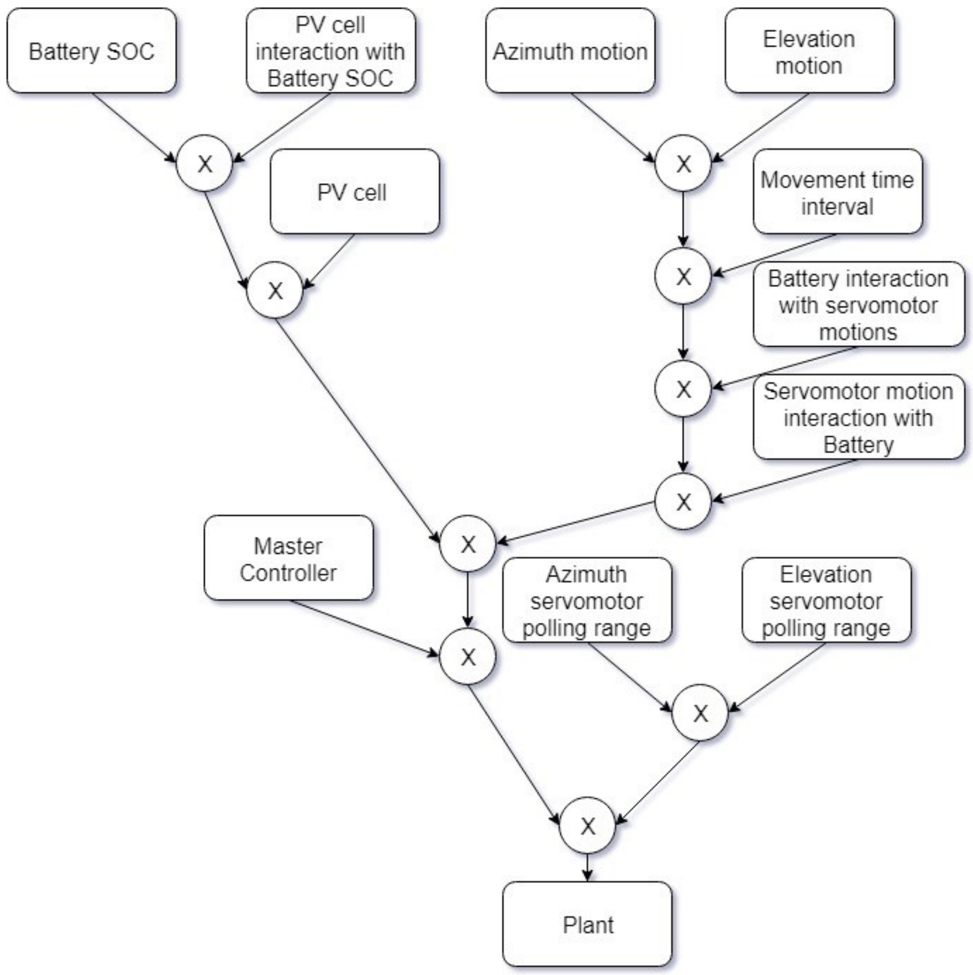

Battery SOC
PV cell interaction with Battery SOC
Azimuth motion
Elevation motion
X
PV cell
Movement time interval
Battery interaction with servomotor motions
Servomotor motion interaction with Battery
Master Controller
Azimuth servomotor polling range
Elevation servomotor polling range
Plant

Figure 4.2-4: The order of sync operation to build plant expansion.

2. Expanding the specification: The same procedure is applied to construct the general specification according to the five specifications (Figure 4.2-5). Since the attitude in this LLP implementation is conservative because of admissible results in all cases, as it is pointed out in the third step of LLP computation in Section 2.2.2, for conservative attitude, the size of expansion is one step less than the plant expansion. Therefore, the size of expansion in this phase is always one step less than the previous phase.

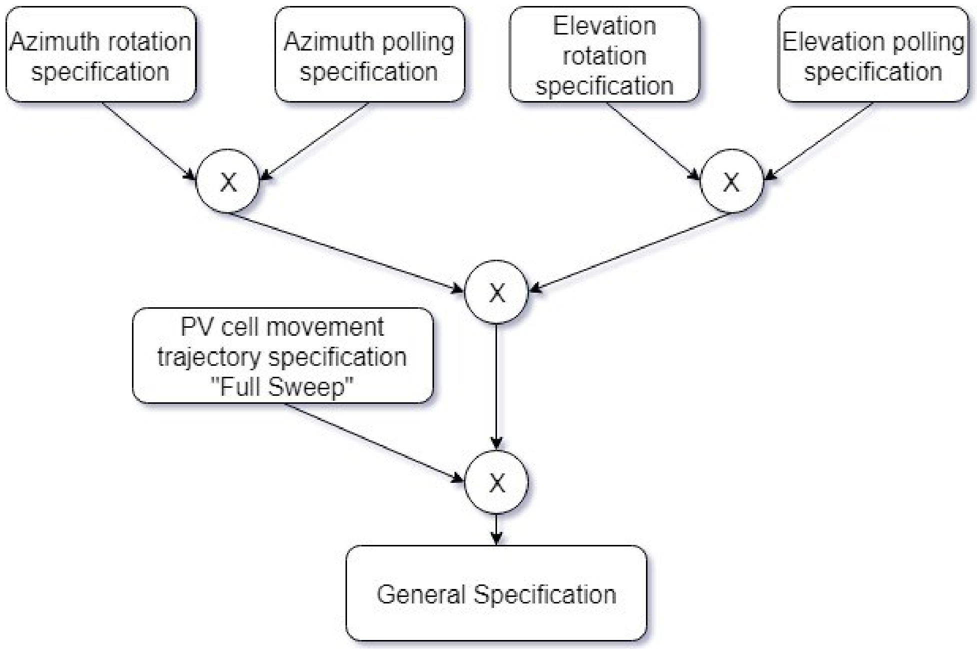

Azimuth rotation specification
Azimuth polling specification
Elevation rotation specification
Elevation polling specification
X
X
X
PV cell movement trajectory specification "Full Sweep"
X
General Specification

Figure 4.2-5: The order of calculation of specification

3. Constructing the supervisor: In this phase, using **supcon,** the supervisor is generated from the plant and specification and then by extracting the events of the initial state, the enabled events are determined.

It is obvious that from the second cycle after detecting an event, all component models need to be updated in order to determine the new state.

One can observe from Table 4.2-1 that the average computation time is bigger than the scan time (139ms > 50ms). Therefore even on average, the code is not fast enough to do LLP calculation between two consecutive events. To evaluate the timing behavior of the software, the computation is defined to be feasible whenever there is enough time to execute all tasks of the processor [36]. As any task has an activation time, deadline and execution time, the aim of LLP is to respond to all events detected in scan time, and the execution of LLP calculation should meet the deadlines.

Since the average LLP computation time (138ms in Table 4.2-1) is quite higher than the time between most events (Table 3.4-2) this type of scheduling is infeasible for the LLP supervisory implementation. Therefore, a new method to address scheduling issues is introduced in the next section.

4.3 Limited Lookahead Supervisory Control with Buffering

The online calculations of LLP have to be executed after every new event is generated. This takes considerable CPU time specially in the case of large expansion window (i.e. large N_w). As we saw in the previous section, the LLP computation time in the solar tracker is typically larger than the scan time (the period of polling process). Sometimes in the single scan time, more than one event may be detected. Therefore the complied C code does not run fast enough to respond to events in a timely fashion. To deal with this issue of LLP, a novel method called LLP with buffering is proposed in this section. In this method, in every LLP calculation, control commands are calculated for multiple future steps (not just the immediate next step). This is achieved by using a larger expansion window.

4.3.1 Extension of the Lookahead window size

As discussed in Sections 2.2.2 and 2.2.3, a valid supervisor has to use an expansion window of at least N_{min} events after the current state to guarantee optimality. But how about using a larger window size? Suppose the size of lookahead window is increased to $N_{min} + \Delta$ as it is illustrated in Figure 4.3-1. (This extension is different from what is outlined in [38], since they consider any string after limited lookahead window to remove the attitude effects on the supervisor.)

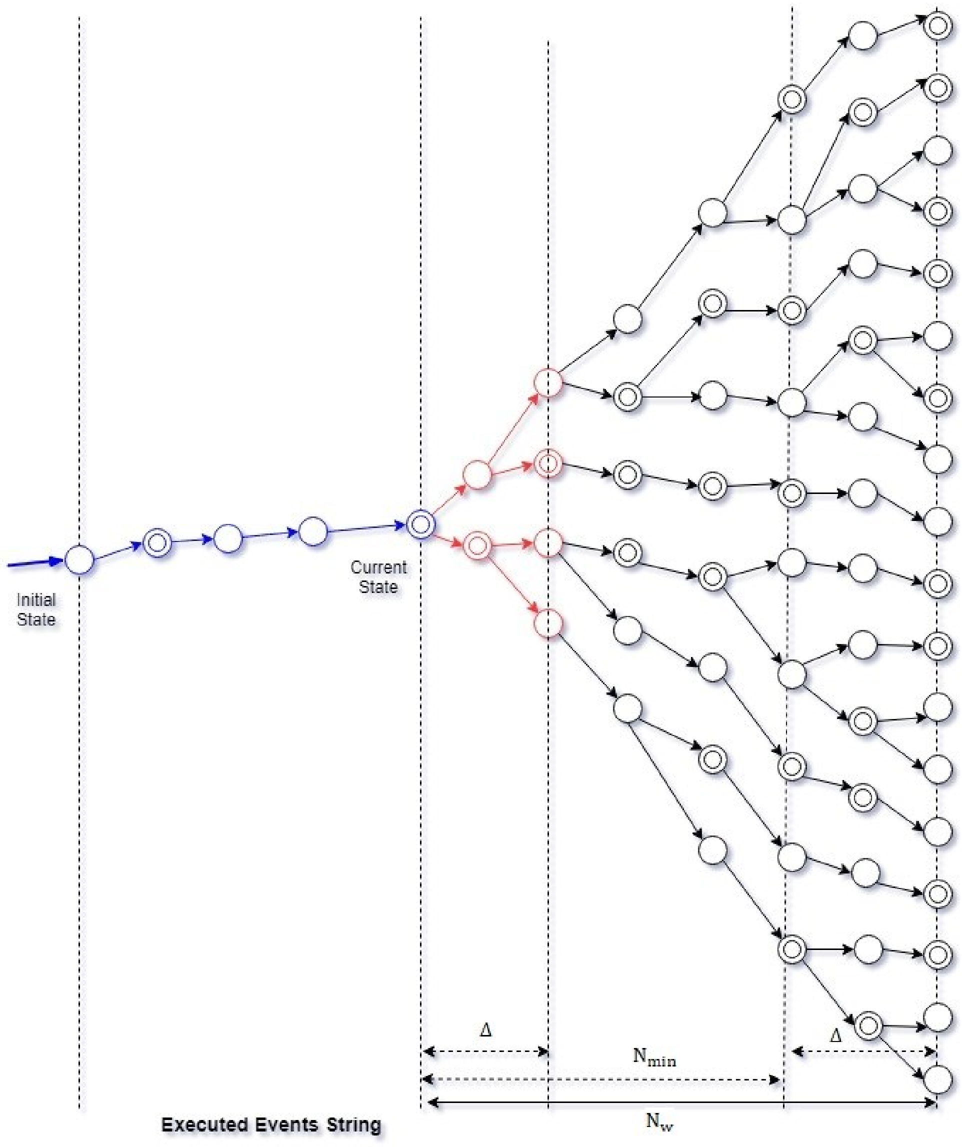


Figure 4.3-1: LLP with extended window size

It will be shown that by considering Δ steps beyond the available window size N_{min}, the calculated supervisor has the control commands of Δ steps after the current state (red zone in Figure 4.3-1). In other words, using an expansion window, enlarged by Δ events, we can calculate and "buffer" control commands for new and Δ events in the future. We will later explain how this property can be useful.

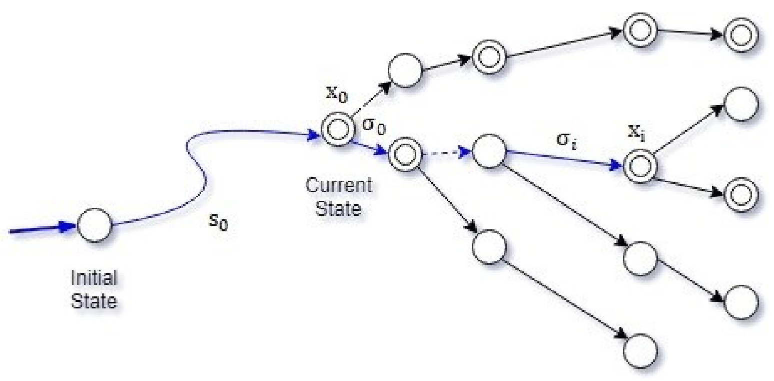


Figure 4.3-2: Tree expansion

The mentioned property can be simply derived as a corollary to the following theorem.

Theorem 4.1: [39]

If $s_0 \in \overline{SupC(K, \mathrm{L(G)})}$, then $SupC(K, \mathrm{L(G)})/s_0 = SupC(K/s_0, \mathrm{L(G)}/s_0)$.

Here the supremal controllable sublanguage of K with respect to the language which is generated by G and uncontrollable events set Σ_{uc}, is shown by $SupC(K, \mathrm{L(G)})$.

The theorem states that the post language of supremal controllable sublanguage of any string which is inside the closure of that language is equivalent to the supremal controllable sublanguage of post language of that string.

Therefore if a string which is executed inside the plant is allowed by the supervisor (s_0 in Figure 4.3-2), then any extension of this string inside the supervisor ($s_0\sigma_0 \dots \sigma_i$) has solely the same properties as it is outlined in Theorem 4.1. Without loss of generality, suppose that the limited lookahead supervisor window size which gives enabled events in the state x_0 in Figure 4.3-2, is extended to $\mathrm{N_{min}} + \Delta$, then the result of the supervisor is valid after the execution of Δ events from x_0 (the next event σ_i is valid until $\mathrm{i} \leq \Delta$).

In other words, as long as after the current state there is a window size with at least $\mathrm{N_{min}}$ events, the calculated supervisor is valid; therefore the calculated supervisor for the plant after any possible event (which is in the valid supervisor) provided that the front window size is greater than or equal to $\mathrm{N_{min}}$ (right-hand side of the equation in Theorem 4.1), is the same as the post language of the supervisor after those events with the window size of those events in addition to $\mathrm{N_{min}}$ (left-hand side of the equation). Simply, Δ events after the current state of the calculated supervisor in all valid possible paths are kept (i.e. buffered) to be used for future events.

We saw in Section 4.2 that the time between consecutive events can be too short for LLP calculations. However often, as it is the case for the solar tracker, events are not generated regularly. Sometimes a sequence of events is generated rapidly, followed by a long gap till the next event. One can take advantage of this sporadic occurrence of events to overcome the challenge of performing LLP calculations. If we calculate and *buffer* a sequence of commands *in advance* (so that the commands at any states are always ready) and perform these calculations on a sufficiently long time (over which the events are spread), then all LLP calculations can be done on time.

To formalize the above discussion, we introduce two functions.

Def. 1 Shortest Duration Function. $\mathrm{T}_{min}: \mathbb{N} \rightarrow \mathbb{R}$

$\mathrm{T}_{min}(n)$: The shortest duration of the execution of a sequence containing at most n events.

Def. 2 Longest LLP computation $C_{max}: \mathbb{N} \rightarrow \mathbb{R}$

$C_{max}(\mathrm{N_w})$: The longest computational time of LLP calculations for an expansion window of length $\mathrm{N_w}$.

$\mathrm{T_{min}}$ depends on the properties of the plant (and its supervisor) while C_{max} depends on the computational algorithms and the computer running the algorithms.

In the following, we will see how the above two functions are determined experimentally for the solar tracker and show how LLP with buffering can be developed to meet all LLP calculation deadlines. This discussion, in turn, leads to a general procedure for designing LLP with buffering.

4.3.2 LLP with Buffering and Its Design Procedure

In Table 4.2-1, the computation time of LLP is shown for $\mathrm{N_{min}}$=6 and obviously just for one step (Δ=1). To have a better understanding about LLP computation time when the window size $\mathrm{N_w}$ expands, the trajectory of a completed full sweep is considered as a sample string to take the LLP computation time and the size of lookahead window size is changed over a range ($\Delta \geq 1$).In this sequence of events, Sweep_Failure event happens because a bright direction is not found. All trajectory and the entire specification are involved. The normalized LLP computation time in second is illustrated in Figure 4.3-3 ($\mathrm{N_w} = \mathrm{N_{min}} + \mathrm{n} - 1$ and $\mathrm{N_{min}} = 6$). The considered sequence consists of 1433 events, and in every state, the LLP computation time is in the range of maximum (red line) and minimum (blue line) time. The computation time distribution for each LLP calculation is depicted by green circles and the average time is shown by pink line. The red line is $C_{max}(\mathrm{N_{min}} + n - 1)$. As it is expected by increasing the n, the size of expanded plant and specification grows dramatically. Although at the $\mathrm{N_w} = 11$ $(n = 6)$ which equals to the PD, there is no growth in the expanded plant, but the specification still expands until $\mathrm{N_w} = 24$ $(n = 19)$(very slowly near the end) which causes bigger size in the constructed supervisor and consequently the computation time increases steadily. After n =20 stable average computation time is clear in this graph. The data for $C_{max}(\mathrm{N_{min}} + n - 1)$ is given in Table 4.3-1.

