Logic Diagram Verification by Modular Supervisory Control of Discrete-Event System

Yong Chang Liu

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 Engineering) at
Concordia University
Montreal, Quebec, Canada

September 2008

Your file Votre référence
ISBN: 978-0-494-45520-3
Our file Notre référence
ISBN: 978-0-494-45520-3

Abstract

Logic Diagram Verification by Modular Supervisory Control of Discrete-Event System

Yong Chang Liu

Control function verification is an important task in current engineering design. Traditional researches usually focus on the final function validation when the control system has already been implemented on a hardware controller. However, it would be more useful if design errors are found in earlier stages of design. Logic diagram, as a popular middle medium, plays a critical role in the current design practices, especially for medium-sized and large-sized control systems. Therefore, verification of the design specifications of the logic diagrams is an interesting topic in order to find and eliminate the design errors in an early stage. In this thesis, we provide a viable approach to verify the design functions of the logic diagrams which is based on the modular supervisory control of Discrete-Event Systems. We create models for basic logic gates and introduce buffers to obtain automaton representation of logic diagrams. After converting the informal verbal specifications to automata, we can verify whether the logic diagram satisfies these specifications with the help of TTCT (a computer program based on automata for analysis and design of supervisory control systems). A formal proof of controllability and a semi-formal proof of nonblocking property are given. An industrial-sized example is studied to demonstrate the feasibility of our methodology.

Acknowledgments

First of all, I would like to thank my supervisor, Dr. Peyman Gohari for his constant support and guidance to my research. This is my first time to do a formal research after over ten years engineering work. Dr. Gohari always shows me a great patience to my course study and thesis review. What I learned from him is not just a new research field but a strict research attitude, which makes my life more meaningful.

I am particularly grateful to Dr. Shahin Hashtrudi Zad for his detailed review and his kindly arrangement of my defense. I also want to express my gratitude to my thesis committee: Dr. John Zhang, Dr. Otmane Ait Mohamed and Dr. Ali Dolatabadi. They gave me lots of insightful advices.

I'd like to extend many thanks to my friend Siamak. We passed the initial difficulties together in learning Discrete-Event Systems and other courses. Our discussions still impress me and built a solid base for my future work. Furthermore, I want to acknowledge Amin. He lent his time and materials to me to explain how to write a paper by LaTeX, which made my thesis exist in front of us.

Finally, I thank my wife, my son and my parents for their loves and supports. They are always my inspiring sources in my entire life.

YONG CHANG LIU

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Control function verification is one of the most routine and important tasks in industrial control system design, especially in safety critical systems, such as nuclear power plants [CC04], [YK05], aerospace industries and toxic chemical plants, where the safety and reliability of control systems are becoming more important. Traditional verification methods usually verify the design function when the control system is nearly built up and mostly depend on manual tests based on the designer's experience. This means that the designer must wait until the control algorithms have been implemented in a controller, such as a Programmable Logic Controller (PLC), a Distributed Control System (DCS) or an industrial PC, and then the design can be tested by some Hardware-In-Loop (HIL) platform with incomplete experience rules. Therefore the test relies on a hardware platform to verify the design function.

In order to understand the context, we need to discuss the typical design philosophy. The generic engineering design currently used in practice proceeds as follows:

1. Process system engineer gives the informal verbal design specifications [1] to ensure the system is always operating in a safe and legal state.

2. Process system engineer and control engineer work together to draw the logic diagrams

[1] Specification, property and requirement are used interchangeably in this thesis.

specifying these requirements.

3. Control engineer implements the logic diagrams in an industrial controller, such as a PLC.

4. Process system engineer and control engineer test and verify the program in the controller by a specific test platform.

5. If any unexpected result happens during the test period, process system engineer and control engineer must trace back to find out the possible errors in the whole cycle.

Many previous and present works overlook the importance of steps 2 and 3. Mostly, they just pay attention to steps 1, 4 and 5. Due to their scales and complexities, many control systems are usually implemented on the basis of logic diagrams, which is a useful and popular middle medium. As a matter of fact, logic diagrams let the technical staff understand the system's operating principle and they make the troubleshooting much easier. A logic diagram generally is an essential document to submit to the customer in current engineering practice as a formal contract between the two parties. Event though we could possibly develop PLC programs directly from the informal verbal design specifications for some small-sized systems, drafting logic diagrams is still a good middle step to design the control system.

Instead of direct verification of the final PLC program, we believe that it will be quite meaningful if we can find a feasible solution to verify the logic diagrams and their associated design functions before hardware implementation. The reason is quite simple: it will let us find the potential errors in an early stage of the design, which means that the errors could be corrected in a timely fashion and the overall design cost could be reduced.

In our work, a logic diagram verification method based on modular supervisory control theory of Discrete-Event Systems (DES) is introduced to verify the logic diagram in the design phase. Discrete-event system framework is a useful modeling abstraction for various manufacturing systems, especially for those systems which can be modeled by boolean logic. A discrete-event system is driven by the occurrence of events (as opposed to time) and occupies a discrete state at each time instant. *Supervisory Control Theory* (SCT) [RW87] is developed for the synthesis of control systems for DES and has received wide acceptance

with a series of complementary works by [WR87], [FW88], [WR88], and, [Won06], to name a few. Our work is based on supervisory control theory. Chapter 2 will give a detailed review of this theory. Another important DES modeling framework is Petri-net and will not be covered in our work.

1.2 Literature Review

It is necessary to review some related topics before we present our research. The verification of control logic diagrams can be rarely found in the existing literature. Many papers exist on logic verification in VLSI design, which is not our interest. However, we can find three topics which we believe are closely related to our research.

1.2.1 DES Supervisor Implementation by PLC.

The objective here is to find an algorithm and directly or indirectly implement a formal DES controller by PLCs. [LW95] and [Led96] have addressed this issue initially. They translate a DES supervisor to an equivalent Clocked Moore Synchronous State Machine (CMSSM). At this point, the state transition can be naturally expressed by boolean logic, which is finally converted to PLC Ladder Logic Diagrams (LLDs). The most interesting aspects of these works are the modular approach and formal model reduction theorems, although the conditions for application of these theorems are very strict. [FH98] offers a good insight into the implementation problems caused by the gap between the event-based asynchronous automata and the synchronous signal-based PLC, but does not provide any solutions. [LD02] proposes a conversion method based on IDEF3 standard and treats a manufacturing system as a set of activities and their required resources. It is quite complicated to apply this method and it also seems impossible to automatically apply it to a large-scale system. [Mon06] presents a quite straightforward solution to convert a supervisor to LLDs, where inputs are regarded as uncontrollable events and outputs are regarded as controllable events. [ZV98] derives LLDs from a Petri-net controller, and provides Petri-net models for basic logic gates. [Hel01] and [HL02] propose a solution to design a DES controller by Petri-net, and then convert it to PLC Sequential Function Charts (SFC). SFC

is a structured PLC language, but SFC is ambiguous in some cases, and is not as formal as LLD, which limits its application scope.

After an analysis from both theoretical and engineering perspectives, we believe that all of the above works have the following drawbacks:

1. Even though the controller is derived from a formal supervisor, the formal supervisor is still designed from some informal verbal requirements. Thus the formal function verification cannot be avoided.

2. Even for a small-scale application, the size of an LLD converted from a DES supervisor is still much bigger than the size of an LLD derived from a conventional boolean logic controller. The actual application value is quite limited in this sense.

3. The PLC program implemented by a DES supervisor cannot be easily maintained by a general maintenance engineer, and its upgrade and troubleshooting will be difficult, too.

1.2.2 PLC Program Verification

One popular industrial approach is direct implementation of the informal specifications using a PLC programming language. Then, the test engineer simply validates the program against the informal specifications. It is a manageable but informal way to verify the correctness of a simple system, but for a large-sized or even a medium-sized system, the problem of incompleteness may happen. As [FL00] defines, the term "informal" refers to "everything that is not based on a strictly composed, syntactically and semantically well-defined form". The main problem with informal specifications is that "they do not facilitate tests for completeness, unambiguity and consistency". Hence, formal verification of PLC programs is an interesting topic which can address the incompleteness, ambiguity and inconsistency issues.

[FL00] has given an overview of the current state of the art of formal methods in PLC program verification. The present approaches are mainly of two types: *model-based* or *non-model-based.* The general choices of formalism are: *Petri-nets*, *automata*, *condition/event systems*, *high order logics*, *synchronous languages*, *general transition systems* and *algebraic*

equations. And the methods of verification and validation are: *simulation*, *model checking*, *reachability analysis* and *theorem proving*.

[RK98] adopts the model-based approach and converts the PLC programs into *Symbolic Model Verifier* (SMV) modules. SMV is a model-checker to verify the specification. [GD06] investigates how to provide an efficient representation of SMV modules. As a pioneering work, [Moo94] developed a non-model-based method for the verification of LLDs. The LLD is reinterpreted using a transition system and the properties to be checked are formalized using *Computation Tree Logic* (CTL). Again, an early version of SMV model checker is employed to determine whether the CTL assertion is satisfied in the model being verified. [DC00] took a similar approach as [Moo94] and extended the results to several PLC languages defined in IEC 61131-3 standards: LLD, SFC and *Structured Text* (ST) (modeled as transition systems synchronized by message exchanges). The properties to be checked are expressed in *Linear Temporal Logic* (LTL). SMV is still the model checker used to verify these properties. As one of the latest works, [LY05] converts the PLC *Instruction List* (IL) program into SMV code and uses the bounded model checking technique to avoid unmanageably large state spaces. [Hu03] develops a comprehensive formal transition system for SFC and IL. It employs the SMV model checker to verify the temporal logic properties of SFC program as well, and combines the abstract interpretation and data flow analysis to detect the run-time errors of IL program.

The above works on PLC program verification, we believe, have the following drawbacks:

1. They directly verify the final PLC program and do not verify the middle results, such as the logic diagrams, and therefore cannot guarantee the correctness of these middle results. In this sense, it is not a full verification and validation of all stages and is just a function validation for a complete design cycle.

2. They generally face the state explosion problem when manipulating large-sized systems.

3. Converting PLC programs to some formal forms, such as Petri-nets, condition/event systems, automata and algebraic equations, is not an easy job and these models could

be difficult to understand for most field engineers.

4. They usually do not pay much attention to the expressions and interpretations of informal design specifications, which will limit their applications in engineering practices. The reason is that we need to get a formal form of these informal design specifications to verify the PLC program.

We found that model checking is an extensively adopted tool in PLC program verification, and is also employed in our method. Therefore it is necessary to understand its principles. Model checking is a three-steps process [LY05], [FL00]:

- In the first step, the system description is converted to a finite formal model;
- The second step is to generate a formal description of properties or specifications to be verified;
- In the third step, a model checker takes the results from the previous two steps and shows whether each property or specification holds or not.

1.2.3 Creating a Formal Specification from an Informal Verbal Requirement.

We will briefly review this topic. [JB93] builds a software environment called *Aries* to manually make this conversion. [Amo95] provids a stepwise evolutionary refining method to establish formal specification from informal requirement. It starts from very general requirements and consists of five steps: *requirements categorization*, *initial specification*, *requirements restatement*, *evolutionary refinement* and *target language specification.* We can see that these steps almost all involve manual efforts as well. Instead of taking these complicated operations, our approach, which is based on the current industrial practices, follows a straightforward path. The informal verbal requirements in our work are constrained to a reduced subset of a natural language, which can cover most of logic control functions. This reduced subset can be readily converted to some formal or semi-formal boolean expressions. The detailed implementation is demonstrated in Chapter 3 and Chapter 4.

Previously, we reviewed PLC implementation of DES supervisors, PLC program verification and creating formal specification from informal verbal requirements, and pointed out their drawbacks and beneficial results from these works. We believe that verification of logic diagrams will be more fruitful. The following tasks will be considered in our research:

- Model the logic diagrams by automata and include the cyclic execution feature of PLC, which makes it possible for the approach to be extended to test PLC programs.
- Apply the well-developed supervisory control theory of discrete-event systems to convert the logic diagram to automata.
- Take the modular approach to avoid the state explosion problem; therefore, we take a suboptimal approach rather than a centralized and optimal one.
- Use the popular model checking method; the properties will be expressed by automata and TTCT (a computer program based on automata for analysis and design of supervisory control systems) is used as a model-checker.
- Take the non-model-based approach to directly verify the properties or specifications of logic diagrams (which will avoid establishing the controlled system model).
- Straightforwardly covert the informal design specifications to Boolean expressions and then to automata.

Our suggested method is illustrated in Fig. 1.1. The left side describes the steps in modeling the logic diagrams, and the right side the system specifications. The last step is to check whether the logic diagrams satisfy the specifications. This diagram precisely represents the three steps of model checking.

1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 will briefly review the background of supervisory control theory of discrete-event systems. A complete logic diagram modeling methodology is presented in Chapter 3. The formal proof of controllability and semi-formal proof of nonblocking are provided as well. An industrial-sized example, logic diagram

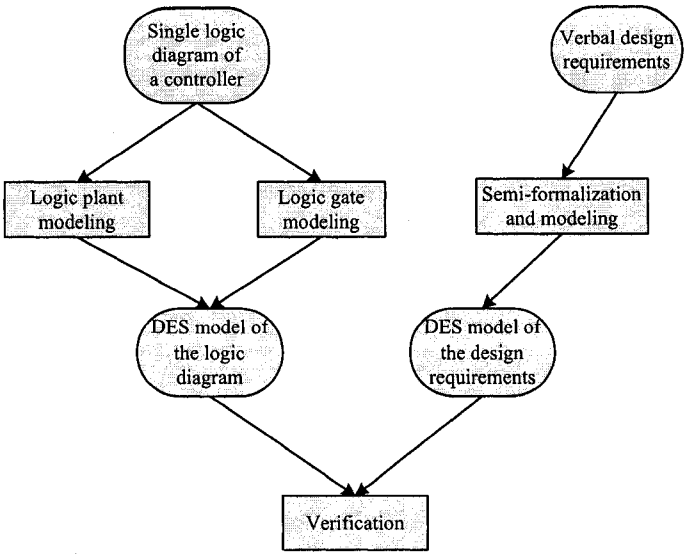

Figure 1.1: Logic diagram verification

verification of an emergency diesel generator startup controller, is studied in Chapter 4. Chapter 5 summarizes our work and points out future research directions.

Chapter 2

Background Review

In this chapter, we will briefly introduce automata theory and supervisory control theory of discrete-event systems [RW87], [WR87], [Won06]. Only the part related to our future discussions will be covered. Since TTCT is our design tool to realize the supervisory control of DES, each TTCT procedure used in our work will be introduced in the context.

2.1 Languages and Automata

2.1.1 Languages

Let Σ be a finite *alphabet*. Σ^+ denotes the set of all finite symbol sequences or *strings* over Σ. Let ϵ be the *empty string* (sequence with no symbols) . Then we define:

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

Obviously, an element of Σ^* is a *string* or *word* over Σ. A *language* over Σ is any subset of Σ^*.

For $s, t \in \Sigma^*$, we say t is a *prefix* of s, denoted by $t \leq s$, if $\exists u \in \Sigma^*$, such that $s = tu$. *Prefix closure* of L, denoted by $\overline{L}$, is defined by:

$$\overline{L} := \{t \in \Sigma^* \mid \exists s \in L, t \leq s\}$$

It can be observed that $\overline{L}$ consists of all prefixes of strings in L, and that $L \subseteq \overline{L}$.

2.1.2 Automata

We use an *automaton* (a finite state machine) to model a discrete-event system. The behavior of an automaton is represented by a pair of languages. A discrete-event system can be modeled by an automaton, which is a 5-tuple:

$$G = (Q, \Sigma, \delta, q_0, Q_m)$$

where Q is the set of states, $q_0 \in Q$ is the *initial state*, $Q_m \subseteq Q$ is the set of *marked states*, and δ is a partial state transition function $\delta : Q \times \Sigma \rightarrow Q$: $\delta(q, \sigma) = q'$ means that there is a transition labeled by event σ from state q to state q'. An automaton with partial state transitions is also termed a *generator*. Throughout this chapter, the DES plant is always denoted by G.

The transition function δ can be recursively extended from $Q \times \Sigma$ to $Q \times \Sigma^*$ by two rules:

1. $\delta(q, \epsilon) = q$
2. $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$

   where $q \in Q$, $s \in \Sigma^*$ and $\sigma \in \Sigma$.

   With the above definitions, we can obtain two kinds of behaviors of G:

- **Closed behavior**, defined by:

  $$L(G) := \{s \in \Sigma^* \mid \delta(q_0, s) \in Q\}$$

  which is the set of all strings generated by G;

- **Marked behavior**, defined by:

  $$L_m(G) := \{s \in \Sigma^* \mid \delta(q_0, s) \in Q_m\}$$

  which is the set of all strings that can take G from the initial state q_0 to a marked state in Q_m.

It can be seen that $L_m(G) \subseteq L(G)$.

Now, we define two other concepts related to G:

- **Reachable states:**

$$Q_r := \{q \in Q \mid \exists s \in \Sigma^*, \delta(q_0, s) = q\}$$

Q_r is the set of states which can be reached from the initial state q_0.

- **Coreachable states:**

$$Q_{cr} := \{q \in Q \mid \exists s \in \Sigma^*, \delta(q, s) \in Q_m\}$$

Q_{cr} is the set of all states which can reach marked states.

G is *reachable* if $Q_r = Q$ and is *coreachable* if $Q_{cr} = Q$. G is *trim* if it is both reachable and coreachable.

G is *nonblocking* if $\overline{L_m(G)} = L(G)$, meaning every reachable state is coreachable.

TTCT procedure `Create` can be used to create a DES.

2.2 Operations on Languages and Automata

We will define some important operations used in our work.

- **Complement**

The complement operation on $L_m(G)$ can be defined as:

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

We see later that it is frequently used in logic function verification. This operation can be realized by TTCT procedure `Complement`.

- **Natural projection**

Let $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$, where in general $\Sigma_1 \cap \Sigma_2 \neq \emptyset$. Let $\Sigma = \Sigma_1 \cup \Sigma_2$. Define a natural projection

$$P_i : \Sigma^* \to \Sigma_i^* \quad (i = 1, 2)$$

according to

1. $P_i(\epsilon) = \epsilon$;

2. $P_i(\sigma) = \begin{cases} \epsilon & \text{if } \sigma \notin \Sigma_i \\ \sigma & \text{if } \sigma \in \Sigma_i \end{cases}$;

3. $P_i(s\sigma) = P_i(s)P_i(\sigma)$, $s \in \Sigma^*$, $\sigma \in \Sigma$.

P_i is *catenative* since $P_i(st) = P_i(s)P_i(t)$. The action of P_i on a string s is just to erase all occurrences of σ in s such that $\sigma \notin \Sigma_i$. P_i is called the *natural projection* of Σ^* onto Σ_i^*. Let

$$P_i^{-1} : Pwr(\Sigma_i^*) \rightarrow Pwr(\Sigma^*)$$

be the inverse image function of P_i, where Pwr denotes *power set.* Then for $H \subseteq \Sigma_i^*$,

$$P_i^{-1}(H) := \{s \in \Sigma^* \mid P_i(s) \in H\}$$

TTCT procedure `Project` is employed to realize this function.

- **Synchronous product**

For $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, the synchronous product $L_1 \parallel L_2 \subseteq \Sigma^*$ is defined according to

$$L_1 \parallel L_2 := P_1^{-1}L_1 \cap P_2^{-1}L_2$$

Thus, $s \in L_1 \parallel L_2$ if and only if $P_1(s) \in L_1$ and $P_2(s) \in L_2$.

If there are two generators $G_1 = (Q_1, \Sigma_1, \delta_1, q_{10}, Q_{m1})$ and $G_2 = (Q_2, \Sigma_2, \delta_2, q_{20}, Q_{m2})$, the synchronous product of G_1 and G_2 is the reachable part of the automation

$$G_1 \parallel G_2 := G = (Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_{10}, q_{20}), Q_{m1} \times Q_{m2})$$

where for $(q_1, q_2) \in Q_1 \times Q_2$ and $\sigma \in \Sigma_1 \cup \Sigma_2$,

$$\delta((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \text{if } \delta_1(q_1, \sigma)! \text{ and } \delta_2(q_2, \sigma)! \\ (\delta_1(q_1, \sigma), q_2) & \text{if } \delta_1(q_1, \sigma)! \text{ and } \sigma \notin \Sigma_2 \\ (q_1, \delta_2(q_2, \sigma)) & \text{if } \delta_2(q_2, \sigma)! \text{ and } \sigma \notin \Sigma_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

TTCT procedure `Sync` is employed to realize this function and is usually employed to combine the modular plants to a single, more complex plant.

- **Meet**

  The meet operation is the same as synchronous product except that its partial transition function is defined as follows:

$$\delta((q_1,q_2),\sigma) = \begin{cases} (\delta_1(q_1,\sigma),\delta_2(q_2,\sigma)) & \text{if } \delta_1(q_1,\sigma)! \text{ and } \delta_2(q_2,\sigma)! \\ \text{undefined} & \text{otherwise} \end{cases}$$

  Note that meet and synchronous product are identical when $\Sigma_1 = \Sigma_2$. We use $\wedge$ to denote meet.

  TTCT procedure `Meet` is used to realize this function. Generally, in cases with more than one specification, the specifications are combined by the `Meet` operation.

- **Selfloop**

  For $G = (Q, \Sigma, \delta, q_0, Q_m)$, let Σ' be a set of events which is disjoined from Σ, or $\Sigma \cap \Sigma' = \emptyset$. The selflooped automaton $\tilde{G}$ is formed from G by adding transitions $\delta(q,\sigma) = q$ for $\forall \sigma \in \Sigma'$ and $\forall q \in Q$.

  TTCT procedure `Selfloop` is employed to realize this function. It is mainly used to extend a modular specification on a subalphabet to a specification for the entire plant.

- **Trim**

  In Section 2.1.2, we have mentioned that trim is the reachable and coreachable part of an automaton. TTCT procedure `Trim` is used to obtain the trim substructure of an automaton. Therefore, if an automaton has no marked state, after trim operation, the result will be an empty automaton.

