Formal Verification and Dynamic Validation of Logic-based Control Systems

by

Taeshin Park

Submitted to the Department of Chemical Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Chemical Engineering

at the

MASSACHUSETTS INTITUTE OF TECHNOLOGY

~~September 1997~~
February 1998

Author ....
Department of Chemical Engineering
September 29 1997

Certified by ....
Paul I. Barton
Assistant Professor
Thesis Supervisor

Accepted by ....
Robert Cohen
St. Laurent Professor of Chemical Engineering
Chairman, Committee on Graduate Students

Formal Verification and Dynamic Validation of Logic-based Control Systems

by

Taeshin Park

Submitted to the Department of Chemical Engineering
on September 29 1997, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Chemical Engineering

Abstract

Safety is an important issue in the CPI (Chemical Processing Industry) because accidents can lead to major loss of life and property. More sophisticated control systems and processes in today's CPI often allow operation at more severe conditions and closer to safety limits, which is often economically desirable. Multiple layers of protection exist in today's chemical plants for risk reduction and control. Among them, the layer of logic-based control systems (or automatic protective systems) provides major protection from potential hazards and risks. Logic-based control systems drive the plant to a safe shutdown state automatically when safety critical conditions occur and prevent the plant from entering a potentially dangerous state. Furthermore, logic-based control systems play an essential role in sequencing operations such as start-up, shut-down, feedstock changeover, etc. The safety-critical nature and increasing complexity of typical applications makes the development of systematic design techniques for logic-based control systems a critical issue. While procedures for design with respect to safety integrity are now available, much less attention has been paid to the design and verification of logic-based control systems with respect to functionality. This thesis therefore develops rigorous techniques for the functional verification of logic-based control systems with respect to design specifications.

A formal verification technology known as implicit model checking is developed that can systematically identify any inconsistency between the functionality of logic-based control systems and their specifications. The formal verification problem exhibits combinatorial complexity due to the exponential growth of the discrete state space with the number of state variables. Implicit model checking completely eliminates this state explosion in the problem formulation. The logic-based control system is modeled compactly as an implicit Boolean state space model. This Boolean state space model can embed implicitly all the details of the functionality exerted by any logic-based control system without encountering the state explosion problem. The language of temporal logic is employed to specify formally the correctness properties to be checked against the model. The verification problem is posed as a Boolean satisfiability problem by combining the model with specifications. The problem is then transformed into its equivalent 0-1 linear integer programming feasibility problem because it allows efficient solution with standard integer programming solution algorithms. The efficiency of this approach is demonstrated through the empirical study of a series of large-scale problems derived from industrial practice. The implicit model checking technology is applied to three industrial burner management systems: the MOD5™ system, the Throx sequence, and a global burner management system consisting of a series of MOD5™ systems. The methodology has successfully verified a problem containing potentially upwards of 5.7×10^{132} states and 4.8×10^{474} transitions.

In real life, the logic-based control system is coupled with its underlying chemical process, and its control actions also depend upon dynamic interaction with this process. Therefore, even if the logic-based control system is fully consistent with its design specifications, it may still fail to function correctly if this dynamic interaction is not properly considered in the design of the logic-based control system. A dynamic validation technique is developed as a tool to identify such potential problems. The dynamic validation problem is formulated as hybrid discrete/continuous dynamic simulation problem. The hybrid model is constructed by coupling the model of the logic-based control system and the model of the physico-chemical process. A solution algorithm is developed to solve the hybrid simulation problem. In particular, a state event location algorithm is developed that guarantees the location of all discontinuities in strict time order with minimal computational efforts. The algorithm supports flexible representation of state conditions in propositional logic, and completely eliminates discontinuity sticking, a numerical phenomena that can cause computational inefficiency and simulation failures. The algorithm has successfully solved a series of test problems from the literature and a number of chemical engineering problems.

Thesis Supervisor : Paul I. Barton

Title : Assistant Professor

Dedicated to Jeong-Yeon

Acknowledgments

Many people have made it possible for me to get where I am now as I reflect on the long trail that has taken me to this thesis.

Let me begin by thanking Professor Paul I. Barton for his guidance and encouragement over the past years spent here at MIT. He has provided an enriching intellectual environment, and his advice has always been helpful for my research and career development.

I would like to thank other members of the Barton research group. In particular, I would like to mention Berit Ahmad, Russell Allgor, Richard Brogle, Christophe Bruneton, Lars Kruel, William Feehery, Santos Galan, Wade Martinson, Denes Sedes, and John Tolsma.

I would like to express my sincere thanks to all my friends for making my life at MIT more pleasant. In particular, I would like to thank Hyungshik Shin, Jaewook Ko, Shinkyu Gang, Chonghun Han, Sung Park, Glen Ko, Yongpil Han, Byunghyuk Sohn, Chunhyuk Lee, Heeyeop Chae, Seokwon Lee, and Sungjoon Yoon.

I would like to thank Professor En Sup Yoon, who convinced me to come to MIT, for his continued support and thoughtful advice for all kinds of matters.

Finally, I would like to thank my parents for their never-ending support and encouragement. Most importantly, I would like to thank my best friend and wife, Jeong-Yeon Choi, whose love, patience, and support have helped me through my years at MIT and to whom I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, the chemical processing industries has experienced significant changes as a result of new safety and environmental regulations, globalization of the world economy and increasing competition, and more stringent manufacturing constraints. In particular, the level of material and energy interaction in plants has increased substantially, which leads to more complex processes, and operation under more severe conditions. Furthermore, the level of plant automation has increased substantially, especially in plants operating in the batch mode.

More attention is being paid to safety in the chemical industries than ever before due to the above trends in the chemical industries, and a number of major accidents in the last two decades resulting in large loss of lives (Flixborough, 1974; Seveso, 1976; Bhopal, 1984; Chernobyl, 1986). As an illustration of the damage that can be caused by a major safety event in a chemical process, Figure 1.1 shows an ethylene oxide plant after a fire and explosion. Safety in chemical plants is concerned with the safety of plant personnel, plant equipment, and the surrounding community. Safe operation of chemical plants is essential because: 1) chemical plants process large quantities of material that is toxic, flammable, and/or explosive, 2) accidental emissions of these materials cause serious safety and environmental problems, and 3) there can be large losses associated with an accident in terms of human life and property, material, and energy losses.

Multiple layers of protection are provided in modern chemical plants to maintain safe operation (AIChE/CCPS, 1993). Figure 1.2 shows the typical protection layers found in a modern chemical plant in order of expected activation as a hazardous condition is approached. Multiple protection layers addressing the same event are often necessary to achieve high levels of certainty that protection will be available when needed (Drake and Thurston, 1993). Each layer is called an *independent protection layer*, which is defined as a system specifically designed to reduce the likelihood or severity of impact of an identified hazardous event by a large factor (AIChE/CCPS, 1993). The action of each protection layer must be independent of other protection layers associated with the identified hazardous event.

Protection begins with the most fundamental elements of process design, i.e., to design inherently safe processes (Kletz, 1991). For example, many incidents are the result of leaks of hazardous materials. The most effective way of preventing such incidents is to use hazardous materials as little as possible, or to substitute them with safer materials. The hazard is avoided rather than controlled and the design is inherently safe. Kletz (1991) discusses design practices for inherently safe process. However, even if a process is designed to promote safe operation according to this design philosophy, potential risks and hazards still remain in many chemical processing plants. These range from unlikely events with minor impact to major risks. Additional protective actions must be applied to control these risks.

The next layer of protection is the Basic Process Control System (BPCS) installed to regulate normal production functions. The BPCS regulates automatically the heat and material flows and provides sufficient information for operator supervision of processing conditions. Although the BPCS functions as a protective layer, its primary purpose is automatic regulation of the process.

Physical protection such as pressure relief devices and/or dikes provides another layer of protection. The general trend is toward less reliance on physical protection (where possible), because activation of the relief system is not preferred in an environmental sense, and relief collection and disposal systems to prevent direct emissions to atmosphere are expensive. Furthermore, it may be difficult to install relief devices that are large enough and operate quickly enough to avoid over-pressuring the equipment if an explosion occurs in a vessel (Lawley and Kletz, 1975).