Table 4.3-1: C_{max} for $n = 1$ to 29

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C_{max}	0.29	0.44	0.55	0.71	0.98	1.36	1.84	2.52	2.79	3.33	3.74	3.91	4.14	3.82	3.79
n	16	17	18	19	20	21	22	23	24	25	26	27	28	29	
C_{max}	4.37	3.44	3.63	4.26	4.18	2.13	2.18	2.11	2.08	2.07	2.05	2.06	2.08	2.11	

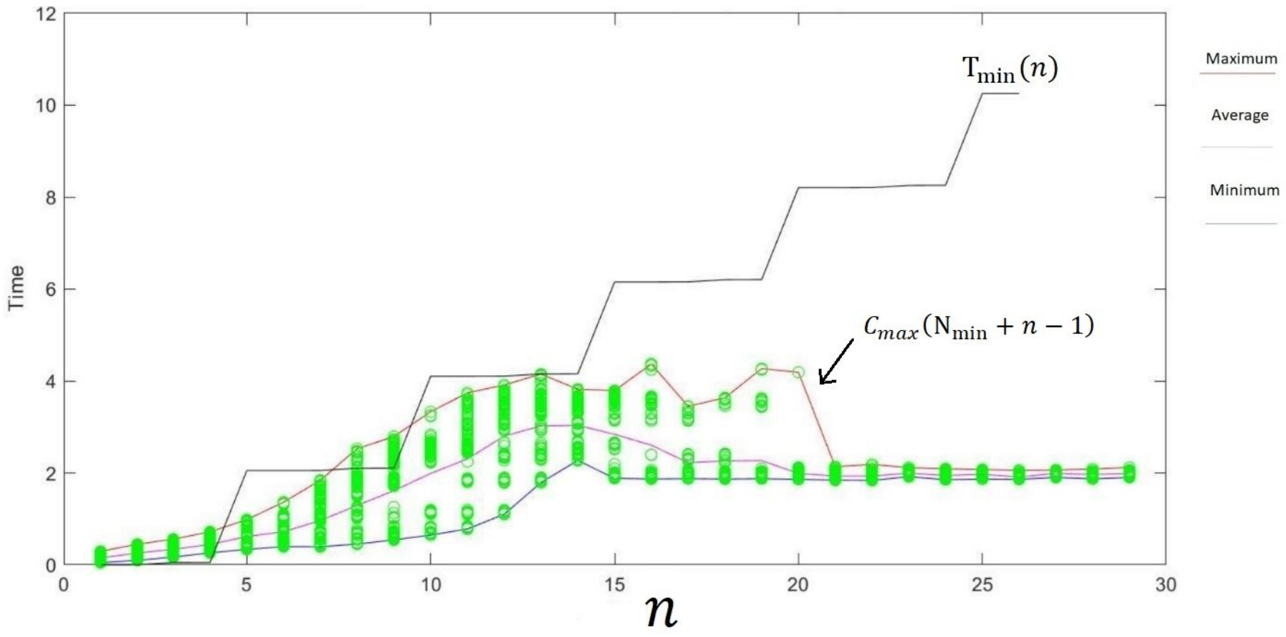


Figure 4.3-3: Normalized LLP computation time and occurrence time of the events vs. the size of LLP extension

The black line which is depicted in the above graph is $\mathrm{T_{min}}(n)$ obtained from plant under supervision of conventional supervisor and based on the same sequence studied for Table 3.4-2. As can be seen in Figure 4.3-3, the addition of every 5 events adds 2 seconds to the execution time. This gap creates a space for precalculating control commands using LLP. The data for $\mathrm{T}_{min}(n)$ is given in Table 4.3-2.

Table 4.3-2: T_{min} for n=1 to 26

n	1	2	3	4	5	6	7	8	9	10	11	12	13
$\mathrm{T}_{min}(n)$	0	0.002	0.05	0.052	2.051	2.051	2.05	2.10	2.10	4.10	4.10	4.10	4.15
n	14	15	16	17	18	19	20	21	22	23	24	25	26
$\mathrm{T}_{min}(n)$	4.15	6.15	6.15	6.15	6.20	6.20	8.20	8.20	8.20	8.25	8.25	10.25	10.25

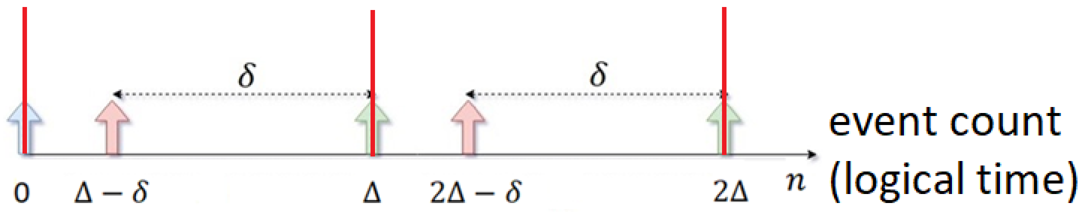


Figure 4.3-4: Timeline of LLP computation with buffering

In the proposed LLP with buffering, the LLP calculations are done in advance, i.e., the control commands are precalculated before they are needed. The timeline is shown in Figure 4.3-4. Note that the horizontal axis is the event count (logical time). The timeline is based on the following:

1. At the starting point ($n = 0$), before sending an initial command to initiate the system (blue arrow in above figure), the LLP should have been computed for $\mathrm{N_w} = \mathrm{N_{min}} + \Delta - 1$ to have enough responses for the upcoming Δ events (i.e, $0 \le n < \Delta$) and then the first command to start the system can be issued.
2. After $\Delta - \delta$ events (pink arrow), the LLP computation begins for the $\mathrm{N_w} = \mathrm{N_{min}} + \delta + \Delta - 1$. This will generate control commands for $\Delta - \delta \le n < 2\Delta$. This way the control commands for $\Delta \le n < 2\Delta$ are precalculated. These require

$$C_{max}(\mathrm{N_{min}} + \delta + \Delta - 1) \le \mathrm{T_{min}}(\delta) \quad \text{and} \quad \delta < \Delta$$

3. At $n = \Delta$ (green arrow), the supervisor calculated in step 2 has to be updated based on the δ events occurred from $n = \Delta - \delta$ to $n = \Delta$ and then it becomes the supervisor from $n = \Delta$ to $n = 2\Delta$. Obviously, after $\Delta - \delta$, at $n = 2\Delta - \delta$, step two has to be repeated, followed by step 3 and so on.

Now as an example, let us consider Figure 4.3-3 of the solar tracker. We have to choose parameter Δ and δ. If $\delta = 10$, then $\mathrm{T_{min}}(\delta)$ will be larger than any $C_{max}(\mathrm{N_{min}} + \delta + \Delta - 1)$. Thus Δ can be any number larger than 10. To leave some margin for error, we choose $\Delta = 15$ (which makes $\Delta - \delta$=5). Thus the LLP window will be $\mathrm{N_w}$=$\mathrm{N_{min}} + \delta + \Delta - 1 = 6 + 10 + 15 - 1 = 30$.

Next we will discuss the software implementation of LLP with buffering.

4.3.3 Generating code for LLP with Buffering

The flowchart for LLP with buffering is similar to that of the Figure 4.2-2. The section of flowchart related to the microcontroller remains the same and the right side of the flowchart is modified in order to respond to the received events by MATLAB at the highest priority. To accomplish this task, the part of the code which responds to received events has to be moved to the callback function which guarantees the readiness of responses in the case of any event occurrence while heavy LLP computation is being performed (The flowchart is depicted in Figure 4.3-5; the microcontroller part remains the same as Figure 4.2-2 and is not shown for brevity). Since the number of events plays a critical role in activating different tasks, the delta variable keeps the number of events in the buffer. Thus, at the starting point, delta equals Δ and after the occurrence of $\Delta - \delta$ events, it reaches δ that is the time to activate LLP computation. Therefore, at delta= δ, the LLP_execution bit is set to true and a copy of all models are kept till in the main function LLP_execution is checked and then the prepared copies are used to compute the supervisor. Meanwhile, the received events from the microcontroller are responded to in "instrcallback" function and are stored to be used at the time of delta=0.

Before delta approaches zero, the supervisor is constructed and a copy of this supervisor is sent to the callback function and at the time of delta=0, the supervisor is updated according to the recorded events very quickly and then it is ready for the next Δ events. At the end of the callback function if there is any controllable event among enabled events, it is sent to the microcontroller. The experimental results are reported in the next chapter.

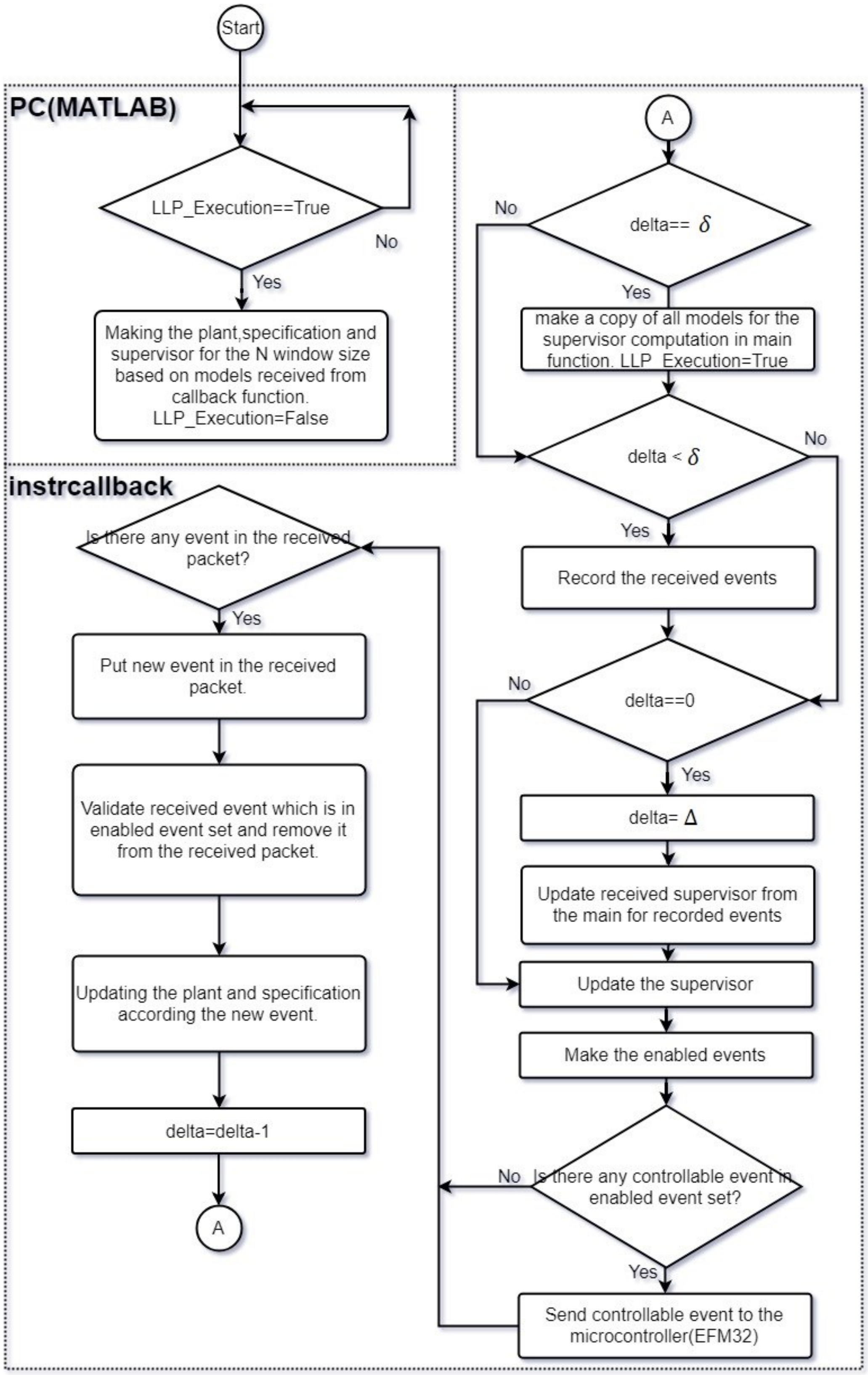

Start
PC(MATLAB)
LLP_Execution==True
No
Yes
Making the plant,specification and supervisor for the N window size based on models received from callback function. LLP_Execution=False
instrcallback
Is there any event in the received packet?
Yes
Put new event in the received packet.
Validate received event which is in enabled event set and remove it from the received packet.
Updating the plant and specification according the new event.
delta=delta-1
A
A
delta== δ
No
Yes
make a copy of all models for the supervisor computation in main function. LLP_Execution=True
delta < δ
No
Yes
Record the received events
delta==0
No
Yes
delta= Δ
Update received supervisor from the main for recorded events
Update the supervisor
Make the enabled events
Is there any controllable event in enabled event set?
No
Yes
Send controllable event to the microcontroller(EFM32)

Figure 4.3-5: Implementation of LLP with buffering flowchart

Chapter 5

Experimental Results

In this section we will discuss the experimental results of implementation of LLP with buffering on the solar tracker. First we review the setup and the tests conducted. Next we will explore the effect of factors that were not considered in the theoretical analysis of Section 4.3. Finally, we will compare the test results of LLP with Buffering with those of the experiences of Chapter 3 which was designed offline and implemented on the microcontroller. This setup under the supervision of Chapter 3 is used as the benchmark for evaluating LLP with buffering.

We implemented the flowchart of the previous chapter in the solar tracker system and performed several tests from various initial positions with respect to the light source. We observe that the LLP supervisor with buffering parameters $\delta = 10$ and $\Delta = 15$ successfully controlled the system, confirming the applicability of the proposed method.

To better understand the interactions between the software codes in the microcontroller and MATLAB, several sample points were added to track the code execution flow. Because the delta variable changes in the callback function and at delta= δ, the LLP_execution is set to true, but the LLP calculation is done inside the main function, in some cases in which two successive events are received by the callback function, LLP computation time starts after $\Delta - \delta$ has occurred (Figure 4.3-4). The actual $\mathrm{N_w}$ and computation time in Table 5-1 indicate that at most just one extra event can occur from the time which is set for the start off LLP computation. Moreover, the average of 29.7 for window size shows that most of the time LLP calculation starts as shown in Figure 4.3-4.

Table 5-1: LLP computation time and window size

	LLP computation time in seconds (MATLAB, Intel(R) Core™ i5-6200U)	**LLP window size (N_w)**
Maximum	2.59	30
Average	2.29	29.7
Minimum	2.03	29

Next let us explore the effects of the factors which were not considered in the previous section.

Callback Functions. Consider the timeline in Figure 5-1 in which the duration of LLP computation is depicted by two-side yellow arrow. The duration of LLP computation in Table 5-1 which contains callback function execution time shows that as expected from Figure 4.3-3, the average LLP computation time for $N_w \geq 29$ is around 2.29 seconds (Since the computation time in Figure 4.3-3 does not depend on sending and receing events from the microcontroller, the computation time in Figure 4.3-3 for $n = \Delta + \delta = 25$ are slightly smaller than the computation time in Table 5-1 in which **instrcallback** function interrupts LLP computation several times). Thus the callback function does not have any significant effect on execution time.

Another point which is observed from the timeline is that in spite of calling **instrcallback** function during the LLP computation, the computation finishes well before the deadline when delta is zero. Depending on the times **instrcallback** function interrupts this computation, the end of computation varies between delta=8 for the shorter interruption and delta=3 for longer interruption.

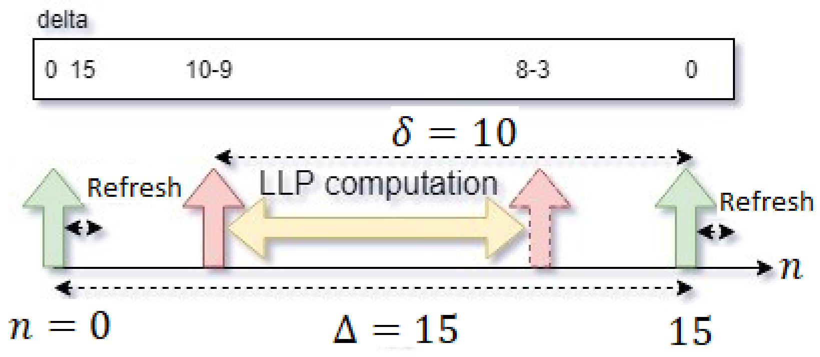


Figure 5-1: LLP implemented timeline

Supervisor Refreshing. As can be seen in Figure 5-1, right after when the delta is zero ($n = 15$), the LLP supervisor that was calculated must be initializd to take over control over the window of $n = 16$ to $n = 30$. We refer to this initialization time as the "refresh time". This refresh time should be quite small in order to avoid any interference for upcoming events. The average refresh time is 4 ms, with a maximum of 10 ms which is negligible. Specially, using Figure 4.3-3 with $\delta = 10$ and $\Delta = 15$, we see that

$$C_{max}(\mathrm{N}_{\min} + \delta + \Delta - 1) = C_{max}(\mathrm{N}_{\min} + 24) \simeq 2.2 \text{ sec.}$$

$$\mathrm{T}_{\min}(\delta) = \mathrm{T}_{\min}(10) \simeq 4.1 \text{ sec.}$$

This creates a margin of 4.1-2.2=1.9 sec. which is much more than 10ms refresh time.

Communication Delay. Now we compare the timelines of events in both LLP (Chapter 4) and conventional (Chapter 3) implementations. In the conventional system, the events are read and processed inside the microcontroller. On the other hand, in the LLP implementation, although uncontrollable events are generated and stamped inside the microcontroller, controllable events are sent from the supervisor in MATLAB to the microcontroller and then they are sent back to the MATLAB to be accepted as controllable events. Consequently, an unwanted communication delay is imposed on the system in LLP supervision. In fact, the communication delay over a window of $\Delta = 15$ events is around 300 ms which is much smaller than the margin of 1.9s mentioned in the previous paragraph.

Comparison with Conventional Supervisor. The plant under the supervision of the standard supervisor, designed offline (Chapter 3) is our benchmark and reference for evaluating LLP supervision with buffering. We note that any two executions of the two implementations may slightly differ from one another since some events such as those of PV cell and battery may occur at slightly different times. To compare the two supervisors, we have decided to use the duration of event sequences as a function of the length of the sequences (Figure 5-2). The execution times for the conventional system (the pink line) are slightly less than those of system supervised by LLP (the green line). The difference is minor and can be attributed mainly to the communication delays.