2.3 Supervisory Control of DES

2.3.1 Basics of Supervisory Control

Ramadge-Wonham [RW87] supervisory control theory provides a framework for control of a DES. Here, plant's behavior is considered as the language generated by a generator. A supervisor, which is also modeled by a generator, is coupled with the plant to restrict its behavior.

To consider the real situation of a control system and impose control on the plant, the events set Σ is partitioned to two disjoined sets $\Sigma = \Sigma_c \cup \Sigma_u$. Σ_c is the set of *controllable* events and Σ_u is the set of *uncontrollable* events. The supervisor can just enable or disable the occurrence of controllable events. The occurrence of uncontrollable events should always be allowed.

A *control pattern* γ is any set within the bounds $\Sigma_u \subseteq \gamma \subseteq \Sigma$. The set of all control patterns is introduced as:

$$\Gamma := \{\gamma \in Pwr(\Sigma) \mid \gamma \supseteq \Sigma_u\}$$

A supervisory control for G is an arbitrary map $V : L(G) \rightarrow \Gamma$. Denote G under supervision of V by V/G and define recursively the closed behavior of V/G, $L(V/G)$, as the smallest set satisfying

1. $\epsilon \in L(V/G)$;
2. If $s \in L(V/G)$, $s\sigma \in L(G)$ and $\sigma \in V(s)$, then $s\sigma \in L(V/G)$;
3. No other strings belong to $L(V/G)$.

From the above definition, we know that $L(V/G)$ is nonempty and closed. The marked behavior of V/G is

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

which consists of the strings of $L_m(G)$ which survive under the supervision of V.

We say V is *nonblocking* if $\overline{L_m(V/G)} = L(V/G)$.

We can use an automaton S to implement V if

1. $L(S \wedge G) = L(S) \cap L(G) = L(V/G)$
2. $L_m(S \wedge G) = L_m(S) \cap L_m(G) = L_m(V/G)$

2.3.2 Controllability

A language $K \subseteq L(G)$ is said to be *controllable* with respect to G if

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

Considering an industrial application, we can treat $L(G)$ as the behavior of a plant G and K as a design specification. If K is controllable with respect to G, then it means that a supervisory control can be found such that the plant under control can meet the design specification. We call the agent that realizes this supervisory control map as a *supervisor*, therefore, the agent or supervisor just enables or disables some controllable events in each state to prevent a plant G reaching any illegal behavior, as shown in Fig. 2.1.

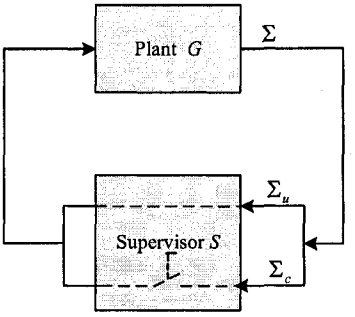

Figure 2.1: The monolithic supervisory control

If $K \subseteq L_m(G)$ which may not in general be controllable, then from [WR87] we know that a *supremal controllable sublanguage* of K can always be found and at this time, the supervisor is *maximally permissive* or *minimally restrictive.*

TTCT procedure **Supcon** is used to get the supremal supervisor S of a given specification K based on a given plant G. If $S = \emptyset$, it means that K is not controllable with respect to G, and does not have any nontrivial sublanguage that is controllable with respect to G.

2.3.3 Nonblocking

In Section 2.1.2, we defined the nonblocking property of an automaton. Here we will give the definition of a *nonblocking* supervisor. A supervisor S is nonblocking for G if

$$\overline{L_m(S \wedge G)} = L(S \wedge G)$$

TTCT procedure `Nonconflict` is used to check if S is nonblocking for G.

2.3.4 Modular Supervisory Control

Up to now, we just presented the centralized supervisor, which means that a plant G is controlled by a single supervisor S. It can only be applied to a small plant. For a large-sized or even a medium-sized plant, this approach will immediately become infeasible due to the combinatory state explosion. [FW88] and [WR88] introduce *decentralized* or *modular supervisory control* and it can solve this problem under certain conditions. Fig. 2.2 illustrates this idea.

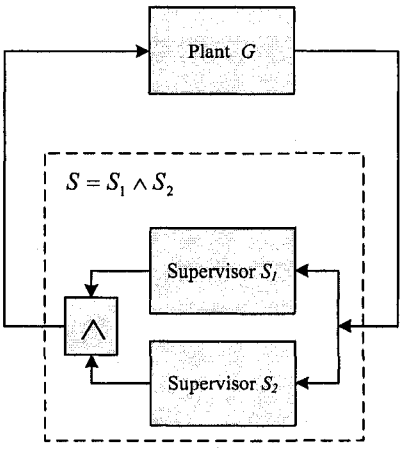

Figure 2.2: The modular supervisory control

To define modular supervision, let's first introduce what a *proper supervisor* is.

We call S a proper supervisor for G when:

1. S is trim,
2. $L_m(S)$ is controllable with respect to G,
3. $S \wedge G$ is nonblocking or $\overline{L_m(S \wedge G)} = L(S \wedge G)$.

[FW88], [WR88] and [Won06] give the following result.

If S_i, i=1, 2, are two proper supervisors for plant G, then for $S = S_1 \wedge S_2$ to be a proper supervisor for G we must have:

1. S_1 and S_2 are controllable with respect to G,

2. $S_1 \wedge S_2$ and G are nonblocking,

3. $S_1 \wedge S_2$ is trim.

The modular supervisory control is adopted in our work as a critical solution to verify logic diagrams.

2.4 Industrial Logic Diagram

Industrial logic diagrams are widely used in engineering design in logic control field. Generally, an industrial logic diagram is a graphic representation of design specifications and is composed of some basic logic gates, for example *and*, *or*, *not*, *flip-flop* and *time delay* as illustrated in Fig. 2.3.

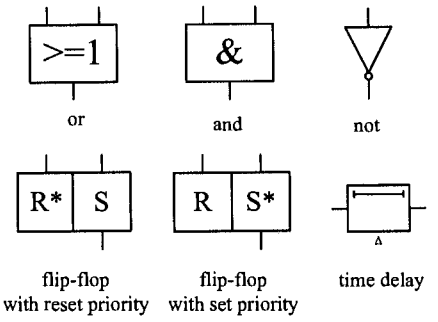

Figure 2.3: Basic logic gates

These basic logic gates are powerful and can realize most design specifications. In Appendix A, an Emergency Diesel Generator Startup Controller is specified based on these logic gates.

2.5 Conclusion

In this chapter, we have given a brief and necessary introduction to languages, automata and supervisory control theory of DES. The corresponding operations and TTCT procedures are provided. The concepts of controllability and nonblocking are presented as well. A brief

introduction to logic diagrams is given. This background knowledge will become the basis of our further discussions in the next two chapters.

Chapter 3

Logic Models And Their Functional Verification

3.1 Introduction

In order to verify the correctness of a logic diagram, the following steps must be taken:

1. Find a controllable and nonblocking supervisor S_i for each logic diagram;
2. Convert the informal design function requirements to one or several formal specification automata V_i;
3. Verify whether $L(S_i) \subseteq L(V_i)$ for each i to check the design functions.

We can use the specification automata in step (2) and the uncontrolled plant model to design a supremal supervisor for each specification automaton. But there are drawbacks to this approach, namely, 1) the supremal supervisor can be too large for efficient implementation by PLCs or other industrial controllers, which is a potential application in the future for our research, and 2) the functionality of such a supervisor may not be very well understood. Instead, we use traditional, suboptimal supervisors designed as logic diagrams. To make sure that what is modeled by a logic diagram can serve as a supervisor, we need to show that it is controllable and nonblocking (step (1)), and that it satisfies the specification (step (3)).

A standard industrial logic diagram, generally, is composed of some basic logic gates, such as AND, OR and Flip-Flop logic elements. Another special element is time delay. In the first part of this chapter we will give a standard plant model and its supervisor for each basic logic gate. A motivating example will be used to show the complete methodology in Section 3.3. Section 3.4 will try to formally prove the controllability of the combined DES supervisor from the controllability of the logic gates in each circuit, and infer its nonblocking nature. The next chapter will apply this method to an industrial diesel generator startup controller to demonstrate its feasibility in real industrial control systems.

Remark: Here we will consider each logic diagram as a DES supervisor of a single DES plant.

In each logic diagram, there are always several basic logic elements. Obtaining an appropriate DES plant and its supervisor is not a trivial task. We probably can design a monolithic plant and supervisor according to the function requirements of each logic diagram, but this will not be easy in most situations because the size of the resulting supervisor becomes unmanageable and it is impossible to be taken as a standard method. Modular approach seems more feasible in real industrial systems, which can take full advantage of modular DES supervisory control approaches.

The design of plant model and its corresponding supervisor for basic logic gates is not difficult. It is therefore natural to find a suitable DES plant model and its supervisor for each basic logic element. Then, a monolithic supervisor of a logic diagram can be obtained with modular supervisory control theory of DES by synchronizing the DES supervisor for each basic logic element. In the next section, DES plant models and their corresponding supervisors for basic logic elements will be presented.

3.2 Basic Logic Models and Supervisors

Before giving the detailed models, we need to make the following important assumption:

Assumption: All input signals in a logic gate will be treated as uncontrollable, and the output signals will be treated as controllable. This assumption is reasonable from the controller's point of view.

In order to model dynamical time-varying behavior and memory capability of a logic controller, we introduce a special timing event T, which represents *the scan cycle of the PLC*. It must be pointed out that despite this our models differ from timed DES models; the convenience of introducing T is to simulate the real controller behavior and to model flip-flop and time delays.

We assume that at every point in time the value of a signal x is either 0 or 1. Events are derived from signals in the following way: at the beginning of the kth cycle the signal x is sampled. If $x(kT) = 1$ then event X is generated at the kth cycle, while if $x(kT) = 0$ event $\overline{X}$ is generated.

NOT gate is a basic logic gate too, but it will not be separately modeled since its nature can be easily denoted by the event itself. For example, when a signal x is 1(0), then event $X(\overline{X})$ is generated. We can directly use $\overline{X}(X)$ to represent the NOT of $X(\overline{X})$.

3.2.1 AND Gate

Fig. 3.1 is a 2-input AND gate and Table. 3.1 gives its truth table. The value 1 represents that the corresponding signal is triggered and the value 0 means that it is not triggered. A multiple-input AND gate can be decomposed into several 2-input AND gates, so a 2-input gate is a basic logic element.

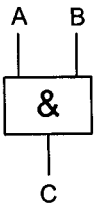

Figure 3.1: AND gate

For each input signal, a basic plant model is shown in Fig. 3.2-(a) and (b). The model is reasonable since an output event can only be generated after an input event. Other than this functional requirement, the plant model does not impose any restriction on the input-output relationship: for *any* value of the input, the output can assume *any* value. It simulates the

Table 3.1: Truth table of AND.

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

operation sequence of a real controller, for example, a PLC. The synchronization of plant models for each input yields

$$G = G_1 \parallel G_2$$

The final logic plant model G is shown in Fig. 3.2-(c). The state after event T is always marked. The reason is that the controller processing time is much shorter than the plant response time, and any functional requirement is always based on a complete processing cycle. Here the marked state is also the initial state.

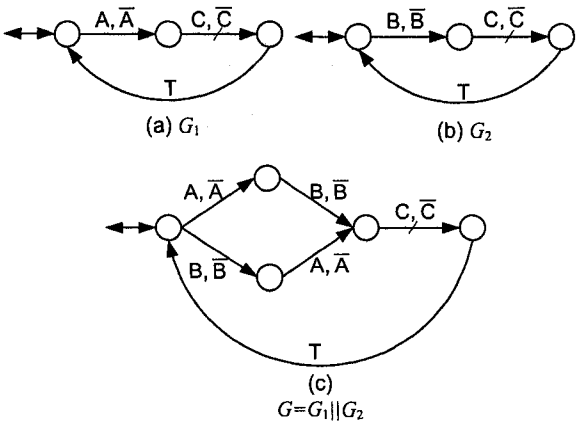

Figure 3.2: Logic plant model

From its truth table, a specification of the AND gate can be derived as in Fig. 3.3. Using the TTCT command `SUPCON`, it is verified that the specification is controllable and nonblocking and can thus serve as a supervisor. From Fig. 3.3 we can see that when $A(B)$

is zero, regardless of the value of $B(A)$ the output signal C must be set to zero.

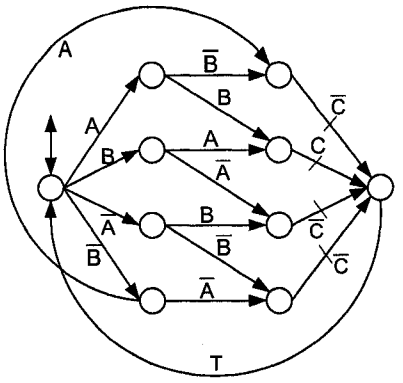

Figure 3.3: AND specification and supervisor

3.2.2 OR Gate

Similarly, Fig. 3.4 is a 2-input OR gate and Table. 3.2 gives its truth table. Due to their symmetry, the plant model of an OR gate is exactly the same as the plant model of an AND gate.

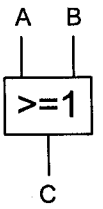

Figure 3.4: OR gate

Based on its truth table, the specification of the OR gate can be derived as in Fig. 3.5. Using TTCT command `SUPCON`, the nonblocking and controllable supervisor satisfying this specification can be shown to be the same as the specification in Fig. 3.5.

From Fig. 3.5 we can see that when A (B) is one, regardless of the value of B (A) the output signal C must be set to one.

Table 3.2: Truth table of OR.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

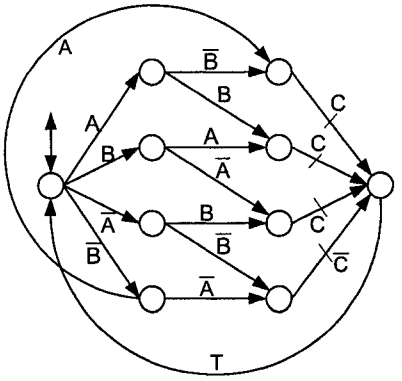

Figure 3.5: OR specification and supervisor

3.2.3 Flip-flop

An RS flip-flop is a special logic element which includes memory function. Depending on the input priority, there are two types of flip-flops.

- RS with *reset* priority.

  Fig. 3.6 is an RS gate with reset priority and Table. 3.3 gives its truth table. R stands for Reset and S stands for Set.

  From the truth table, we can observe that when both inputs are 0, the output will keep its previous value. Output will always be 0 as long as R is 1, which means that *reset* has priority over *set*. It will be quite complicated if we just try to model the behavior of a flip-flop with ordinary DES models without the timing event. Fortunately, this

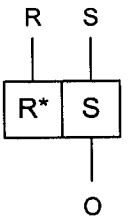

Figure 3.6: RS gate with reset priority

Table 3.3: Truth table of RS.

R	S	O
0	0	*No change*
0	1	1
1	0	0
1	1	0

problem can be easily dealt with when timing event T is introduced. The plant model will be a little bit different now as illustrated in Fig. 3.7.

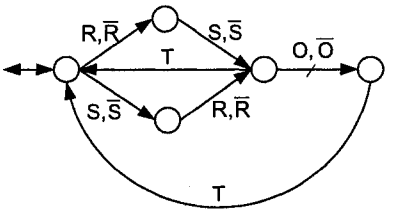

Figure 3.7: RS plant model

In this model, controllable outputs O and $\overline{O}$ are forcible and can preempt the time event T, so they can be either both disabled, indicating that the previous value of the output must be kept, or when either one is enabled it can preempt the transition labeled with T back to the initial state.

The specification is derived from Table 3.3 in Fig. 3.8. We can see that when both

inputs are 0, O and $\overline{O}$ are disabled and only T can happen back to the initial state. In this case, the output will keep its previous value. It can be easily verified using TTCT that the nonblocking and controllable supremal supervisor satisfying this specification is isomorphic to the specification in Fig. 3.8.

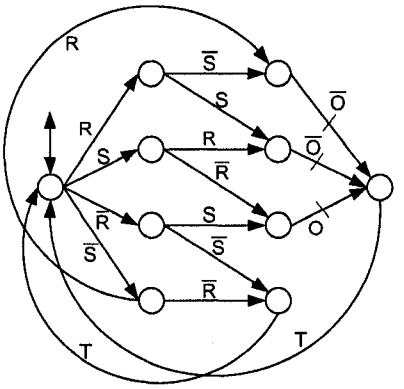

Figure 3.8: RS specification and supervisor

- RS with *set* priority.

  RS with set priority (also called SR) is almost identical to RS with reset priority and is shown in Fig. 3.9. The only difference is that the *set* signal has priority over the *reset* signal as shown in Table. 3.4. Note that the output will always be 1 when S is 1. The plant model is unchanged (Fig. 3.7).

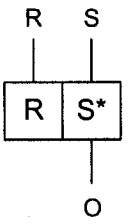

Figure 3.9: RS gate with set priority

The controllable and nonblocking specification, which is isomorphic to its supremal

Table 3.4: Truth table of SR.

R	S	O
0	0	*No change*
0	1	1
1	0	0
1	1	1

supervisor, is depicted in Fig. 3.10.

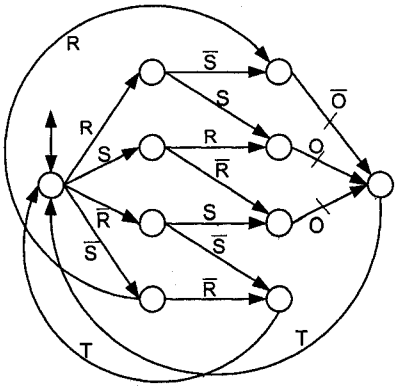

Figure 3.10: SR specification and supervisor

3.2.4 Time Delay

Time delay can be categorized to two types, *Power-On Time Delay* and *Power-Off Time Delay*. The timing event introduced earlier can be thought of as having a *qualitative* nature—it represents the passage of *some* units of time. Time delay duration Δ can be one cycle time, or millions of cycle times. It is impossible to model the exact duration of delay with a simple untimed DES model. In this work, the duration of all time delays will be abstracted as one cycle time T, and the delay circuit will be modeled as a buffer. As a result, we can only verify the existence of time delay, but cannot verify its duration.

- *Power-On Time Delay.*

Fig. 3.11 is the power-on time delay and Fig. 3.12 describes its exact function. Parameter Δ is the delay duration. Obviously, when the input I jumps from 0 to 1, the controller will start a timer with preset time Δ and the output O will change from 0 to 1 if the timer expires while the input has been continuously 1; otherwise, the output will stay at 0. Once the input I changes from 1 to 0, the output O will immediately change from 1 to 0 without any delay.

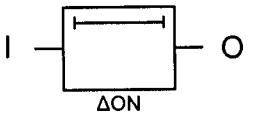

Figure 3.11: Power-on time delay

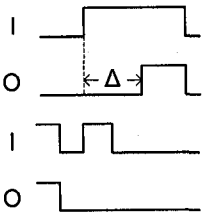

Figure 3.12: Power-on time delay sequence

As mentioned above, we will abstract any arbitrary duration Δ to a single cycle time T and model the delay circuit as a buffer; Fig. 3.13 is the model we use in this work for power-on time delay.

- *Power-Off Time Delay.*

  Power-off time delay is just the opposite of power-on time delay. Fig. 3.14 is the power-off time delay and Fig. 3.15 describes its exact sequence function.

  Fig. 3.16 shows the model which will be used in this thesis.

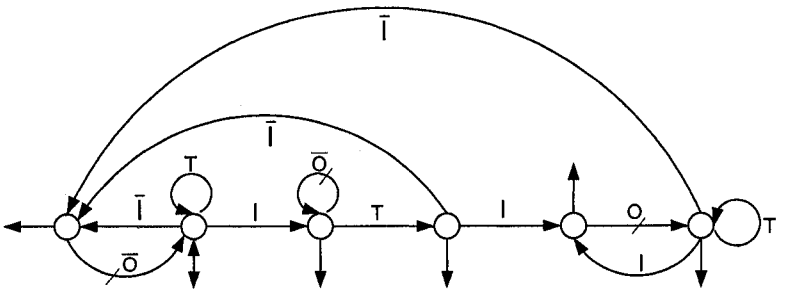

Figure 3.13: Power-on time delay model

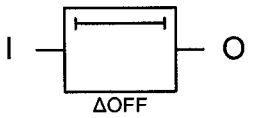

Figure 3.14: Power-off time delay

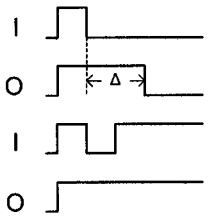

Figure 3.15: Power-off time delay sequence

3.3 A Motivating Example

The previous section has covered all basic logic elements in a standard industrial control logic diagram. However, it is still impossible to get a monolithic supervisor from the modular supervisors for basic logic elements because the interconnections between basic logic elements have not been defined yet. In this section we use a motivating example to describe how different basic elements in a logic diagram are interconnected.

First, we will give a typical industrial memory logic as in Fig. 3.17. To give the example

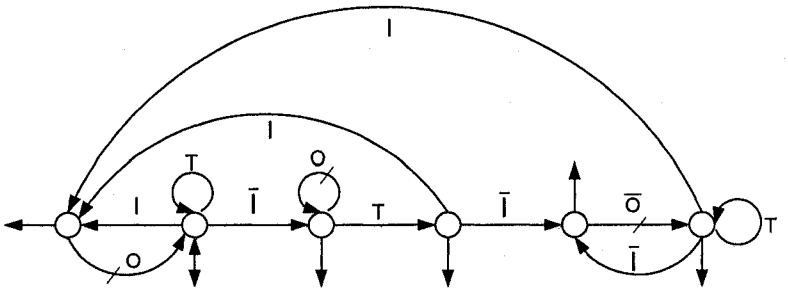

Figure 3.16: Power-off time delay model

a practical flavor, assume that input B can be considered as a high-level signal from a level switch in a tank, and B' is the alarm acknowledgement signal from a pushbutton. Once the level switch sends a high-level signal, the output C' will be set to 1 and an alarm will sound, even if B becomes 0, until pushbutton B' is manually pressed to 0 to reset the alarm.