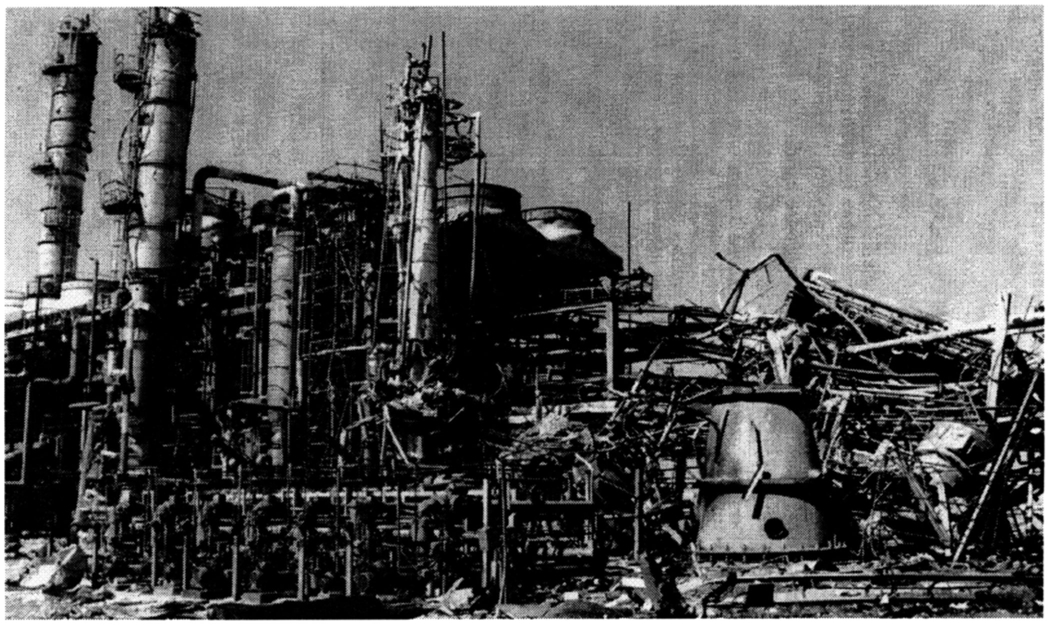


Figure 1.1: An ethylene oxide plant after a fire and explosion[1]

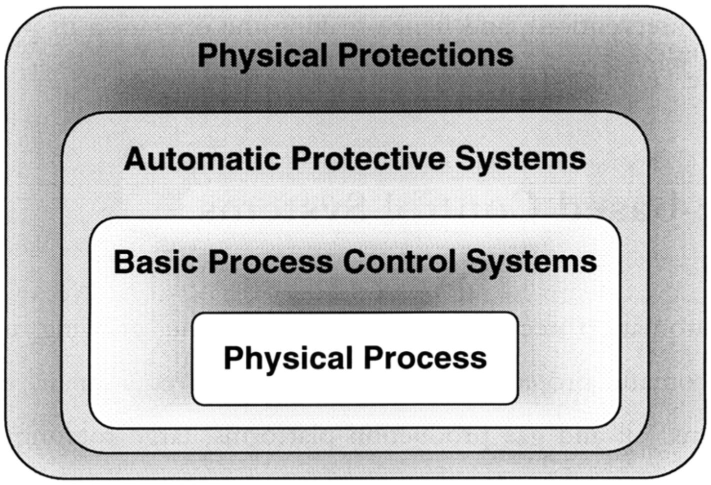


Figure 1.2: Typical protection layers found in modern chemical plants (AIChE/CCPS, 1993)

[1] From What Went Wrong, 3rd Edition, by Trevor Kletz. Copyright 1994 by Gulf Publishing Company. Used with permission. All rights reserved.

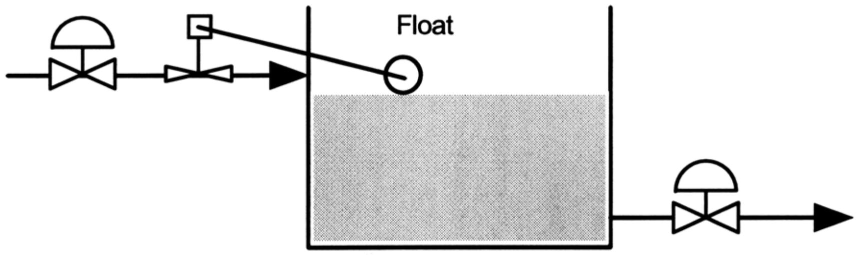


Figure 1.3: Mechanical interlock to prevent tank overflow

Major protection in modern chemical plants is provided by automatic protective systems, an intermediate layer of protection between the BPCS and physical protection. Automatic protective systems assume greater importance when hazards cannot be avoided by inner protection layers. Typically, an automatic protective system is installed as a dedicated defense against potential hazards. Each automatic protective system is usually expected to reduce the likelihood of a particular hazard by a factor of at least 100. This is achieved by taking automatic action to correct an abnormal plant event which has not been controlled and/or caused by actions in the previous level of protection (the BPCS and/or operator intervention), and hence making the operation of physical relief devices as rare as possible.

1.1 Logic-based Control Systems

Advances in automation technology have led to the growing use of increasingly sophisticated automatic protective devices on process equipment, especially in power generation systems, oil and gas production platforms, large rotating machinery, nuclear and certain chemical reactors, and boilers and furnaces (Frederickson, 1990). Automatic protective systems range from simple mechanical devices to complex interlock devices. Figure 1.3 is a simple mechanical interlock, which couples tank level and inlet flow to prevent tank overflow. Pressure and temperature trips are commonly used in chemical processing industries. Interlocks are also used to ensure the synchronized operation of equipment such as in a conveyor system. For large rotating machinery, interlocks are used to control the start-up so as to ensure that all the pre-start conditions are satisfied and that the correct sequence is followed. Similarly, sophisticated shutdown interlocks are implemented in chemical reactors to prevent thermal runaway. Complex interlocks known as burner management systems are implemented in furnace equipment to help ensure safe operation.

In modern plants, these automatic protective devices are referred to collectively as *safety interlock systems* (SISs). The term safety interlock system can refer to many different definitions and applications. NFPA Standard 85A (NFPA, 1982) defines a safety interlock as follows:

> "A device or group of devices arranged to sense a limit or off-limit condition or improper sequence of events and to shut down the offending or related piece of equipment, or to prevent proceeding in an improper sequence in order to avoid a hazardous condition."

According to this definition, safety interlock systems embed *shutdown logic*, *permissive logic*, and *sequences*. Shutdown action shuts the plant or part of it down if an out-of-limits or abnormal condition is detected. Permissive function checks the existence of a prerequisite for a specific action before it is actually taken. Embedded sequences force the plant or part of it to be operated according to pre-determined sequences. These multiple functionalities are embedded in a coupled manner in the SIS, therefore many modern SISs exhibit considerable complexity (e.g., a burner management system, the safety system for a toxic and/or exothermic chemical reactor, the interlock system for the top side of an offshore production platform).

Even though automatic protective systems are commonly referred to as safety interlock systems, this term is somewhat confusing since, as we have discussed above, modern automatic protective systems will typically embed other functionalities coupled to and in addition to shutdown interlock functionality. Therefore, we argue that modern automatic protective systems are more properly regarded as general *logic-based control systems* (LCSs). The term logic-based control systems will be used throughout this thesis.

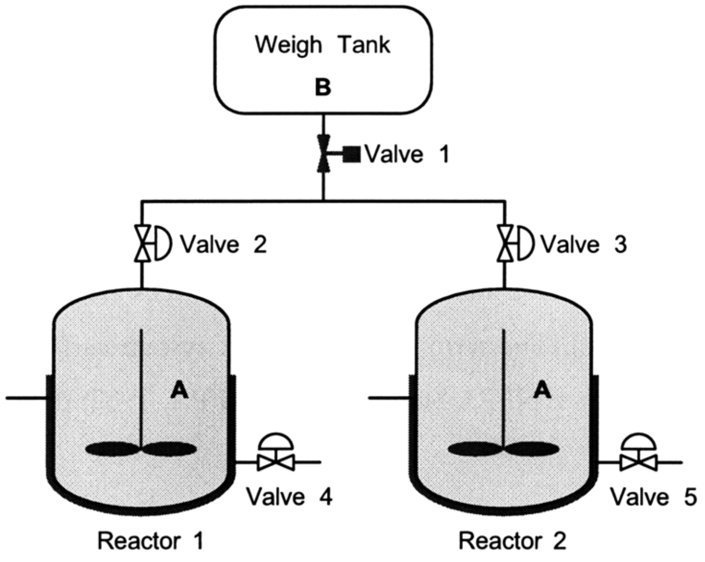


Figure 1.4: Batch reactor system

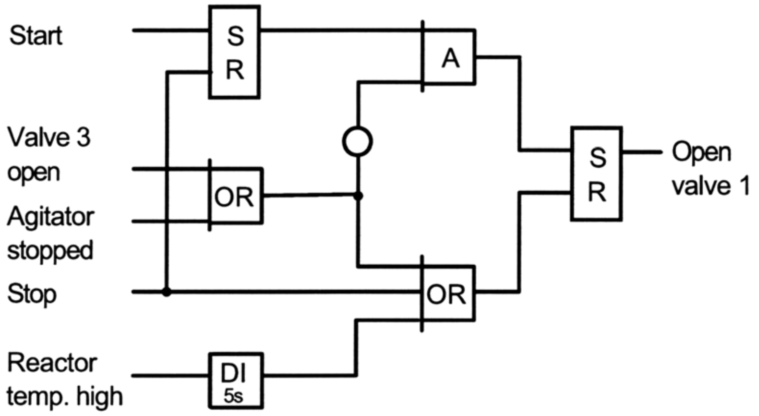


Figure 1.5: Logic-based control system for batch reactor operation

As an example, consider the batch reactor system (Lees, 1980) in Figure 1.4. A reactor is charged with chemical A, and then chemical B is fed gradually from a weigh tank as the reaction proceeds. Adequate mixing and cooling is necessary because of the exothermic nature of the reaction. Some prerequisites must be satisfied before opening valve 1 to start the reaction in reactor 1. Valve 3 must be closed and the agitator in reactor 1 should be working to prevent a lack of mixing. This functionality illustrates the permissive action of LCSs. During normal operation, shutdown action is required to cut off the supply of chemical B from the weigh tank if the valve 3 of reactor 2 is opened, agitator malfunction is detected, or the reactor temperature has risen above a fixed threshold for more than 5 seconds. Figure 1.5 shows the LCS as a *binary logic diagram* (ANSI/ISA, 1981). The LCS installed in this plant protects the plant against equipment failures, control failures from the BPCS or higher levels, and operator errors. The flowrate of chemical B and cooling water is controlled by the BPCS and/or an operator by manipulating valve 2 and valve 4. In the event of BPCS and/or operator failures that issue incorrect actions to these valves, or plant equipment failures such as an agitator malfunction, the LCS is activated to protect the plant.