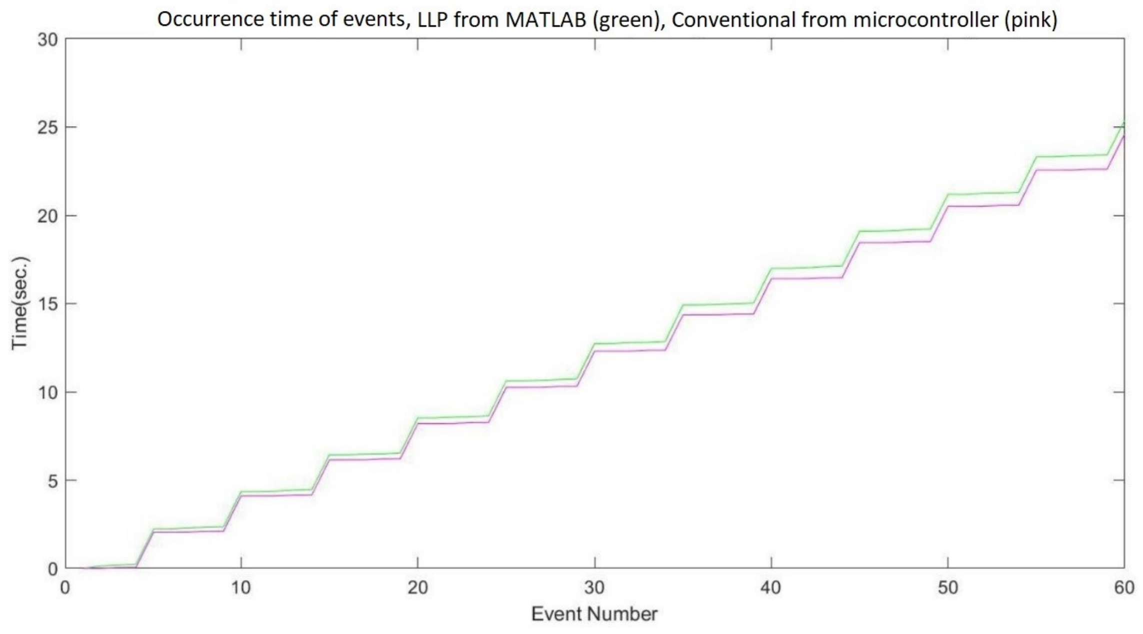


Figure 5-2: Occurrence time of events in LLP and conventional implementation

Chapter 6

Conclusion

6.1 Summary

In this thesis, the implementation issues of LLP supervisors are studied. To address the issue of computational delay in LLP supervision, a novel technique is proposed in which supervisory control commands are calculated in advance (and online) for a window of events in the future and buffered. When the window starts, the commands would be ready after each event. This eliminates the delay due to online calculations, and reduces the delay in responding to new events to levels close to those of conventional supervisors (designed "offline").

In an effort to assess the proposed methodology and better understand the implementation issues of SCT, a two degree-of-freedom solar tracker with two servo motors is selected as the plant. Previously, a conventional supervisor had been designed for this solar tracker to guide the tracker and perform a sweep to find a sufficiently bright direction to charge the battery and other parts of the system (from its Photo Voltaic cell). In this thesis, the conventional supervisor was improved. Next the LLP with buffering was implemented. Several experimental results confirmed that the plant under the supervision of LLP supervisor with buffering can match the behavior of the plant under the supervision of conventional supervisor.

To compare the offline and online implementations, firstly, memory requirement in both cases should be examined. The conventional supervisor is stored in the memory using a **struct** format. Thus, the minimum required flash memory which IDE dedicates for storing the full sweep supervisor is:

Memory Size= (Number of states x 6) + (Number of transitions x 4) +10

= (2061 x 6) + (9527 x 4) +10 =50,484 bytes.

Each element of **struct** is composed of a variable "len" and a two dimensional array. All variables in the format of an unsigned integer each requiring 2 bytes. Therefore, for any state 2 bytes for **len** have to be allocated, but the way IDE stores this **struct** has an overhead which adds 4 other bytes to each state. Furthermore, an array starts from 0 but the supervisor states start from 1 in TL, and to keep the same number a null element is added to **struct** array which takes 10 bytes. There are other methods (e.g. memory safe in [40]) for memory management but they do not have any specific advantage compared with the State Transition Table.

The amount of memory for code in LLP implementation is different from the conventional but it is negligible and therefore, since there is no need for allocation of specific amount of flash memory to store supervisor, it has a significant advantage over conventional implementation. One can argue that the required memory for conventional supervisor compared to the capacity of the flash memory in this microcontroller (256 Kbyte) is enough; nevertheless, for larger systems which have more components, much larger memory will be needed. For example, for a system with 100,000 states and 1,000,000 transitions, unsigned double integer has to be used to store numbers and then the minimum required memory is more than 8.7 Mbytes while the maximum flash memory of EFM32 series is 2 Mbytes.

6.2 Future Work

To realize an efficient implementation of LLP on a microcontroller (so as to have a more realistic comparison of conventional and LLP implementations), a fast and customized code for LLP computation for the microcontroller (as opposed to code generated from MATLAB) should be prepared in a way to decrease the computation time. Then after optimizing the computation time (in comparison to the scan time), the window size, N_w, can be chosen to be lower than PD and the LLP implementation will be beneficial compared with the conventional supervisor.

In this thesis, the sequence duration function T_{min} (Chapter 4) was obtained experimentally. It would be interesting to find a formal procedure to determine this function using a timed model of the plant under supervision.

Since in SCT, the uncontrollable events cannot be prevented from occurring, at any state, if an uncontrollable event in the system occurs, the supervisor must also follow that transition. Suppose at some state, the LLP calculation starts and before the enabled events set becomes ready, an uncontrollable event is detected. In this case, the LLP computation can be abandoned and the information about the current plant state can be updated. The LLP then should start its calculations from this new state. This enhances its performance. The result of an experiment with use of this method in solar tracker system shows that almost 20% of LLP computations can be avoided; this could considerably boost the performance of LLP.

Moreover, some other variations of LLP which are mentioned in Section 1.4.3 such as VLP could be equipped with buffering to make calculations more efficient in terms of computation time.

Even though the choice problem does not exist in the solar tracker, a rigorous solution must be found to address cases in which more than one controllable event can be enabled in one state.

References

[1] 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, *(12),* pp. 1921-1935, 1992. DOI: 10.1109/9.182478.

[2] C. R. Frost, "Challenges and opportunities for autonomous systems in space," Presented at the National Academy of Engineering's U.S. Frontiers of Engineering Symposium, 2010, pp. 17.

[3] N. Muscettola, P. Pandurang Nayak, Barney Pell, Brian C. Williams, "Remote Agent: To Boldly Go Where No AI System Has Gone Before," *Artif. Intell.,* vol. 103, *(1-2),* pp. 5-47, 1998. DOI: 10.1016/S0004-3702(98)00068-X.

[4] B. C. Williams and P. P. Nayak, "A model-based approach to reactive self-configuring systems,"*Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2,* 1996, pp. 971-978.

[5] M. Pekala, G. Cancro and J. Moore, "Verifying executable specifications of spacecraft autonomy," *Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space,* 2008, pp. 8.

[6] B. C. Williams, M.D. Ingham, S.H. Chung, P.H. Elliott "Model-based programming of intelligent embedded systems and robotic space explorers," *Proceedings of the IEEE,* vol. 91, *(1),* pp. 212-237, 2003. DOI: 10.1109/JPROC.2002.805828.

[7] S. Bensalem, L. Silva, M. Gallien, R. Yan, ""Rock solid" software: A verifiable and correct-by-construction controller for rover and spacecraft functional levels," International Symposium on *Artificial Intelligence*, Robotics and Automation for Space, 2010, pp. 9.

[8] P. J. Ramadge and W. M. Wonham, "The control of discrete event systems," *Proceedings of the IEEE,* vol. 77, *(1),* pp. 81-98, 1989. DOI: 10.1109/5.21072.

[9] P. J. Ramadge and W. M. Wonham, "Supervisory Control of a Class of Discrete Event Processes," *SIAM J.Control Optim.,* vol. 25, *(1),* pp. 206-230, 1987. DOI: 10.1137/0325013.

[10] C. M. Ozveren and A. S. Willsky, "Observability of discrete event dynamic systems," *IEEE Transactions on Automatic Control,* vol. 35, *(7),* pp. 797-806, 1990. DOI: 10.1109/9.57018.

[11] F. Lin and W. M. Wonham, "On Observability of Discrete-event Systems," *Inf. Sci.,* vol. 44, *(3),* pp. 173-198, 1988. DOI: 10.1016/0020-0255(88)90001-1.

[12] P. Malik, "Generating controllers for discrete-event models." Proceedings of MOdelling and VErification of parallel Processes, Nantes, France, 2002.

[13] S. R. Mohanty, V. Chandra and R. Kumar, "A computer implementable algorithm for the synthesis of an optimal controller for acyclic discrete event processes," Proceedings 1999 IEEE International Conference on Robotics and Automation1, 1999, pp. 126-130, DOI: 10.1109/ROBOT.1999.769942.

[14] V. Chandra, Z. 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, *(2),* pp. 284-289, 2003. DOI: 10.1109/TSMCC.2003.813152.

[15] J. W. P. Geurts, "Supervisory Control of MRI Subsystems", Master's Thesis, Eindhoven University of Technology, Eindhoven, 2012.

[16] R. H. J. 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] A. D. Vieira , E. A. P. Santos, M. H. de Queiroz, A.B. Leal, A. D. de P. Neto, J. E. R. Cury "A Method for PLC Implementation of Supervisory Control of Discrete Event Systems," *IEEE Transactions on Control Systems Technology,* vol. 25, *(1),* pp. 175-191, 2017. DOI: 10.1109/TCST.2016.2544702.

[18] M. H. de Queiroz and J. E. R. Cury, "Synthesis and implementation of local modular supervisory control for a manufacturing cell," Proceedings of the Sixth International Workshop on Discrete Event Systems, Zaragoza, Spain, 2002, pp. 6, DOI: 10.1109/WODES.2002.1167714.

[19] F. Göbe, T. Timmermanns, O. Ney, S. Kowalewski "Synthesis tool for automation controller supervision," 2016 13th International Workshop on Discrete Event Systems, Xi'an, China, 2016, pp. 424-431, DOI: 10.1109/WODES.2016.7497883.

[20] J. N. Tsitsiklis, "On the control of discrete-event dynamical systems," *Mathematics of Control, Signals and Systems,* vol. 2, *(2),* pp. 95-107, 1989. DOI: 10.1007/BF02551817.

[21] Z. A. Banaszak and B. H. Krogh, "Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows," *IEEE Transactions on Robotics and Automation,* vol. 6, *(6),* pp. 724-734, 1990. DOI: 10.1109/70.63273.

[22] C. A. Brooks, R. Cieslak and P. Varaiya, "A method for specifying, implementing, and verifying media access control protocols," *IEEE Control Systems Magazine,* vol. 10, *(4),* pp. 87-94, 1990. DOI: 10.1109/37.56282.

[23] P. Gawrychowski, D. Krieger, N. Rampersad, J. Shalli, "Finding the growth rate of a regular of context-free language in polynomial time," International Conference on Developments in Language Theory, Kyoto, Japan, 2008, pp. 339-358, DOI: 10.1007/978-3-540-85780-8_27.

[24] C. Winacott, B. Behinaein and K. Rudie, "Methods for the estimation of the size of lookahead tree state-space," *Discrete Event Dynamic Systems,* vol. 23, *(2),* pp. 135-155, 2013. DOI: 10.1007/s10626-012-0138-y.

[25] S. L. Chung, S. Lafortune and F. Lin, "Recursive computation of limited lookahead supervisory controls for discrete event systems," *Proceedings of the 31st IEEE Conference on Decision and Control, Tucson, AZ, USA,* 1992, pp. 3764-3769, DOI: 10.1109/CDC.1992.370956.

[26] S. Chung, S. Lafortune and F. Lin, "Supervisory control using variable lookahead policies," *Discrete Event Dynamic Systems,* vol. 4, *(3),* pp. 237-268, 1994. DOI: 10.1007/BF01438709.

[27] N. B. Hadj-Alouane, S. Lafortune and F. Lin, "Variable lookahead supervisory control with state information," *IEEE Transactions on Automatic Control,* vol. 39, *(12),* pp. 2398-2410, 1994. DOI: 10.1109/9.362854.

[28] M. Fabian and A. Hellgren, "PLC-based implementation of supervisory control for discrete event systems," Proceedings of the 37th IEEE Conference on Decision and Control, Tampa, FL, USA, 1998, pp. 3305-3310, DOI: 10.1109/CDC.1998.758209.

[29] A. B. Leal, D. L. L. da Cruz and M. d. S. Hounsell, "Supervisory control implementation into programmable logic controllers," IEEE Conference on Emerging Technologies & Factory Automation, Mallorca, Spain, 2009, pp. 7, DOI: 10.1109/ETFA.2009.5347090.

[30] K. Searle and S. Hashtrudi-Zad, "Microcontroller based supervisory control of a solar tracker," IEEE 30th Canadian Conference on Electrical and Computer Engineering, Windsor, ON, Canada, 2017, pp. 6, DOI: 10.1109/CCECE.2017.7946686.

[31] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems.* (2nd ed.), New York, Springer, 2010.

[32] W. M. Wonham and P. J. Ramadge, "On the supremal controllable sublanguage of a given language," The 23rd IEEE Conference on Decision and Control, Las Vegas, Nevada, USA, pp. 1073-1080, 1984. DOI: 10.1109/CDC.1984.272178.

[33] S. Lafortune and E. Chen, "The infimal closed controllable superlanguage and its application in supervisory control," *IEEE Transactions on Automatic Control,* vol. 35, *(4),* pp. 398-405, 1990. DOI: 10.1109/9.52291.

[34] Mathworks Inc, "MATLAB," Available: *https://www.mathworks.com/products/matlab.html.*

[35] S. Balemi, "*Control of Discrete Event Systems: Theory and Application," Ph.D. dessertation, ETH Zurich,* 1992.

[36] A. C. Shaw, *Real-Time Systems and Software. New York, NY, USA: John Wiley \& Sons, Inc.* (1st ed.) 2000.

[37] S. Hashtrudi Zad, S. Zahirazami and F. Boroomand, "Discrete Event Control Kit," *DECK,* 2013, Department of Electrical and Computer Engineering, Concordia University, Available: http://users.encs.concordia.ca/~shz/deck.

[38] R. Kumari, H. M. Cheung and S. I. Marcus, "Extension based limited lookahead control for discrete event systems," Proceedings of 35th IEEE Conference on Decision and Control, Kobe, Japan, 1996, pp. 2225-2230, DOI: 10.1109/CDC.1996.572975.

[39] S. L. Chung, S. Lafortune and F. Lin, "Addendum to 'Limited lookahead policies in supervisory control of discrete event systems':Proofs of technical results," College of Eng. Contr. Group Reps., Univ. of Michigan, Tech. Rep. Tech. Rep. CGR-92-6, Apr. 1992.

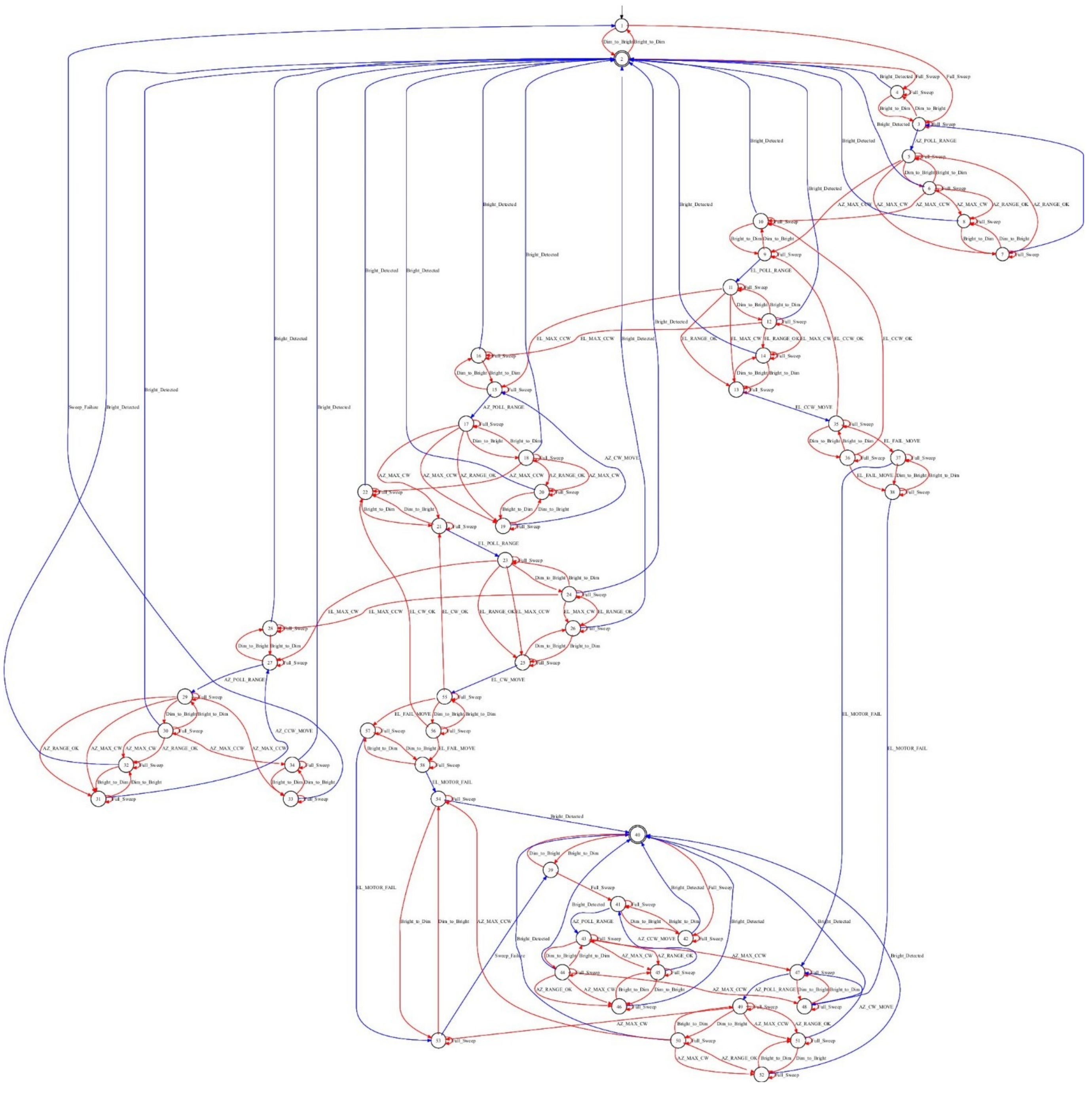
[40] Y. Kaszubowski Lopes, A. Leal, R. Rosso Jr., E. Harbs "Local modular supervisory implementation in microcontroller," 9 th International Conference of Modeling, Optimization and Simulation, Bordeaux, France, 2012, pp. 6.