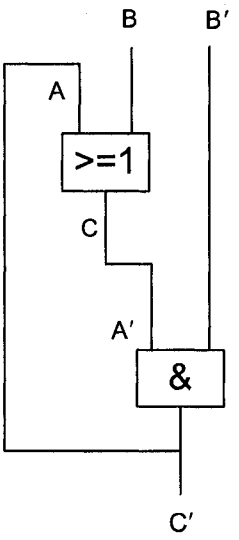

Figure 3.17: A typical memory logic diagram

Second, we will define the buffer model. A logic diagram is a network consisting of basic logic elements interconnected by wires. Not only should each logic element be modeled, but

also their interconnections must be specified. After analyzing Fig. 3.17, we can find that there are two types of interconnections, *forward* connection and *feedback* connection. In Fig. 3.17, the connection from C to A' is a forward connection, and the connection from C' to A is a feedback connection. In the next part we will model these two types of connections with two types of buffers: for forward connection, we require that the value of the input be consumed by the output in the *same* cycle, while in a feedback connection the value of the input can be consumed by the output *only* in the *next* scan cycle.

Finally, we will show how to get a monolithic plant model from the basic logic plant models and their interconnection buffers.

3.3.1 Buffer Models

All buffer models are to be synchronized with basic logic plant models to obtain a monolithic plant model.

Forward Buffer Model.

Fig. 3.18 shows the forward buffer model in which all states are marked. After receiving a controllable event C or $\overline{C}$ from the output of a gate, forward buffer will *immediately* (i.e. in the same scan cycle) sends an uncontrollable event A' or $\overline{A'}$ to the input of a *subsequent* gate (i.e. a gate whose output in no way influences the output of the current gate).

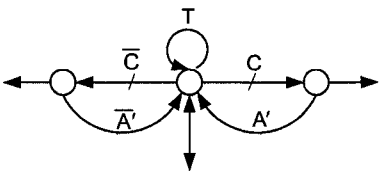

Figure 3.18: Forward buffer model

Feedback Buffer Model.

Fig. 3.19 shows the feedback buffer model in which all states are marked. After receiving a controllable event C' or $\overline{C'}$ from the output of a gate, feedback buffer must wait for the

next scan cycle to send an uncontrollable event A or $\overline{A}$ to the input of a *preceding* gate (i.e. a gate whose output affects the output of the current gate).

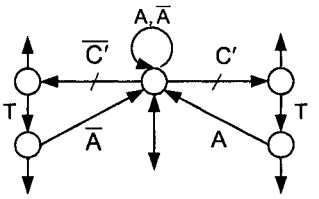

Figure 3.19: Feedback buffer model

3.3.2 Case Analysis

Monolithic Supervisor

In this subsection we will obtain a monolithic supervisor for the system of Fig. 3.17 and show the isomorphism between the monolithic supervisor and the meet of modular supervisors.

First, we define the following notations:

- P_1 is the plant model of OR gate.
- P_2 is the plant model of AND gate.
- $Spec_1$ is the supervisor of OR gate.
- $Spec_2$ is the supervisor of AND gate.
- $\tilde{Spec}_1$ is $Spec_1$ with all of events not in OR selflooped.
- $\tilde{Spec}_2$ is $Spec_2$ with all of events not in AND selflooped.
- B_1 is the forward buffer model from C to A',
- B_2 is the backward buffer model from C' to A.

Naturally, the monolithic plant automaton model P of Fig. 3.17 can be obtained as:

$$P = P_1 \parallel P_2 \parallel B_1 \parallel B_2$$

In addition, the modular supervisor of the OR gate, S_1, can be obtained by

$$\tilde{S_1} = \texttt{SUPCON}(P, \tilde{Spec_1})$$

The modular supervisor of the AND gate, S_2, can be obtained by

$$\tilde{S_2} = \texttt{SUPCON}(P, \tilde{Spec_2}).$$

Actually, we can directly obtain the result since $\tilde{S_1}$ is isomorphic to $\tilde{Spec_1}$ (similarly, $\tilde{S_2}$ is isomorphic to $\tilde{Spec_2}$). The reason is that $Spec_1$ is already controllable and nonblocking with respect to P_1, and we selfloop events that are not in P_1 to obtain $\tilde{Spec_1}$, so $\tilde{Spec_1}$ should remain controllable and nonblocking with respect to P.

The monolithic specification can be obtained by:

$$Spec = \tilde{Spec_1} \bigcap \tilde{Spec_2}$$

Then, the monolithic supervisor S will be:

$$S = \texttt{SUPCON}(P, Spec)$$

We can verify that $\tilde{S_1} \wedge \tilde{S_2}$ is controllable and nonblocking, by using TTCT; in addition, we can verify that

$$L(\tilde{S_1}) \cap L(\tilde{S_2}) \cap L(P) = L(S) \cap L(P)$$

$$L_m(\tilde{S_1}) \cap L_m(\tilde{S_2}) \cap L_m(P) = L_m(S) \cap L_m(P)$$

which means the joint supervision of the modular supervisors induces the same behavior as the monolithic supervisor does.

Logic Design Function Verification

The design specification of the logic diagram of Fig. 3.17 is informally stated as follows.

1. When tank level switch B sends a high-level signal and alarm-acknowledge pushbutton is not pressed, the alarm output C' should be set;

2. Once C' is set, it should remain set without caring about the status of the level switch B. The alarm signal can only be reset by alarm-acknowledge pushbutton B'.

In general, the translation of informal English requirements to formal automata models is a challenging task which requires logical thinking and sufficient knowledge about the system. We will try to give a reasonable and feasible way to obtain the models.

For function 1, the alarm output C' must be generated (i.e. alarm should sound) when the following two conditions are both satisfied: level switch B sends a high signal, and alarm-acknowledge pushbutton B' is not pressed. So we summarize this by a logic *and* equation $C' = B \wedge B'$. This equation will be the basis of the automaton model. Fig. 3.20 is the automaton model V_1 based on the above analysis. In this model, after B and B', only C' is permitted (i.e. $\overline{C'}$ is disabled). Note that the automaton must be carefully designed to faithfully translate the English statement: no new behavior should be introduced, and no legal behavior should be removed.

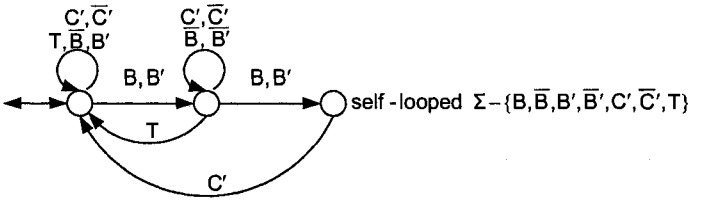

Figure 3.20: Specification 1

For function 2, it is a memory function which is very convenient to be described by an automaton. Fig. 3.21 shows the memory automaton model V_2. In this model, once C' is generated, only $\overline{B'}$ can reset the automaton to its initial state.

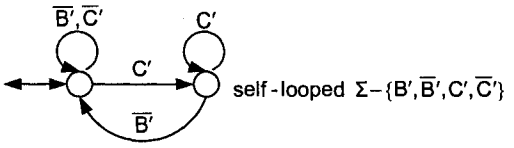

Figure 3.21: Specification 2

The next step is to verify that the supervisor S satisfies these two function requirements, which means:

$L_m(S) \subseteq L_m(V_1)$ and $L_m(S) \subseteq L_m(V_2)$

or equivalently,

$L_m(S) \cap \overline{L_m(V_1)} = \emptyset$ and $L_m(S) \cap \overline{L_m(V_2)} = \emptyset$

TTCT `COMPLEMENT` and `MEET` commands have been used to verify the above equalities.

A Reduced Verification Technique

The motivating example described in this section is not a complicated example. If a logic diagram has more basic logic elements and inputs, the design of specification model will be quite difficult, which will make this method not so applicable to real industrial applications. In this subsection, we will try to simplify the verification of a complicated logic diagram by fixing some inputs to either *true* or *false* when specifying a certain requirement, which will make the specification design much easier and can directly yield a reduced supervisor from the monolithic supervisor.

In a complicated logic diagram, we usually assume some fixed initial conditions to simplify the specification of a requirement. For instance, in the above example, we can assume that alarm-acknowledge pushbutton B' is not pressed when specifying the first requirement. Then the first specification will be simplified as: When tank level switch B sends a high-level signal, the alarm output C' will send a signal. This simplified specification V_1' is shown in Fig. 3.22.

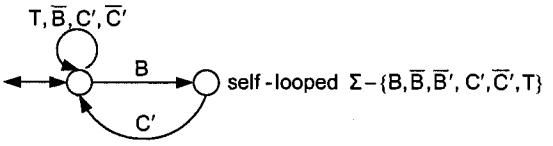

Figure 3.22: Simplified specification 1

If the condition that alarm-acknowledge pushbutton B' is not pressed is assumed, it follows that the event $\overline{B'}$ is not present in the alphabet of the supervisor. We call the resulting supervisor a reduced supervisor, denoted by S'. This reduced supervisor S' can be

obtained by taking the meet of the supervisor S with the automaton C_1 shown in Fig. 3.23, i.e. $S' = S \wedge C_1$. It can be observed that C_1 is a single state automaton with all events self-looped except the event $\overline{B'}$ which is to be removed. According to the definition of meet, S' will be the same as S with all transitions labeled with $\overline{B'}$ removed.

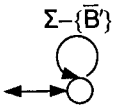

Figure 3.23: Assumed condition $B' = 0$

Then we can use the same logic to verify that the reduced supervisor S' satisfies the specification V_1', or $L_m(S') \cap \overline{L_m(V_1')} = \emptyset$.

The key observation in the above example is that the original requirements V_1 and V_2 have *redundancies*, in the sense that part of the plant's behavior is restricted by both V_1 and V_2. The requirement specification V_1' is more efficient in that it doesn't duplicate but complements V_2.

For any signal a, break every (supervisor, specification) pair into two pairs: one in which a is assumed to be 0, and the other in which a is assumed to be 1. If both pairs are verified, the original pair is verified. Sometimes only one pair need to be verified. The other pair is not necessary to be verified since it is contained in other specifications (as in the above example), or it is not required by the specification (as in the next chapter).

It can be observed that V_1' is less complex than V_1 while jointly with V_2 it restricts the plant's behavior in an identical fashion. This benefit will be much more significant if the logic diagram is more complicated as shown in the next chapter's real industrial example. The operational procedure can be summarized as follows:

1. Decide the input signals to be fixed as triggered (1) or not-triggered (0) according to the informal function requirements, which needs correct understanding of the system operation;

2. Design a simplified specification V_i' without considering the fixed events;

3. Design a single state condition automaton C_i with all events self-looped except the fixed input events;

4. Synchronize the monolithic supervisor S with C_i to get a reduced supervisor S';

5. Verify $L_m(S') \subseteq L_m(V_i')$ to verify the design function.

3.4 Controllability and Nonblocking

The previous section has given a detailed description of basic plant models and their supervisors. A motivating example has shown that the monolithic supervisor is isomorphic to the composition of the modular supervisors, and is controllable and nonblocking.

However, it is necessary to give a formal proof for the general case to make sure that this method can be applied to industrial control systems. This section presents a formal proof of the controllability of the composition of modular supervisors, and makes informal arguments in support of the nonblocking of the plant controlled by the modular supervisors.

3.4.1 Definitions

Each plant P in a logic diagram is composed of $n \geq 1$ basic plant models and $m \geq 0$ basic buffer models. The event set of P is denoted by Σ, and consists of disjoint controllable event set Σ_c and uncontrollable event set Σ_u.

Let $I = \{1, 2, \cdots, n\}$ be an index set for the n basic plant models. Define a basic plant model as:

$$G_i = (X_i, \Sigma_i, \xi_i, x_{0i}, X_{mi}),\ i \in I$$

Let $J = \{1, 2, \cdots, m\}$ be an index set for the m basic buffer models. Define a basic buffer model as:

$$B_j = (Y_i, \Sigma_j', \eta_j, y_{0i}, Y_{mi}),\ j \in J$$

Here,

$$\Sigma'_1 \cup \Sigma'_2 \cup \ldots \cup \Sigma'_m \subseteq \Sigma_1 \cup \Sigma_2 \cup \ldots \cup \Sigma_n = \Sigma$$

The monolithic plant model P is defined by taking the parallel composition of all basic plant models and all basic buffer models:

$$P = G_1 \parallel G_2 \parallel \ldots \parallel G_n \parallel B_1 \parallel B_2 \parallel \ldots \parallel B_m$$

For each basic plant model G_i, define the corresponding supervisor as:

$$S_i = (Z_i, \Sigma_i, \zeta_i, z_{0i}, Z_{mi}),\ i \in I$$

Let $\tilde{S}_i$ be a supervisor that is identical to S_i except that events not in Σ_i are self-looped:

$$\tilde{S}_i = (Z_i, \Sigma, \tilde{\zeta}_i, z_{0i}, Z_{mi}),\ i \in I$$

where

$$\forall z \in Z_i, \forall \sigma \in \Sigma.\ \tilde{\zeta}_i(z,\sigma) = \begin{cases} \zeta_i(z,\sigma) & ;\ \sigma \in \Sigma_i \\ z & ;\ \sigma \notin \Sigma_i \end{cases}$$

We denote all the uncontrollable events in Σ_i by $\Sigma_{i,u}$; the set of all other uncontrollable events is denoted by $\Sigma_{i,ru}$: $\Sigma_{i,ru} = \Sigma_u - \Sigma_{i,u}$.

3.4.2 Controllability

In this section, we show that the controllability of gate supervisors implies the controllability of the composition of all gate supervisors. First we will give Lemma 1, then we will prove Theorem 3.4.1 based on this Lemma.

Lemma 1 Suppose that S_i, $\tilde{S}_i$ and P are defined according to Section 3.4.1 and S_i is constructed following the procedure in Section 3.2. Then $\tilde{S}_i$ is controllable with respect to plant P.

Proof:

Since S_i is controllable with respect to G_i, and $L(S_i)$ is a prefix-closed language, we have:

$$L(S_i)\Sigma_{i,u} \cap L(G_i) \subseteq L(S_i) \tag{3.1}$$

We must show that $\tilde{S}_i$ is controllable with respect to plant P, i.e.:

$$L(\tilde{S}_i)\Sigma_u \cap L(P) \subseteq L(\tilde{S}_i) \tag{3.2}$$

Let P' be the plant without buffers, i.e.:

$$P' = G_1 \parallel G_2 \parallel \ldots \parallel G_n$$

It is obvious that

$$L(P) \subseteq L(P') \tag{3.3}$$

First, we show that $\tilde{S}_i$ is controllable with respect to P', i.e.:

$$L(\tilde{S}_i)\Sigma_u \cap L(P') \subseteq L(\tilde{S}_i) \tag{3.4}$$

Then our target equation (3.2) immediately follows from equations (3.3) and (3.4).

Let $P_i : \Sigma^* \rightarrow \Sigma_i^*$ denote the natural projection for $i \in I$. Let $s \in L(\tilde{S}_i)$, $\sigma \in \Sigma_u$ and $s\sigma \in L(P')$. We must show that $s\sigma \in L(\tilde{S}_i)$.

We consider two cases: 1) $\sigma = T$, 2) $\sigma \in \{\Sigma_u - T\}$.

1. For $\sigma = T$, since

$$L(P') = P_1^{-1}(L(G_1)) \cap P_2^{-1}(L(G_2)) \cap \ldots \cap P_n^{-1}(L(G_n)) \tag{3.5}$$

then $sT \in L(P')$ implies that for all $i \in I$, $sT \in P_i^{-1}(L(G_i))$. It follows that

$$P_i(sT) \in L(G_i)$$

In each basic plant model $i \in I$ we have $T \in \Sigma_i$, so

$$P_i(s)T \in L(G_i) \tag{3.6}$$

From the definition of $\tilde{S}_i$ it follows from $s \in L(\tilde{S}_i)$ that

$$P_i(s) \in L(S_i) \tag{3.7}$$

From equations (3.6) and (3.7), and the controllability of S_i it follows that

$$P_i(s)T \in L(S_i)$$

which implies $P_i(sT) \in L(S_i)$, and therefore $sT \in P_i^{-1}(L(S_i)) = L(\tilde{S}_i)$ for all $i \in I$, as desired.

2. For $\sigma \in \{\Sigma_u - T\}$, for any $i \in I$, there are two possible sub-cases,

   2-1) $\sigma \in \Sigma_{i,u}$, or 2-2) $\sigma \in \Sigma_{i,ru}$.

   - **Sub-case 2-1:** $\sigma \in \Sigma_{i,u}$

     The proof is very similar to case 1.

     From equation (3.5), $s\sigma \in L(P')$ implies that for all $i \in I$, $s\sigma \in P_i^{-1}(L(G_i))$. It follows that

     $$P_i(s\sigma) \in L(G_i)$$

     In each basic plant model $i \in I$ we have $\sigma \in \Sigma_i$, so

     $$P_i(s)\sigma \in L(G_i) \tag{3.8}$$

     From equations (3.7) and (3.8), and the controllability of S_i it follows that

     $$P_i(s)\sigma \in L(S_i)$$

     which implies $P_i(s\sigma) \in L(S_i)$, and therefore $s\sigma \in P_i^{-1}(L(S_i)) = L(\tilde{S}_i)$ for all $i \in I$, as desired.

   - **Sub-case 2-2:** $\sigma \in \Sigma_{i,ru}$

     It is quite obvious in this situation.

     Since $\sigma \notin \Sigma_i$, it is apparent that

     $$P_i(s\sigma) = P_i(s)$$

     From equation (3.7), it follows that

     $$P_i(s\sigma) \in L(S_i)$$

     which implies $s\sigma \in P_i^{-1}(L(S_i)) = L(\tilde{S}_i)$ for all $i \in I$, as desired. ■

Now we will prove that the monolithic supervisor is controllable with respect to P.

Theorem 3.4.1 Suppose that $S_1,\ S_2 \ldots S_n,\ P$ are defined according to Section 3.4.1 and $S_1,\ S_2 \ldots S_n$ are constructed following the procedure in Section 3.2. Then $S = S_1 \parallel S_2 \parallel \ldots \parallel S_n$ is controllable with respect to plant P.

Proof: According to Lemma 1, we know that $\tilde{S}_i$ is controllable with respect to plant P. We also notice that $L(\tilde{S}_i)$ is a prefix-closed language, and

$$L(S) = L(\tilde{S}_1) \cap L(\tilde{S}_2) \cap \ldots \cap L(\tilde{S}_n) \tag{3.9}$$

so, $L(S)$ is also a prefix-closed language. We only need to show

$$L(S)\Sigma_u \cap L(P) \subseteq L(S)$$

Let $s \in L(S),\ \sigma \in \Sigma_u$, and $s\sigma \in L(P)$. We must show that $s\sigma \in L(S)$.

From equation (3.9), we know that

$$s \in L(\tilde{S}_i),\ \forall i \in I$$

Since $\tilde{S}_i$ is controllable w.r.t. P (Lemma 1), it follows that

$$s\sigma \in L(\tilde{S}_i),\ \forall i \in I$$

so

$$s\sigma \in L(S)$$

as desired. ■

We have formally proved the controllability of S. The next subsection will argue in favor of S being nonblocking.

3.4.3 Nonblocking

For the nonblocking property of the monolithic supervisor, at first we will prove that the monolithic supervisor S is nonblocking for P'.

Lemma 2 Suppose that $S_1,\ S_2 \ldots S_n,\ P'$ are defined according to Section 3.4.1 and $S_1,\ S_2 \ldots S_n$ are constructed following the procedure in Section 3.2. Then $S = S_1 \parallel S_2 \parallel \ldots \parallel S_n$ is nonblocking for P'.

Proof:

We already know that S_i is nonblocking for G_i from Section 3.2. And $\tilde{S}_i$ is identical to S_i except that the events not in Σ_i are self-looped, so $\tilde{S}_i$ is nonblocking for P'.

In order to prove this Lemma, we need to show:

$$\overline{L_m}(S/P') = L(S/P')$$

Since $\overline{L_m}(S/P') \subseteq L(S/P')$ is automatic, we only need to show

$$L(S/P') \subseteq \overline{L_m}(S/P')$$

We will prove it by contradiction. If we assume that

$$L(S/P') \nsubseteq \overline{L_m}(S/P')$$

then there exists $s \in \overline{L_m}(S/P')$ and $\sigma \in \Sigma$ such that $s\sigma \in L(S/P')$ but $s\sigma \notin \overline{L_m}(S/P')$. Two cases need to be considered: 1) $\sigma \in \Sigma_i - \{T\}$ for some $i \in I$, 2) $\sigma = T$.

1. For $\sigma \in \Sigma_i - \{T\}$, $\exists i \in I$,

   It follows from the assumption that for some $i \in I$,

   $$s\sigma \in L(\tilde{S}_i) \cap L(P')$$

   and since $\tilde{S}_i$ is nonblocking for P', then

   $$s\sigma \in \overline{L_m(\tilde{S}_i) \cap L_m(P')}$$

   Since σ is self-looped in all states of $\tilde{S}_j$, $j \neq i$, $(i,\ j \in I)$ it follows that

   $$s\sigma \in \overline{L_m(\tilde{S}_1) \cap \ldots \cap L_m(\tilde{S}_n) \cap L_m(P')} = \overline{L_m}(S/P')$$

   which is a contradiction.