Consider the tank filling process in Figure 1.6, and its LCS represented by a binary logic diagram in Figure 1.7. The LCS shown is responsible for decisions concerning the running of the pump (the control element). Input signals to the LCS include hand-operated control signals (e.g., HS1, HS2, HS7), and signals that indicate threshold crossings in the system's dynamic state (e.g., LSH3, LSH4, PSL5). The logic of the control system embeds shutdown logic (e.g., the pump is shut down regardless of the system state if the suction pressure remains low for more than five seconds), permissive logic (e.g., the pump cannot be started if the switches HS1 and HS2 are ON simultaneously), and sequences (e.g., to start filling a tank, open the inlet valve and close the inlet valve on the other tank, wait until the valves have responded, start the pump). Note that even in this simple example, multiple functionalities are embedded in a coupled manner.

Figure 1.8 shows the generic structure of a LCS and its interaction with its underlying physico-chemical process. The LCS consists of three components, input module, logic module, and output module. The input module monitors analog and discrete signals and converts them to logic signals for the logic module. The logic module determines the action of the LCS by solving the logic and generating output signals. Finally, the output signals of the LCS are linked to the final control elements by the output module.

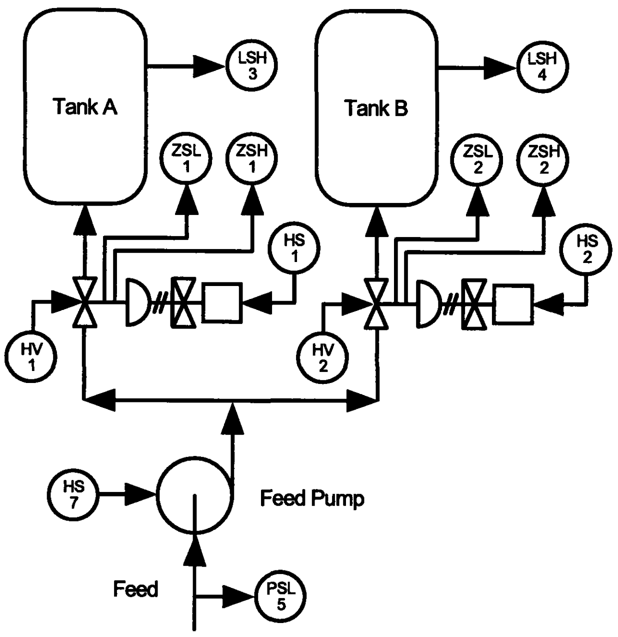


Figure 1.6: Tank filling process

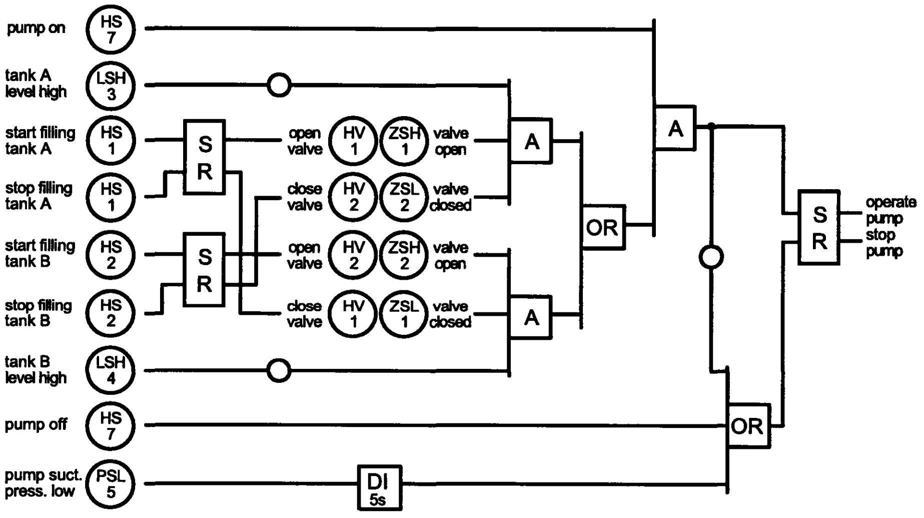


Figure 1.7: Logic-based control system for tank filling process

Note that the LCS is normally activated by the change in status of discrete input signals (e.g., `Start_filling_tank_A`) and/or by the value of analog input signals (e.g., `Tank_A_level`) passing through a critical or pre-determined threshold. An input interface between the analog input signals and the LCS is necessary because the LCS is a discrete system. Due to the nature of the input interface between the process and the LCS, the operation of the LCS is *static*: it continuously monitors selected variables but remains unoperative until abnormal conditions occur. However, if activated, the LCS imposes actions to final control elements (e.g., valves), which causes discontinuities in the process behavior.

Due to the nature of this operation, the LCS can be considered to be a member of the class of *reactive systems*, which are defined as computing systems that continuously react to their environment at a speed determined by this environment (Halbwachs, 1993). In the case of the LCS, the environment is the underlying chemical process. Note that reactive systems should be distinguished from *transformational systems* (whose inputs are available at the beginning of the execution and which deliver their outputs when terminating) or *interactive systems* (which continuously interact with their environment, but at their own rate).

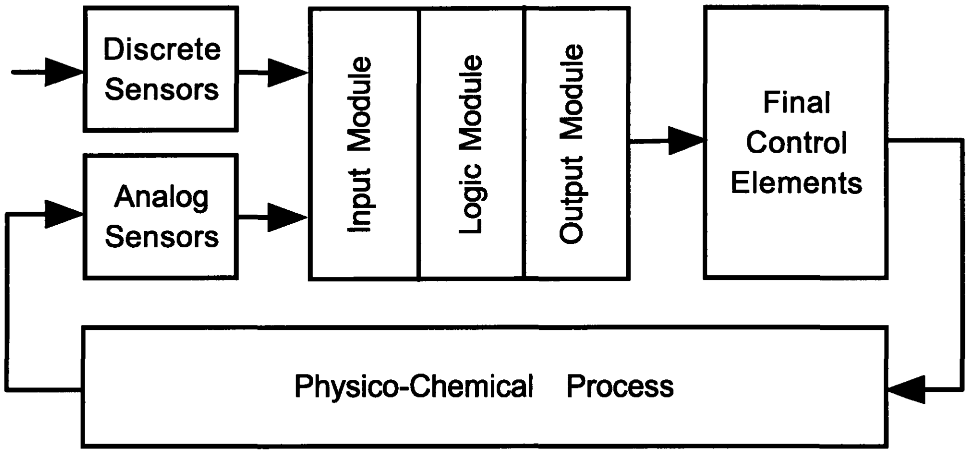


Figure 1.8: Structure of the LCS and its interaction with a chemical process

1.2 Survey of Hazardous Incidents

According to recent data on the primary causes of control system failures in safety-related applications based on 34 incidents (Bell, 1994) shown in Figure 1.9, many accidents in chemical processing plants have occurred because appropriate automatic protective systems were not installed, they were not tested thoroughly, they were modified without verification, or they were made inoperative when needed.

The worst catastrophe in the history of the chemical industry occurred in Bhopal on December 3, 1984. A leakage of toxic methyl isocyanate spread beyond the plant boundary and caused the death over 2,000 people. The direct cause of the accident was the contamination of an methyl isocyanate storage tank by several tons of water and chloroform. A runaway reaction occurred, and the methyl isocynate vapor was discharged to atmosphere. The protective systems, which could have prevented or minimized the release, were not in full working order. The high temperature alarm on the methyl isocyanate tank did not operate, as the setpoint had been raised too high. One of the main lessons of Bhopal is the need to keep protective systems in good working order (Kletz, 1994).

The following sections survey hazardous incidents that have occurred in the chemical processing industries, whose causes are related to automatic protective systems.

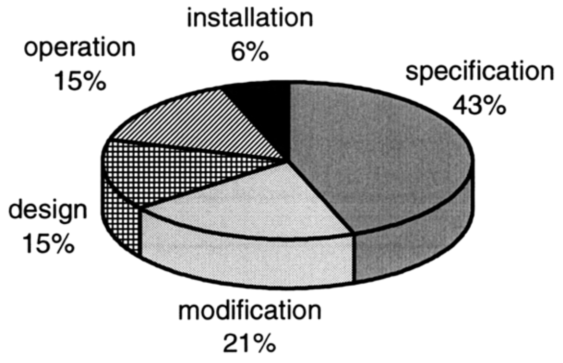


Figure 1.9: Primary causes of control system failures

1.2.1 Incidents Caused by the Lack of Appropriate Automatic Protective Systems

This section discusses three examples of accidents that occurred because appropriate automatic protective systems were not installed. The first example is a failure of furnace tubes due to overheating. The pump feeding an oil stream to the tubes of a furnace failed. The operator closed the oil valve and intended to open a steam valve to purge the furnace tubes. He opened the wrong valve, there was no flow to the furnace, and the tubes were overheated and collapsed (Kletz, 1994).