[41] W.M.Wonham, Supervisory Control of Discrete Event Systems, ECE Dept., Univ. of Toronto, 2015.

[42] K. Searle, "Microcontroller Based Supervisory Control of a Solar Tracker", Master's Thesis, Concordia University, Montreal, 2016.

Appendix A

Full Sweep Specification Model



Appendix B

List of customized C language files

Table appendix B: List of customized C language files

Filename	Function	Changes
thesis_adc.c	Analog to Digital Converting of input signals	-Adding moving average
thesis_dma.c	Direct Memory Access	-
thesis_events.c	Generating uncontrollable events according to the input signals and sending controllable events as out signals	-Events are generated inside stored models. -Time stamping
thesis_i2c.c	Reading the SOC and voltage of the battery	-
thesis_pwm.c	Sending moving commands to the servomotors	-
thesis_supervisor_new	Finding next state of offline supervisor	-Using of generated array events (offline mode only)
timer.c	To call functions in specific timer interrupts	-All functions are executed in this routine.
thesis_uart.c	Sending and receiving data between microcontroller and PC	-Some new packets are added -Adding functions in the time of receiving controllable events
app.c	Main function	-No task is defined

Appendix C

Communication between the microcontroller and the PC

To send and receive data between the microcontroller and the PC, the data are collected in some packets in the following frame:

!XX@YYYYY&

Each packet has ten ASCII characters inside. The ! and & character denote for initial and end of the packets respectively. XX as a header identifies the data type and value of data is in the format of YYYY and they are separated by @ character. List of all packets which are used in this thesis is shown in Table Appendix C-0-1. According to this table, each packet is sent from microcontroller to the PC or in reverse direction and they are used in each mode of offline or online or both. For packets which have to deliver a specific value (e.g. voltage, current or a number), the second part of the packet which consists of four characters is used but for other packets which convey an event (e.g. Full_Sweep), just the header is set and recognized in the packets.

Table Appendix C-0-1: List of packets in communication between the microcontroller and the PC.

Header	**Packet Name**	**Sender**	**Used in Mode**
00	Start events	PC	Online
01	Elevation servomotor current	Microcontroller	Offline/Online
02	Azimuth servomotor current	Microcontroller	Offline/Online
03	PV cell voltage	Microcontroller	Offline/Online
04	System Time	Microcontroller	Offline/Online
05	Battery voltage	Microcontroller	Offline/Online
06	Battery SOC	Microcontroller	Offline/Online
07	Offline supervisor state number	Microcontroller	Offline
08	Event number	Microcontroller	Online
15	Full_Sweep	PC	Offline/Online
20	Bright_Detected	Microcontroller	Offline
21	Sweep_Failure	Microcontroller	Offline
22	EL_MOTOR_FAIL	Microcontroller	Offline
23	AZ_POLL_RANGE	PC	Online
24	EL_POLL_RANGE	PC	Online
25	AZ_CW_MOVE	PC	Online
26	AZ_CCW_MOVE	PC	Online
27	EL_CW_MOVE	PC	Online
28	EL_CCW_MOVE	PC	Online
29	Bright_Detected	PC	Online
30	Sweep_Failure	PC	Online
31	EL_MOTOR_FAIL	PC	Online
32	ONLINE	PC	Online

Appendix D

MATLAB Code

Online Supervisory control computation:

```
% Online supervisory control of 2 Degrees Solar Tracker with buffering
% Ehsan Ghaheri, June 2018
% Number of offline supervisor states     :2061
% Number of offline supervisor transitions:9527
%
clc;
clear;
%If a variable with the same name as the global variable already exists in the current workspace,
%MATLAB issues a warning and changes the value of that variable and its scope to match the global variable.
clear global variable;
global Online Nw delta delta_N small_delta;
delta=0;
Auto=input('Auto(For predefined N)1=Yes 0=No ?\n');
if Auto==1
    Online=1;
    all_marked=1;
    delta_N=15
    small_delta=10
else
    Online=input('Online(LLP Computation)1=Yes 0=No?\n');
    if Online==1
        all_marked=input('All states marked? 1=Yes 0=No\n');
        delta_N=input('Big Delta?\n');
        Nw=delta_N+5;%It has been tested as depth of lookahead window with plenty of runs.
        small_delta=input('Small Delta N?\n');
    else
        all_marked=0;
    end
end
global Serial;
Serial=input('Serial Port Available? 1=Yes 0=No\n');
%
% Plant Components
%
% Battery State of Charge

Safe_to_Full = 601;
Full_to_Safe = 602;
Crit_to_Safe = 603;
Safe_to_Crit = 604;
if all_marked==1
    Bat_SOC_Marked_States =(1:3) ;
else
    Bat_SOC_Marked_States =[2];
end
Bat_SOC_STT = [1 Safe_to_Full 2 ; 2 Full_to_Safe 1 ; 3 Crit_to_Safe 1 ; 1 Safe_to_Crit 3];
Bat_SOC = automaton(3, Bat_SOC_STT, Bat_SOC_Marked_States);

%
% PV Cell Illumination
%

Dark_to_Dim = 301;
Dim_to_Bright = 302;
Dim_to_Dark = 303;
Bright_to_Dim = 304;
if all_marked==1
    PV_Marked_States =(1:3) ;
else
    PV_Marked_States = [3];
```

```
end
PV_STT = [1 Dark_to_Dim 2 ; 2 Dim_to_Bright 3 ; 3 Bright_to_Dim 2 ; 2 Dim_to_Dark 1];
PV_Cell = automaton(3, PV_STT, PV_Marked_States);

%
% Motor  Motion
%

% Azimuth
AZ_CCW_OK = 401;
AZ_CW_OK = 402;
AZ_CCW_MOVE = 403;
AZ_CW_MOVE = 404;
if all_marked==1
    AZ_Motor_Motion_Marked_States =(1:3) ;
else
    AZ_Motor_Motion_Marked_States =[1];
end
AZ_Motor_Motion_STT = [1 AZ_CW_MOVE 2 ; 2 AZ_CW_OK 1 ; 1 AZ_CCW_MOVE 3 ; 3 AZ_CCW_OK 1];
AZ_Motor_Motion = automaton(3, AZ_Motor_Motion_STT, AZ_Motor_Motion_Marked_States);

% Elevation
EL_CCW_OK = 451;
EL_CW_OK = 452;
EL_CCW_MOVE = 453;
EL_CW_MOVE = 454;
EL_FAIL_MOVE = 455;

if all_marked==1
    EL_Motor_Motion_Marked_States =[1:4] ;
else
    EL_Motor_Motion_Marked_States =[1,4];
end
EL_Motor_Motion_STT = [1 EL_CW_MOVE 2 ; 2 EL_CW_OK 1 ; 1 EL_CCW_MOVE 3 ; 3 EL_CCW_OK 1; 2 EL_FAIL_MOVE 4; 3 ↙
EL_FAIL_MOVE 4];
EL_Motor_Motion = automaton(4, EL_Motor_Motion_STT, EL_Motor_Motion_Marked_States);

% Timing delay  for 2 seconds for each  servomotors movement.
%
wait_2sec=480;
Wait_Marked_States=[1:3];
Wait_STT=[1 AZ_CCW_MOVE 2;1 AZ_CW_MOVE 2;1 EL_CCW_MOVE 2;1 EL_CW_MOVE 2;2 wait_2sec 3 ;3 AZ_CW_OK 1;3 ↙
AZ_CCW_OK 1;3 EL_CW_OK 1;3 EL_CCW_OK  1];
Wait=automaton(3,Wait_STT,Wait_Marked_States);

%
% Motor  Range
%

% Azimuth
AZ_MAX_CW = 410;
AZ_MAX_CCW = 411;
AZ_RANGE_OK = 412;
AZ_POLL_RANGE = 425;
if all_marked==1
    AZ_Motor_Range_Marked_States = (1:4);
else
    AZ_Motor_Range_Marked_States = [1,2,3];
end
AZ_Motor_Range_STT = [1 AZ_POLL_RANGE 4; 4 AZ_RANGE_OK 1; 4 AZ_MAX_CW 2; 4 AZ_MAX_CCW 3; 3 AZ_POLL_RANGE 4; 2 ↙
AZ_POLL_RANGE 4];
AZ_Motor_Range = automaton(4, AZ_Motor_Range_STT, AZ_Motor_Range_Marked_States);

% Elevation
EL_MAX_CW = 460;
EL_MAX_CCW = 461;
EL_RANGE_OK = 462;
EL_POLL_RANGE = 435;
if all_marked==1
    EL_Motor_Range_Marked_States =(1:4) ;%
else
    EL_Motor_Range_Marked_States = [1,2,3];
end
EL_Motor_Range_STT = [1 EL_POLL_RANGE 4; 4 EL_RANGE_OK 1; 4 EL_MAX_CW 2; 4 EL_MAX_CCW 3; 3 EL_POLL_RANGE 4; 2 EL_POLL_RANGE 4];
EL_Motor_Range = automaton(4, EL_Motor_Range_STT, EL_Motor_Range_Marked_States);

%
% Master  Controller
%

Full_Sweep = 504;
Bright_Detected = 510;
Sweep_Failure = 511;
EL_MOTOR_FAIL = 512;

MC_Marked_States = [1];
MC_STT = [1 Full_Sweep 1; 1 Bright_Detected 1; 1 Sweep_Failure 1 ; 1 EL_MOTOR_FAIL 1];
MC = automaton(1, MC_STT, MC_Marked_States);

%
% Interactions
%

%The Motors  cannot move  when  the  battery state  of charge is Critical.
if all_marked==1
    Mot_Motion_f_Bat_SOC_Marked_States =(1:4) ;
else
    Mot_Motion_f_Bat_SOC_Marked_States =[1, 2] ;
end
Mot_Motion_f_Bat_SOC_STT = [Bat_SOC_STT; 2 AZ_CCW_OK 2; 2 AZ_CW_OK 2; 2 AZ_CW_MOVE 2; 2 AZ_CCW_MOVE 2; 3 AZ_CCW_OK 3; 3 AZ_CW_OK 3; 1 AZ_CCW_OK 1; 1 AZ_CW_OK 1; 1 AZ_CW_MOVE 1; 1 AZ_CCW_MOVE 1];
Mot_Motion_f_Bat = automaton(4, Mot_Motion_f_Bat_SOC_STT, Mot_Motion_f_Bat_SOC_Marked_States);

% The Battery  State  of Charge  varies  as  a  function  of the  brightness on  the
% PV Cell.

% Battery  State  of charge as  a  function  of Solar  Cell Brightness
if all_marked==1
    Bat_SOC_f_PV_Marked_States =(1:3) ;
else
    Bat_SOC_f_PV_Marked_States = [3];
end
Bat_SOC_f_PV_STT = [PV_STT; 1 Safe_to_Crit 1; 1 Full_to_Safe 1; 2 Safe_to_Crit 2; 2 Full_to_Safe 2; 2 Crit_to_Safe 2; 2 Safe_to_Full 2; 3 Safe_to_Crit 3; 3 Full_to_Safe 3; 3 Crit_to_Safe 3; 3 Safe_to_Full 3];
Bat_SOC_f_PV = automaton(3, Bat_SOC_f_PV_STT, Bat_SOC_f_PV_Marked_States);

% The Battery  State  of Charge  varies  as  a  function  of the  motor state  (both
% EL and  AZ)
% Battery  State  of charge as  a  function  of Solar  Cell Brightness.
```

```

[Bat_SOC_f_Motor_Motions, states]  = sync(AZ_Motor_Motion, EL_Motor_Motion);
for i=1:size(states,1)

  if (states(i,1)  ==  1  && states(i,2)  ==  1)
    Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL;  i  Safe_to_Full  i];
    Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL;  i  Full_to_Safe  i];
    Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL;  i  Crit_to_Safe  i];
    Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Safe_to_Crit i];

  elseif (states(i,1)  ==  1  && states(i,2)  ==  4)
    Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL;  i  Safe_to_Full  i];
  Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL;  i  Full_to_Safe  i];
  Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL;  i  Crit_to_Safe  i];
  Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL;  i  Safe_to_Crit  i];
  else
    Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL;  i  Safe_to_Crit  i];
    Bat_SOC_f_Motor_Motions.TL=[Bat_SOC_f_Motor_Motions.TL; i Full_to_Safe i];

  end
end

% Specifications:

% Motor  Motion  and  Motor  Range  need to  be  synchronized - This is performed
% as  a  Specification  as  we  are  imposing  a  set  of  rules  on  how  the  system
% behaves.

% The motors cannot move  when  CCW when  in the  Max CCW state.
% The motors cannot move  CW when  in the  Max CW state.
% When  the  motors are  "In Range"  there  is no  limit on  the  motion

% Azimuth
if all_marked==1
    Spec_AZ_M_Motion_f_M_Range_MarkedStates =(1:4) ; %In the  previous  ver. it was [1,2,3]
else
    Spec_AZ_M_Motion_f_M_Range_MarkedStates =[1,2,3];
end
Spec_AZ_M_Motion_f_M_Range_STT = [AZ_Motor_Range_STT; 1 AZ_CW_MOVE 1; 1 AZ_CCW_MOVE 1; 2 AZ_CCW_MOVE 2; 3  ↙
AZ_CW_MOVE 3];
Spec_AZ_M_Motion_f_M_Range = automaton(4, Spec_AZ_M_Motion_f_M_Range_STT,  ↙
Spec_AZ_M_Motion_f_M_Range_MarkedStates);

E_Not_In_AZ_M_Motion_f_M_Range_Spec = [Dark_to_Dim, Dim_to_Dark, Bright_to_Dim, Dim_to_Bright, Safe_to_Full,  ↙
Full_to_Safe, Crit_to_Safe, Safe_to_Crit, Full_Sweep, Bright_Detected, Sweep_Failure, EL_MOTOR_FAIL, AZ_CCW_OK,  AZ_CW_OK,  ↙
EL_POLL_RANGE, EL_RANGE_OK, EL_MAX_CW, EL_MAX_CCW, EL_FAIL_MOVE, EL_CW_MOVE,  EL_CCW_MOVE,  EL_CW_OK,  ↙
EL_CCW_OK,wait_2sec];

global  Spec_AZ_M_Motion_f_M_Range_SelfLooped Spec_AZ_M_Motion_f_M_Range_SelfLooped_tmp;
Spec_AZ_M_Motion_f_M_Range_SelfLooped = selfloop(Spec_AZ_M_Motion_f_M_Range,  ↙
E_Not_In_AZ_M_Motion_f_M_Range_Spec);

% Elevation
if all_marked==1
    Spec_EL_M_Motion_f_M_Range_MarkedStates =(1:4) ;
else
    Spec_EL_M_Motion_f_M_Range_MarkedStates =[1,2,3];
end
Spec_EL_M_Motion_f_M_Range_STT = [EL_Motor_Range_STT; 1 EL_CW_MOVE 1; 1 EL_CCW_MOVE 1; 2 EL_CCW_MOVE 2; 3  ↙
EL_CW_MOVE 3];
```

```
Spec_EL_M_Motion_f_M_Range = automaton(4, Spec_EL_M_Motion_f_M_Range_STT, ↙
Spec_EL_M_Motion_f_M_Range_MarkedStates);

E_Not_In_EL_M_Motion_f_M_Range_Spec = [Dark_to_Dim, Dim_to_Dark, Bright_to_Dim, Dim_to_Bright, Safe_to_Full, Full_to_Safe, ↙
Crit_to_Safe, Safe_to_Crit, Full_Sweep, Bright_Detected, Sweep_Failure, EL_MOTOR_FAIL, EL_FAIL_MOVE, AZ_CCW_OK, AZ_CW_OK, ↙
AZ_CW_MOVE, AZ_CCW_MOVE, AZ_POLL_RANGE, AZ_RANGE_OK, AZ_MAX_CW, AZ_MAX_CCW, EL_CCW_OK, EL_CW_OK,wait_2sec];

global Spec_EL_M_Motion_f_M_Range_SelfLooped Spec_EL_M_Motion_f_M_Range_SelfLooped_tmp;
Spec_EL_M_Motion_f_M_Range_SelfLooped = selfloop(Spec_EL_M_Motion_f_M_Range, ↙
E_Not_In_EL_M_Motion_f_M_Range_Spec);

% Motor Range as a function of Motor Motion
% When the Motors are Turning CW, it can generate the "AZ_CW_OK" signal, all other signals are suppresed. (Cannot add - will ↙
make it non-deterministic)
% When the Motors are Turning CCW, it can generate the "AZ_CCW_OK" signal, all other signals are suppresed. (Cannot add - ↙
will make it non-deterministic)
% When the Motors are Idle, it can generate the AZ_MAX_CCW and AZ_MAX_CW signals


% Azimuth
if all_marked==1
    Spec_AZ_M_Range_f_M_Motion_MarkedStates =(1:3) ; %In the previous ver. it was [1]
else
    Spec_AZ_M_Range_f_M_Motion_MarkedStates =[1];
end
Spec_AZ_M_Range_f_M_Motion_MarkedStates_STT = [AZ_Motor_Motion_STT; 1 AZ_MAX_CW 1; 1 AZ_MAX_CCW 1; 1 ↙
AZ_RANGE_OK 1; 1 AZ_POLL_RANGE 1];
Spec_AZ_M_Range_f_M_Motion = automaton(3, Spec_AZ_M_Range_f_M_Motion_MarkedStates_STT, ↙
Spec_AZ_M_Range_f_M_Motion_MarkedStates);

E_Not_In_AZ_M_Range_f_M_Motion_Spec = [Dark_to_Dim, Dim_to_Dark, Bright_to_Dim, Dim_to_Bright, Safe_to_Full, ↙
Full_to_Safe, Crit_to_Safe, Safe_to_Crit, Full_Sweep, Bright_Detected,Sweep_Failure, EL_MOTOR_FAIL, EL_CW_MOVE, EL_CCW_MOVE, ↙
EL_CW_OK, EL_CCW_OK, EL_FAIL_MOVE, EL_POLL_RANGE, EL_RANGE_OK, EL_MAX_CW, EL_MAX_CCW,wait_2sec];

global Spec_AZ_M_Range_f_M_Motion_SelfLooped Spec_AZ_M_Range_f_M_Motion_SelfLooped_tmp;
Spec_AZ_M_Range_f_M_Motion_SelfLooped = selfloop(Spec_AZ_M_Range_f_M_Motion, ↙
E_Not_In_AZ_M_Range_f_M_Motion_Spec);

% Elevation
if all_marked==1
    Spec_EL_M_Range_f_M_Motion_MarkedStates =(1:4) ; %In the previous ver. it was [1,4]
else
    Spec_EL_M_Range_f_M_Motion_MarkedStates =[1,4];
end
Spec_EL_M_Range_f_M_Motion_MarkedStates_STT = [EL_Motor_Motion_STT; 1 EL_MAX_CW 1; 1 EL_MAX_CCW 1; 1 ↙
EL_RANGE_OK 1; 1 EL_POLL_RANGE 1];
Spec_EL_M_Range_f_M_Motion = automaton(4, Spec_EL_M_Range_f_M_Motion_MarkedStates_STT, ↙
Spec_EL_M_Range_f_M_Motion_MarkedStates);

E_Not_In_EL_M_Range_f_M_Motion_Spec = [Dark_to_Dim, Dim_to_Dark, Bright_to_Dim, Dim_to_Bright, Safe_to_Full, Full_to_Safe, ↙
Crit_to_Safe, Safe_to_Crit, Full_Sweep, Bright_Detected,Sweep_Failure, EL_MOTOR_FAIL, AZ_CCW_OK, AZ_CW_OK, AZ_CW_MOVE, ↙
AZ_CCW_MOVE, AZ_POLL_RANGE, AZ_RANGE_OK, AZ_MAX_CW, AZ_MAX_CCW,wait_2sec];

global Spec_EL_M_Range_f_M_Motion_SelfLooped Spec_EL_M_Range_f_M_Motion_SelfLooped_tmp;
Spec_EL_M_Range_f_M_Motion_SelfLooped = selfloop(Spec_EL_M_Range_f_M_Motion, ↙
E_Not_In_EL_M_Range_f_M_Motion_Spec);

% The final spec defines the behaviour of the system should take when a
% Sweep command is received.