2. For $\sigma = T$,

   It also follows from the assumption that for all $i \in I$,

   $$sT \in L(\tilde{S}_i) \cap L(P')$$

and similarly, since $\tilde{S}_i$ is nonblocking for P', then

$$sT \in \overline{L_m(\tilde{S}_i) \cap L_m(P')},\ \forall i \in I$$

It follows that

$$sT \in \overline{L_m(\tilde{S}_1) \cap \ldots \cap L_m(\tilde{S}_n) \cap L_m(P')} = \overline{L_m}(S/P')$$

which is a contradiction, too. ■

Next, we will give an informal explanation why S is also nonblocking for P.

Since $P = P' \parallel B_1 \parallel B_2 \parallel \ldots \parallel B_m$, we need to consider whether B_i will introduce a blocking situation. After checking the structures of forward buffer and feedback buffer, we can state that both buffers will unlikely cause blocking when plant P is controlled by S. The reason is that they don't disable any controllable events and all states are marked. Both buffers permit two controllable events which are generated from the preceding logic gate. After generating a controllable event, the forward buffer will generate a corresponding uncontrollable event and return to the initial, marked state. After generating a controllable event and waiting for the next cycle, the feedback buffer will also generate a corresponding uncontrollable event and return to the initial marked state.

We conclude that the monolithic supervisor S is nonblocking for P based on the above proof and explanation.

3.5 Conclusion

In this chapter, at first we described the plant model and its supervisor of each basic logic gate. Then a motivating example was used to show the general logic verification process. At the same time, a reduced verification technique was introduced. Finally we gave a formal proof of controllability of the monolithic supervisor and an informal argument of its nonblocking property.

Chapter 4

Diesel Generator Startup Controller Function Verification

4.1 Introduction

Emergency Diesel Generator Set (GS) is a critical backup power supply system in many industries, especially in nuclear power plants. A typical GS consists of several subsystems (Fig. 4.1) [Fr96], including:

- *Diesel engine and generator*, is the machine body itself.
- *Compressed air system*, produces high-pressure compressed air to start GS.
- *Preheating system*, maintains GS in hot state when GS is shutdown.
- *Lubrication oil system*, lubricates GS.
- *Cooling water system*, dissipates the heat of lubrication oil into cooling water. The cooling water is cooled by air cooler fan.
- *Fuel oil system*, provides fuel oil.
- *Ventilation system*, ventilates GS building when GS is running.

When GS is in standby mode, the cooling water is heated by the electrical heater of preheating system to keep GS in hot state. A pre-lubricating oil pump is also running to

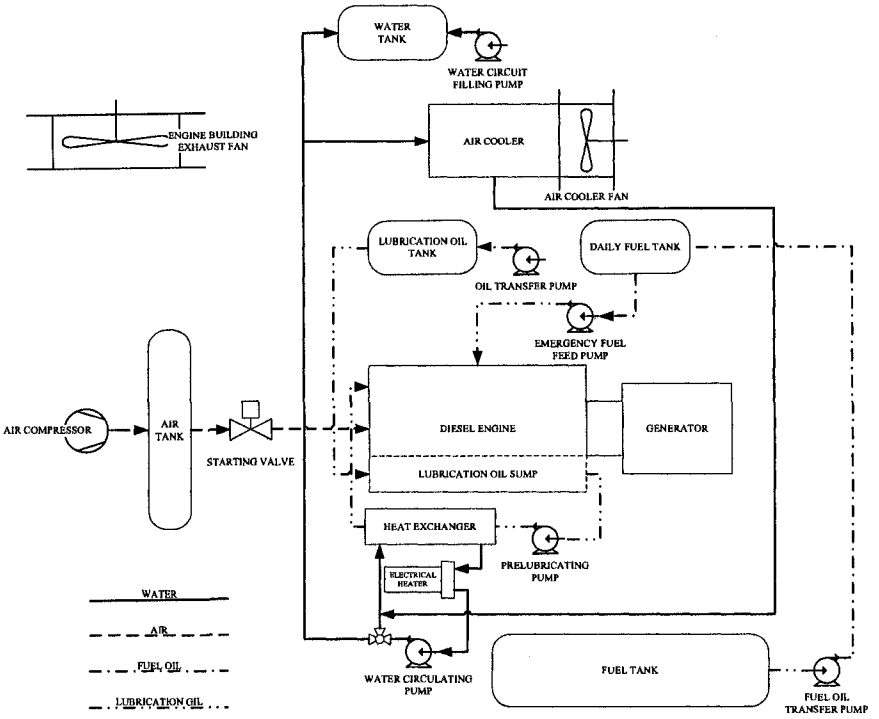

Figure 4.1: Diesel generator flow diagram

lubricate the machine. At this time, lubrication oil is heated by the water through the oil/water heat exchanger. However, when GS is running, it is the opposite, which means that lubrication oil is cooled by the water. The water itself is cooled by the air cooler fan located on top of the building when its temperature is higher than 80 degrees Celsius to prevent it from boiling.

Once GS receives a starting order, it should start automatically by the high-pressure compressed air. The starting order will open a starting solenoid valve to conduct the high-pressure compressed air to the engine cylinders. The maximum duration of starting order is 5 seconds. When GS is started, the controller will simultaneously start the fuel oil transfer pump, emergency fuel feeding pump and building exhaust fan. The function of fuel oil transfer pump is to transfer the fuel oil from a large underground storage tank to a smaller and high-positioned daily tank. Emergency fuel feeding pump is used to supply fuel oil from the daily tank to the diesel engine.

The start order can be classified to two types, *remote start order* and *local start order*. The remote start order has higher priority and is triggered by a low voltage signal across a bus bar, which can only be reset by an operator or four emergency signals: *overspeed*, *ground fault*, *under-voltage* and *emergency stop* pushbutton. The local start order is usually generated by a test procedure which is performed periodically to check the availability of GS, so it has a lower priority. Local start order can be reset by a number of local alarm signals as well as a local operator. The stop order is sent to the engine governor to stop the supply of fuel oil.

In this chapter we use the GS logic diagrams in a typical pressurized water reactor, and will verify the function of each logic diagram by the methodology presented in Chapter 3. The reduced verification technique in Section 3.3.2 will be widely used in this chapter to simplify the verifications.

4.2 GS Logic Diagram Verification

4.2.1 GS Logic Diagram Verification Sequences

The GS logic diagrams are attached in Appendix A [Fr96]. In each logic diagram, the following steps are taken:

1. Events and their acronyms are listed. Each event is assigned two unique numbers, which correspond to the *on* and *off* conditions. An uncontrollable event represents an input signal while a controllable event represents an output signal.

2. The monolithic supervisor SUP_{mon} specifying the function of the current logic diagram is generated by the methodology of Chapter 3 as shown in Fig. 4.2.

3. The design requirements of the current logic diagram, given by GS system engineers, are formalized by automata.

4. For each verification automaton model V_i we verify

$$L_m(SUP_{mon}) \subseteq L_m(V_i)$$

which means that the current logic diagram satisfies each of its requirements.

5. In some cases, the reduced verification technique may be applied and we will give a single state condition automaton C_i and a simplified verification automaton V_i' to verify the requirement. The scope of applicability of the reduced verification technique will be discussed in Section 4.2.2.

Note: TTCT will be used throughout these steps.

In order to make the above steps clearer, two flow charts are developed to show the detailed process. Fig. 4.2 specifies the exact operations of steps 1 and 2. The inputs of these operations are: logic diagram, basic logic plant models, buffer models, and basic logic gates specifications. The output is a monolithic supervisor representing the function of the current logic diagram. We will also verify that the monolithic supervisor is isomorphic to the combination of modular supervisors, a fact that has been proven in Chapter 3.

Fig. 4.3 specifies the exact operations of steps 3, 4 and 5. The output of Fig. 4.2, the monolithic supervisor SUP_{mon}, is actually one of the two inputs in this flow chart. The only remaining input is the listed verbal design requirements of the logic diagram. These requirements could be given by GS system engineers who may not know DES or automata theory at all. How to interpret these requirements and convert them to automata is difficult and error-prone. In order to overcome this difficultly, we suggest that these requirements are described by some standard forms:

- If XXX then YYY
- Once XXX then YYY
- When XXX then YYY

XXX could be an input condition given by certain verbal logic expressions, such as A *and* $\overline{B}$, $\overline{A}$ *or* B. YYY usually is the state of output signal. These forms can be easily expressed by boolean expressions, and consequently can be converted to automata without much difficulty.

We can see that most of the operations in Fig. 4.2 and Fig. 4.3 can be taken into two parallel lines. One line is to obtain the monolithic supervisor and the other line is to convert the design requirements to specification automata. The two final steps in Fig. 4.3 integrates

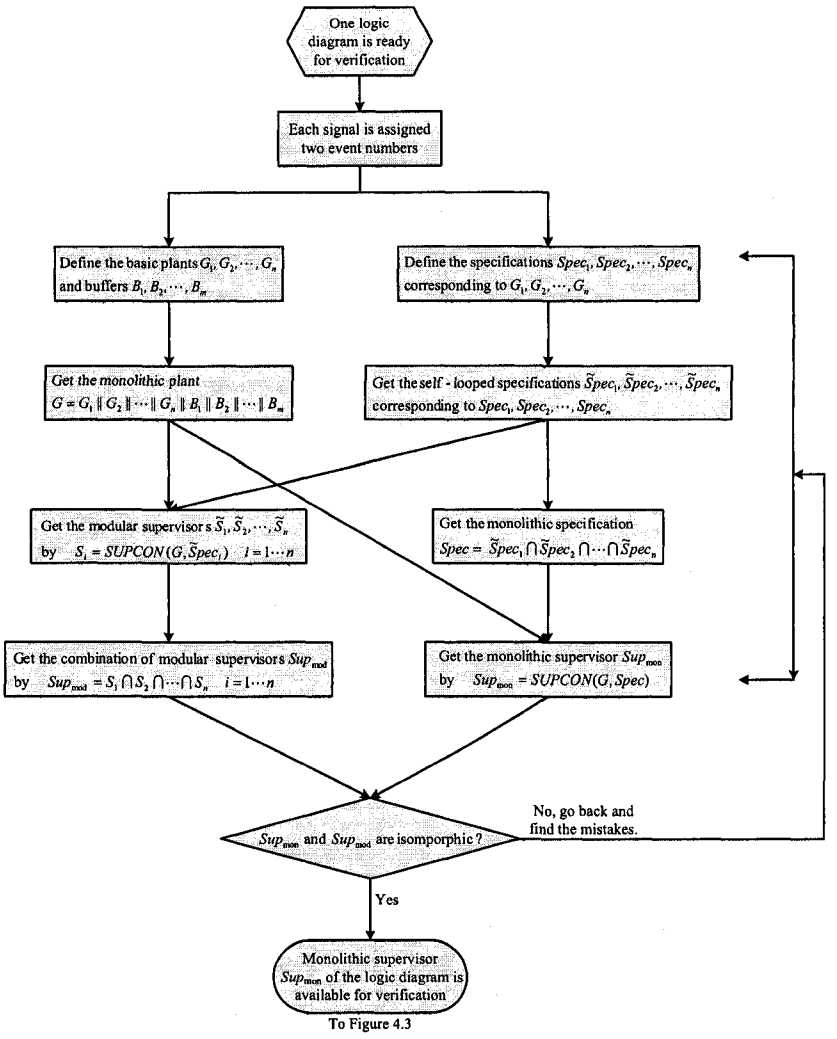

Figure 4.2: Monolithic supervisor generation process

these two lines together and delivers the result of verification: whether the logic diagram satisfies the design requirements or not.

Once the logic diagram cannot satisfy the design requirement, we can know the illegal behavior with the help of TTCT procedures and thus, can judge the possible following reasons.

1. The automaton model is wrong and is not accordance with the logic diagram or design requirement. The model should be established again.

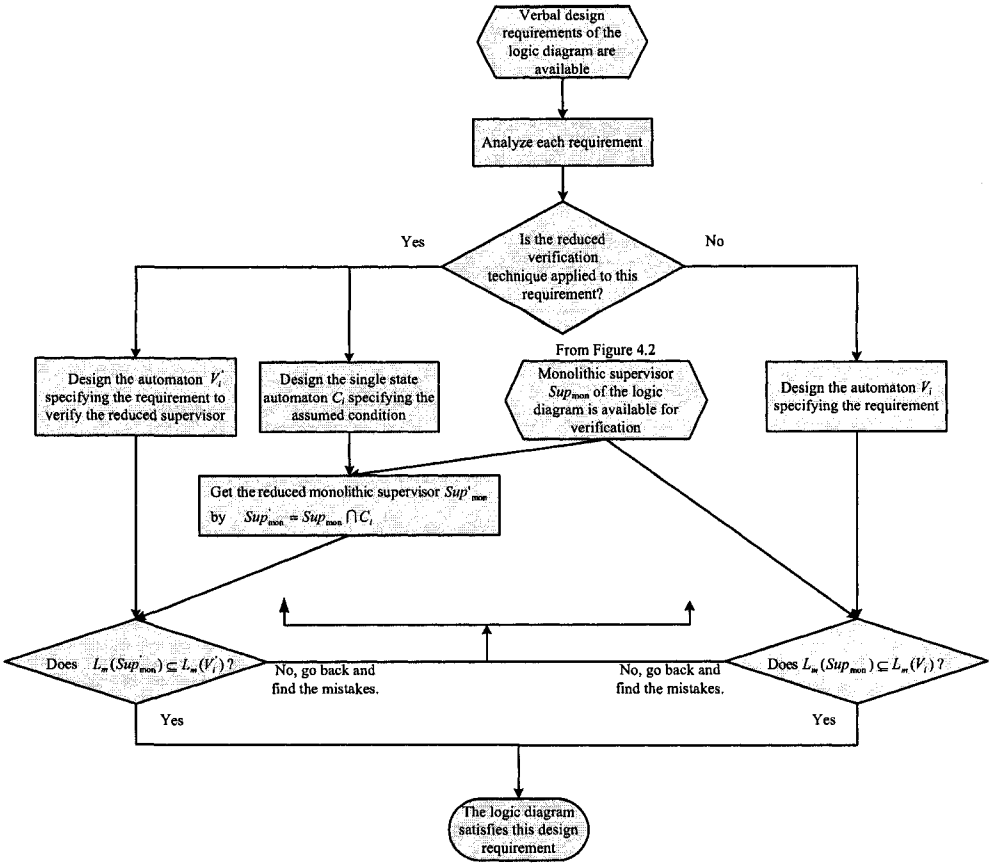

Figure 4.3: Logic Diagram verification process

2. The logic diagram has problem and needs to be modified.

3. The design requirement is not right and needs to be given again.

Since the automata models are designed according to the verbal design requirements, the completeness of verification will mostly depend on the completeness of such verbal design requirements.

4.2.2 Applicable Scope of Reduced Verification Technique

As we mentioned earlier, the reduced verification technique is an efficient tool and will be frequently utilized in this chapter. Hence, a natural question to ask is under what kind of circumstances it should be applied. We believe this should be an engineering question

rather than a theoretical one, which implies that the answer could be given by different engineering scenarios instead of formal justification.

We know that the reduced monolithic supervisor is the composition of a monolithic supervisor and an assumed condition with a single state, so the reduced supervisor is just the monolithic supervisor with all of the events not in the assumed condition removed. The critical questions are:

- How can we know that there should be an assumed condition for a specification?
- How can we make sure that the removed portion of the monolithic supervisor can still satisfy the design requirements?

After carefully looking into the logic diagram, we list four scenarios to apply the reduced technique.

1. The removed portion is already covered by other design requirements. The assumed condition should be defined based on the analysis of all design requirements.
2. It's a symmetrical structure, so the removed portion can be verified in an identical manner. The assumed condition can be defined based on the symmetrical structure of the design requirement and the logic diagram.
3. The removed portion is not restricted by the design requirement. The assumed condition should be defined according to the current design requirement.
4. The assumed condition is a unique situation, which depends on the result of analyzing all design requirements.

In the following section, it will be pointed out which scenario is applied when the reduced technique is utilized.

4.2.3 The Complete GS Logic Diagram Verification

Now, we will start analyzing the logic diagrams page by page.

1. **Logic diagram 1.** [1]

[1] Please refer to the corresponding logic diagram in Appendix A.

The design requirements are given below.

(a) If *fault in remote and local control bistable memory set (FRS)* signal is on, *GS starting remote control reset (RSR)* signal should be on.

(b) If *remote start order (RSO)* is off and *stop authorization (SA)* pushbutton is pressed, *GS starting remote control reset (RSR)* signal should be on.

(c) Once *GS starting remote control reset (RSR)* signal is triggered by the situation described in (b), it can only be reset by one of the following two signals:

- *Remote start order (RSO)* is on.
- *GS starting order bistable memory set (SOS)* signal is off and *stop authorization (SA)* pushbutton is not pressed.

Table. 4.1 lists the events in this logic diagram.

Table 4.1: Events table of logic diagram 1.

Acronym	Description	On	Off
SA	Stop authorization	2	4
SOS	GS starting order bistable memory set	12	14
FRS	Fault in remote and local control bistable memory set	22	24
RSO	Remote start order	32	34
RSR	GS Starting remote control reset	41	43

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 381 states and 1460 transitions.

The formal automata models specifying the above requirements are designed as follows:

- Requirement (a) is very straightforward and can be expressed in logic as $RSR = FRS$. Fig. 4.4 shows the automaton model expressing this requirement.
- Requirement (b) can be expressed as $RSR = \overline{RSO} \wedge SA$. Fig. 4.5 shows the automaton model in accordance with this requirement.

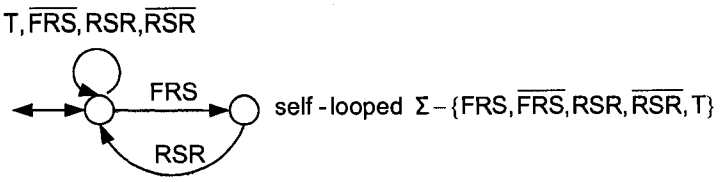

Figure 4.4: Specification 1 of logic diagram 1

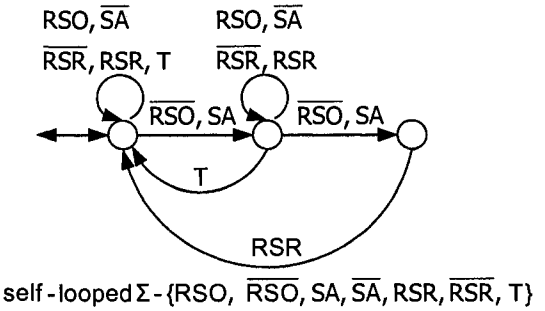

Figure 4.5: Specification 2 of logic diagram 1

- Requirement (c) can be expressed as $\overline{RSR} = RSO \vee (\overline{SA} \wedge \overline{SOS})$. Observe that by requirement (a), RSR can also be set by FRS. Fig. 4.6 shows the condition automaton model, which means that FRS signal is removed to make sure that RSR is triggered by the mechanism described in (b) only. Fig. 4.7 shows the automaton model based on this assumed condition.

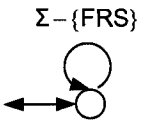

Figure 4.6: Assumed condition for specification 3 of logic diagram 1

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design

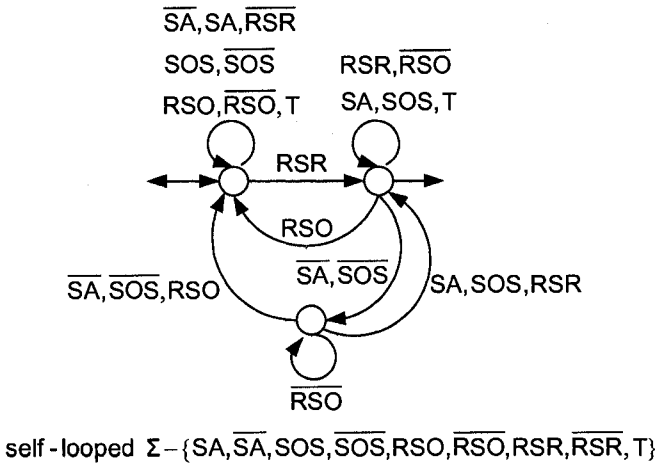

Figure 4.7: Specification 3 of logic diagram 1

requirements.

Please notice that the reduced verification technique has been employed in verification of design requirement (c).

2. **Logic diagram 2.**

   The design requirements are given below.

   (a) If *remote start order (RSO)* is on and *fault in remote and local control bistable memory set (FRS)* signal is off and *GS starting remote control reset (SRR)* signal is off, *GS starting remote control bistable memory set (SRS)* signal should be on and maintained until *GS starting remote control reset (SRR)* signal is on.

   (b) While *GS starting remote control reset (SRR)* signal is on, *GS starting remote control bistable memory set (SRS)* signal should always be off.

   Table. 4.2 is the events list in this logic diagram.

   By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 32 states and 73 transitions.

   The formal automata models specifying the above requirements are designed as follows:

Table 4.2: Events table of logic diagram 2.

Acronym	Description	On	Off
FRS	Fault in remote and local control bistable memory set	22	24
RSO	Remote start order	32	34
SRR	GS starting remote control reset	82	84
SRS	GS starting remote control bistable memory set	61	63

- Requirement (a) can be expressed as $SRS = \overline{FRS} \wedge RSO \wedge \overline{SRR}$. Since SRR on signal will be verified in requirement (b), we can design the condition automaton model in Fig. 4.8 to *combine* with the monolithic supervisor and get a reduced supervisor, which means that SRR is off. In this situation, the design requirement can be simplified as $SRS_{new} = SRS_{old} \vee (\overline{FRS} \wedge RSO)$. Here SRS_{old} is the current state of SRS and SRS_{new} is the state of next cycle. Fig. 4.9 shows the automaton model in accordance with this requirement. We can see that only output SRS is permitted in the last state.