The second example is a runaway reaction. A reactor was filled with one reactant, and an operator started to add another reactant, gradually increasing the flow. The operator intended to start a flow of cooling water as soon as the temperature started to rise. The temperature rise was not shown in the recorder even though the temperature actually rose as indicated by a temperature sensor. A runaway reaction occurred (Kletz, 1994).

Finally, an accident occurred at a plant in which ethylene oxide and aqueous ammonia were reacted to produce ethanolamine. Some ammonia leaked into the ethylene oxide storage tank, past a series of check valves. The ammonia reacted with ethylene oxide in the storage tank. The resulting explosion caused damage and destruction over a wide area. It is not sufficient to rely on check valves. The pressure drop in the pipeline should be measured and pressure trips should be installed to close the valve automatically if the pressure is too low (Troyan and Vine, 1968).

In all three cases, appropriate valve interlocks and/or trips could have prevented the accidents.

1.2.2 Incidents Caused by Incomplete Automatic Protective Systems

This section discusses two examples of accidents that occurred because existing automatic protective systems were not complete and thus failed to provide protective actions when needed. The first example is an explosion of a furnace. A decrease in fuel oil pressure caused the burner in an oil-fired furnace to go out, and the flame failure device closed the solenoid valve in the fuel oil line. When the fuel oil pressure was restored, the operator ignited the furnace after testing the atmosphere in the furnace. Even though the fuel oil supply was isolated, an explosion occurred. When the burner went out, the solenoid valve took a few seconds to close, and during this time some oil entered the furnace. The flash point of the fuel oil was 65°C, too high for the oil to be detected by the gas detector (Kletz, 1994). An automatic protective system that forces a mandatory purging step for a certain period of time before relighting a hot furnace could have prevented the incident.

The second example is a reactor rupture due to uncontrolled reaction. A reactor was initially charged with a batch of glycerol. The glycerol inside the reactor was circulated through a heat exchanger that acts as a heater. When the temperature reached 115°C, ethylene oxide is added, and the reaction mixture is cooled by the heat exchanger because the reaction is exothermic. An automatic protective system prevents the running of the ethylene oxide pump unless the heat exchanger circulation pump was running, the reaction mixture temperature is above 115°C (otherwise, the ethylene oxide would not react), and the reaction mixture temperature is below 125°C (otherwise, the reaction is too fast). Despite the protective system, an explosion occurred. When ethylene oxide addition was started, the pressure in the reactor rose, which indicates that the ethylene oxide was not reacting. The operator decided that more heat was required to start the reaction, so he adjusted the trip setting and allowed the indicated temperature to rise to 200°C. However, the pressure did not fall. Later, he found out that he had forgotten to open the valve at the base of the reactor, and so he opened it. A violent uncontrolled reaction occurred (Kletz, 1994). A high-pressure trip or valve interlock could have prevented the incident.

1.2.3 Incidents Caused by Process Modification without Rigorous Verification

This section discusses two accidents that occurred in batch reactors because operating procedures were modified without rigorous verification. In one incident, the recommended operating procedure was to add a reactant at 45°C over a period of 1½ hours. However, the operators decided to add it at a lower temperature and then heat the material in the reactor because the feed heater was not powerful enough to feed the reactant at this temperature.

In the second incident, a nitration reaction was carried out at low temperature. The approved operating procedure was to heat the reactor to 90°C, to keep it at this temperature for 30 minutes, and then to cool it down. After a year of successful operation, an operator decided to let the batch cool by heat loss to the surroundings as soon as the temperature reached 90°C.

In both cases, explosions occurred due to runaway reactions (Kletz, 1994). The moral of these stories is that any change to a process including operating procedures should be verified rigorously in order to prevent such accidents.

1.3 Design of Logic-based Control Systems

The previous section clearly illustrates that a rigorous design and verification technology for automatic protective systems is essential to prevent or reduce accidents. This section presents the prototypical design procedure for logic-based control systems in detail.

Two types of LCS failure have been identified: *random hardware failures* and *systematic failures* (HSE, 1987). Any implementation of a LCS consists of a large number of linked electronic and mechanical components. Each component will break down or wear out after a different length of time randomly. Because of this, it is not possible to predict exactly when a system will break down due to the failure of any one of its components. These random hardware failures can be effectively prevented by employing redundancy. Since failures occur randomly, it is extremely unlikely that redundant systems break down at the same time. On the other hand, systematic failures always occur under the same set of conditions. The associated faults will remain hidden until a particular set of conditions arises and the system breaks down. Unlike random hardware failures, a redundancy is not effective against systematic failures. The ultimate objective in the design of a LCS is to eliminate or minimize both these categories of failures. The two key attributes, *safety integrity* and *functionality*, are identified in the design of the LCS to deal with random hardware failures and systematic failures respectively (AIChE/CCPS, 1993). Figure 1.10 shows a prototypical design procedure for the LCS with respect to these attributes.

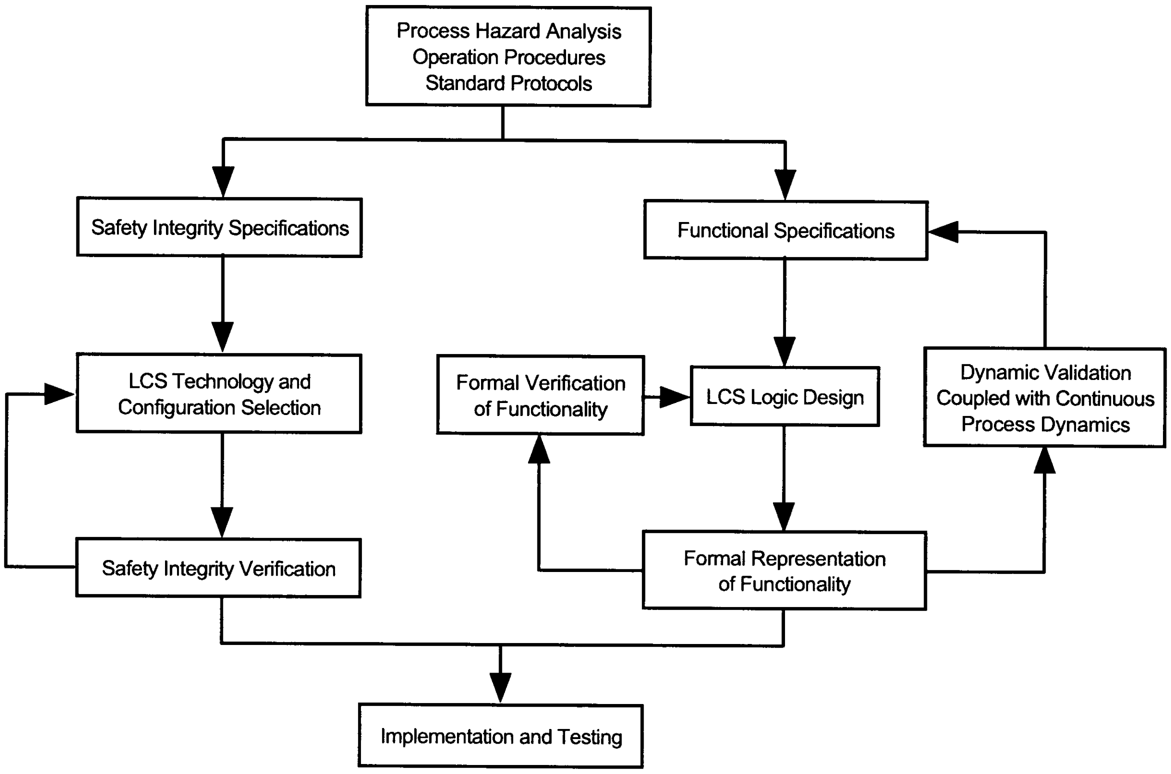

Process Hazard Analysis
Operation Procedures
Standard Protocols
Safety Integrity Specifications
Functional Specifications
LCS Technology and Configuration Selection
Formal Verification of Functionality
LCS Logic Design
Dynamic Validation Coupled with Continuous Process Dynamics
Safety Integrity Verification
Formal Representation of Functionality
Implementation and Testing

Figure 1.10: Prototypical design procedure for logic-based control systems

Table 1.1: Safety integrity level and availability

Safety integrity level	Hazardous event severity	Target availability
1	minor	0.99
2	serious	0.999
3	extensive	0.999 to 0.9999

Table 1.2: Safety integrity level and LCS configurations

Safety integrity level	Recommended LCS configuration
1	Non-redundant: best single path design.
2	Partially redundant: redundant independent paths for elements with lower availability
3	Totally redundant: redundant, independent paths for total LCS.

A basic set of information is necessary before the design of a LCS, which is provided by the *Process Hazard Analysis* (PHA). PHA identifies process risks, estimates potential consequences, and determines the need for LCSs. If a process to be controlled is similar to an existing system, experience with currently operating systems can be used for hazard analysis. Standards and codes of practice may also be used. In addition, a hazard and operability (HAZOP) study is frequently used to identify hazards (AIChE/CCPS, 1992). In a HAZOP study, an established procedure is followed in which the P&IDs are systematically analyzed component by component. For each component, a set of guidewords is used to help identify any process deviations that could lead to a hazard. Alternatively, failure mode and effect analysis (FMEA) can be used to analyze all the failure modes of a given item for their effects on other components and the final effect on process safety (AIChE/CCPS, 1992). Fault trees are logic diagrams that display sequences of failures of the process leading to top events. Fault tree analysis is useful in identifying causes of top events and evaluating the frequency of top events as a function of the magnitude of causes (AIChE/CCPS, 1989). Event trees start with a single event and then branch into all possible sequences of subsequent failures that might lead to a hazard. Event tree analysis provides systematic coverage of the time sequence of event propagation (AIChE/CCPS, 1989).