```

```
Spec_Total_States = 58;
if all_marked==1
   Sweep_Spec_MarkedStates =(1:58) ;
else
   Sweep_Spec_MarkedStates =[2,40];
end
Sweep_Spec_STT = [1 Dim_to_Bright 2 ; 2 Bright_to_Dim 1 ; 1 Full_Sweep 3 ; 2 Full_Sweep 4 ; 4 Bright_Detected 2 ; 3 ↙
Dim_to_Bright 4 ; 4 Bright_to_Dim 3 ; 3 AZ_POLL_RANGE 5 ; 5 Dim_to_Bright 6 ; 6 Bright_to_Dim 5 ; 5 AZ_RANGE_OK 7 ; 5 ↙
AZ_MAX_CW 7 ; 7 AZ_CCW_MOVE 3 ; 7 Dim_to_Bright 8 ; 8 Bright_to_Dim 7 ; 5 AZ_MAX_CCW 9 ; 9 Dim_to_Bright 10 ; 10 ↙
Bright_to_Dim 9 ; 9 EL_POLL_RANGE 11 ; 11 Dim_to_Bright 12 ; 12 Bright_to_Dim 11 ; 11 EL_RANGE_OK 13 ; 11 EL_MAX_CW 13 ; 13 ↙
Dim_to_Bright 14 ; 13 EL_CCW_MOVE 35 ; 14 Bright_to_Dim 13 ; 11 EL_MAX_CCW 15 ; 15 Dim_to_Bright 16 ; 16 Bright_to_Dim 15 ; ↙
15 AZ_POLL_RANGE 17 ; 17 Dim_to_Bright 18 ; 18 Bright_to_Dim 17 ; 17 AZ_RANGE_OK 19 ; 17 AZ_MAX_CCW 19 ; 19 Dim_to_Bright ↙
20 ; 19 AZ_CW_MOVE 15 ; 20 Bright_to_Dim 19 ; 17 AZ_MAX_CW 21 ; 21 Dim_to_Bright 22 ; 22 Bright_to_Dim 21 ; 21 ↙
EL_POLL_RANGE 23 ; 23 Dim_to_Bright 24 ; 24 Bright_to_Dim 23 ; 23 EL_RANGE_OK 25 ; 23 EL_MAX_CCW 25 ; 25 Dim_to_Bright 26 ; ↙
25 EL_CW_MOVE 55 ; 26 Bright_to_Dim 25 ; 23 EL_MAX_CW 27 ; 27 Dim_to_Bright 28 ; 28 Bright_to_Dim 27 ; 27 AZ_POLL_RANGE 29 ↙
; 29 Dim_to_Bright 30 ; 30 Bright_to_Dim 29 ; 29 AZ_RANGE_OK 31 ; 29 AZ_MAX_CW 31 ; 29 AZ_MAX_CCW 33 ; 31 AZ_CCW_MOVE ↙
27 ; 31 Dim_to_Bright 32 ; 32 Bright_to_Dim 31 ; 33 Dim_to_Bright 34 ; 34 Bright_to_Dim 33 ; 35 EL_CCW_OK 9 ; 35 Dim_to_Bright 36 ↙
; 36 Bright_to_Dim 35 ; 36 EL_CCW_OK 10 ; 36 EL_FAIL_MOVE 38 ; 35 EL_FAIL_MOVE 37 ; 37 Dim_to_Bright 38 ; 38 Bright_to_Dim 37 ; ↙
38 EL_MOTOR_FAIL 48 ; 37 EL_MOTOR_FAIL 47 ; 55 EL_CW_OK 21 ; 55 Dim_to_Bright 56 ; 56 Bright_to_Dim 55 ; 56 EL_CW_OK 22 ; ↙
55 EL_FAIL_MOVE 57 ; 57 Dim_to_Bright 58 ; 58 Bright_to_Dim 57 ; 56 EL_FAIL_MOVE 58 ; 58 EL_MOTOR_FAIL 54 ; 57 ↙
EL_MOTOR_FAIL 53 ; 39 Full_Sweep 41 ; 40 Full_Sweep 42 ; 41 AZ_POLL_RANGE 43 ; 43 AZ_RANGE_OK 45 ; 43 AZ_MAX_CW 45 ; 45 ↙
AZ_CCW_MOVE 41 ; 43 AZ_MAX_CCW 47 ; 47 AZ_POLL_RANGE 49 ; 49 AZ_RANGE_OK 51 ; 49 AZ_MAX_CCW 51 ; 39 Dim_to_Bright ↙
40 ; 40 Bright_to_Dim 39 ; 41 Dim_to_Bright 42 ; 42 Bright_to_Dim 41 ; 43 Dim_to_Bright 44 ; 44 Bright_to_Dim 43 ; 45 Dim_to_Bright ↙
46 ; 46 Bright_to_Dim 45 ; 47 Dim_to_Bright 48 ; 48 Bright_to_Dim 47 ; 49 Dim_to_Bright 50 ; 50 Bright_to_Dim 49 ; 51 Dim_to_Bright ↙
52 ; 51 AZ_CW_MOVE 47; 52 Bright_to_Dim 51 ; 49 AZ_MAX_CW 53 ; 53 Dim_to_Bright 54 ; 54 Bright_to_Dim 53 ; 53 Sweep_Failure ↙
39 ; 33 Sweep_Failure 1 ; 6 Bright_Detected 2 ; 8 Bright_Detected 2 ; 10 Bright_Detected 2 ; 12 Bright_Detected 2 ; 14 ↙
Bright_Detected 2 ; 16 Bright_Detected 2 ; 18 Bright_Detected 2 ; 20 Bright_Detected 2 ; 22 Bright_Detected 2 ; 24 Bright_Detected 2 ↙
; 26 Bright_Detected 2 ; 28 Bright_Detected 2 ; 30 Bright_Detected 2 ; 32 Bright_Detected 2 ; 34 Bright_Detected 2 ; 42 ↙
Bright_Detected 40 ; 44 Bright_Detected 40 ; 46 Bright_Detected 40 ; 48 Bright_Detected 40 ; 50 Bright_Detected 40 ; 52 ↙
Bright_Detected 40 ; 54 Bright_Detected 40 ; 3 Full_Sweep 3 ; 4 Full_Sweep 4 ; 5 Full_Sweep 5 ; 6 Full_Sweep 6 ; 7 Full_Sweep 7 ; 8 ↙
Full_Sweep 8 ; 9 Full_Sweep 9 ; 10 Full_Sweep 10 ; 11 Full_Sweep 11 ; 12 Full_Sweep 12 ; 13 Full_Sweep 13 ; 14 Full_Sweep 14 ; 15 ↙
Full_Sweep 15 ; 16 Full_Sweep 16 ; 17 Full_Sweep 17 ; 18 Full_Sweep 18 ; 19 Full_Sweep 19 ; 20 Full_Sweep 20 ; 21 Full_Sweep 21 ; 22 ↙
Full_Sweep 22 ; 23 Full_Sweep 23 ; 24 Full_Sweep 24 ; 25 Full_Sweep 25 ; 26 Full_Sweep 26 ; 27 Full_Sweep 27 ; 28 Full_Sweep 28 ; 29 ↙
Full_Sweep 29 ; 30 Full_Sweep 30 ; 31 Full_Sweep 31 ; 32 Full_Sweep 32 ; 33 Full_Sweep 33 ; 34 Full_Sweep 34 ; 35 Full_Sweep 35 ; 36 ↙
Full_Sweep 36 ; 37 Full_Sweep 37 ; 38 Full_Sweep 38 ; 41 Full_Sweep 41 ; 42 Full_Sweep 42 ; 43 Full_Sweep 43 ; 44 Full_Sweep 44 ; 45 ↙
Full_Sweep 45 ; 46 Full_Sweep 46 ; 47 Full_Sweep 47 ; 48 Full_Sweep 48 ; 49 Full_Sweep 49 ; 50 Full_Sweep 50 ; 51 Full_Sweep 51 ; 52 ↙
Full_Sweep 52 ; 53 Full_Sweep 53 ; 54 Full_Sweep 54 ; 55 Full_Sweep 55 ; 56 Full_Sweep 56 ; 57 Full_Sweep 57 ; 58 Full_Sweep 58; 6 ↙
AZ_RANGE_OK 8; 6 AZ_MAX_CW 8; 6 AZ_MAX_CCW 10; 12 EL_RANGE_OK 14; 12 EL_MAX_CW 14; 12 EL_MAX_CCW 16; 18 ↙
AZ_RANGE_OK 20; 18 AZ_MAX_CW 20; 18 AZ_MAX_CCW 22; 24 EL_RANGE_OK 26; 24 EL_MAX_CW 26; 24 EL_MAX_CCW 28; 30 ↙
AZ_RANGE_OK 32; 30 AZ_MAX_CW 32; 30 AZ_MAX_CCW 34; 44 AZ_RANGE_OK 46; 44 AZ_MAX_CW 46; 44 AZ_MAX_CCW 48; 50 ↙
AZ_RANGE_OK 52; 50 AZ_MAX_CW 52; 50 AZ_MAX_CCW 54];
Sweep_Spec_Automata = automaton(Spec_Total_States, Sweep_Spec_STT, Sweep_Spec_MarkedStates);

E_Not_In_Sweep_Spec = [AZ_CCW_OK, AZ_CW_OK, Dark_to_Dim, Dim_to_Dark, Safe_to_Full, Full_to_Safe, Crit_to_Safe, ↙
Safe_to_Crit,wait_2sec];

global  Sweep_Spec_Automata_SelfLooped Sweep_Spec_Automata_SelfLooped_tmp;
Sweep_Spec_Automata_SelfLooped = selfloop(Sweep_Spec_Automata, E_Not_In_Sweep_Spec);

global  Euc;
Euc = [Full_Sweep, Dark_to_Dim, Dim_to_Bright, Dim_to_Dark, Bright_to_Dim, AZ_CCW_OK, AZ_CW_OK, EL_CW_OK, ↙
EL_CCW_OK, EL_FAIL_MOVE, Safe_to_Full, Full_to_Safe, Crit_to_Safe, Safe_to_Crit, AZ_RANGE_OK, AZ_MAX_CCW, AZ_MAX_CW, ↙
EL_RANGE_OK, EL_MAX_CCW, EL_MAX_CW,wait_2sec];

global  Ec;
Ec=[AZ_CW_MOVE, AZ_CCW_MOVE,EL_CW_MOVE,  EL_CCW_MOVE,EL_POLL_RANGE,AZ_POLL_RANGE, Bright_Detected, ↙
Sweep_Failure,EL_MOTOR_FAIL];
%
%-----------------------------------------------
Plant  = sync(PV_Cell, Bat_SOC, AZ_Motor_Motion, EL_Motor_Motion, AZ_Motor_Range, EL_Motor_Range, MC, ↙
Mot_Motion_f_Bat, Bat_SOC_f_PV, Bat_SOC_f_Motor_Motions,Wait);
Xmc_verify(Plant,Ec);%To check marked states with just controllable events to determine NB.

[SPEC,Sstates] = product(Sweep_Spec_Automata_SelfLooped, Spec_EL_M_Range_f_M_Motion_SelfLooped, ↙
Spec_AZ_M_Range_f_M_Motion_SelfLooped, Spec_EL_M_Motion_f_M_Range_SelfLooped, ↙
Spec_AZ_M_Motion_f_M_Range_SelfLooped);

%The legal language
legal=product(Plant,SPEC);
Xmc_verify(legal,Ec);%To check marked states with just controllable events to determine NB.
%-------------------------
Supervisor = supcon(SPEC, Plant, Euc);%Offline supervisor
%To recheck consistency of defined events.
Modules_events=[PV_Cell.TL(:,2)', Bat_SOC.TL(:,2)', AZ_Motor_Motion.TL(:,2)', EL_Motor_Motion.TL(:,2)', AZ_Motor_Range.TL(:,2)', ↙
EL_Motor_Range.TL(:,2)', MC.TL(:,2)', Mot_Motion_f_Bat.TL(:,2)', Bat_SOC_f_PV.TL(:,2)', Bat_SOC_f_Motor_Motions.TL(:,2)',Wait.TL(:,2)'];
Events_diff=setdiff([Ec,Euc],unique(Modules_events));
if ~isempty (Events_diff)
    fprintf('There is a different between availabe events and Ec,Euc :%d      \n ',Events_diff);
    return;
end

%The product function(instead of sync) is used to make a Plant inside lookehead window,therefore, all events will be added as ↙
a
%selfloop to the models(..._s) and then by using Product of these modules within Nw
%steps,the online plant will be constructed.
global Bat_SOC_s PV_Cell_s AZ_Motor_Motion_s EL_Motor_Motion_s AZ_Motor_Range_s EL_Motor_Range_s MC_s ...
       Mot_Motion_f_Bat_s Bat_SOC_f_PV_s Bat_SOC_f_Motor_Motions_s Wait_s;
global Bat_SOC_s_tmp PV_Cell_s_tmp AZ_Motor_Motion_s_tmp EL_Motor_Motion_s_tmp AZ_Motor_Range_s_tmp ↙
EL_Motor_Range_s_tmp MC_s_tmp ...
       Mot_Motion_f_Bat_s_tmp Bat_SOC_f_PV_s_tmp Bat_SOC_f_Motor_Motions_s_tmp Wait_s_tmp;

Bat_SOC_s=selfloop(Bat_SOC,setdiff(Modules_events,unique(Bat_SOC.TL(:,2)))); %
PV_Cell_s=selfloop(PV_Cell,setdiff(Modules_events,unique(PV_Cell.TL(:,2)))); %
AZ_Motor_Motion_s=selfloop(AZ_Motor_Motion,setdiff(Modules_events,unique(AZ_Motor_Motion.TL(:,2)))); %
EL_Motor_Motion_s=selfloop(EL_Motor_Motion,setdiff(Modules_events,unique(EL_Motor_Motion.TL(:,2)))); %
AZ_Motor_Range_s=selfloop(AZ_Motor_Range,setdiff(Modules_events,unique(AZ_Motor_Range.TL(:,2)))); %
EL_Motor_Range_s=selfloop(EL_Motor_Range,setdiff(Modules_events,unique(EL_Motor_Range.TL(:,2)))); %
MC_s=selfloop(MC,setdiff(Modules_events,unique(MC.TL(:,2)))); %
Mot_Motion_f_Bat_s=selfloop(Mot_Motion_f_Bat,setdiff(Modules_events,unique(Mot_Motion_f_Bat.TL(:,2)))); %
Bat_SOC_f_PV_s=selfloop(Bat_SOC_f_PV,setdiff(Modules_events,unique(Bat_SOC_f_PV.TL(:,2)))); %
Bat_SOC_f_Motor_Motions_s=selfloop(Bat_SOC_f_Motor_Motions,setdiff(Modules_events,unique(Bat_SOC_f_Motor_Motions.TL ↙
(:,2))));%
Wait_s=selfloop(Wait,setdiff(Modules_events,unique(Wait.TL(:,2)))); %

%To transform Automaton to Struct(The struct data type is used in mex files).
%______________________________________________________________
    PV_Cell_s=Change(PV_Cell_s,1);
    Bat_SOC_s=Change(Bat_SOC_s,1);
    AZ_Motor_Motion_s=Change(AZ_Motor_Motion_s,1);
    EL_Motor_Motion_s=Change(EL_Motor_Motion_s,1);
    AZ_Motor_Range_s=Change(AZ_Motor_Range_s,1);
    EL_Motor_Range_s=Change(EL_Motor_Range_s,1);
    MC_s=Change(MC_s,1);
    Mot_Motion_f_Bat_s=Change(Mot_Motion_f_Bat_s,1);
    Bat_SOC_f_PV_s=Change(Bat_SOC_f_PV_s,1);
    Bat_SOC_f_Motor_Motions_s=Change(Bat_SOC_f_Motor_Motions_s,1);
    Wait_s=Change(Wait_s,1);

    %----
    %Modules which make SPEC
```

```
  Sweep_Spec_Automata_SelfLooped=Change(Sweep_Spec_Automata_SelfLooped,1); %
  Spec_EL_M_Range_f_M_Motion_SelfLooped=Change(Spec_EL_M_Range_f_M_Motion_SelfLooped,1); %
  Spec_AZ_M_Range_f_M_Motion_SelfLooped=Change(Spec_AZ_M_Range_f_M_Motion_SelfLooped,1); %
  Spec_EL_M_Motion_f_M_Range_SelfLooped=Change(Spec_EL_M_Motion_f_M_Range_SelfLooped,1); %
  Spec_AZ_M_Motion_f_M_Range_SelfLooped=Change(Spec_AZ_M_Motion_f_M_Range_SelfLooped,1); %
  %______________________________________________________________________________
global  String;
String=[];%It keeps  the  trajectory of selected events  which are  taken  during  online  looahead.
global  Supervisor_m new_enable_event CallbackRunning;
Supervisor_m=Supervisor;%The offline  supervisor which  is tracked by taken  events.
new_enable_event=false;
%------------------------------------------------------------------------------------------

Lookahead_tot_time=[];%Keep record  of total  time  of generating Plant and  Supervisor  in online  mode.
global  serialOne;%Used in serial_com
del_time=[];
del_t=tic;
global  totall_e_time;
totall_e_time=[0 0];
global  first;
first=true;%For the  first run  of LLP.
Lookahead_Comp_cntr=0;
buff_cntr=0;%Buffer counter to  keep  the  unsyncronized received  event  in the  buffer  for further
        %steps  use.
global  rec_e_t buff_cntr;
Plant_size=[];
SPEC_size=[];
Online_Supervisor_size=[];
global  Enable_Events_offline Enable_Events_first;
global  Online_Supervisor Online_Supervisor_at_small_delta;
global  Lookahead_Copm_time_Intervals_tot Lookahead_Copm_time_Intervals LLP_at_delta LLP_exe miss_event del_t del_time ↙
String_tmp callback_times_matrix;
Lookahead_Copm_time_Intervals_tot=[];

LLP_at_delta=false;
LLP_exe=false;%When selta=samll delta,the LLP computation has  to  be  executed.
miss_event=0;%The nimber  of events  which is past  since  LLP_exe=True and  real  start  of LLP computation.
  while(Online)
    %LLP_at_delta would  be  "false" at  delta=0 to  prevent LLP
    %calculation be  repeated at  delta_N-small_delta
  if first==true || (LLP_exe==true && LLP_at_delta==false)
    %The copy  of last available  modeles should  be  used  for LLP.
    %the  manipulation of these modules should  be  prevented by receing  event  interrupt.
    if first==true
        Lookahead_Copm_time_Intervals=tic;%To record  the  time  intervals  of supcon occurence
        Nw_tmp=Nw;
        copy_modules();%Make a copy  of all models as  ..._tmp
    elseif LLP_exe==true
        %If due  to  some  consequitive events,the point  of
        %"delta_N-small_delta" for LLP calculation is missed,the length of
        %LLP window  should  be  less as "delta_N+delta"
        Nw_tmp=5+delta_N+delta;
        LLP_at_delta=true;
    end
    %Making  a Plant  with depth of Nw by producting of every  two  modeles
    check_t_LookaheadComTime=tic;
    Lookahead_Copm_time_Intervals_t=toc(Lookahead_Copm_time_Intervals);
    Lookahead_Copm_time_Intervals_tot=[Lookahead_Copm_time_Intervals_tot;Lookahead_Copm_time_Intervals_t 1];
    p1=product_c_v2_n_mex(Nw_tmp,Bat_SOC_f_PV_s_tmp,Bat_SOC_s_tmp); %
    p2=product_c_v2_n_mex(Nw_tmp,p1,PV_Cell_s_tmp); %
    p3=product_c_v2_n_mex(Nw_tmp,AZ_Motor_Motion_s_tmp,EL_Motor_Motion_s_tmp);%
    p4=product_c_v2_n_mex(Nw_tmp,p3,Wait_s_tmp);%
    p5=product_c_v2_n_mex(Nw_tmp,p4,Bat_SOC_f_Motor_Motions_s_tmp);%
    p6=product_c_v2_n_mex(Nw_tmp,Mot_Motion_f_Bat_s_tmp,p5);%
    p7=product_c_v2_n_mex(Nw_tmp,AZ_Motor_Range_s_tmp,EL_Motor_Range_s_tmp);%
    p8=product_c_v2_n_mex(Nw_tmp,p2,p6);%
    p9=product_c_v2_n_mex(Nw_tmp,p8,MC_s_tmp);%
    p10=product_c_v2_n_mex(Nw_tmp,p9,p7);

    %Modeles  which  make  SPEC
    s1=product_c_v2_n_mex(Nw_tmp-1,Spec_EL_M_Motion_f_M_Range_SelfLooped_tmp, ↙
Spec_EL_M_Range_f_M_Motion_SelfLooped_tmp);%
    s2=product_c_v2_n_mex(Nw_tmp-1,Spec_AZ_M_Range_f_M_Motion_SelfLooped_tmp, ↙
Spec_AZ_M_Motion_f_M_Range_SelfLooped_tmp);%
    s3=product_c_v2_n_mex(Nw_tmp-1,s1,s2);%
    s4=product_c_v2_n_mex(Nw_tmp-1,s3,Sweep_Spec_Automata_SelfLooped_tmp);%

   if first==true
       Online_Supervisor=supcon_c_mex(s4,p10,Euc);
       rec_e_t=tic;%To start  timer  of recording events  once  the  computation time  starts.
       delta=delta_N;
       Enable_Events_offline=unique(Supervisor_m.TL(Supervisor_m.TL(:,1)==1,2));%
       Enable_Events_first=unique(Online_Supervisor.TL(Online_Supervisor.TL(:,1)==1,2));
       fprintf('Size of lookahead buffer:%d       \n ',delta);
       fprintf('First Enable Events of online  Supervisor:%d       \n ',Enable_Events_first);
       Event_diff1=setdiff(Enable_Events_offline,Enable_Events_first);
       Event_diff2=setdiff(Enable_Events_first,Enable_Events_offline);
       if ~isempty(union(Event_diff1,Event_diff2))
           fprintf('Not Valid Nw because of %d       \n ',Event_diff1)%The validity of LLP supervisor is checked in every single  step.
           break;
       end
    elseif LLP_exe==true
       Online_Supervisor_at_small_delta=supcon_c_mex(s4,p10,Euc);
       miss_event=small_delta-delta
       LLP_exe=false;
       CallbackRunning=false;
    end
    p_t_s_t=toc(check_t_LookaheadComTime);
    uint64  check_t_LookaheadComTime=0;
    Lookahead_tot_time=[Lookahead_tot_time; p_t_s_t delta  Nw_tmp];
    %--------------------------------------------
    Lookahead_Copm_time_Intervals_t=toc(Lookahead_Copm_time_Intervals);
    Lookahead_Copm_time_Intervals_tot=[Lookahead_Copm_time_Intervals_tot;Lookahead_Copm_time_Intervals_t 2];
    %-------------------------------------------------
    %To record  the  size of main  automatons in every  LLP computation.
    t=size(p10.TL);
    Plant_size=[Plant_size;p10.N t(1,1)];
    t=size(s4.TL);
    SPEC_size=[SPEC_size;s4.N t(1,1)];
    t=size(Online_Supervisor.TL);
    Online_Supervisor_size=[Online_Supervisor_size;Online_Supervisor.N t(1,1)];
    %------------------------------------------------
    if Serial==1 && first==true
       serial_com_v2();%To Config  the  serial  port  and  received  events  from  "uc"  after  sending  "START_EVENT".
       first=false;
    end
    Lookahead_Comp_cntr=Lookahead_Comp_cntr+1;
  end%if  delta==0 ||...
  if isempty(Online_Supervisor.TL)
     disp('e(SE or  RTE)');%It means SE or  RTE.
```

```
        return;
    end
  end  %while
  if Serial==1
      RESET=input('Reset the  position of motors?\n');
      if RESET==true
          fprintf(serialOne,'!33@      &');%
      end
  end
  %To save  the  results  of execution in related files.
  dlmwrite('String for small  delta'+string(small_delta)+'DelN'+string(delta_N)+'.txt',String);
  dlmwrite('Time Exucation for Plant  & Supcon   for small  delta'+string(small_delta)+'DelN'+string(delta_N)+'.txt', ↙
Lookahead_tot_time);
  dlmwrite('Time of Event Reception to  MATLAB for small  delta'+string(small_delta)+'DelN'+string(delta_N)+'.txt',totall_e_time);
  dlmwrite('Time of serial  data  delivery  for small  delta'+string(small_delta)+'DelN'+string(delta_N)+'.txt',del_time);
  dlmwrite('Time of Supcon  Execution Intervals for small  delta'+string(small_delta)+'DelN'+string(delta_N)+'.txt', ↙
Lookahead_Copm_time_Intervals_tot);
  dlmwrite('Lookahead Plant  Size of small  delta'+string(small_delta)+'DelN'+string(delta_N)+'.txt',Plant_size);
  dlmwrite('Lookahead SPEC Size of small  delta'+string(small_delta)+'DelN'+string(delta_N)+'.txt',SPEC_size);
  dlmwrite('Lookahead Supervisor  Size of small  delta'+string(small_delta)+'DelN'+string(delta_N)+'.txt',Online_Supervisor_size);
  dlmwrite('Callback Function  execution time'+string(small_delta)+'DelN'+string(delta_N)+'.txt',callback_times_matrix);
```