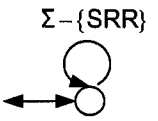

Figure 4.8: Assumed condition for specification 1 of logic diagram 2

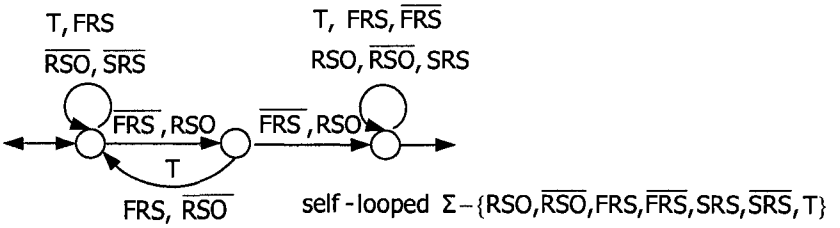

Figure 4.9: Specification 1 of logic diagram 2

- Requirement (b) can be expressed as $\overline{SRS} = SRR$. Fig. 4.10 shows the desired

automaton model.

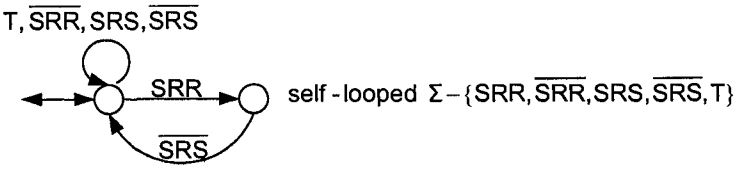

Figure 4.10: Specification 2 of logic diagram 2

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design requirements.

3. **Logic diagram 3.**

   The design requirements are given below.

   (a) If *fault in remote and local control bistable memory set (FRS)* signal is on, *GS starting order reset (SOR)* signal should be on.

   (b) If *GS starting remote control bistable memory set (SRS)* signal is off, either *fault in local control monostable memory set (FLS)* signal or *local stop (LS)* pushbutton should make *GS starting order reset (SOR)* signal on.

   (c) If *GS starting remote control bistable memory set (SRS)* signal is on and *fault in remote and local control bistable memory set (FRS)* signal is off, *GS starting order reset (SOR)* signal should always be off.

   Table. 4.3 is the events list in this logic diagram.

   By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 92 states and 277 transitions.

   The formal automata models specifying the above requirements are designed as follows:

Table 4.3: Events table of logic diagram 3.

Acronym	Description	On	Off
FRS	Fault in remote and local control bistable memory set	22	24
FLS	Fault in local control monostable memory set	102	104
LS	Local stop	112	114
SRS	GS starting remote control bistable memory set	122	124
SOR	GS starting order reset	91	93

- Requirement (a) can be expressed as $SOR = FRS$. Fig. 4.11 shows the automaton model in accordance with this requirement.

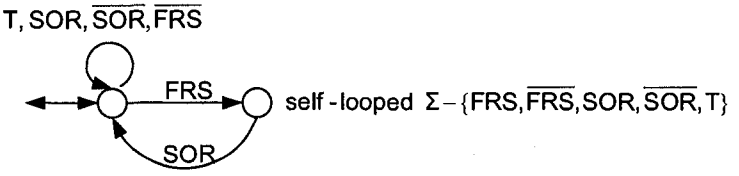

Figure 4.11: Specification 1 of logic diagram 3

- Requirement (b) can be expressed as $SOR = \overline{SRS} \wedge (FLS \vee LS)$. Fig. 4.12 shows the condition automaton model used to get a reduced supervisor, which means that SRS is off, since SRS on signal has been covered by requirements (a) and (c). In this situation, the design requirement can be simplified as $SOR = FLS \vee LS$. Fig. 4.13 shows the desired automaton model.

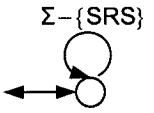

Figure 4.12: Assumed condition for specification 2 of logic diagram 3

- Requirement (c) can be expressed as $\overline{SOR} = SRS \wedge \overline{FRS}$. Fig. 4.14 shows the

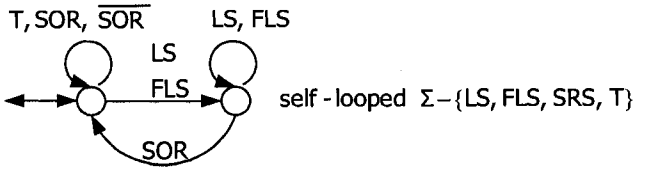

Figure 4.13: Specification 2 of logic diagram 3

desired automaton model.

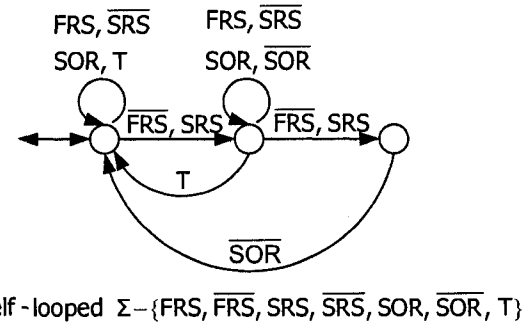

Figure 4.14: Specification 3 of logic diagram 3

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design requirements.

4. **Logic diagram 4.**

   The design requirements are given below.

   (a) *GS starting order reset (SOR)* on signal should always reset *GS starting order bistable memory set (SOS)* signal.

   (b) If *GS starting order reset (SOR)* signal is off, then *GS starting remote control bistable memory set (SRS)* signal should make *GS starting order bistable memory set (SOS)* signal on.

(c) If *GS starting order reset (SOR)* signal is off and *local start (LT)* pushbutton is pressed and *fault in local control monostable set (FLS)* signal is off and *local stop (LS)* pushbutton is not pressed and *GS starting remote control bistable memory set (SRS) signal is off*, then *GS starting order bistable memory set (SOS)* signal should be on.

(d) If *GS starting order bistable memory set (SRS)* signal is on, then it should be maintained until *GS starting order reset (SOR)* signal is on.

Table. 4.4 is the events list in this logic diagram.

Table 4.4: Events table of logic diagram 4.

Acronym	Description	On	Off
FLS	Fault in local control monostable memory set	102	104
LS	Local stop	112	114
SRS	GS starting remote control bistable memory set	122	124
LT	Local start	152	154
SOR	GS starting order reset	182	184
SOS	GS starting order bistable memory set	131	133

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 271 states and 983 transitions.

The formal automata models specifying the above requirements are designed as follows:

- Requirement (a) can be expressed as $\overline{SOS} = SOR$. Fig. 4.15 shows the automaton model in accordance with this requirement.
- Requirement (b) can be expressed as $SOS = \overline{SOR} \wedge SRS$. Fig. 4.16 shows the condition automaton model used to get a reduced supervisor, which means that SOR signal is off, since SOR on signal has been dealt with in requirement (a). In this situation, the design requirement can be simplified as $SOS = SRS$.

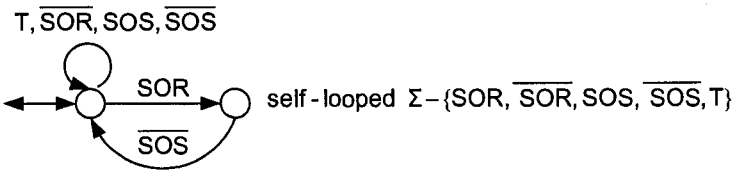

Figure 4.15: Specification 1 of logic diagram 4

Fig. 4.17 shows the desired automaton model.

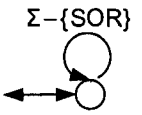

Figure 4.16: Assumed condition for specification 2 of logic diagram 4

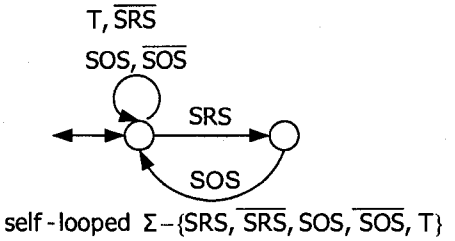

Figure 4.17: Specification 2 of logic diagram 4

- Requirement (c) can be expressed as $SOS = \overline{SOR} \wedge \overline{SRS} \wedge LT \wedge \overline{FLS} \wedge \overline{LS}$. Fig. 4.18 shows the condition automaton model used to get a reduced supervisor, which means that SOR, SRS, FLS, LS will be discarded. In this situation, the design requirement can be simplified as $SOS = LT$. Fig. 4.19 shows the desired automaton model. The assumed condition here is a mixed situation. Since SOR has been covered by requirement (a) and $\overline{SOR} \wedge SRS$ has been covered by requirement (b), we only need to verify the all combinations of LT, $\overline{FLS}$ and $\overline{LS}$. It is a symmetrical structure and the other two cases can be verified similarly as follows:

(a) SOR, SRS, $\overline{LT}$, LS will be discarded and the design requirement will be $SOS = \overline{FLS}$.

(b) SOR, SRS, $\overline{LT}$, FLS will be discarded and the design requirement will be $SOS = \overline{LS}$.

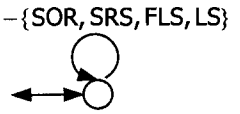

Figure 4.18: Assumed condition for specification 3 of logic diagram 4

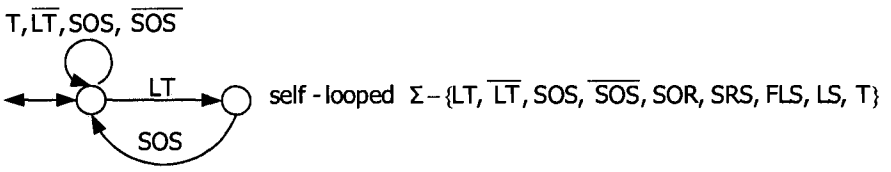

Figure 4.19: Specification 3 of logic diagram 4

- Requirement (d) can be expressed as

  If $SOS_{old} = 1$, then $SOS_{new} = \overline{SOR}$.

  Fig. 4.20 shows the automaton model in accordance with this requirement. We can notice that once SOS is set it will be maintained until SOR is on.

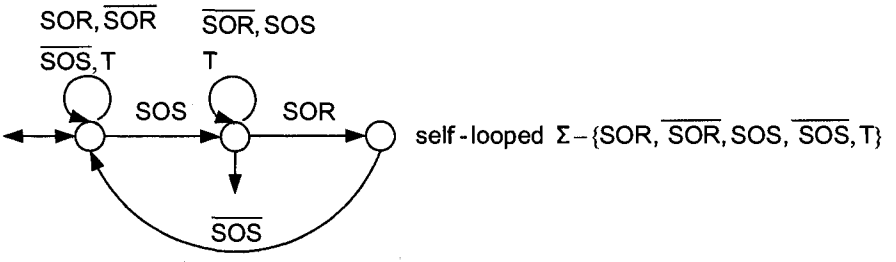

Figure 4.20: Specification 4 of logic diagram 4

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by

each of these automata, which means that the logic diagram satisfies the listed design requirements.

5. **Logic diagram 5.**

The design requirement is given below.

If *started lubrication oil pressure greater than 1.5 bar (LOP)* signal is off and *GS starting order bistable memory set (SOS)* signal is on, *start pulse (SP)* signal should be on. If *started lubrication oil pressure greater than 1.5 bar (LOP)* signal is on when *GS starting order bistable memory set (SOS)* signal is on, the *start pulse* should be off immediately, otherwise the *start pulse (SP)* should last at most 5 seconds.

Table. 4.5 is the events list in this logic diagram.

Table 4.5: Events table of logic diagram 5.

Acronym	Description	On	Off
LOP	Started lubrication oil pressure > 1.5 bar	202	204
SOS	GS starting order bistable memory set	12	14
SP	Start pulse to solenoid	151	153

There is a power-on delay in this logic diagram and it will be modeled by one cycle (T) delay, so we can only verify that the time duration is one cycle time (T) instead of 5 seconds.

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 56 states and 132 transitions.

The design requirement can be expressed as $SP = (\overline{LOP} \wedge SOS)\uparrow$, where $\uparrow$ means that the output SP will be set only one cycle time (T) even though LOP is off and SOS is on for more than 2 cycles. Fig. 4.21 shows the desired automaton model. We can see that SP is maintained only for one cycle time.

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by this

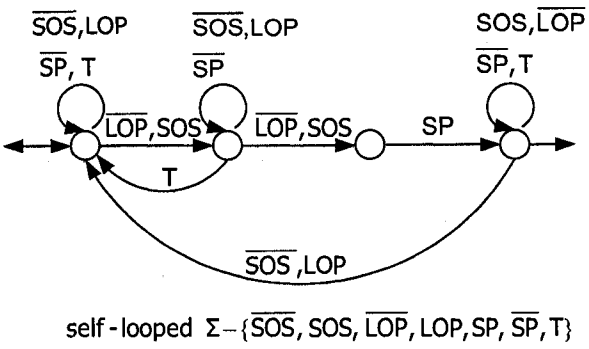

Figure 4.21: Specification 1 of logic diagram 5

automaton, which means that the logic diagram satisfies the design requirements.

6. **Logic diagram 6.**

   The design requirements are given below.

   (a) If *generator exciter contactor (ECC)* closed signal is off, then the real *under-voltage set (UVS)* signal should never be generated.

   (b) If *generator exciter contactor (ECC)* closed signal is on and any 2 out of 3 *under-voltage sensors (UV1, UV2, UV3)* send under-voltage signal, the real *under-voltage set (UVS)* signal should be generated.

   Note: The power-on delays in this logic diagram will not be modeled.

   Table. 4.6 is the events list in this logic diagram.

   By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 535 states and 1589 transitions.

   The formal automata models specifying the above requirements are designed as follows:

   - Requirement (a) can be expressed as $\overline{UVS} = \overline{ECC}$. Fig. 4.22 shows the automaton model in accordance with this requirement.
   - Requirement (b) can be expressed as:

Table 4.6: Events table of logic diagram 6.

Acronym	Description	On	Off
ECC	Generator exciter contactor closed	242	244
UV1	Under-voltage 1	252	254
UV2	Under-voltage 2	262	264
UV3	Under-voltage 3	272	274
UVS	Under-voltage set	211	213

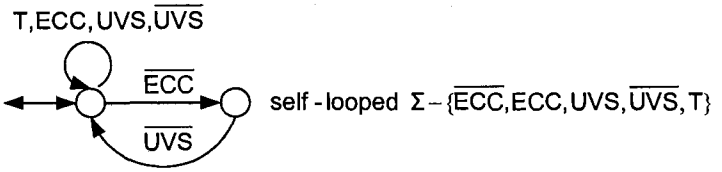

Figure 4.22: Specification 1 of logic diagram 6

$UVS = ECC \wedge ((UV1 \wedge UV2) \vee (UV1 \wedge UV3) \vee (UV2 \wedge UV3))$

Fig. 4.23 shows the automaton model used to get a reduced supervisor, which means that ECC signal is on. The ECC off signal has been verified by requirement (a). In this situation, the design requirement can be simplified as: $UVS = (UV1 \wedge UV2) \vee (UV1 \wedge UV3) \vee (UV2 \wedge UV3)$

Fig. 4.24 shows the desired automaton model.

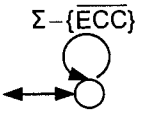

Figure 4.23: Assumed condition for specification 2 of logic diagram 6

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design

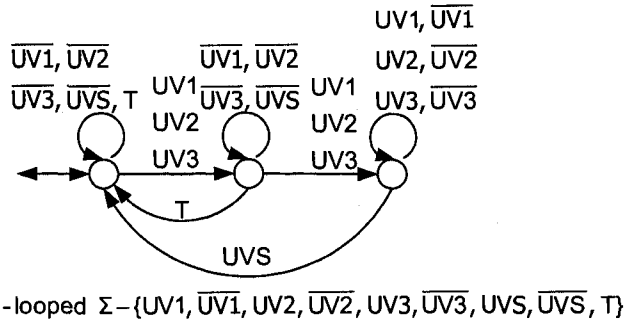

Figure 4.24: Specification 2 of logic diagram 6

requirements.

7. **Logic diagram 7.**

   The design requirements are given below.

   (a) Any of the following signals should make *fault in remote and local control bistable memory set (FRS)* signal on:

   - *Overspeed (OS)*
   - *Emergency stop (ES)*
   - *Under-voltage set (UVS)*
   - *Ground fault (GF)*

   (b) If *fault in remote and local control bistable memory set (FRS)* signal is on and none of the above signals exists, then it can be reset by *fault acknowledge (FA)* pushbutton.

   Table. 4.7 is the events list in this logic diagram.

   By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 268 states and 1043 transitions.

   The formal automata models specifying the above requirements are designed as follows:

   - Requirement (a) can be expressed as:

Table 4.7: Events table of logic diagram 7.

Acronym	Description	On	Off
OS	Overspeed RPM > 1725	332	334
ES	Emergency stop	342	344
UVS	Under-voltage set	352	354
GF	Ground fault	362	364
FA	Fault acknowledge	392	394
FRS	Fault in remote and local control bistable memory set	251	253

$FRS = OS \vee ES \vee UVS \vee GF$

Fig. 4.25 shows the automaton model in accordance with this requirement.

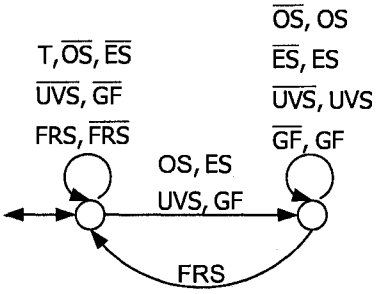

Figure 4.25: Specification 1 of logic diagram 7

- Requirement (b) can be expressed as:

  $\overline{FRS} = \overline{OS} \wedge \overline{ES} \wedge \overline{UVS} \wedge \overline{GF} \wedge FA$

  Due to its symmetric structure, we only verify one case. Fig. 4.26 shows the condition automaton model used to get a reduced supervisor, which means that ES, UVS, and GF are assumed to be off. In this situation, the design requirement can be simplified as:

  $\overline{FRS} = \overline{OS} \wedge FA$. Fig. 4.27 shows the desired automaton model.

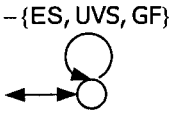

Figure 4.26: Assumed condition for specification 2 of logic diagram 7

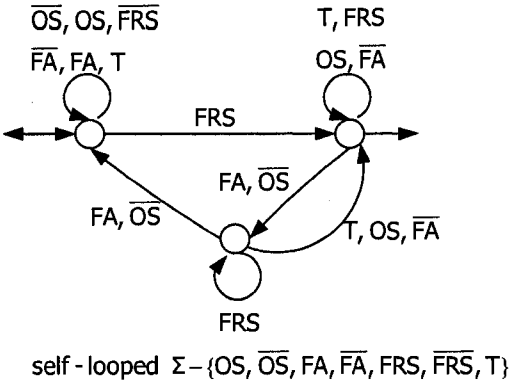

Figure 4.27: Specification 2 of logic diagram 7

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design requirements.

8. **Logic diagram 8.**

The design requirements are given below.

Any of the following signals should generate *fault in local control alarm (FLO)*:

- *Crankcase pressure high (CPH)*
- *Exhaust gas temperature high (ETH)*
- *Oil sump level low (OLL)*
- *Lubrication oil pressure low (OPL)*
- *Lubrication oil temperature high (OTH)*
- *Cooling water temperature high (WTH)*

- *Cooling water tank level low (WLL)*
- *Cooling water pressure low (WPL)*

Table. 4.8 is the events list in this logic diagram.

Table 4.8: Events table of logic diagram 8.

Acronym	Description	On	Off
CPH	Crankcase pressure high	412	414
ETH	Exhaust gas temperature high	422	424
OLL	Oil sump level low	432	434
OPL	Oil pressure low	442	444
OTH	Oil temperature high	452	454
WTH	Cooling water temperature high	462	464
WLL	Cooling water tank level low	472	474
WPL	Cooling water pressure low	482	484
FLO	Fault in local control alarm	321	323

By applying the operations of Fig. 4.2, we can obtain a supervisor [2] which has 13072 states and 78679 transitions.

The requirement can be expressed as:

$FLO = CPH \vee ETH \vee OLL \vee OPL \vee OTH \vee WTH \vee WLL \vee WPL$

Fig. 4.28 gives the automaton to realize this function.

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by this automaton, which means that the logic diagram satisfies the design requirements.

9. **Logic diagram 9.**

The design requirements are given below.

[2]The computing scale of monolithic supervisor is over the limit of TTCT, so the supervisor is generated by the combination of modular supervisors.

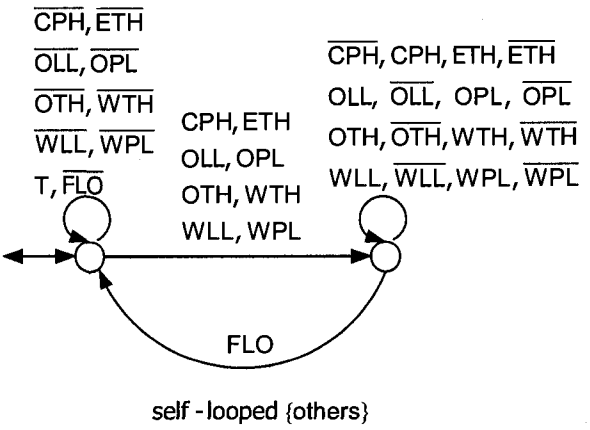

Figure 4.28: Specification 1 of logic diagram 8

(a) *GS starting remote control bistable memory set (SRS)* signal should always make *fault in local control monostable memory set (FLS)* signal off.

(b) If *GS starting remote control bistable memory set (SRS)* signal is off, then *fault in local control alarm (FLO)* signal should make *fault in local control monostable memory set (FLS)* signal on. While *FLS* is on only *fault in local control alarm (FLO)* off signal together with *fault acknowledge (FA)* pushbutton pressed signal can make it off.

Table. 4.9 is the events list in this logic diagram.

Table 4.9: Events table of logic diagram 9.

Acronym	Description	On	Off
FA	Fault acknowledge	392	394
FLO	Fault in local control alarm	552	554
SRS	GS starting remote control bistable memory set	122	124
FLS	Fault in local control monostable memory set	351	353

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 62 states and 135 transitions.

The formal automata models specifying the above requirements are designed as follows:

- Requirement (a) can be expressed as: $\overline{FLS} = SRS$. Fig. 4.29 shows the automaton model in accordance with this requirement.