The outcomes of the PHA provide *safety integrity specifications* and *functional specifications*. Safety integrity specifications assign an integrity level classification to each LCS, and provide target *reliability* and *availability* for each LCS for the quantitative verification of the integrity level at a later step. Reliability is the probability that a system will function correctly under stated conditions for a stated period of time. Reliability is important to minimize hazards associated with spurious upsets. Availability is the probability that a system will be able to perform its designated function when required for use. Availability is important for LCSs that must function on demand to perform a protective function. Three distinct levels of safety integrity are used in chemical plant applications. Safety integrity levels are assigned to each potential hazardous event according to its severity. Table 1.1 shows each safety integrity level, the severity of its associated hazard, and target availability (AIChE/CCPS, 1993). The integrity level 1 is assigned to prevent minor equipment damage and insignificant production losses. The integrity level 2 is assigned to protect against damage to major equipment or substantial production costs. The integrity level 3 is assigned to the risk involving personnel injury and a significant environmental release of hazardous materials.

In the next step, the LCS technology and configuration (the way in which all components including input and output devices of the LCS are arranged and interconnected) are designed to achieve the specified safety integrity level. A number of technologies can be selected for implementation of the LCS, including electromechanical relays, modular solid state logic, programmable electronic system (PES), and hybrid systems (Rosenof and Ghosh, 1987). The technology selected should be capable of attaining the required reliability and availability criteria. Then, an acceptable configuration of the LCS is selected to achieve the target availability and reliability. Typical LCS configurations include a dual redundant system and a triple-modular redundant system (Frederickson and Beckman, 1991). Table 1.2 lists recommended system configurations for each safety integrity level (AIChE/CCPS, 1993).

The next step is safety integrity verification. Verification is the process of determining whether or not the product of the design process fulfills all the requirements specified in the previous step. Availability and reliability are evaluated using well established techniques (AIChE/CCPS, 1989). Reliability block diagrams, diagrammatic representations of the reliability characteristics of a system, are simple techniques to approximate the reliability of a system. Markov models are also widely used in analyzing the reliability of complex systems (AIChE/CCPS, 1989). A model is constructed as a discrete-state, continuous-time process, and numerical techniques are used to analyze the model. Availability analysis techniques are similar to those used in reliability analysis, except that the component/system is allowed to be repaired from a detected failed state when estimating the availability. The calculated availability and reliability are compared with target values to verify whether the configuration selected satisfies the specified integrity level. If not, another configuration should be considered.

Functional specifications define each LCS required for risk reduction and control, and define its inputs and functional actions to be taken when activated. The next step is the design of the LCS logic module. The design of the LCS logic is to transform functional specifications into its equivalent configurable and/or procedural description depending on the particular implementation technology chosen for the LCS. This transformation is generally known as *logic synthesis* in digital circuit design (Devadas et al., 1994). A digital circuit design starts from a *behavioral model* of the circuit that describes the behavioral functionality of the circuit without regard to exact bit-level behavior. The behavioral models are commonly written in programming languages. As a next step, the behavioral model is transformed into a RTL (*register-transfer level*) model that describes the operation of the circuit as synchronous transfers between functional units such as adders. Currently, RTL models are written using hardware design languages such as VERILOG (Thomas and Moorby, 1991) and VHDL (IEEE, 1987). The transformation of a behavioral model into an RTL model is very difficult in general, and almost all RTL models are presently generated manually. Finally, the RTL model is automatically translated into a configurable logic-gate level model, which is then optimized to minimize the circuit area while satisfying the speed constraints. The LCS logic design step is related to the mapping of a behavioral model into an RTL model in digital circuit design, which is performed manually.

Since the mapping of a behavioral description of a LCS into logic design is extremely difficult in general (Kern, 1990), at present designs tend to be based on informal information collected from several sources, such as national standards, more detailed company interpretations of these standards, and process specific documentation. Once the logic has been designed, formal verification against functional specifications is essential because the previous design step (transformation of specifications into designs) is not automatic. For the purposes of unambiguous information flow and communication throughout the design cycle, the functionality of the LCS logic must be represented and documented adequately and formally. The role of a formal implementation-independent representation is pivotal to the rigor of these design steps, particularily because it allows automation of many tasks, eliminating the possibility of human error when transcribing information.

Even if the functionality of the LCS is formally verified with respect to its complete set of specifications, the LCS may fail to provide an appropriate action when necessary. This can happen because the functionality of the LCS has not been verified considering its dynamic interaction with the underlying chemical process. In other words, formal verification of the LCS in isolation is necessary but not sufficient for safe operation of the overall system. Therefore, as a further step, it is desirable to verify formally the functionality of the LCS considering its dynamic coupling with the underlying chemical process. The notion of *hybrid discrete/continuous dynamic systems* (Barton and Park, 1997) can be used to formulate this problem. However, only a very limited class of hybrid systems with respect to model complexity can be verified in a formal manner (Alur et al., 1993; Kestne et al., 1993). A typical sub-problem in the verification problem is reachability analysis determining whether or not the system has a sequence of transitions from a given initial state to a final state. This reachability problem is undeciadable for very restricted classes of hybrid systems, e.g., constant slope hybrid systems, in which the right hand side of all differential equations is an integer constant. Note that dynamic models of typical chemical processes require (at least) the general nonlinear form of differential-algebraic equations. Therefore, the hybrid system of the LCS and the underlying nonlinear process cannot be verified formally. However, the performance of the LCS with its underlying process can be validated. Hybrid discrete/continuous dynamic simulation technologies can be used to study the overall system response in a set of key upset scenarios identified by the PHA. The results of such a validation step may identify problems with the original functional specifications, and thus require revision and redesign of the logic.

After successful verification, the system can then be implemented using particular hardware or software (in principle automatically, if a formal representation is used). This final step includes a field test and evaluation of the integrated system to ensure compliance with the safety, functional, performance, and interface requirements. Testing of the complete LCS is critical prior to the start-up of the plant and after any modifications have been made.

1.4 Motivations and Objectives

Much progress on reliable procedures for achieving the desired safety integrity level or minimizing random hardware failures has been made in recent years, and these are now well documented in several standards and guidelines (AIChE/CCPS, 1993; HSE, 1987; IEC, 1995; IEE, 1996). On the other hand, much less attention has been paid to the design of LCSs with respect to functionality. Most standards and guidelines have neglected the development of a rigorous design procedure to reduce systematic failures. In addition, note that most incidents listed in section 1.2 were caused by the lack of rigorous design with respect to functionality. We believe that the occurrence of such incidents could be greatly reduced if the original design or design modification could be verified formally according to the prototypical design procedure in Figure 1.10. In particular, the formal verification and dynamic validation steps will be able to reduce the occurrence of systematic failures significantly. Even though the use of formal methods is recommended for high safety integrity level LCSs (IEE, 1996) and the use of dynamic simulation technology is recommended to test thoroughly the performance of a LCS including its interaction with the underlying process prior to its actual use (AIChE/CCPS, 1993), no techniques are currently available that can verify and validate the functionality of the LCS correctly and efficiently. The current industrial practice is to rely on qualitative testing such as checklists and/or walkthroughs (HSE, 1987; IEE, 1996) and/or to use extensive simulations of the LCS to check the functionality of the LCS against its specifications. However, qualitative testing cannot replace formal verification step, and verification via extensive simulations of the LCS becomes an intractable combinatorial task for complex LCSs. Furthermore, the dynamic validation step is frequently neglected due to the lack of appropriate tools. Our studies and discussions with industry leaders indicate that there is currently an urgent demand for the development of such techniques.

The objective of this thesis is to develop a unified framework for the formal verification of LCSs. In order to accomplish this objective, two main tasks are identified, which are two major sub-problems in the functional design procedure for the LCS in Figure 1.10. The first task is to develop a formal methodology that can verify the functionality of the LCS in isolation against its formal specifications. Due to the coupled embedding of multiple functionalities in the LCS, the complexity and size of LCS applications is ever increasing. Therefore, it is important to develop a methodology that can deal with large-scale, complex, and coupled systems. A formal verification technique can eliminate any inconsistency between the design and the specification systematically, and consequently reduce the occurrence of the LCS failures. The second task is to develop a hybrid discrete/continuous simulation methodology that can validate the performance of the LCS considering its dynamic interaction with its underlying process. With such a methodology, the hybrid modeling of the LCS and its underlying process will be facilitated and the hybrid simulation will be performed correctly and efficiently. Even if the LCS is completely consistent with its specifications, as proven by formal verification, such dynamic validation technology is still valuable because it can identify problems in the original specifications. Furthermore, the development of such a technique will promote the use of dynamic validation in the overall design procedure.

1.5 Thesis Outline

The thesis is divided into two parts. Each part develops a methodology for the formulation and solution of one of the two sub-problems involved in the unified verification framework. The first part addresses the formal verification of the LCS with respect to its formal specifications. The second part addresses dynamic validation of the LCS with its underlying process.

In the first part, chapter 2 describes modeling and implementation-independent representation of the functionality of a LCS, which is the first step to achieve the objectives of the thesis. Chapter 3 develops a formal verification technology that can check the LCS against its formal specifications. The methodology developed, known as *implicit model checking*, can solve large-scale problems due to a novel model formulation and solution algorithm. Chapter 4 illustrates the capability of implicit model checking by solving industrial-scale problems.