Serial communication between MATLAB and microcontroller:

```
function  []=serial_com_v2()
%To send  an receive  data  from\to microcontroller through serial  port.
%clc;
%clear;
global  frst_full_sweep Lookahead_Copm_time_Intervals_tot;
frst_full_sweep=true;
global  serialOne  E_rec_str Online;
E_rec_str=[];%keep the  current received  events
global  E_rec_n;
E_rec_n=[];%The number of events  which are  received  in one  packet.
global  Event;
Event=[];
global  E_rec_number;
E_rec_number=0;%Event number
global  E_rec_cmplt;
E_rec_cmplt=false;
global  Euc Jump_Lokkahead_com Euc_received_p rec_e_t delta  delta_N small_delta new_enable_event;
global  Supervisor_m Online_Supervisor Online_Supervisor_at_small_delta;
global  Ec_t_b_delivered;%True if there  is an  Ec to  be  delivered.
Ec_t_b_delivered=false;
global  String_tmp del_time callback_times_matrix;
callback_times_matrix=[];
String_tmp=[];
del_time=[];%
global  CallbackRunning String_test LLP_at_delta first LLP_exe Enable_Events Enable_Events_first Enable_Events_offline Ec_deliver del_t;
Ec_deliver=0;%The Ec which  should  be  delivered.
String_test=[];
first_t=true;
reapeated_n=0;
%To initialize  the  serial  port.
if ~isempty(instrfind)
     fclose(instrfind);
      delete(instrfind);
end
serialOne=serial('COM3','BaudRate', 115200,'Timeout',10,'InputBufferSize',2048,'Terminator','&');
serialOne.ReadAsyncMode='continuous';
serialOne.BytesAvailableFcnMode =  'terminator';
%If BytesAvailableFcnMode  is terminator,
%the  callback  function  executes every  time  the  character specified  by the  Terminator property is read.
serialOne.BytesAvailableFcn = @instrcallback;
%The MATLAB® file callback  function  specified  for the  OutputEmptyFcn property is executed when  the  output buffer  is empty.
fopen(serialOne);
fprintf(serialOne,'!00@      &');%Sending start  signal(START_EVENT)

if Online==1
    fprintf(serialOne,'!32@      &');%ending online  mode signal(ONLINE)
end
if frst_full_sweep==true
    fprintf(serialOne,'!15@      &');%Full sweep  command
    frst_full_sweep=false;
end
global  totall_e_time Lookahead_Copm_time_Intervals;

%Equivalent events_array numbers in the  vector.
Dark_to_Dim_v = 0;
Dim_to_Bright_v = 1;
Dim_to_Dark_v = 2;
Bright_to_Dim_v = 3;