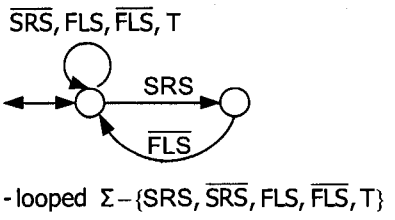

Figure 4.29: Specification 1 of logic diagram 9

- Requirement (b) can be shown by two expressions:

  $FLS = \overline{SRS} \wedge FLO$

  $\overline{FLS} = \overline{SRS} \wedge \overline{FLO} \wedge FA$

  Fig. 4.30 gives the condition automaton model to get the reduced supervisor when SRS is off. SRS on signal has been covered by requirement (a). In this case, the above two expressions can be simplified as:

  $FLS = FLO$

  $\overline{FLS} = \overline{FLO} \wedge FA$

  Fig. 4.31 gives the automaton to realize the function of the first expression and Fig. 4.32 gives the automaton to realize the function of the second one. Please notice that the priority of FLO over FA has been covered by requirement (a), since FLO on signal will always make FLS signal off as demonstrated in Fig. 4.31, which does not care the state of FA.

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design requirements.

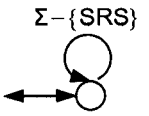

Figure 4.30: Assumed condition for specification 2 and 3 of logic diagram 9

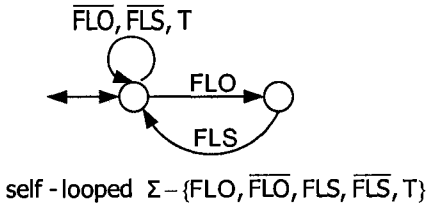

Figure 4.31: Specification 2 of logic diagram 9

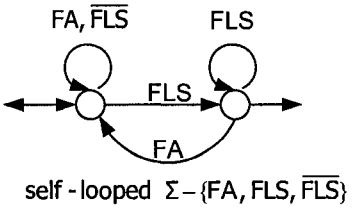

Figure 4.32: Specification 3 of logic diagram 9

10. **Logic diagram 10.**

The design requirements are given below.

(a) If *no start (NS)* signal is off or *GS starting order bistable memory set (SOS)* signal is off, then *no starting fault alarm (NSA)* should always be off.

(b) When *GS starting order bistable memory set (SOS)* signal is on for 10 seconds or more and *no start (NS)* signal is still on, *no starting fault alarm (NSA)* should be on.

Table. 4.10 is the events list in this logic diagram.

The 10 seconds power-on time delay will be modeled by one cycle time (T) power-on time delay, so requirement (b) will be changed to: When *GS starting order bistable*

Table 4.10: Events table of logic diagram 10.

Acronym	Description	On	Off
NS	No start	582	584
SOS	GS starting order bistable memory set	12	14
NSA	No starting fault alarm	361	363

memory set (SOS) signal is on for at least two cycles and *no start (NS)* signal is still on, *no starting fault alarm (NSA)* will be on.

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 42 states and 79 transitions.

The formal automata models specifying the above requirements are designed as follows:

- Requirement (a) can be expressed by: $\overline{NSA} = \overline{NS} \vee \overline{SOS}$. Fig. 4.33 shows the automaton model in accordance with this requirement.

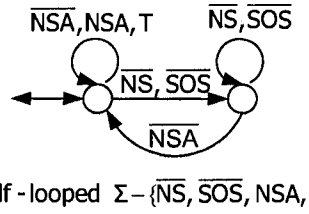

Figure 4.33: Specification 1 of logic diagram 10

- Requirement (b) can be shown by the following expression:

  $NSA = NS \wedge (SOS \nearrow)$

  $\nearrow$ denotes power-on time delay. Fig. 4.34 gives the condition automaton model to get the reduced supervisor under the assumption that NS is on. NS off signal has been dealt with by requirement (a). In this case, the above two expressions can be simplified as:

  $NSA = SOS \nearrow$

Fig. 4.35 gives the automaton to realize the function. We can see that NSA is generated after SOS happens twice in succession.

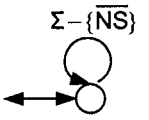

Figure 4.34: Assumed condition for specification 2 of logic diagram 10

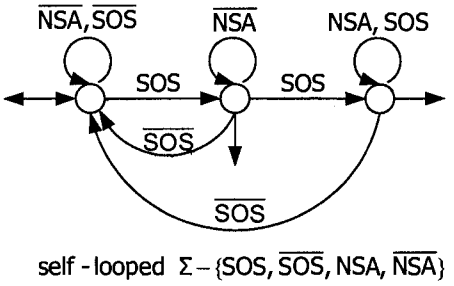

Figure 4.35: Specification 2 of logic diagram 10

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design requirements.

11. **Logic diagram 11.**

The design requirements are given below.

(a) When *air tank pressure low (APL)* signal is on, *air compressor starting order* should be issued.

(b) When *air tank pressure low (APL)* signal is on for more than 8 seconds and *air compressor lubrication oil pressure low (LOL)* signal is still on, *air compressor stopping order (ACS)* should be issued.

(c) When *air tank pressure high (APH)* signal is on, *air compressor stopping order (ACS)* should be issued.

Table. 4.11 is the events list in this logic diagram.

Table 4.11: Events table of logic diagram 11.

Acronym	Description	On	Off
APH	Air tank pressure high	592	594
APL	Air tank pressure low	602	604
LOL	Compressor lubrication oil pressure low	612	614
ACS	Air compressor stopping order	381	383

Because *air tank pressure low* signal directly drives *air compressor starting order* signal and no logic gate is involved, requirement (a) will not be verified. And for simplicity, the 8 seconds power-on time delay in requirement (b) will be skipped as well.

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 30 states and 71 transitions.

The formal automata models specifying requirements (b) and (c) are designed as follows:

- Requirement (b) can be expressed as:

  $ACS = APL \vee LOL$

  Fig. 4.36 shows the automaton model in accordance with this requirement.

- Requirement (c) can be shown by the expression: $ACS = APH$. Fig. 4.37 gives the automaton needed to realize the function.

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design requirements (b) and (c).

12. **Logic diagram 12.**

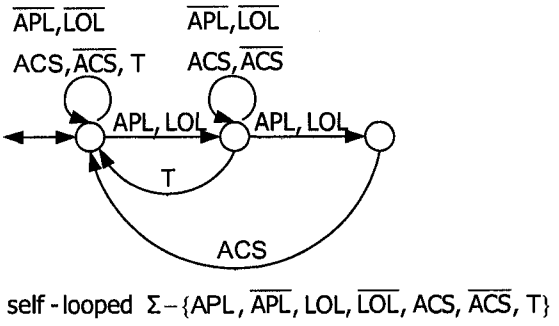

Figure 4.36: Specification 1 of logic diagram 11

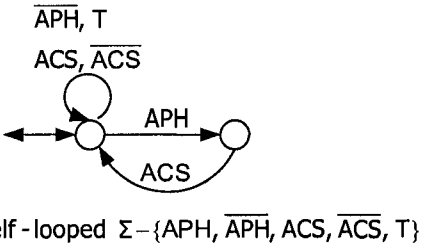

Figure 4.37: Specification 2 of logic diagram 11

The design requirements are given below.

(a) When *GS starting order bistable memory set (SOS)* signal is on, *fuel oil transfer pump starting order* should be issued.

(b) When *GS starting order bistable memory set (SOS)* signal is off or *fuel oil transfer pump stop pushbutton (FPP)* is pressed, *fuel oil transfer pump stopping order (FPS)* should be issued.

Table. 4.12 is the events list in this logic diagram.

Because *GS starting order bistable memory set* signal directly drives *fuel oil transfer pump starting order* signal and no logic gate is involved, requirement (a) will not be verified.

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 8 states and 15 transitions.

Table 4.12: Events table of logic diagram 12.

Acronym	Description	On	Off
FPP	Fuel oil transfer pump stop pushbutton	632	634
SOS	GS starting order bistable memory set	12	14
FPS	Fuel oil transfer pump stopping order	391	393

Requirement (b) can be expressed as:

$FPS = FPP \vee \overline{SOS}$

Fig. 4.38 gives the automaton needed to realize the function.

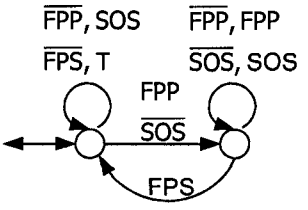

Figure 4.38: Specification 1 of logic diagram 12

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by this automaton, which means that the logic diagram satisfies requirement (b).

13. **Logic diagram 13.**

The verification of this logic diagram is skipped due to its simplicity.

14. **Logic diagram 14.**

The design requirements are given below.

(a) When *started lubrication oil pressure greater than 1.5 bar* signal is off for more than 120 seconds, *pre-lubricating pump and water circulating pump starting order* should be issued. When *started lubrication oil pressure greater than 1.5 bar* is

on, *pre-lubricating pump and water circulating pump stopping order* should be issued.

(b) When both *preheating water temperature lower than 80 degrees (PTL)* signal and *water circulating pump started (WPS)* signal are on, *electrical water heater starting order (WHS)* should be issued.

(c) When *preheating water temperature lower than 80 degrees (PTL)* signal is off or *water circulating pump* stopped (WPP) signal is on, *electrical water heater stopping order (WHP)* should be issued.

Table. 4.13 is the events list in this logic diagram.

Table 4.13: Events table of logic diagram 14.

Acronym	Description	On	Off
WPP	Water circulating pump stopped	642	644
WPS	Water circulating pump started	652	654
PTL	Preheating temperature low < 80	662	664
WHP	Electrical water heater stopping order	401	403
WHS	Electrical water heater starting order	411	413

Requirement (a) has no logic gate involved, so it will not be verified.

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 30 states and 77 transitions.

The formal automata models specifying the requirements (b) and (c) are designed as follows:

- Requirement (b) can be expressed as:

  $WHS = WPS \wedge PTL$

  Fig. 4.39 shows the automaton model specifying this requirement.

- Requirement (c) can be expressed as:

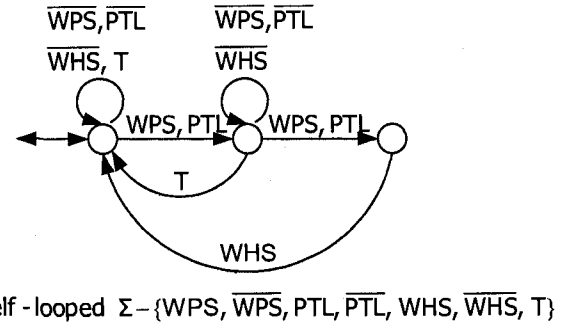

Figure 4.39: Specification 1 of logic diagram 14

$$WHP = WPP \vee \overline{PTL}$$

Fig. 4.40 gives the automaton needed to realize the function.

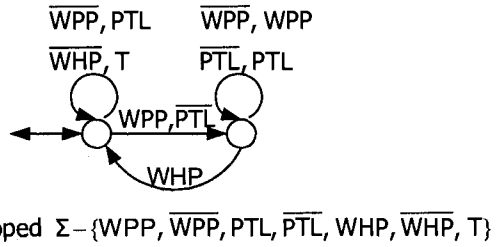

Figure 4.40: Specification 2 of logic diagram 14

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design requirements.

15. **Logic diagram 15.**

The design requirements are given below.

(a) When *GS starting order bistable memory set (SOS)* signal is on and *air cooler temperature high (ATH)* signal is on, *air cooler fan starting order (AFS)* should be issued.

(b) When *GS starting order bistable memory set (SOS)* signal is off, *air cooler fan stopping order* should be issued.

Table. 4.14 is the events list in this logic diagram.

Table 4.14: Events table of logic diagram 15.

Acronym	Description	On	Off
SOS	GS starting order bistable memory set	12	14
ATH	Air cooler temperature high	672	674
AFS	Air cooler fan starting order	421	423

Requirement (b) has no logic gate involved, so it will not be verified.

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 8 states and 15 transitions.

Requirement (a) can be expressed as:

$AFS = SOS \wedge ATH$

Fig. 4.41 gives the automaton needed to realize the function.

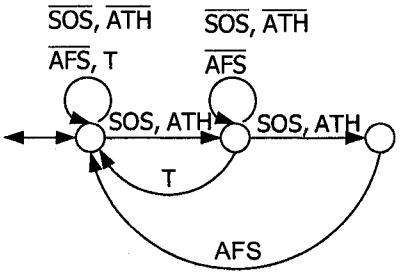

Figure 4.41: Specification 1 of logic diagram 15

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by this automaton, which means that the logic diagram satisfies requirement (a).

16. **Logic diagram 16.**

The design requirements are given below.

(a) When *GS starting order bistable memory set (SOS)* signal is on, *diesel engine building exhaust fan starting order* should be issued.

(b) When *exhaust fan stop pushbutton (EFS)* is pressed or *GS starting order bistable memory set (SOS)* signal is off, *diesel engine building exhaust fan stopping order (EFP)* should be issued.

Table. 4.15 is the events list in this logic diagram.

Table 4.15: Events table of logic diagram 16.

Acronym	Description	On	Off
SOS	GS Starting order bistable memory set	12	14
EFS	Exhaust fan stop pushbutton	682	684
EFP	Exhaust fan stopping order	431	433

Requirement (a) has no logic gate involved, so it will not be verified.

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 8 states and 15 transitions.

Requirement (b) can be expressed as:

$EFP = EFS \vee \overline{SOS}$

Fig. 4.42 gives the automaton needed to realize the function.

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by this automaton, which means that the logic diagram satisfies requirement (b).

17. **Logic diagram 17.**

The verification of this logic diagram is skipped due to its simplicity.

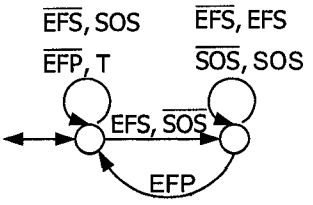

Figure 4.42: Specification 1 of logic diagram 16

18. **Logic diagram 18.**

The design requirements are given below.

(a) When *lubrication oil tank level high (OLH)* signal is off and *oil transfer pump pushbutton (OPO)* is pressed, *oil transfer pump starting order (OPS)* should be issued.

(b) When *lubrication oil tank level high (OLH)* signal is on or *oil transfer pump pushbutton (OPO)* is released, *oil transfer pump stopping order (OPP)* should be issued.

Table. 4.16 is the events list in this logic diagram.

Table 4.16: Events table of logic diagram 18.

Acronym	Description	On	Off
OLH	Lubrication oil tank level high	692	694
OPO	Oil transfer pump pushbutton	702	704
OPP	Oil transfer pump stopping order	441	443
OPS	Oil transfer pump starting order	451	453

By applying the operations of Fig. 4.2, we can obtain a monolithic supervisor, which has 12 states and 21 transitions.

The formal automata models specifying these requirements are designed as follows:

- Requirement (a) can be expressed as:

  $OPS = \overline{OLH} \wedge OPO$

  Fig. 4.43 shows the automaton model specifying this requirement.

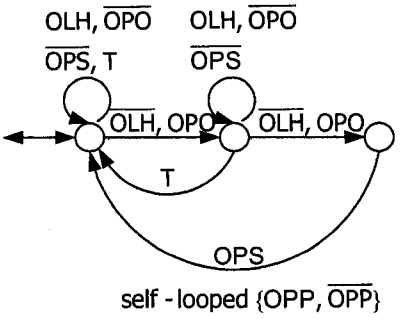

Figure 4.43: Specification 1 of logic diagram 18

- Requirement (b) can be shown by the expression:

  $OPP = OLH \vee \overline{OPO}$

  Fig. 4.44 gives the automaton needed to realize the function.

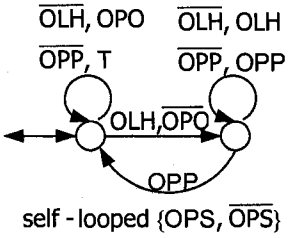

Figure 4.44: Specification 2 of logic diagram 18

Through TTCT, we can verify that the language generated by the system under the supervision of the monolithic supervisor is a subset of the language generated by each of these automata, which means that the logic diagram satisfies the listed design requirements.

4.3 Conclusion

In this chapter, first we described the composition and basic operation principle of an emergency diesel generator set. Then we presented the detailed logic diagram verification process based on the methodology developed in Chapter 3. The applicable scope of reduced verification technique is discussed, too. The final part of this chapter is to verify GS logic diagrams page by page. The purpose is to show that logic function verification by modular supervisory control of DES is feasible and has potential industrial application value. One thing we need to point out again is that since the verification automata are designed based on verbal design requirements, the completeness of the logic verification depends not only on how to convert the verbal requirements to automata, but also on how to give and interpret these verbal requirements. Our efforts in this chapter is to ensure that each informal requirement is correctly expressed by an automaton.

Chapter 5

Conclusion and Future Research

5.1 Conclusion

This dissertation has proposed a feasible way to verify the industrial logic diagrams by modular supervisory control of discrete-event system.

Our approach is based on a single logic diagram. In the first step we establish the basic plants for basic logic gates. Even though we have used TTCT to establish these models, we just partially adopt the timed discrete-event models to make the forcible events preempt the time cycle (T) when necessary and thus, satisfy the function requirements of flip-flops. The duration of the time delay is not modeled.

Next we create the specification model of each logic gate according to its truth table. Using TTCT procedures, we verify their controllability and nonblocking. To obtain a centralized supervisor for a logic diagram, it is necessary to introduce the forward and feedback buffers to form a monolithic plant. We also treat the power-on and power-off time delays as buffers to simplify our models. After we convert the verbal design requirements to automata, the verification can be easily conducted by TTCT. If the logic diagram cannot satisfy certain requirement, with the help of TTCT procedures, we can quickly capture the illegal behaviors and then find out whether the problems originate in the logic diagram, design requirement or the implementation process.

Next, a formal proof of controllability and a semi-formal proof of nonblocking are presented. These are important and necessary steps if we would like to apply this method

to a real industrial application.

Finally, an industrial example, an emergency diesel generater startup controller is taken to fully demonstrate our methodology. The reduced verification technique is frequently employed to reduce the complexity of specification model.

One critical work in Chapter 4 is how to interpret each informal verbal requirement and convert it to a specification automaton. All of the verification steps can be automatically performed by computer except the informal verbal requirements interpretation and conversion, which need manual effort.

We believe that our method has its unique advantage over other methods. Nevertheless, it surely has some limitations pointing out possible directions for future research. We have verified a single logic diagram; verification of the high-level requirements for the whole system is still an open issue. A complete timed model should also be introduced to verify the time function.

5.2 Future Research

Our work is the initial step. Some of potential future challenges are described in the following :

1. Based on our present work, by incorporating the modular and hierarchy approaches, the centralized controller for the whole system could possibly be built to verify multiple logic diagram or high-level specifications. The interconnections between the logic diagrams can be similarly modeled as buffers; we can also consider that there is a communication mechanism to coordinate the information exchange between these logic diagrams.

2. Many control systems have time delays and other timing functions. To include the real-time feature, we must extend our models to timed models. The suggested buffer models for the time delays in our work are too simple to verify their actual time values.

3. We can develop a software platform which is embedded with the TTCT procedures and our methodology. Then this platform could be used to verify the logic diagrams by the design engineer.

4. Our methodology could also be extended to test real industrial controller programs, such as those of PLCs. Since we adopted the program execution cycle (T) in each model, the final centralized controller model already reflects the execution sequence of a real industrial controller.

5. Creating formal requirements from informal verbal requirements still needs manual effort in our thesis. However, this research field is quite active currently. By combining these research results with our method, the new method could possibly become a completely automatic solution.

Bibliography

[Amo95] E. G. Amoroso, "Creating formal specifications from informal requirements documents," *SIGSOFT Software Engineering Notes*, vol. 20, pp. 67-71, 01. 1995.

[CC04] S. W. Cheon, K. H. Cha and K. C. Kwon, "The Software Verification and Validation Tasks for a Safety Critical System in Nuclear Power Plants," *International Journal of Safety*, vol. 3, pp. 38-46, 2004.

[DC00] O. De Smet, S. Couffin, "Safe programming of PLC using formal verification methods," *Proc. 4th Int. PLCopen Conf. on Industrial Control Programming (ICP'2000)*, Utrecht, the Netherlands, pp. 73-74. 2000.

[FH98] M. Fabian and A. Hellgren, "PLC-based implementation of supervisory control for discrete events ystems," *Decision and Control, 1998. Proceedings of the 37th IEEE Conference on*, vol. 3, 1998.

[FL00] G. Frey and L. Litz, "Formal methods in PLC programming," *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 4, 2000.

[Fr96] "Emergency Diesel Generator System Design Manual," *Framatome, Paris, France*, 1996.

[FW88] F. Lin and W. M. Wonham, "Decentralized supervisory control of discrete-event systems," *Information Sciences*, vol. 44, pp. 199-224, 4. 1988.

[GD06] V. Gourcuff, O. De Smet and J. M. Faure, "Efficient representation for formal verification of PLC programs," *Discrete Event Systems, 2006 8th International Workshop on*, pp. 182-187, 2006.

[Hel01] A. Hellgren, "Modelling and Implementation Aspects of Supervisory Control," *Licentiate Thesis, Chlmers University of Technology, Sweden*, 2001.

[HL02] A. Hellgren, B. Lennartson and M. Fabian, "Modelling and PLC-based implementation of modular supervisory control," *Discrete Event Systems, 2002.Proceedings. Sixth International Workshop on*, pp. 371-376, 2002.

[Hu03] R. Huuck, "Software Verification for Programmable Logic Controllers," *Ph.D. dissertation, Kiel University, Germany*, 2003.

[JB93] W. Johnson, K. Benner and D. Harris, "Developing formal specifications from informal requirements," *Expert, IEEE [See also IEEE Intelligent Systems and their Applications]*, vol. 8, pp. 82-90, 1993.

[LD02] J. Liu and H. Darabi, "Ladder logic implementation of Ramadge-Wonham supervisory controller," *Discrete Event Systems, 2002.Proceedings.Sixth International Workshop on*, pp. 383-389, 2002.