In the second part, chapter 5 formulates the dynamic validation problem as a hybrid discrete/continuous simulation problem and develops solution algorithms. Chapter 6 develops an efficient algorithm for state event location, which is a sub-problem required to solve the dynamic validation problem.

Finally, chapter 7 presents preliminary studies on combinational logic verification using implicit model checking, and chapter 8 summarizes the main conclusions derived from this thesis and identifies directions for future research.

Chapter 2

Modeling of Logic-based Control Systems as Binary Logic Diagrams

In this chapter, a model is developed to represent the functionality of a LCS. First, the qualifications for LCS models are discussed considering their area of applications. The binary logic diagram is adopted as a LCS model. The ambiguity inherent in the functionality of a LCS represented by a binary logic diagram is identified by drawing analogies between binary logic diagrams and sequential logic systems. Finally, the problem of ambiguity in binary logic diagrams is resolved.

2.1 Desirable Properties of Models for Logic-based Control Systems

This section discusses the desirable properties for a formal representation of the functionality of a LCS, and compares several models with respect to these properties.

Note that the LCS is a class of *discrete event dynamic systems* (DEDSs). A DEDS is a system whose state space is discrete, and whose state evolution is determined by *discrete events* or transitions between states. The trajectory of a DEDS is characterized as piecewise constant states linked by events or state transitions occurring at deterministic or random instants of time. Systems encountered in the chemical process industries are conveniently modeled as DEDSs. Examples include discrete manufacturing systems, batch control systems, and start-up and shutdown systems (Alsop et al., 1996; Felder, 1983; Felder et al., 1985; Yamalidou et al., 1990). Finite state machines (FSMs), which describe discrete transitions between a finite set of states explicitly, can be used to model the behavior of a DEDS. In the case of a LCS, a FSM describes transitions between states encoded by a set of Boolean variables.

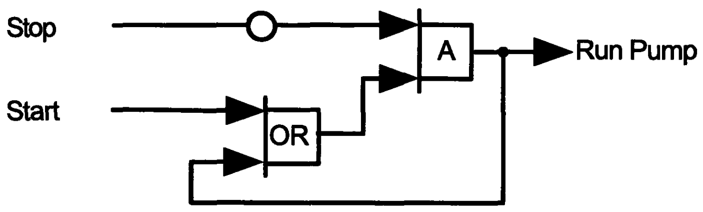


Figure 2.1: Pump interlock system

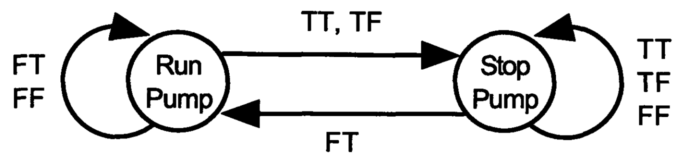


Figure 2.2: State transition graph for pump interlock system

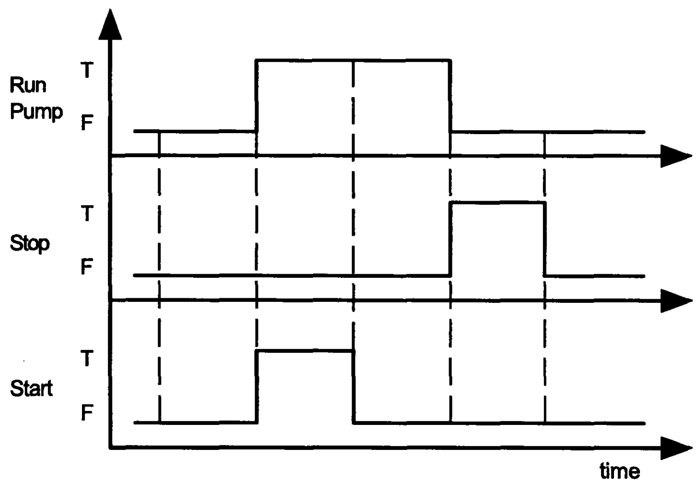


Figure 2.3: Sample trajectory of pump interlock system

As an example, consider the pump interlock system represented by a binary logic diagram in Figure 2.1. Depending upon the values of input signals `Stop` and `Start`, the pump is running or stopped, and/or the status of the pump is retained. Figure 2.2 is a finite state machine represented as a state transition graph, which describes transitions between the two states explicitly. Figure 2.3 is a trajectory of the system for a particular scenario. The pump is first turned on by pressing the `Start` button, and keeps running until the `Stop` button is pressed. Note that this trajectory can be constructed easily from the finite state machine in Figure 2.2.

2.1.1 Determinism

A crucial property of any formal representation of LCS functionality is that it be unambiguous. In particular, an implementation with unpredictable functionality as a consequence of the use of an ambiguous representation in the design is highly undesirable. Determinism is the most important property for the LCS considering its safety-critical applications. A system is said to be deterministic if a given sequence of inputs always produce the same sequence of outputs. Any sensible functional description of real-time systems should be obviously deterministic in this sense. There is no reason the designer should want its system to behave in some unpredictable manner. An example will be discussed in detail in section 2.4.3.

2.1.2 Implicitness

Any LCS model basically describes a set of states and transitions between them either explicitly or implicitly. Explicit models enumerate the states and transitions between them explicitly, e.g., state transition graphs, while implicit models will describe the relationship between states and transitions without referring to them explicitly, e.g., a system of Boolean equations (see chapter 3). As an example, for the pump interlock system above, the state transition graph of Figure 2.2 is an explicit model while the binary logic diagram of Figure 2.1 is an implicit model. Alternatively, an implicit model in terms of Boolean equations can be derived from the binary logic diagram as in Eqn (2.1):

$$\text{Run\_Pump}(t_k) \leftrightarrow \neg\text{Stop}(t_k) \wedge \left(\text{Start}(t_k) \vee \text{Run\_Pump}(t_{k-1})\right) \tag{2.1}$$

where the signal Run_Pump is regarded as a state variable, and t_{k-1} and t_k represent previous and current time respectively. Note that the implicit model for the pump interlock system, the Boolean equation and the binary logic diagram, do not refer to any particular state or transition explicitly. Instead, they state the strongest invariant property of each state variable in a closed form. In order to retrieve a particular state and transition, an additional algorithmic procedure (e.g., logic simulation) should be applied to the implicit models.

Note again that the LCS is a class of discrete event dynamic systems. Figure 2.4 depicts state spaces for discrete event and continuous dynamic systems. Compared to the infinite state space of continuous dynamic systems, the state space of discrete event dynamic systems is finite but combinatorial, i.e., the maximum number of discrete states is exponential in the number of state variables, although all may not be reachable in a particular system. This exponential growth of the state space is commonly known as the *state explosion problem*. Because of this state explosion problem, it is impossible to include all states of a LCS of reasonable size in any explicit model. For example, Figure 2.5 shows a state transition graph of a discrete event dynamic system containing 4 state variables. Note that there are 16 reachable states. It can be seen that the number of states will increase exponentially with the problem size or the number of state variables. Consequently, any explicit model can represent only a subset of all possible states. On the other hand, implicit models can represent the set of all reachable states and possible transitions in a compact form regardless of the problem size. For example, the size of the implicit model in terms of Boolean equations (or the number of logical propositions) increases linearly in the number of state variables, as will be proved in chapter 3.

The inclusion of all reachable states and possible transitions in the model is particularly important for the purposes of analysis including simulation and formal verification because the functionality (or the subset of states) relevant to a particular situation is never known in advance. In particular, formal verification requires all possible states and transitions to be checked. In this respect, the implicit formalism is preferred over the explicit formalism.