```

```
AZ_CCW_OK_v = 4;
AZ_CW_OK_v = 5;
AZ_CCW_MOVE_v = 6;
AZ_CW_MOVE_v= 7;

EL_CCW_OK_v = 8;
EL_CW_OK_v = 9;
EL_CCW_MOVE_v = 10;
EL_CW_MOVE_v = 11;
EL_FAIL_MOVE_v = 12;

AZ_POLL_RANGE_v = 13;
AZ_MAX_CW_v = 14;
AZ_MAX_CCW_v = 15;
AZ_RANGE_OK_v = 16;

EL_POLL_RANGE_v = 17;
EL_MAX_CW_v = 18;
EL_MAX_CCW_v = 19;
EL_RANGE_OK_v = 20;

Safe_to_Full_v = 21;
Full_to_Safe_v = 22;
Crit_to_Safe_v = 23;
Safe_to_Crit_v = 24;

Bat_Discharging_v = 25;
Bat_Charging_v = 26;

AZ_Sweep_CW_v = 27;
AZ_Sweep_CCW_v = 28;
EL_Sweep_CW_v = 29;
EL_Sweep_CCW_v = 30;

Full_Sweep_v = 31;
Bright_Detected_v = 32;
Sweep_Failure_v = 33;
EL_MOTOR_FAIL_v = 34;

wait_2sec_v=35;
 send_AZ_CCW_HW_v=36;
 send_AZ_CW_HW_v=37;
 send_EL_CCW_HW_v=38;
 send_EL_CW_HW_v=39;

 %--------------------------------
 Dark_to_Dim_var = 301;
 Dim_to_Bright_var = 302;
 Dim_to_Dark_var = 303;
 Bright_to_Dim_var = 304;

 AZ_CCW_OK_var = 401;
 AZ_CW_OK_var = 402;
 AZ_CCW_MOVE_var = 403;
 AZ_CW_MOVE_var = 404;

 EL_CCW_OK_var = 451;
 EL_CW_OK_var = 452;
 EL_CCW_MOVE_var = 453;
 EL_CW_MOVE_var = 454;
```

```
EL_FAIL_MOVE_var = 455;

AZ_POLL_RANGE_var = 425;
AZ_MAX_CW_var = 410;
AZ_MAX_CCW_var = 411;
AZ_RANGE_OK_var = 412;

EL_POLL_RANGE_var = 435;
EL_MAX_CW_var = 460;
EL_MAX_CCW_var = 461;
EL_RANGE_OK_var = 462;

Safe_to_Full_var = 601;
Full_to_Safe_var = 602;
Crit_to_Safe_var = 603;
Safe_to_Crit_var = 604;

AZ_Sweep_CW_var = 500;
AZ_Sweep_CCW_var = 501;
EL_Sweep_CW_var = 502;
EL_Sweep_CCW_var = 503;
Full_Sweep_var = 504;
Bright_Detected_var = 510;
Sweep_Failure_var = 511;
EL_MOTOR_FAIL_var = 512;

wait_2sec_var=480;
send_AZ_CCW_HW_var=483;
send_AZ_CW_HW_var=481;
send_EL_CCW_HW_var=493;
send_EL_CW_HW_var=494;
%__________________________________________________
%instrcallback  function
function  instrcallback(serialOne,BytesAvailable)
callback_time=tic;
E_rec_cmplt=false;
strrec='';
strrec=fscanf(serialOne);
chr=0;
if ( strcmp(strrec(1:4),'!08@') && length(strrec)==10 )%strcmp(s1,s2)  compares s1 and  s2 and  returns 1 (true)  if the  two  are identical  and  0 (false) otherwise
    Event=[];
    for i=1:5
        chr=strrec(4+i);
        if chr~='A'
            Event(i)=uint8(chr)-66 ;
        end  end%for
   E_rec_number=length(Event);
   for i=1:E_rec_number
      event_Number=Event(i);
    if(event_Number ==  Dark_to_Dim_v)
        E_rec_str=[E_rec_str; Dark_to_Dim_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Dark_to_Dim_var e_t_rec];
    elseif(event_Number ==  Dim_to_Bright_v)
        E_rec_str=[E_rec_str; Dim_to_Bright_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Dim_to_Bright_var e_t_rec];
    elseif(event_Number ==  Dim_to_Dark_v)

    E_rec_str=[E_rec_str; Dim_to_Dark_var E_rec_number];
    e_t_rec=toc(rec_e_t);
    totall_e_time=[totall_e_time;Dim_to_Dark_var e_t_rec];
elseif(event_Number ==  Bright_to_Dim_v)
    E_rec_str=[E_rec_str; Bright_to_Dim_var E_rec_number];
    e_t_rec=toc(rec_e_t);
    totall_e_time=[totall_e_time;Bright_to_Dim_var e_t_rec];
elseif(event_Number ==  AZ_CCW_OK_v)
        E_rec_str=[E_rec_str; AZ_CCW_OK_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_CCW_OK_var e_t_rec];
elseif(event_Number ==  AZ_CW_OK_v)
        E_rec_str=[E_rec_str; AZ_CW_OK_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_CW_OK_var e_t_rec];
elseif(event_Number ==  AZ_CCW_MOVE_v)
        E_rec_str=[E_rec_str; AZ_CCW_MOVE_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_CCW_MOVE_var e_t_rec];
elseif(event_Number ==  AZ_CW_MOVE_v)
        E_rec_str=[E_rec_str; AZ_CW_MOVE_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_CW_MOVE_var e_t_rec];
elseif(event_Number ==  EL_CCW_OK_v)
        E_rec_str=[E_rec_str; EL_CCW_OK_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_CCW_OK_var e_t_rec];
elseif(event_Number ==  EL_CW_OK_v)
        E_rec_str=[E_rec_str; EL_CW_OK_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_CW_OK_var e_t_rec];
elseif(event_Number ==  EL_CCW_MOVE_v)
         E_rec_str=[E_rec_str; EL_CCW_MOVE_var E_rec_number];
         e_t_rec=toc(rec_e_t);
         totall_e_time=[totall_e_time;EL_CCW_MOVE_var e_t_rec];
elseif(event_Number ==  EL_CW_MOVE_v)
        E_rec_str=[E_rec_str; EL_CW_MOVE_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_CW_MOVE_var e_t_rec];
elseif(event_Number ==  EL_FAIL_MOVE_v)
        E_rec_str=[E_rec_str; EL_FAIL_MOVE_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_FAIL_MOVE_var e_t_rec];
elseif(event_Number ==  AZ_POLL_RANGE_v)
        E_rec_str=[E_rec_str; AZ_POLL_RANGE_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_POLL_RANGE_var e_t_rec];
elseif(event_Number ==  AZ_MAX_CW_v)
        E_rec_str=[E_rec_str; AZ_MAX_CW_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_MAX_CW_var e_t_rec];
elseif(event_Number ==  AZ_MAX_CCW_v)
        E_rec_str=[E_rec_str; AZ_MAX_CCW_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_MAX_CCW_var e_t_rec];
elseif (event_Number ==  AZ_RANGE_OK_v)
        E_rec_str=[E_rec_str; AZ_RANGE_OK_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_RANGE_OK_var e_t_rec];
elseif(event_Number ==  EL_POLL_RANGE_v)
```

```
        E_rec_str=[E_rec_str; EL_POLL_RANGE_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_POLL_RANGE_var e_t_rec];
elseif(event_Number == EL_MAX_CW_v)
        E_rec_str=[E_rec_str; EL_MAX_CW_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_MAX_CW_var e_t_rec];
elseif(event_Number == EL_MAX_CCW_v)
        E_rec_str=[E_rec_str; EL_MAX_CCW_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_MAX_CCW_var e_t_rec];
elseif(event_Number == EL_RANGE_OK_v)
        E_rec_str=[E_rec_str; EL_RANGE_OK_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_RANGE_OK_var e_t_rec];
elseif(event_Number == Safe_to_Full_v)
        E_rec_str=[E_rec_str; Safe_to_Full_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Safe_to_Full_var e_t_rec];
elseif(event_Number == Full_to_Safe_v)
        E_rec_str=[E_rec_str; Full_to_Safe_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Full_to_Safe_var e_t_rec];
elseif(event_Number == Crit_to_Safe_v)
        E_rec_str=[E_rec_str ;Crit_to_Safe_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Crit_to_Safe_var e_t_rec];
elseif(event_Number == Safe_to_Crit_v)
        E_rec_str=[E_rec_str; Safe_to_Crit_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Safe_to_Crit_var e_t_rec];
elseif(event_Number == AZ_Sweep_CCW_v)
        E_rec_str=[E_rec_str; AZ_Sweep_CCW_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;AZ_Sweep_CCW_var e_t_rec];
elseif(event_Number == EL_Sweep_CW_v)
        E_rec_str=[E_rec_str; EL_Sweep_CW_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_Sweep_CW_var e_t_rec];
elseif(event_Number == wait_2sec_v)%EL_Sweep_CCW_v
        E_rec_str=[E_rec_str; wait_2sec_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;wait_2sec_var e_t_rec];
elseif(event_Number == Full_Sweep_v)
        E_rec_str=[E_rec_str; Full_Sweep_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Full_Sweep_var e_t_rec];
elseif(event_Number == Bright_Detected_v)
        E_rec_str=[E_rec_str; Bright_Detected_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Bright_Detected_var e_t_rec];
elseif(event_Number == Sweep_Failure_v)
        E_rec_str=[E_rec_str; Sweep_Failure_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;Sweep_Failure_var e_t_rec];
elseif(event_Number == EL_MOTOR_FAIL_v)
        E_rec_str=[E_rec_str; EL_MOTOR_FAIL_var E_rec_number];
        e_t_rec=toc(rec_e_t);
        totall_e_time=[totall_e_time;EL_MOTOR_FAIL_var e_t_rec];
end%if event_Number..
```

```
 end%for  i:...
fprintf('Event Received   ')
E_rec_str
%-------------------------------------------------------------------------------
   %In this  part,  by receiving  any  event,the online  supervisor responds by
   %generating new  Enable_events.
   [E_rec_str_del,E_rec_n_del]=get_rec_event();
   %Receiving  Ec after  sendig to  uc.
   delivery_time=toc(del_t);%It is assumed that  the  received  event  is what  is sent  right  before.
   if Ec_deliver~=0 && ~isempty(intersect(E_rec_str_del,Ec_deliver))
             Ec_deliver=0
             Ec_t_b_delivered=false;
             del_time=[del_time delivery_time];
             % To send  again  any  lost  packet.
          elseif Ec_t_b_delivered==false && Ec_deliver~=0 && delivery_time>2 && isempty(intersect(E_rec_str_del,Ec_deliver))
             Ec_t_b_delivered=true;
             reapeated_n=reapeated_n+1
             del_time=[del_time delivery_time];
             %
   end
   %------------------------------------------------------------------------
   for  l=1:length(E_rec_str_del(:,1))
   if first_t==true
        Es=det_Es_v1(E_rec_str_del,Enable_Events_first);
        first_t=false;
   else
        Es=det_Es_v1(E_rec_str_del,Enable_Events);
   end
   if ~isempty(Es)
       Change_state(Es);%From the  delta_N -  small_delta point  aferwards the  modeles which  are  used  in LLP should  not  be updated.
       delta=delta-1;
       if delta==small_delta
          LLP_exe=true;
          copy_modules();
          CallbackRunning=true;
       end
       if delta<small_delta
          String_tmp=[String_tmp Es];
       end
       if first==false && delta==0 && LLP_exe==false
       %The last  condition is added in order  to  avoid  picking  up  the  online
       %at small  dalta  which  maight not  be  ready(  delta  in time  stamping=bigdelta)
          delta=delta_N; LLP_at_delta=false;
          Lookahead_Copm_time_Intervals_t=toc(Lookahead_Copm_time_Intervals);
          Lookahead_Copm_time_Intervals_tot=[Lookahead_Copm_time_Intervals_tot;Lookahead_Copm_time_Intervals_t 3];
          for  k=1:small_delta-1
             String_tmp(k);

             Online_Supervisor_at_small_delta=Change(Online_Supervisor_at_small_delta,Next_si(Online_Supervisor_at_small_delta,String_tmp(k)));
          end
          Lookahead_Copm_time_Intervals_t=toc(Lookahead_Copm_time_Intervals);
          Lookahead_Copm_time_Intervals_tot=[Lookahead_Copm_time_Intervals_tot;Lookahead_Copm_time_Intervals_t 4];
          String_tmp=[];
          Online_Supervisor=Online_Supervisor_at_small_delta; end
       Supervisor_m=Change(Supervisor_m,Next_si(Supervisor_m,Es));
       Enable_Events_offline=unique(Supervisor_m.TL(Supervisor_m.TL(:,1)==1,2)); %
       if first==false %Not  First time

```

```
357          if delta~=0
358             Online_Supervisor=Change(Online_Supervisor,Next_si(Online_Supervisor,Es)); %For delta=0 update has been done in above for loop
359             String_test=[String_test Es];
360          end
361          Enable_Events=unique(Online_Supervisor.TL(Online_Supervisor.TL(:,1)==1,2));
362          Event_diff1=setdiff(Enable_Events_offline,Enable_Events);
363          Event_diff2=setdiff(Enable_Events,Enable_Events_offline);
364          if ~isempty(union(Event_diff1,Event_diff2))
365             fprintf('Not Valid Nw because of %d      \n ',Event_diff1);%To check the validity of the online supervisor
366             return;
367          end
368       end
369       fprintf('Size of lookahead buffer:%d      \n ',delta);
370       fprintf('Enable Events of online Supervisor:%d      \n ',Enable_Events);
371       new_enable_event=true;
372       Ec_t_b_sent=intersect(Enable_Events',[510,511,512,425,435,404,403,454,453]);
373        if  ~isempty(Ec_t_b_sent)
374          Ec_t_b_delivered=true;
375          send_Ec(Ec_t_b_sent);
376        end
377
378       if length(Ec_t_b_sent)>1
379          fprintf('Choice Problem       \n ')%If there is more than one controllable event in enable events.
380          return;
381       end
382    else
383      fprintf('No Event has been selected among these enabled events:%d      \n ',Enable_Events);
384
385    end
386    end%for
387
388    E_rec_cmplt=true;
389      if (Jump_Lokkahead_com==1)
390       if ~isempty(intersect(E_rec_str(:,1),Euc))
391          Euc_received_p=true;
392       else
393          Euc_received_p=false;
394       end
395      end
396 end% if event packet recieved(line 163)
397 callback_times_matrix=[callback_times_matrix;toc(callback_time) delta];
398 end%call back function
399
400 end% Base function
401

401
```

Sending controllable events to the microcontroller:

```
function  []=send_Ec(Ec_t_b_sent)
%To send  controllable events  to  the  micrcontroller.
global  Serial;
global  Ec_t_b_delivered;
global  E_rec_str;
global  serialOne;%Used in serial_com
global  Ec_sent Ec_deliver del_t;
    if (Serial==1  && (Ec_t_b_delivered==true) && isempty(E_rec_str))
                Ec_deliver=0;
                if (Ec_t_b_sent== 425) && strcmp(serialOne.TransferStatus,'idle')%AZ_POLL_RANGE_var = 425;
                   fprintf(serialOne,'!23@      &');%
                   del_t=tic;
                   Ec_sent=[Ec_sent 425 ];
                   Ec_sent(end)
                   Ec_deliver=425;
                   Ec_t_b_delivered=false;
                elseif (Ec_t_b_sent== 435) && strcmp(serialOne.TransferStatus,'idle')%EL_POLL_RANGE_var = 435;
                   fprintf(serialOne,'!24@      &');%
                   del_t=tic;
                   Ec_sent=[Ec_sent 435 ];
                   Ec_sent(end)
                   Ec_deliver=435;
                   Ec_t_b_delivered=false;
                elseif (Ec_t_b_sent== 404) && strcmp(serialOne.TransferStatus,'idle')%AZ_CW_MOVE_var = 404;
                  fprintf(serialOne,'!25@      &');%
                  del_t=tic;
                  Ec_sent=[Ec_sent 404 ];
                  Ec_sent(end)
                  Ec_deliver=404;
                  Ec_t_b_delivered=false;
                elseif (Ec_t_b_sent== 403) && strcmp(serialOne.TransferStatus,'idle')%AZ_CCW_MOVE_var = 403;
                   fprintf(serialOne,'!26@      &');%
                   del_t=tic;
                   Ec_sent=[Ec_sent 403 ];
                   Ec_sent(end)
                   Ec_deliver=403;
                   Ec_t_b_delivered=false;
                elseif (Ec_t_b_sent== 454) && strcmp(serialOne.TransferStatus,'idle')%EL_CW_MOVE_var = 454;
                    fprintf(serialOne,'!27@      &');%
                    del_t=tic;
                    Ec_sent=[Ec_sent 454 ];
                    Ec_sent(end)
                    Ec_deliver=454;
                    Ec_t_b_delivered=false;
                elseif (Ec_t_b_sent== 453) && strcmp(serialOne.TransferStatus,'idle')%EL_CCW_MOVE_var = 453;
                    fprintf(serialOne,'!28@      &');%
                    del_t=tic;
                    Ec_sent=[Ec_sent 453 ];
                    Ec_sent(end)
                    Ec_deliver=453;
                    Ec_t_b_delivered=false;
                 elseif (Ec_t_b_sent== 510) && strcmp(serialOne.TransferStatus,'idle')%Bright_Detected_var = 510;
                    fprintf(serialOne,'!29@      &');%
                    del_t=tic;
                    Ec_sent=[Ec_sent 510 ];
                    Ec_sent(end)
                    Ec_deliver=510;
                    Ec_t_b_delivered=false;
                    fprintf('Bright_Detected       \n ')
                  elseif (Ec_t_b_sent== 511) && strcmp(serialOne.TransferStatus,'idle')%Sweep_Failure_var = 511;
```

```
                    fprintf(serialOne,'!30@      &');%
                    del_t=tic;
                    Ec_sent=[Ec_sent 511 ];
                    Ec_sent(end)
                    Ec_deliver=511;
                    Ec_t_b_delivered=false;
                    fprintf('Sweep_Failure        \n ')
                  elseif (Ec_t_b_sent== 512) && strcmp(serialOne.TransferStatus,'idle')%EL_MOTOR_FAIL_var = 512;
                    fprintf(serialOne,'!31@      &');%
                    del_t=tic;
                    Ec_sent=[Ec_sent 512 ];
                    Ec_sent(end)
                    Ec_deliver=512;
                    Ec_t_b_delivered=false;
                    fprintf('EL_MOTOR_FAIL        \n ')

                end
    end%Sending Ec
end
```

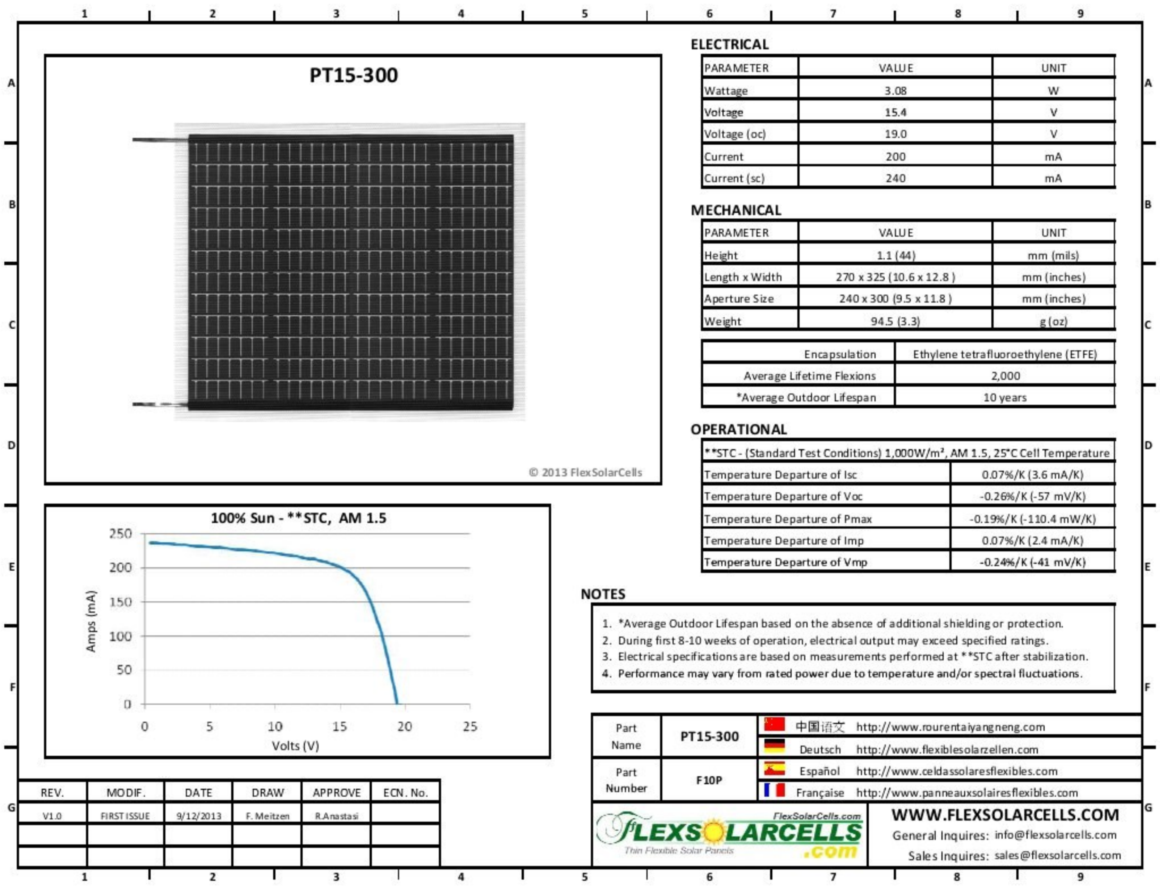
Selection of an event from enable events:

```
function  [Es]=det_Es_v1(E_rec_str_del,Enable_Events)
%To choose an  event  among enable events  according to  received  events.
global  Serial Euc Ec new_enable_event E_rec_str String  Ec_deliver;
Es=[];
if Serial==1 && ~isempty(E_rec_str_del)
            Cmm_Event=intersect(E_rec_str_del,Enable_Events','stable');
            Cmm_Event_Euc=intersect(Cmm_Event,Euc,'stable');
            Cmm_Event_Ec=intersect(Cmm_Event,Ec,'stable');
            if ~isempty(Cmm_Event)
               Cmm_Event_ln=length(Cmm_Event);
               if Cmm_Event_ln==1 && Ec_deliver==0
                    Es=Cmm_Event(1);
               elseif Cmm_Event_ln>=2 && ~isempty(Cmm_Event_Euc) && isempty(Cmm_Event_Ec) && Ec_deliver==0
                    Es=Cmm_Event_Euc(1);
               elseif Cmm_Event_ln>=2 && ~isempty(Cmm_Event_Euc) && ~isempty(Cmm_Event_Ec)
                    Es_u=Cmm_Event_Euc(1);
                    Es_c=Cmm_Event_Ec(1);
                    [~,ind3]=intersect(E_rec_str_del,Es_u);
                    [~,ind4]=intersect(E_rec_str_del,Es_c);
                    Es=Es_c;

               elseif Cmm_Event_ln>=2 && ~isempty(Cmm_Event_Ec)&& isempty(Cmm_Event_Euc)
                     Es=Cmm_Event_Ec(1);
               end
               if ~isempty(Es)
                  new_enable_event=false; end
               [E_rec_str_Es,i1,~]=intersect(E_rec_str(:,1),Es);
               %While selected event  is being  removed from  the
               %receiving  array,another event  with the  same  number
               %might  be  received,so just  one  event  has  to  be  removed.
               if ~isempty(E_rec_str_Es)
                  E_rec_str(min(i1),:)=[];
               end
               ind7=E_rec_str(:,2)==0;
               fprintf('Event Remained  ')
               E_rec_str=E_rec_str(~ind7,:)
               String=[String,Es];
             else
               end%~isempty(Cmm_Event)
             if ~isempty(E_rec_str) ind6=E_rec_str(:,2)>1;
                   E_rec_str(ind6,2)=E_rec_str(ind6,2)-1;
             end


end%There is a received  event
end
```