[Led96] R. Leduc, "PLC Implementation of a DES Supervisor for a Manufacturing Testbed: An Implementation Perspective," *M.A.Sc, Toronto University, Canada*, 1996.

[LW95] R. Leduc and W. Wonham, "Discrete event systems modeling and control of a manufacturing testbed," *Electrical and Computer Engineering, 1995.Canadian Conference on*, vol. 2, 1995.

[LY05] K. Loeis, M. B. Younis and G. Frey, "Application of symbolic and bounded model checking to the verification of logic control systems," in *10th IEEE International Conference on Emerging Technologies and Factory Automation*, PP. 4, 2005.

[Mon06] M. Moniruzzaman, "Implementing supervisory control maps with PLC," *M.A.Sc, Concordia University, Canada*, 2006.

[Moo94] I. Moon, "Modeling programmable logic controllers for logic verification," *Control Systems Magazine, IEEE*, vol. 14, pp. 53-59, 1994.

[RK98] M. Rausch and B. Krogh, "Formal verification of PLC programs," *American Control Conference, 1998.Proceedings of the 1998*, vol. 1, 1998.

[RW87] P.J.Ramadge and W.M.Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Control and Optimization*, 25(1):206-230,1987.

[Won06] W.M.Wonham, "Supervisory control of discrete-event systems," http://www.control.toronto.edu/people/profs/wonham/wonham.html, 2006.

[WR87] W.M.Wonham and P.J.Ramadge, "On the supremal controllable sublanguage of a given language," *SIAM J. Control and Optimization*, 25(3):637-659, May 1987.

[WR88] W. M. Wonham and P. J. Ramadge, "Modular supervisory control of discrete-event systems," *Mathematics of Control, Signals, and Systems (MCSS)*, vol. 1, pp. 13-30, 1988.

[YK05] J. Yoo, T. Kim, S. Cha, J. S. Lee and H. Seong Son, "A formal software requirements specification method for digital nuclear plant protection systems," *The Journal of Systems & Software*, vol. 74, pp. 73-83, 2005.

[ZV98] M. C. Zhou and K. Venkatesh, *Modeling, Simulation, and Control of Flexible Manufacturing Systems: A Petri Net Approach.* Singapore: World Scientific, 1999.

Appendix A

Diesel Generator Startup Controller Logic Diagrams

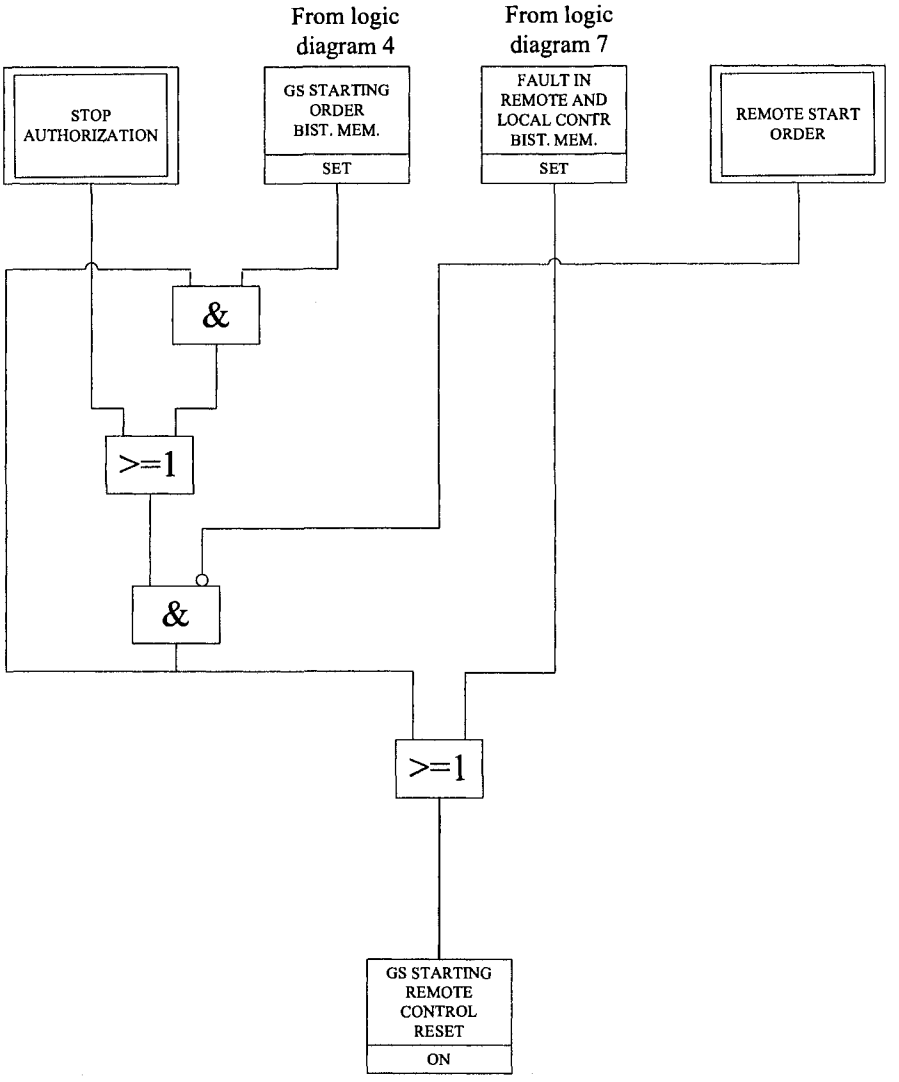

Figure A.1: Logic diagram 1

1

[1]The double-framed block is a physical input signal. The bold-framed block is a physical output signal. The other type blocks are the intermediate logic boolean signals.