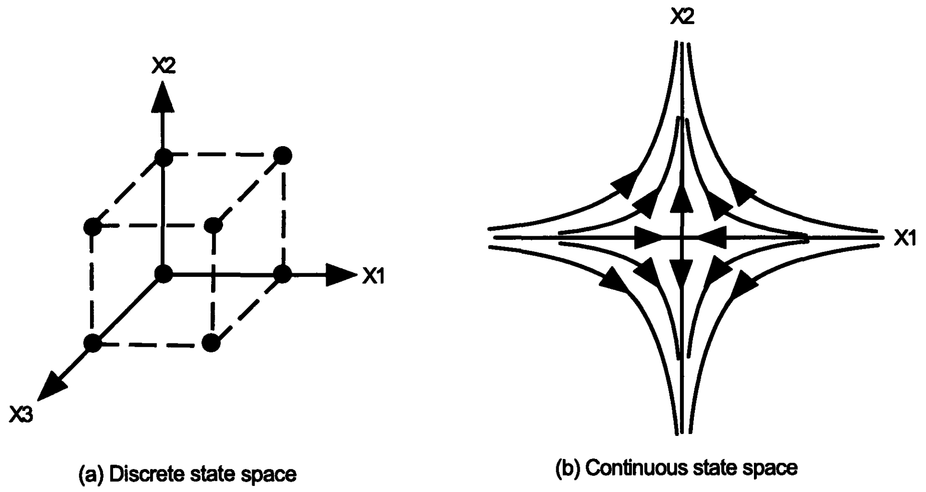


Figure 2.4: State spaces of discrete and continuous dynamic systems

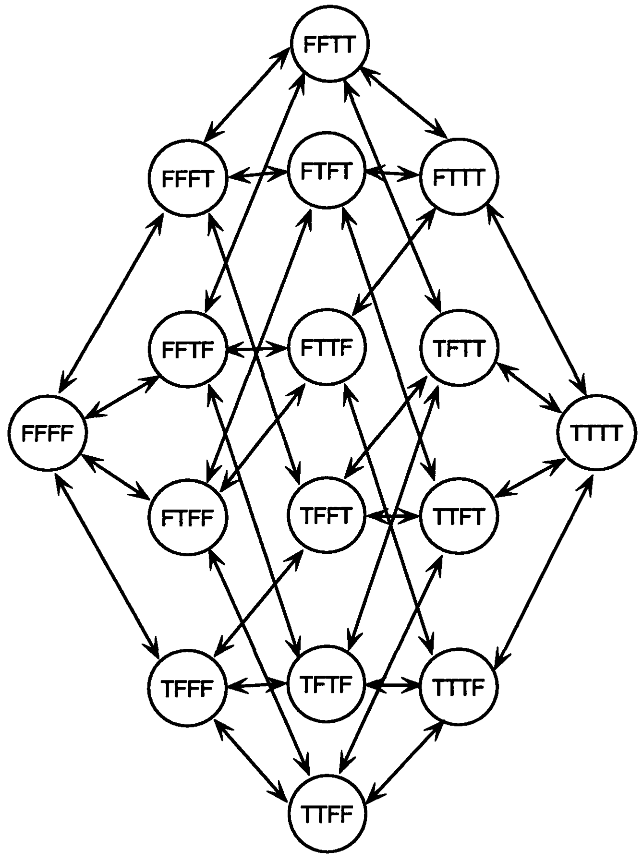


Figure 2.5: State transition graph of a discrete event dynamic system with 4 state variables

2.1.3 Declarativity

The size and complexity of LCS applications is increasing due to the coupled embedding of multiple functionalities in the LCS. Therefore, it is important to develop a LCS model suitable for modeling of complex and large-scale application. There are at least two types of modeling framework: imperative and declarative. Imperative or procedural models include the notion of control, and thus types of variables, evaluation orders, etc. are already determined. However, there is neither the notion of control nor that of sequentiality in declarative models.

Declarative models have several advantages over imperative models for large-scale modeling. First, the declarative modeling framework allows easy, incremental, and modular descriptions of systems, while within the imperative modeling framework the slightest modification to the system specifications may involve a complete modification and rewriting of the entire model. Second, the reuse of the same model for other applications is straightforward in the declarative modeling framework. Third, modular and hierarchical notions are directly supported within the declarative modeling framework. Fourth, an additional specification in the form of assertions (e.g., for formal verification purpose) is easy to incorporate within the model. Finally, the only constraints on the evaluation order arise from the dependencies between variables. As a consequence, any implementation (hardware independent), be it sequential or highly parallel, can be easily derived.

As an example, consider the pump interlock system. The binary logic diagram of Figure 2.1 and the Boolean equation of Eqn (2.1) both correspond to a declarative model. There is no notion of procedure in these models. These declarative models naturally support modular and hierarchical modeling as will be demonstrated in section 2.3. For the pump interlock system, it is reasonable to impose an additional constraint in order to prevent the input signals `Stop` and `Start` from being TRUE simultaneously. Eqn (2.2) is a modified model of Boolean equations after adding this additional constraint:

$$\begin{gathered}\text{Run\_Pump}(t_k) \leftrightarrow \neg\text{Stop}(t_k) \wedge \left(\text{Start}(t_k) \vee \text{Run\_Pump}(t_{k-1})\right) \\ \neg\left(\text{Stop}(t_k) \wedge \text{Start}(t_k)\right)\end{gathered} \tag{2.2}$$

```
Program Pump_Interlock_1
BEGIN
   LOAD Stop
   Z1 := NOT Stop
   LOAD Start
   LOAD Run_Pump
   Z2 := Start OR Run_Pump
   Run_Pump := Z1 AND Z2
   STORE Run_Pump
END
```
(a) procedural model of pump interlock of Eqn (2.1)

```
Program Pump_Interlock_2
BEGIN
   LOAD Stop
   Z1 := NOT Stop
   LOAD Start
   LOAD Run_Pump
   IF NOT (Stop AND Start) THEN
      Z2 := Start OR Run_Pump
      Run_Pump := Z1 AND Z2
      STORE Run_Pump
   ELSE
      Error_Message
   END
END
```
(b) procedural model of pump interlock of Eqn (2.2)

Figure 2.6: Imperative model for pump interlock system

On the other hand, Figure 2.6 (a) and (b) show procedural models for the pump interlock system before and after adding the additional constraint respectively. Note that the introduction of the additional constraint requires rewriting of the model even for this simple example.

2.1.4 Concurrency

The LCS is a reactive system that continuously reacts to the process it is controlling at a speed determined by the process. This reactive nature of the LCS operation requires parallel interaction between the LCS and the process so that events can be processed simultaneously. The nature of this interaction demands concurrency (or parallelism) to be supported within the LCS model. A purely sequential model will be limited in its application because of parallel interaction between the LCS and its environment. Note that the concurrent or sequential property is with respect to expressiveness or representation of the model and is independent of any execution scheme. The concurrent model can be efficiently executed in a sequential manner.

For example, concurrency is supported in the binary logic diagram for the pump interlock in Figure 2.1. Note that there is no restriction on the processing of transitions of the input signals `Stop` and `Start`. On the other hand, the procedural models in Figure 2.6 do not support concurrency, and are represented as a sequence of statements, which will be executed sequentially.

Table 2.1: Comparison of models for logic-based control systems

Models	Determinism	Implicitness	Declarativity	Concurrency
Programming Language	Supported	Supported	Not supported	Not supported
Deterministic FSMs	Supported	Not supported	Not supported	Not supported
Petri Nets or GRAFCET	Not supported	Not supported	Not supported	Supported
Binary Logic Diagrams	Not supported	Supported	Supported	Supported
Ladder Logic Diagrams	Not supported	Supported	Supported	Supported

2.2 Comparison of Models for Logic-based Control Systems

Table 2.1 compares various models for LCSs with respect to the properties discussed in the previous section. Programming languages can be considered as the most flexible model for LCSs. However, they do not support declarativity and concurrency. Furthermore, they are not suitable for large-scale programs since they make understanding of behavior and analysis almost impracticable.

Deterministic finite state machines are efficient in terms of execution, and are well-known mathematical objects for which analysis techniques are available. However, they do not support concurrency; they are purely sequential. Furthermore, they are explicit and imperative. Furthermore, they are *flat* objects without any hierarchical or parallel structure.

Consequently, they are not suitable to model complex large-scale systems. Writing a finite state machine with even a small number of states is a difficult and error-prone task.

Petri Nets or GRAFCET naturally support concurrency, but they lack modular structure. Because of this lack of hierarchy, it is difficult to apply them to large-scale systems. Even though two extensions of GRAFCET, *macrostep* and *forcing order*, offer interesting possibilities for the structuring and the hierarchicalization of the models (Lesage and Roussel, 1993), the semantics of these extensions is not well-established yet (André and Péraldi, 1993). They are also imperative and explicit. These formalisms are commonly used for comparatively small applications, and do not scale up well to large-scale applications (David, 1993). They often lack determinism. For example, the evolution rules of GRAFCET are not sufficient to guarantee the unicity of interpretation of a GRAFCET (André and Péraldi, 1993).

Binary logic diagrams (BLDs) (ANSI/ISA, 1981) are currently used to represent the functionality of the LCS by many companies throughout the chemical industries. A major advantage of BLDs is that they are not tied to a particular brand of hardware or software, and their transformation into a particular implementation is straightforward (AIChE/CCPS, 1993; Fisher, 1989). Furthermore, BLDs support implicitness, declarativity, and concurrency. However, BLDs *may* be non-deterministic in some cases.

Note that ladder logic diagrams (Otter, 1988) are equivalent to binary logic diagrams with respect to the criteria used in Table 2.1. However, the ladder logic diagrams are tied to the peculiarities of electromechanical relay circuits. For the purposes of this thesis, binary logic diagrams will be used instead of ladder logic diagrams.

The ambiguous behavior of the BLD is briefly described in the next section and is corrected by identifying the causes of non-deterministic behavior in subsequent sections. The notion of revised BLDs will be adopted to represent the functionality of the LCS.

2.3 Binary Logic Diagrams

LCS logic represented as BLDs are interconnections between logic building blocks, which operate on and produce logic signals. Figure 1.5 shows an example of LCS logic represented using a BLD. The flow of logic is represented by lines that interconnect logic blocks. Topologically, the binary logic diagram is a directed graph in which the logic building blocks and signals are mapped, respectively, into nodes and edges of the graph. Figure 2.7 shows some basic building blocks of BLDs. These can be classified into three categories:

- Combinational Logic Element - AND, OR, NOT.
- Memory Element - SRFF (Set-Reset Flip-Flop), T (Toggle Flip-Flop), etc.
- Timing Element - DI (Delay Initiation), DT (Delay Termination), etc.

Combinational logic elements establish the logical relationship between input and output signals, memory elements retain information concerning the previous state of the system, and timing elements establish temporal relationships between logic signals.