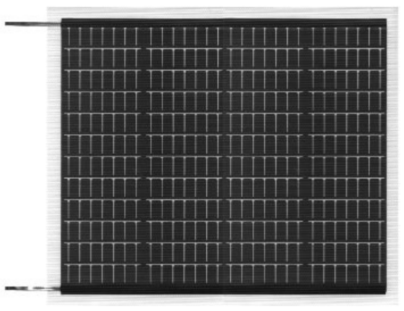
Appendix E

Data sheets

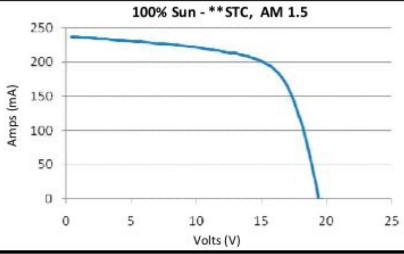
Photovoltaic Cell



PT15-300



© 2013 FlexSolarCells



ELECTRICAL

PARAMETER	VALUE	UNIT
Wattage	3.08	W
Voltage	15.4	V
Voltage (oc)	19.0	V
Current	200	mA
Current (sc)	240	mA

MECHANICAL

PARAMETER	VALUE	UNIT
Height	1.1 (44)	mm (mils)
Length x Width	270 x 325 (10.6 x 12.8)	mm (inches)
Aperture Size	240 x 300 (9.5 x 11.8)	mm (inches)
Weight	94.5 (3.3)	g (oz)

Encapsulation	Ethylene tetrafluoroethylene (ETFE)
Average Lifetime Flexions	2,000
*Average Outdoor Lifespan	10 years

OPERATIONAL

**STC - (Standard Test Conditions) 1,000W/m², AM 1.5, 25°C Cell Temperature	
Temperature Departure of Isc	0.07%/K (3.6 mA/K)
Temperature Departure of Voc	-0.26%/K (-57 mV/K)
Temperature Departure of Pmax	-0.19%/K (-110.4 mW/K)
Temperature Departure of Imp	0.07%/K (2.4 mA/K)
Temperature Departure of Vmp	-0.24%/K (-41 mV/K)

NOTES

1. *Average Outdoor Lifespan based on the absence of additional shielding or protection.
2. During first 8-10 weeks of operation, electrical output may exceed specified ratings.
3. Electrical specifications are based on measurements performed at **STC after stabilization.
4. Performance may vary from rated power due to temperature and/or spectral fluctuations.

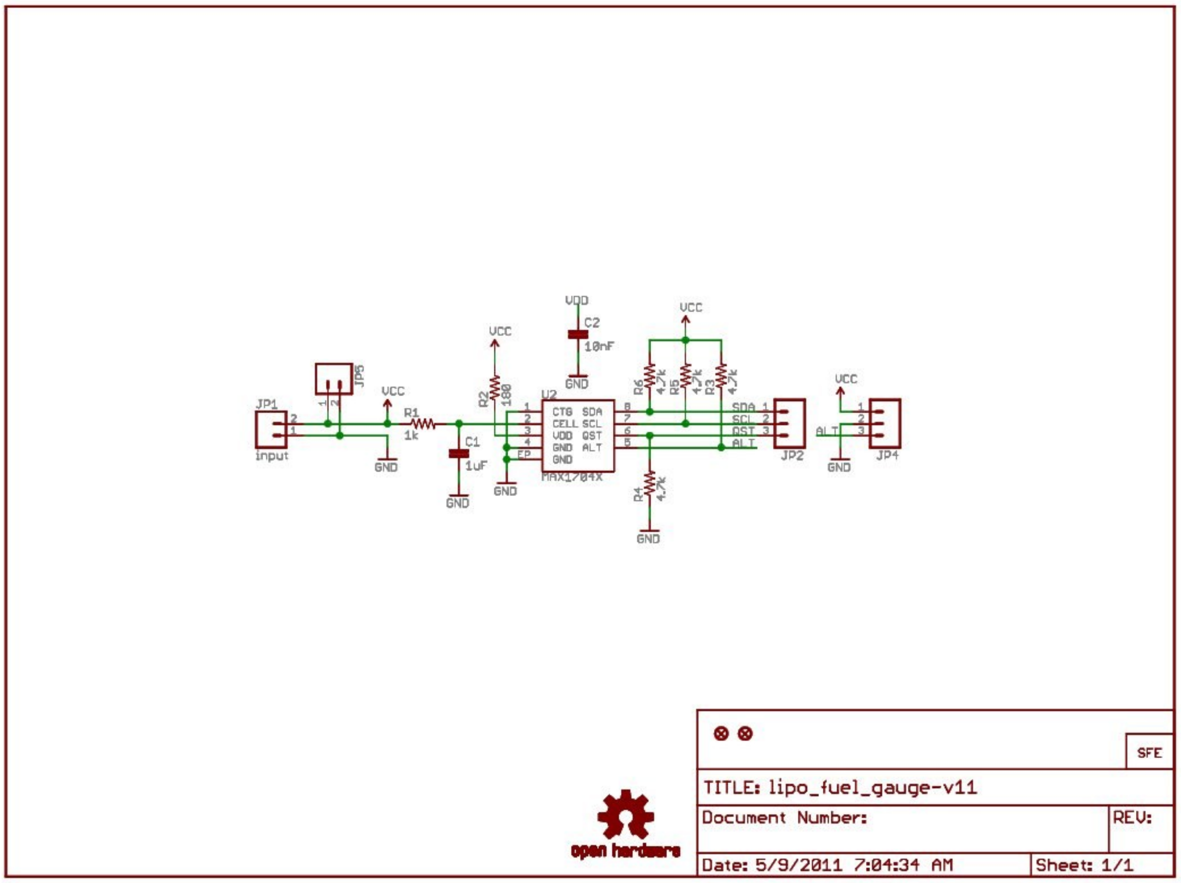
Part Name	PT15-300	中国语文	http://www.rourentaiyangneng.com
		Deutsch	http://www.flexiblesolarzellen.com
Part Number	F10P	Español	http://www.celdassolaresflexibles.com
		Française	http://www.panneauxsolairesflexibles.com

REV.	MODIF.	DATE	DRAW	APPROVE	ECN. No.
V1.0	FIRST ISSUE	9/12/2013	F. Meitzen	R.Anastasi	

FlexSolarCells.com FLEXSOLARCELLS .com Thin Flexible Solar Panels

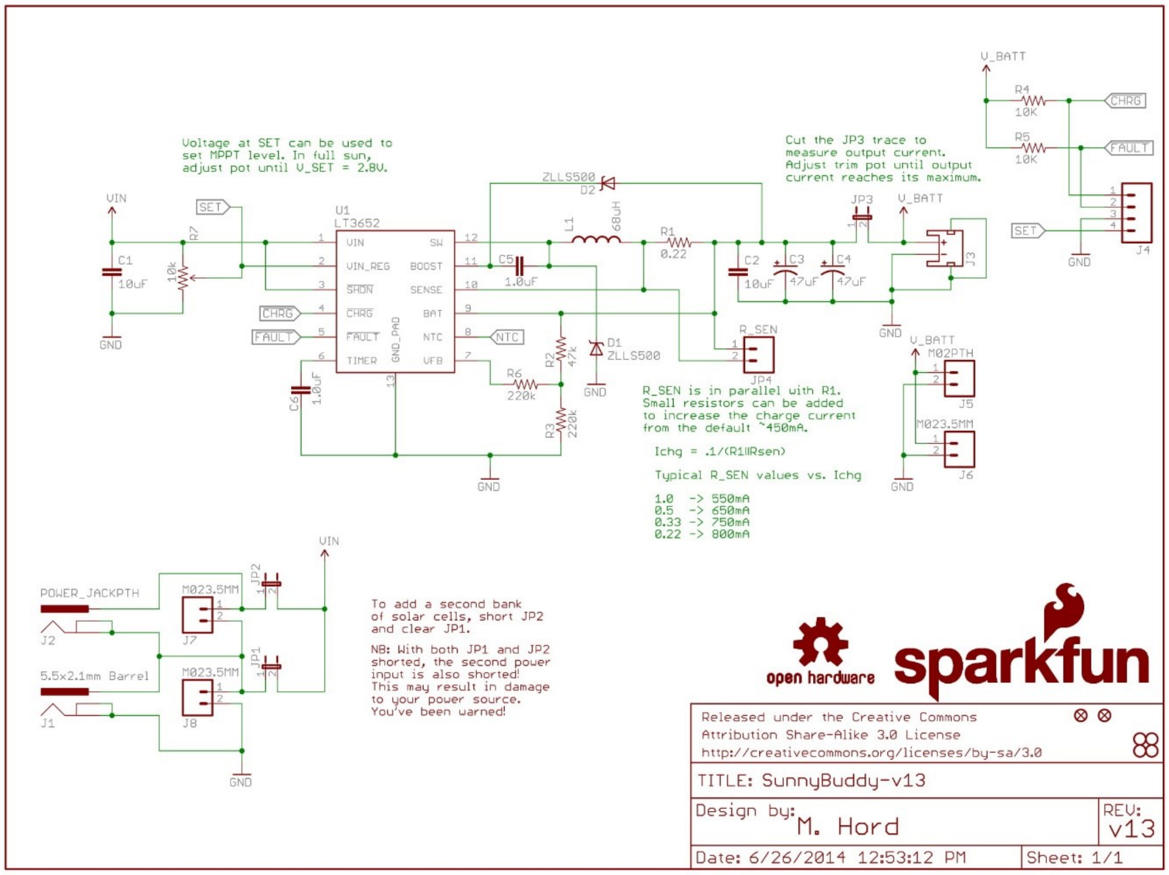
WWW.FLEXSOLARCELLS.COM
General Inquires: info@flexsolarcells.com
Sales Inquires: sales@flexsolarcells.com

Fuel gauge



JP1
input
JP5
VCC
R1
1k
GND
C1
1uF
R2
100
VDD
C2
10nF
U2
CTG SDA
CELL SCL
VDD QST
GND ALT
GND
EP
MAX1704X
R6
R5
R3
R4
4.7k
SDA
SCL
QST
ALT
JP2
JP4
SFE
TITLE: lipo_fuel_gauge-v11
Document Number:
REV:
Date: 5/9/2011 7:04:34 AM
Sheet: 1/1
open hardware

Maximum Power Point Tracker


Voltage at SET can be used to set MPPT level. In full sun, adjust pot until V_SET = 2.8V.

Cut the JP3 trace to measure output current. Adjust trim pot until output current reaches its maximum.

R_SEN is in parallel with R1. Small resistors can be added to increase the charge current from the default ~450mA.

Ichg = .1/(R1||Rsen)

Typical R_SEN values vs. Ichg

1.0 -> 550mA
0.5 -> 650mA
0.33 -> 750mA
0.22 -> 800mA

To add a second bank of solar cells, short JP2 and clear JP1.

NB: With both JP1 and JP2 shorted, the second power input is also shorted! This may result in damage to your power source. You've been warned!

Released under the Creative Commons Attribution Share-Alike 3.0 License http://creativecommons.org/licenses/by-sa/3.0

TITLE: SunnyBuddy-v13

Design by: M. Hord

REV: v13

Date: 6/26/2014 12:53:12 PM

Sheet: 1/1

USB to serial converter

XBee - USB Board
990.002

V2.0

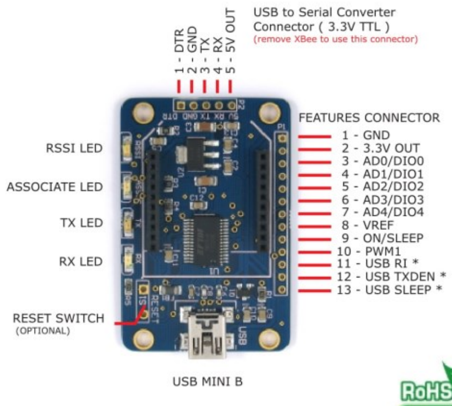
Technical Specifications

- **XBee Modules:** XBee and XBeePRO are fully supported. XBee Series 2 are also supported
- **USB Port:** a powered (500mA) USB 1.0/2.0 port is required.
- **3.3V OUT - Output Power:** depends on the Xbee module used. About 300mA available with XBee, and 140mA with XBeePRO modules.
- **Dimensions:** 52 x 35 mm - max height 7 mm without Xbee module.

Features

RSSI and ASSOCIATE LEDs to monitor Xbee activity
TX and RX LEDs to monitor serial activity between the Xbee module and the USB chip
3.3V OUT to power external devices (MCU's, boards, etc)
Soft Start circuitry to maximize reliability
Features Connector to access the most important XBee and USB connections
USB to Serial Connector: use this board as a USB to Serial Converter

Layout



* Refer to the FTDI FT232RL Datasheet (www.ftdichip.com)

RoHS Compliant

Description

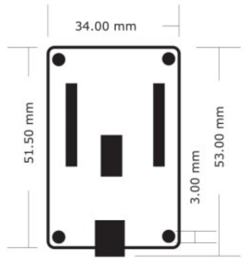
The XBee - USB Board has been designed to allow an easy and reliable connection between the Xbee's modules and PC. XBee and XBeePRO modules are fully supported.
The rock solid construction, the soft start circuitry and the high power voltage regulator; ensure the maximum reliability and full performances of the XBee module.
Four useful LEDs allow constant monitoring of the board activity and fast troubleshooting.
A Features Connector provides the most important connections with the XBee module and the USB chip adding great flexibility to this board.
The USB Mini B connector ensures a compact and easy to use board. Now with a USB to Serial connector, the 990.002 can be used as an USB to Serial converter!

XBee and XBeePRO are Trademarks of Maxstream Inc.

DROIDS SAS
VIA NANNARELLI 41
00139 ROMA ITALY
www.droids.it
info@droids.it

Notes

Dimensions



Applications

USB to Zigbee

USB to Bluetooth

USB to RS232 TTL

Links

XBee Modules and Documentation:
www.maxstream.net
USB Drivers and Documentation:
www.ftdichip.com

RF module

XBee 802.15.4 RF 2.4GHz Module, 1 mW, U.FL Antenna Connector

XB24-AUI-001

RoHS Lead Free

XBee 802.15.4 OEM RF modules are embedded solutions providing wireless end-point connectivity to devices. These modules use the IEEE 802.15.4 networking protocol for fast point-to-multipoint or peer-to-peer networking. They are designed for high-throughput applications requiring low latency and predictable communication timing.

Part of the XBee family of wireless products, these modules are easy to use, share a common footprint, and are fully interoperable with other XBee products utilizing the same technology. This allows different XBee technologies to be drop-in replacements for each other; you can switch from 802.11 to 802.15.4, ZigBee, DigiMesh and proprietary long-range with minimal development time or risk.

2.4GHz 802.15.4 RF Module Features

- Power output: 1mW (+0 dBm)
- Indoor/Urban range: Up to 100 ft (30 m)
- Outdoor/RF line-of-sight range: Up to 300 ft (90 m)
- RF data rate: 250 Kbps
- Interface data rate: Up to 115.2 Kbps
- Operating frequency: 2.4 GHz
- Receiver sensitivity: -92 dBm
- Spread Spectrum type: DSSS (Direct Sequence Spread Spectrum)
- Networking topology: Point-to-point, point-to-multipoint, & peer-to-peer
- Error handling: Retries & acknowledgments
- Filtration options: PAN ID, Channel, and 64-bit addresses
- Channel capacity: 16 channels
- Addressing: 65,000 network addresses available for each channel
- Supply voltage: 2.8V - 3.4V DC (Footprint Recommendation: 3.0V - 3.4V DC)
- Transmit current: 45 mA (@ 3.3V) boost mode; 35 mA (@ 3.3V) normal mode
- Receive current: 50 mA (@ 3.3V)
- Power-down sleep current: <10 µA at 25°C
- Frequency band: 2.4000 - 2.4835 GHz
- Interface options: 3V CMOS UART
- Size: 0.960 in × 1.087 in (2.438 cm × 2.761 cm)
- Weight: 0.10 oz (3g)
- Operating temperature: -40°C to 85°C (industrial)
- Certifications: FCC (USA), IC (Canada), ETSI (Europe), C-TICK (Australia), Telec (Japan)