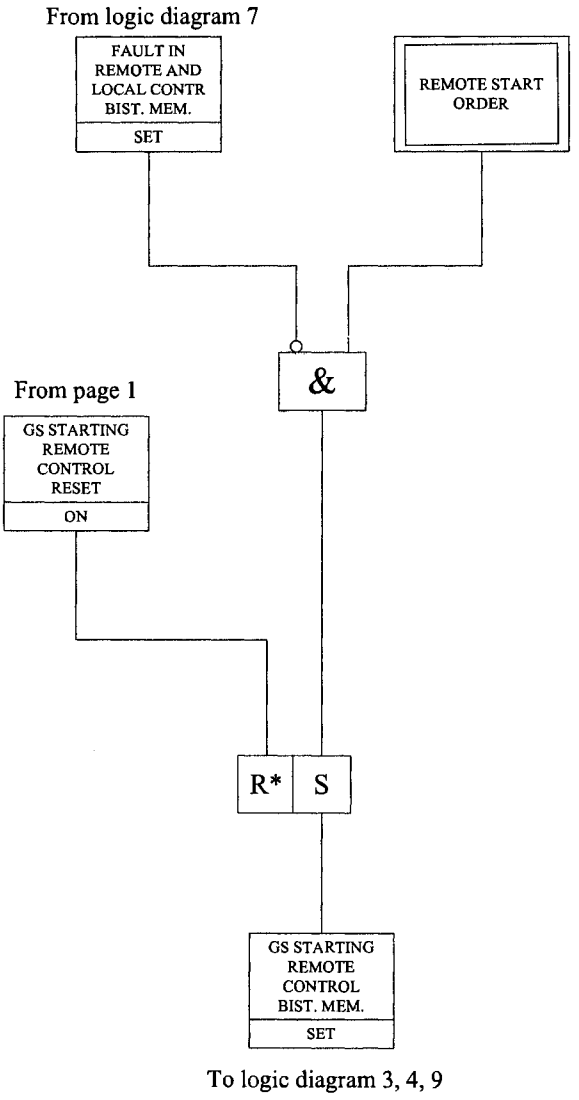

Figure A.2: Logic diagram 2

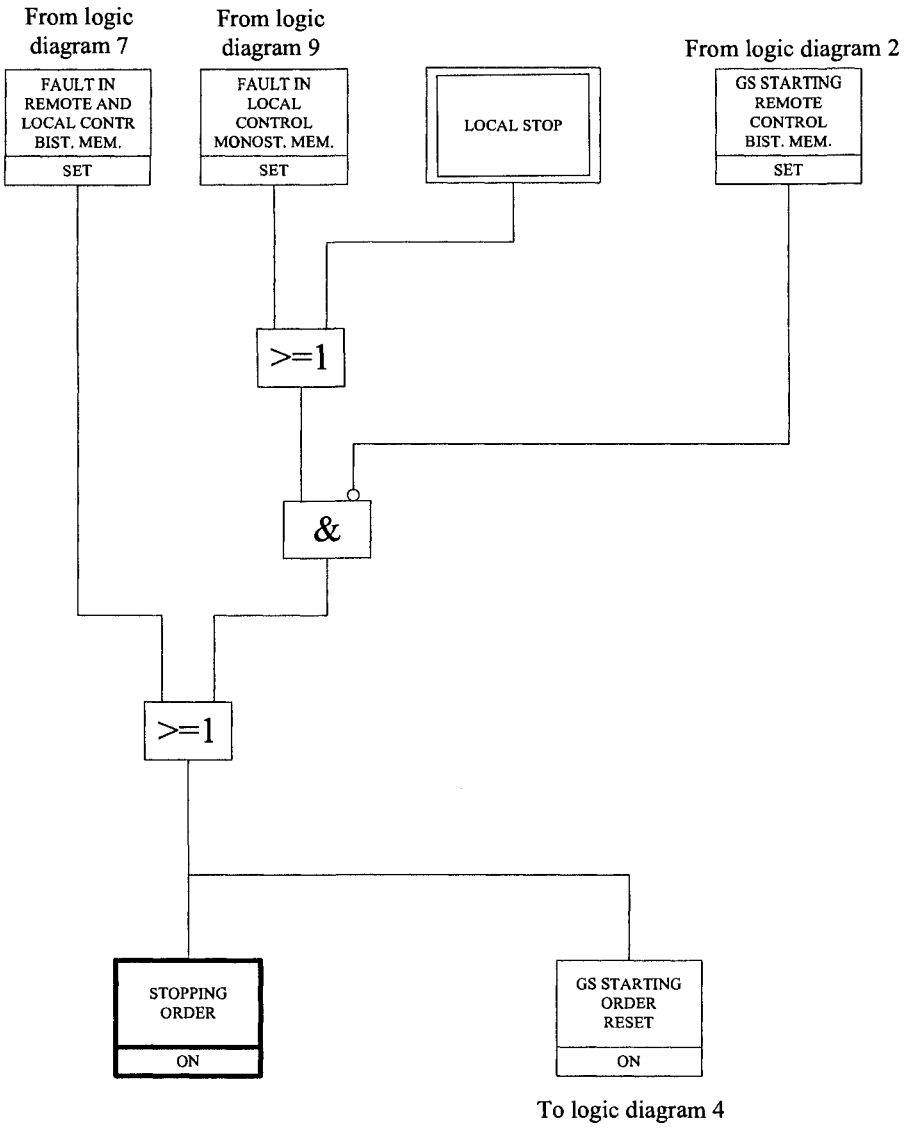

Figure A.3: Logic diagram 3

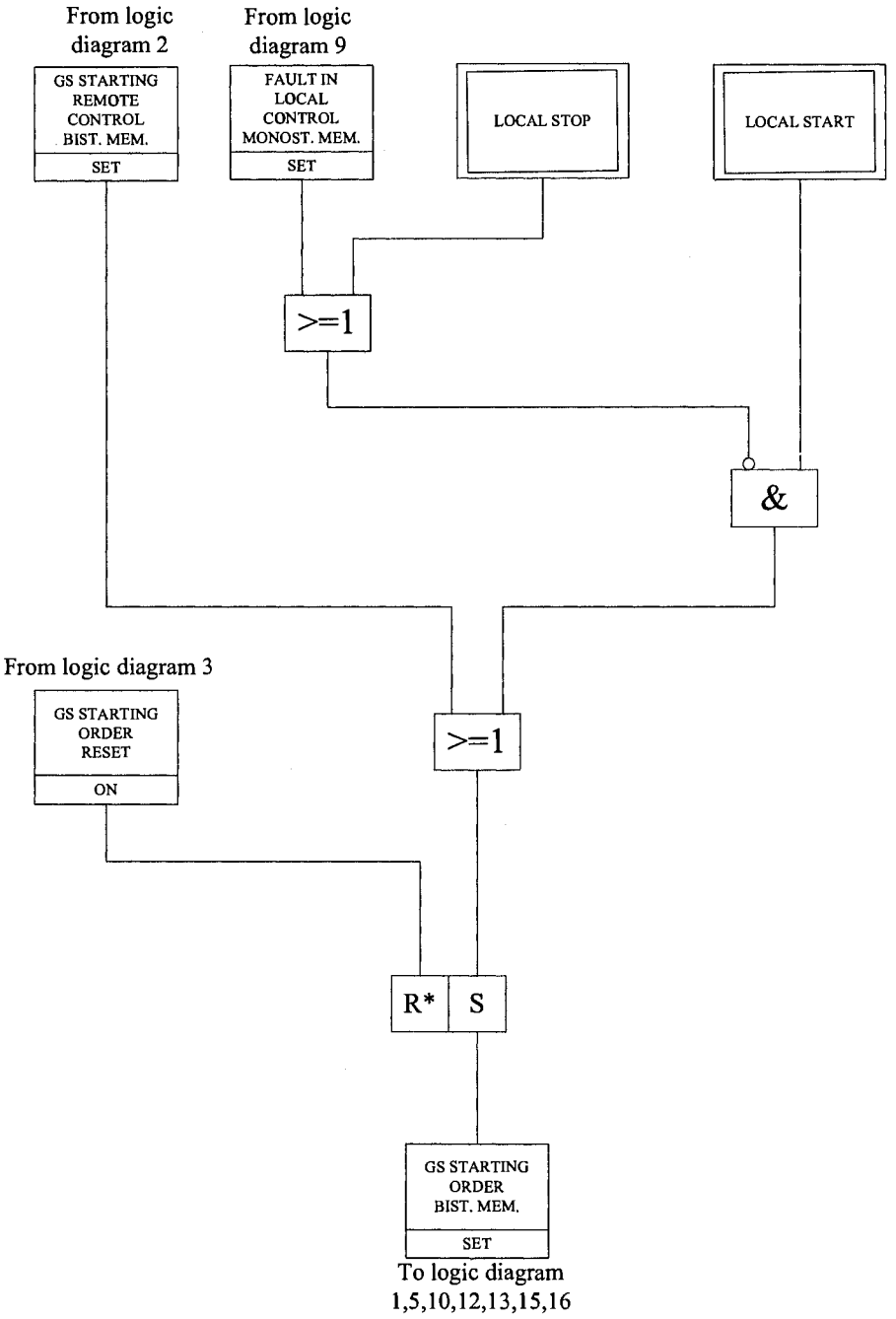

Figure A.4: Logic diagram 4

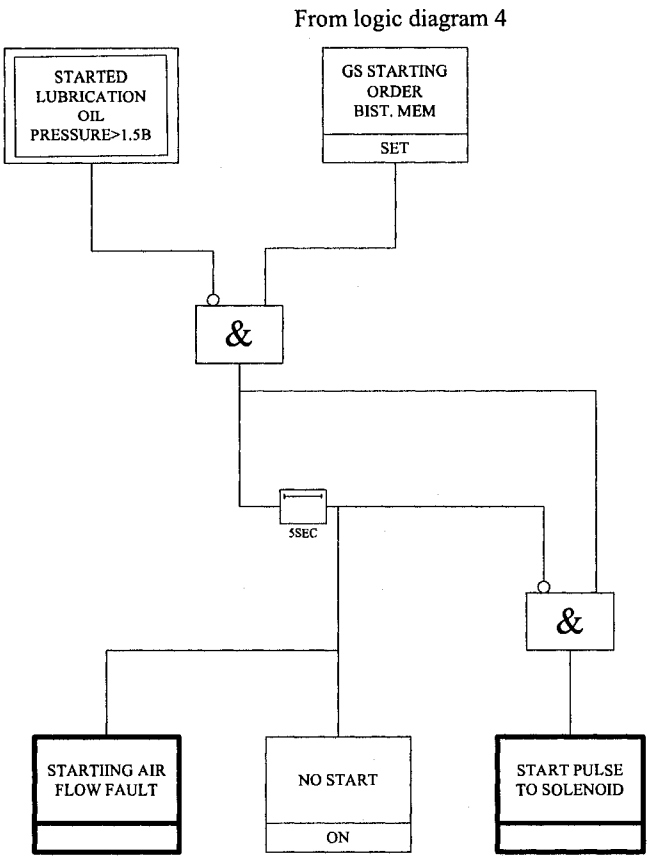

Figure A.5: Logic diagram 5

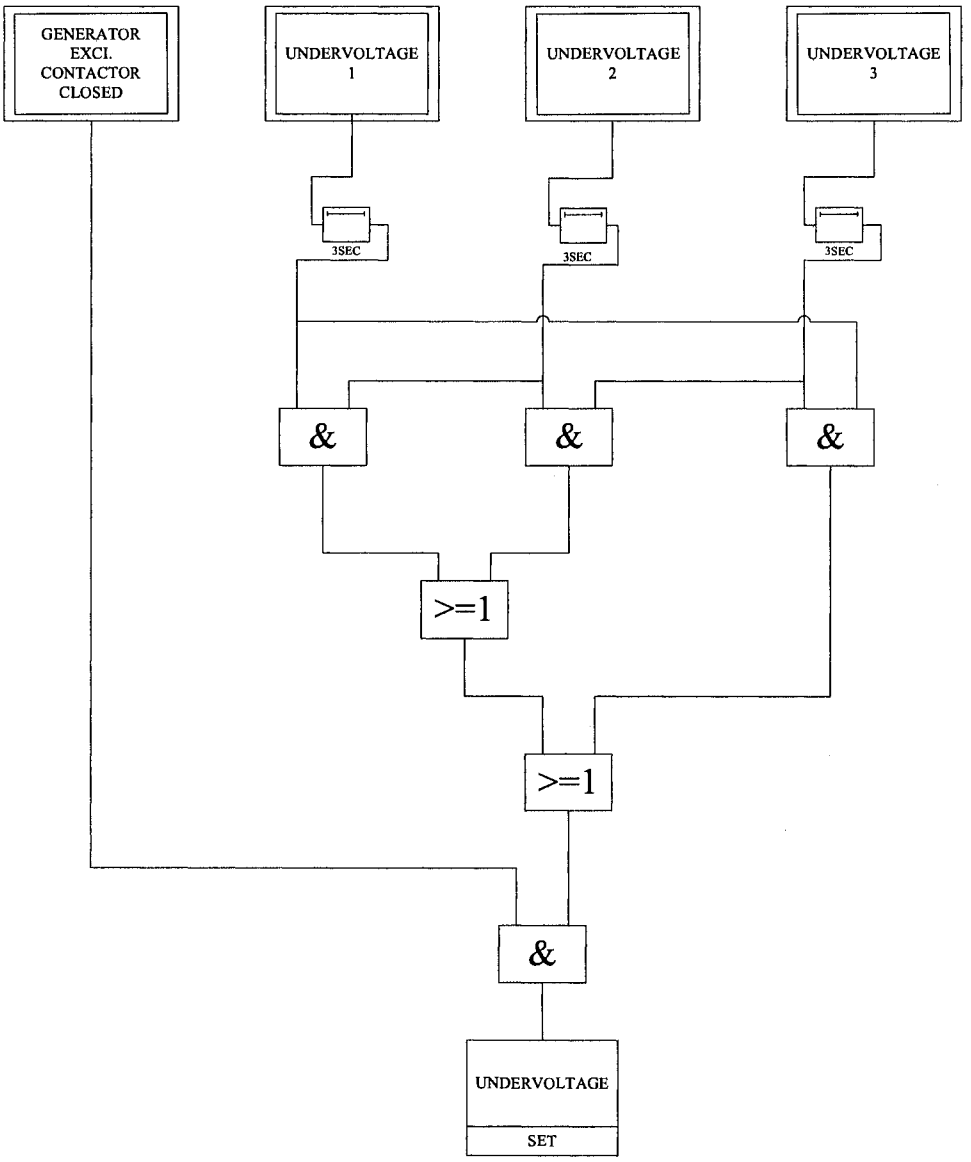

Figure A.6: Logic diagram 6

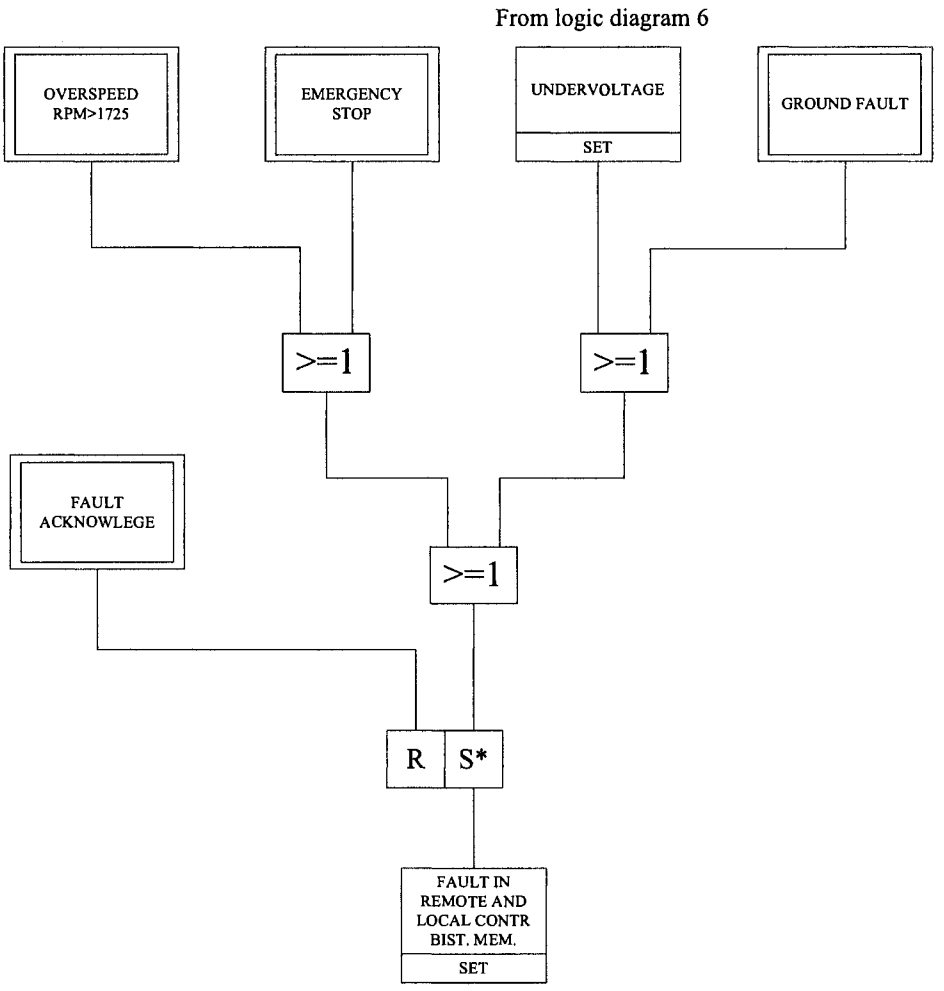

Figure A.7: Logic diagram 7

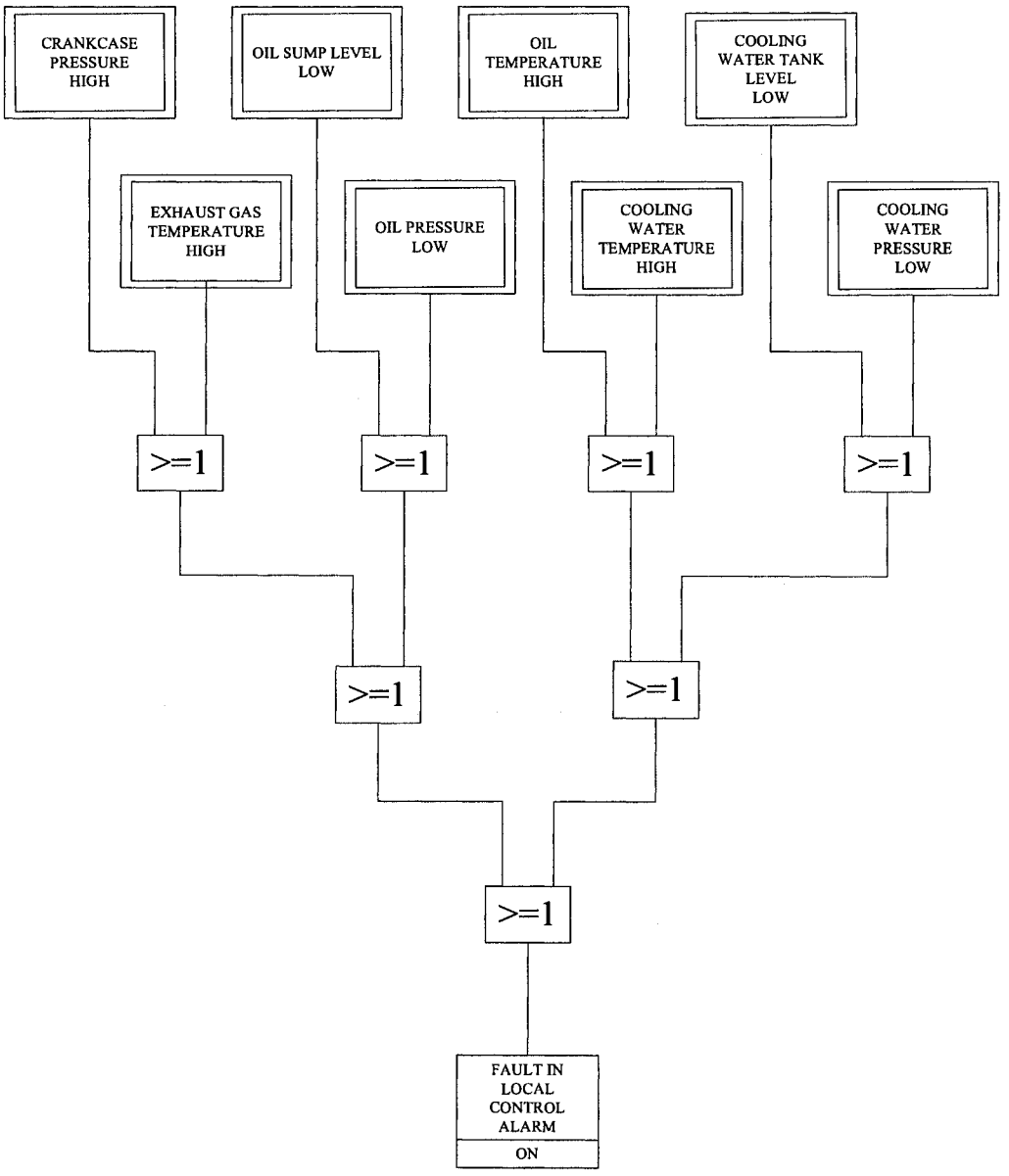

Figure A.8: Logic diagram 8

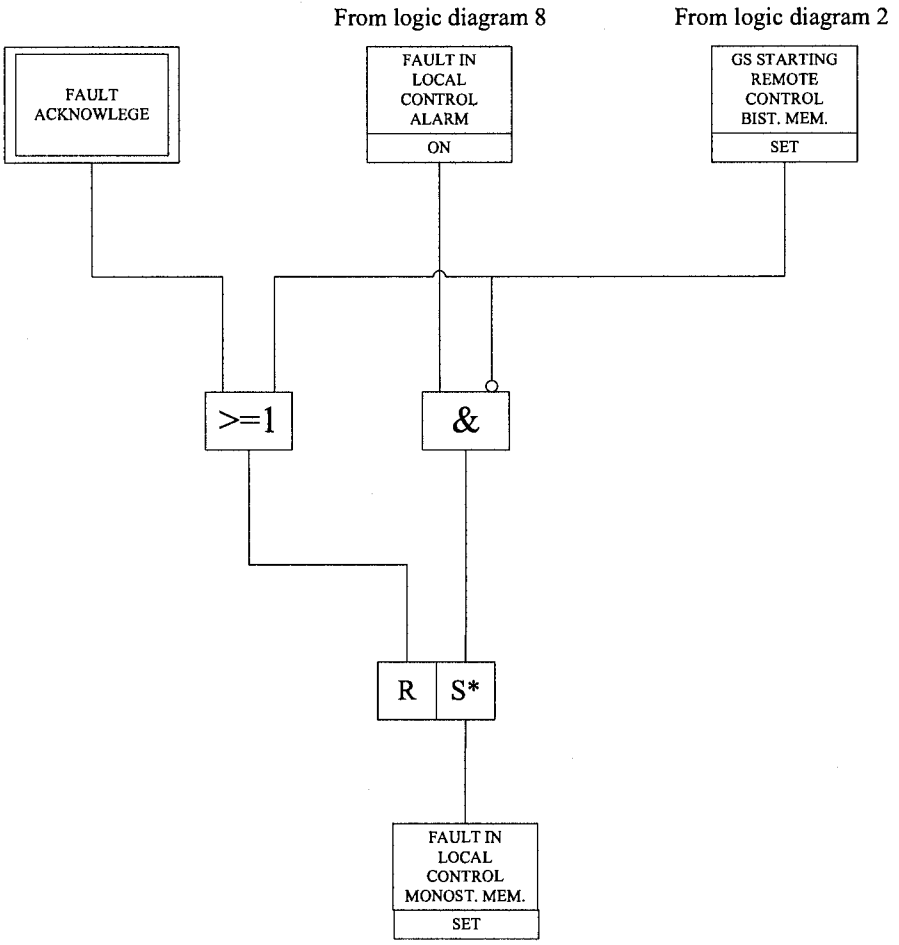

Figure A.9: Logic diagram 9

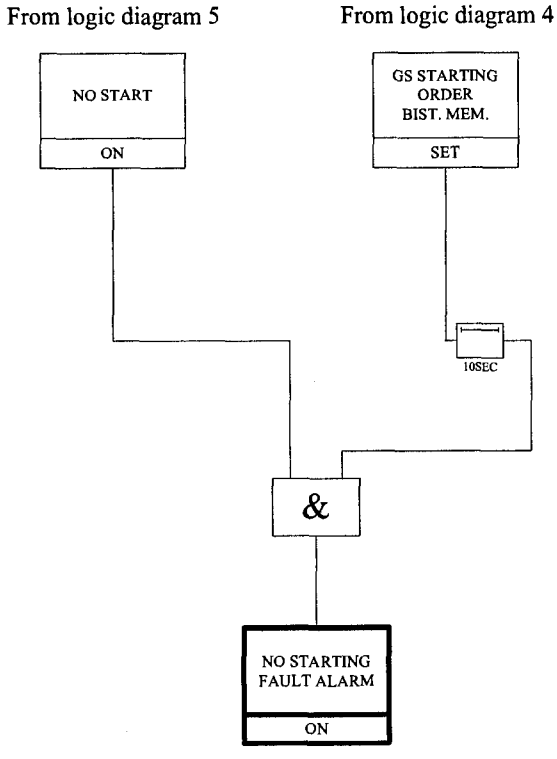

Figure A.10: Logic diagram 10

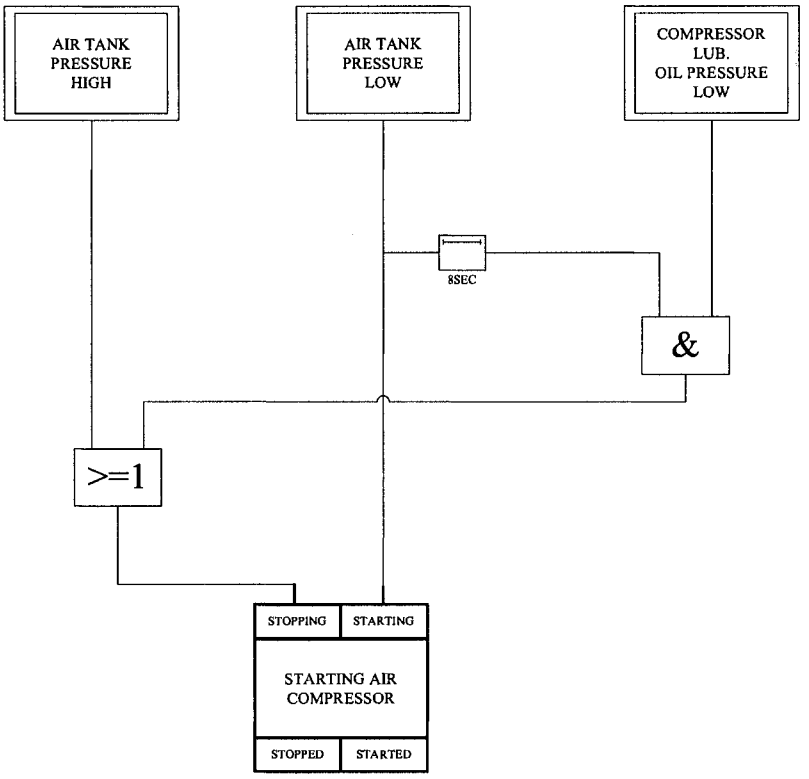

Figure A.11: Logic diagram 11

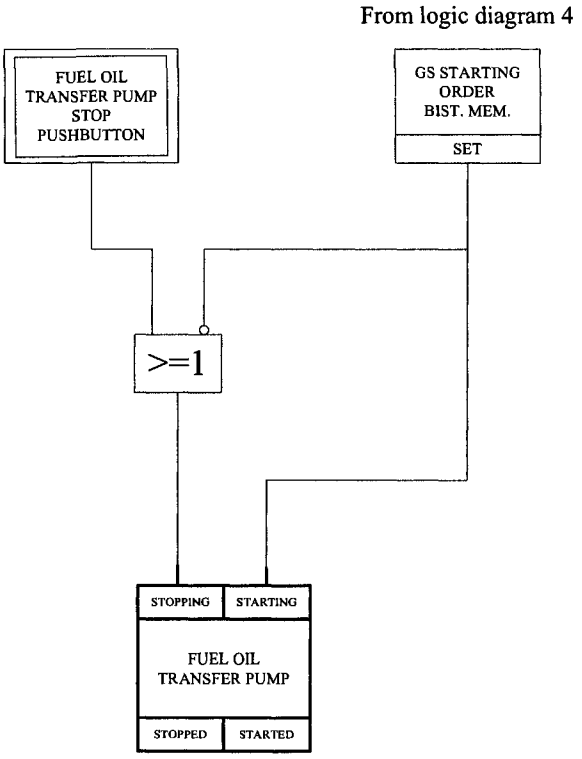

Figure A.12: Logic diagram 12

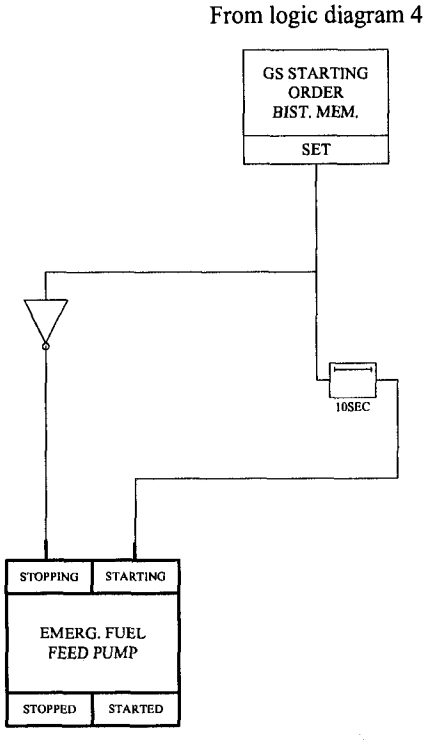

Figure A.13: Logic diagram 13

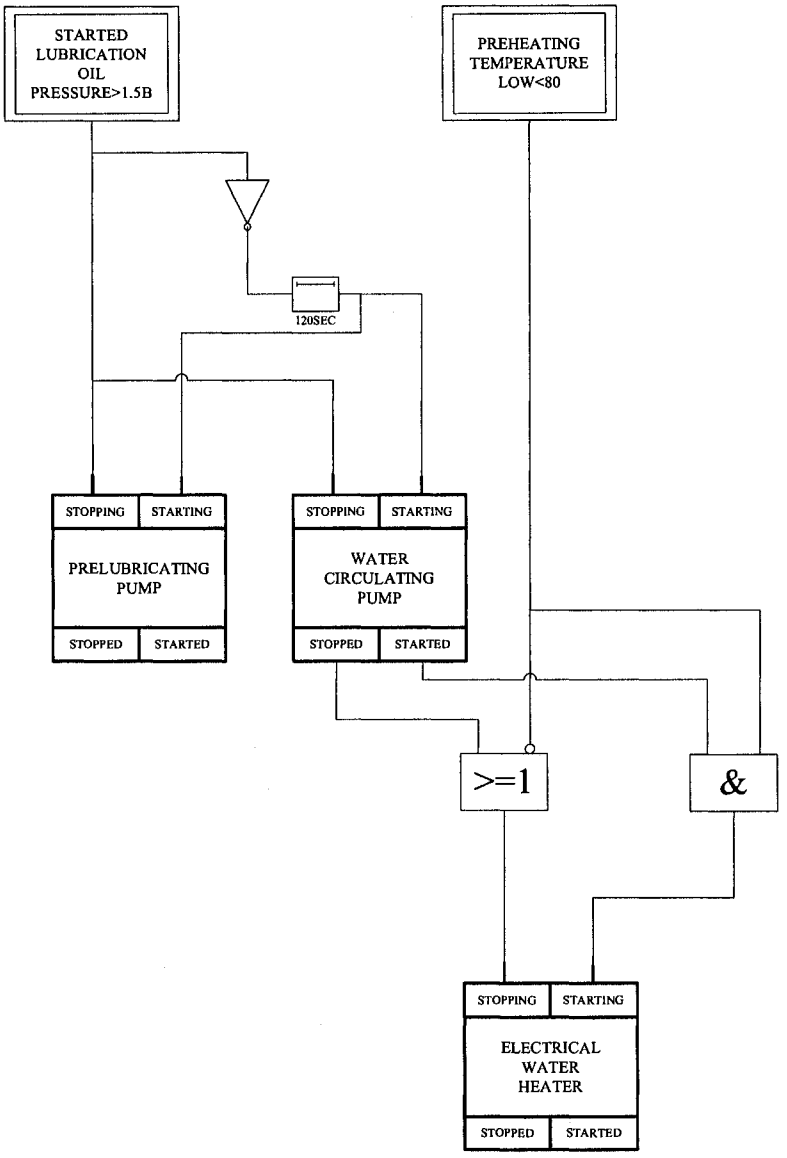

Figure A.14: Logic diagram 14

From logic diagram 4

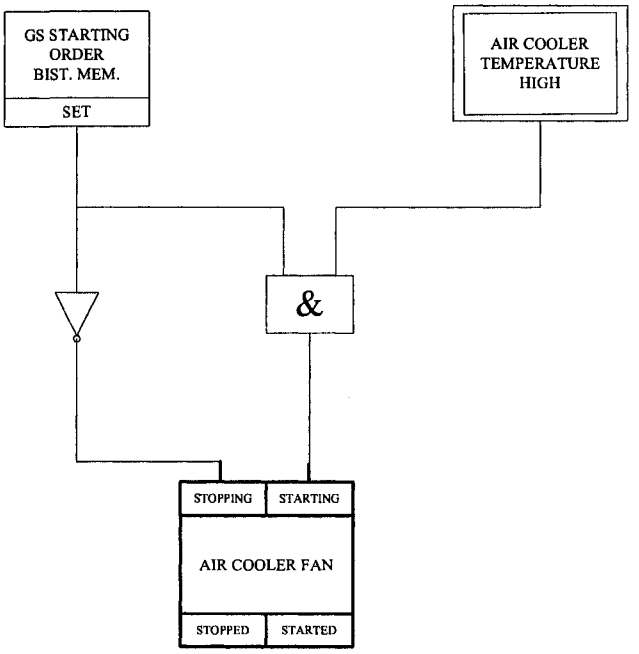

Figure A.15: Logic diagram 15

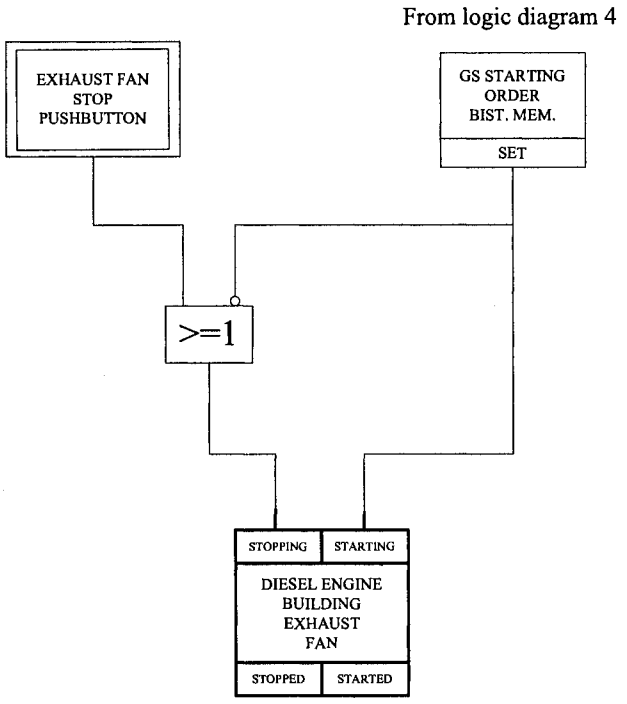

Figure A.16: Logic diagram 16

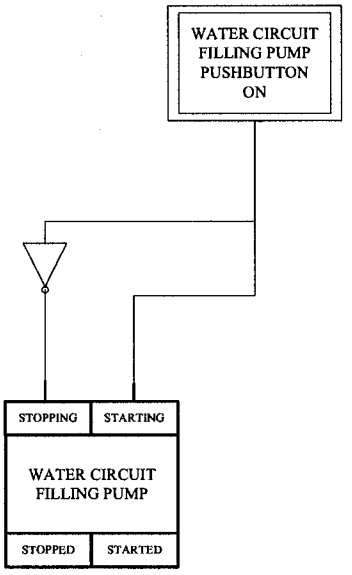

Figure A.17: Logic diagram 17

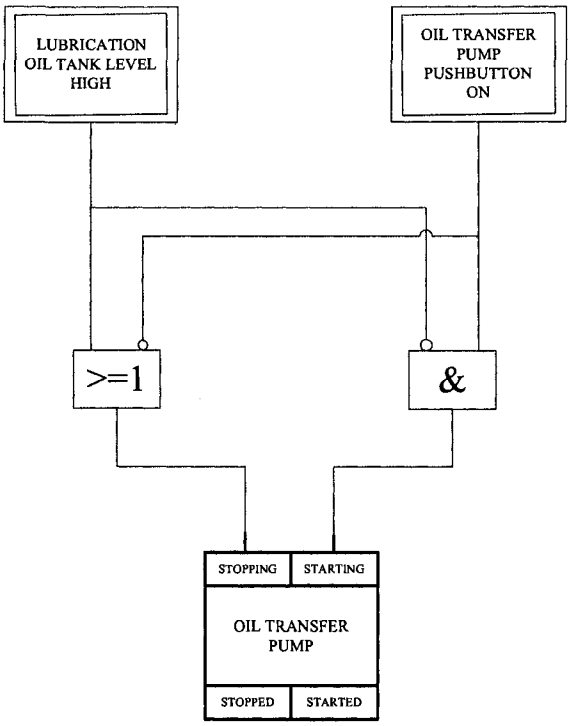

Figure A.18: Logic diagram 18