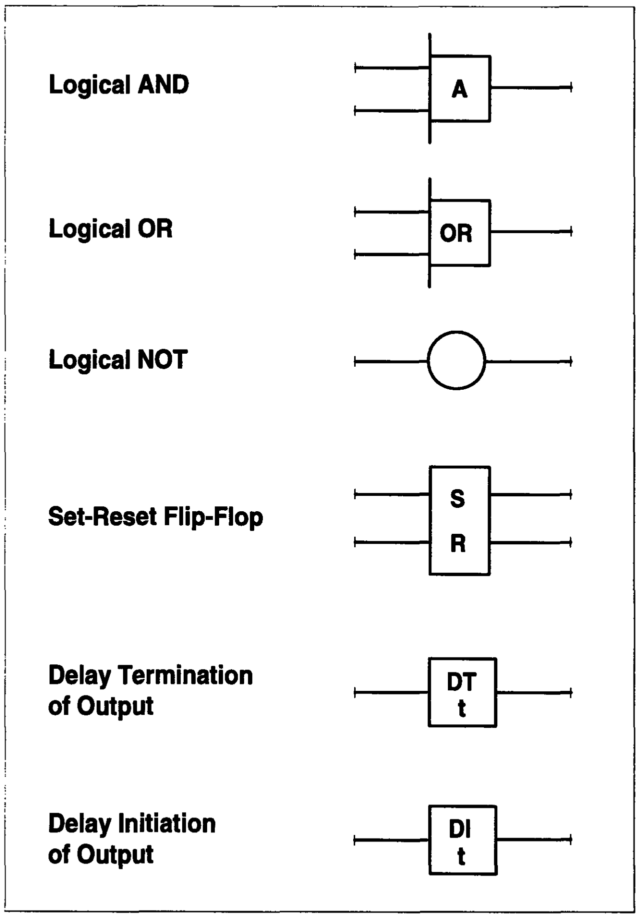


Figure 2.7: Basic building blocks of binary logic diagrams

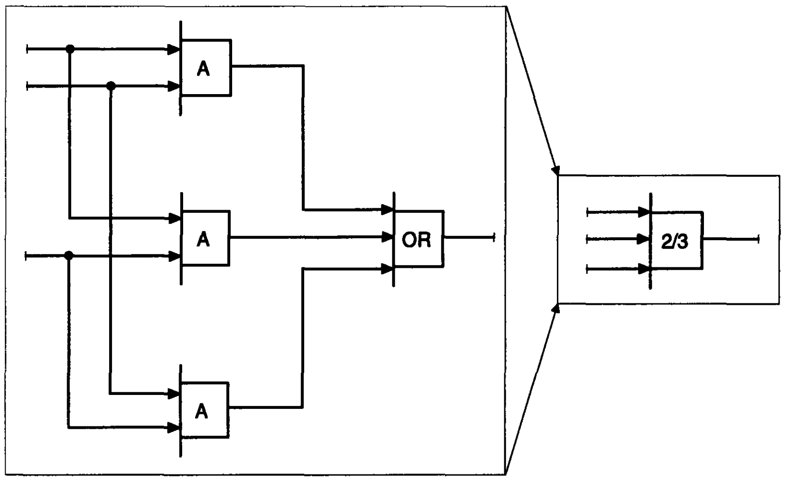


Figure 2.8 : Realization of 2-out-of-3 voting logic

For the purposes of modeling, the identification of a complete set of primitive elements for each category is essential, so that more complex elements can be defined in terms of these primitive elements through the introduction of the notion of hierarchical sub-model decomposition (Elmquist, 1978). This is illustrated in Figure 2.8, which shows the realization of 2-out-of-3 voting logic in terms of primitive elements. Given this definition, the 2-out-of-3 voting block may now be considered as a component that may be inserted in a larger structure. A complete set of primitive elements for combinational logic is $\{\text{AND},\text{OR},\text{NOT}\}$. Memory elements are sequential logic systems that can be implemented by combinational logic elements connected with feedback loops. For timing elements, ANSI/ISA-S5.2 (1981) proposes various kinds of timing elements.

2.3.1 Non-determinism in Binary Logic Diagrams

Most binary logic diagrams for industrial applications contain feedback logic signals, and certain binary logic diagrams containing feedback paths represent a class of several different possible functionalities, rather than a unique predictable functionality, due to the existence of *hazards* and *races*. A major consequence of this fact is that two separate implementations based on the same binary logic diagram can exhibit different functional behaviors, neither of which can be guaranteed to conform with the original functional specifications. For example, the same binary logic diagram can be translated into two different programmable logic controller (PLC) programs implementing functionalities that depend on the order in which the logic blocks are solved in each controller (e.g., from left to right and from top to bottom of the diagram, or vice versa). Current industrial practice is to implement a convenient functionality (from the point of view of implementation) selected from the multiple functionalities embedded in a binary logic diagram, and to test validity against specifications through extensive simulations of the LCS in isolation (known as factory testing), changing the logic design if necessary. For complex logic-based control systems, validation via simulation quickly becomes an intractable combinatorial task because the number of simulations required for complete validation is increasing exponentially with the problem size or the number of state variables and inputs, and even for smaller problems appears to be very crude and inefficient if extensive redesign of the logic becomes necessary. It should further be noted that simulation can only be used for validation purposes, and does not amount to formal verification. On the other hand, a binary logic diagram that is guaranteed to be unambiguous obviates the unnecessary factory testing step in the design cycle, since, potentially, the binary logic diagram can be used for all modifications to the design, formal verification of the design after each modification, and then automated implementation from the verified BLD.

In the subsequent sections, the ambiguous behavior of the BLD will be analyzed by drawing analogies between BLDs and the well documented properties of sequential logic systems.

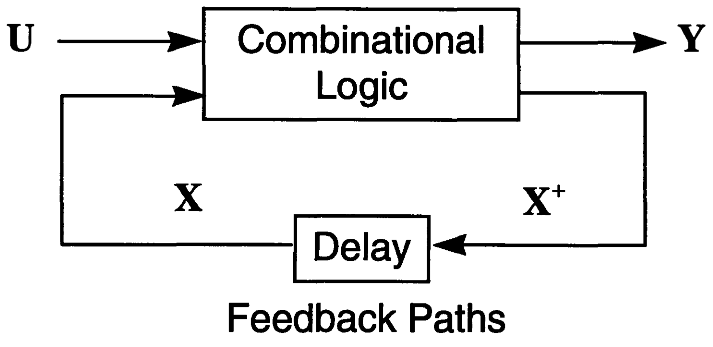


Figure 2.9 : Huffman model of sequential logic systems

2.4 Sequential Logic Systems

Most BLDs contain feedback logic signals, either because memory blocks contain implicit feedback signals internally, or because it is still common industrial practice to use explicit feedback signals to represent memory. These properties make binary logic diagrams equivalent to *sequential logic systems.*

Sequential logic systems (Abramovici et al., 1990) are defined as logic systems whose outputs $\mathbf{Y}$ depend both on the present input signals $\mathbf{U}$ and the past history of input signals condensed in state variables $\mathbf{X}$. Sequential logic systems are widely used to model digital circuits at the logic level. The *total* state of a sequential logic system is completely specified by a set of values for inputs and state variables. The properties of sequential logic systems can be conveniently studied through use of the *Huffman model* (Miczo, 1986) depicted in Figure 2.9, which consists of a combinational part and feedback paths, which pass through delay elements and then act as additional inputs to the combinational part. The delay elements may be delay inherent in the implementation of the logic blocks (*stray delays*) or may be inserted to ensure the proper operation of the system (*inserted delays*). If the values of all feedback signals (state variables) are changed at the same time, the system is called a *synchronous sequential logic system.* Otherwise, it is called an *asynchronous sequential logic system.* A sequential logic system has reached a *stable* state if none of the state variables are changing with the primary input held constant. The set of stable states characterizes the *static* or long term behavior of the system, while the *transient* behavior of the system includes both stable and unstable states of the system. An asynchronous sequential logic system is said to be operating in a *fundamental mode* if and only if the primary inputs are never changed unless the system is in a stable state (Friedman, 1986). Fundamental mode operation is assumed throughout this thesis, and this will be justified in chapter 3.

Binary logic diagrams are asynchronous sequential logic systems, i.e., there in which feedback signal transitions are not synchronized. This property of asynchronicity arises due to the fact that the binary logic diagram is an implementation-independent representation, and hence does not contain any information concerning the processing of transitions in feedback signals, even though a particular implementation may or may not be synchronized. In order to describe an asynchronous sequential logic system with a

Huffman model, a set of *state branches* (or feedback paths), which define the secondary outputs $\mathbf{X}^+$ and the state variables $\mathbf{X}$, must be specified. It is necessary that a feasible set of state branches should have at least one member in every closed path of the circuit (Unger, 1959). In other words, the primary output $\mathbf{Y}$ and secondary output $\mathbf{X}^+$ can be expressed as a combinational function of the primary input $\mathbf{U}$ and state variable $\mathbf{X}$ by selecting a feasible set of state branches. In general, there are multiple feasible sets of state branches, with varying numbers of state branches (Unger, 1959). More than one state branch can be located in the same feedback loop. One possible solution is to find a minimal set of state branches (or a minimal feedback cut set). However, even the minimal feedback cut set is not unique for a given circuit (Unger, 1959). In general, each model derived from a different feasible set of state branches will show different static and transient behavior due to the existence of *hazards* and *races*.

2.4.1 Hazard and Race

A logic system is said to contain a *hazard* if there exists some possible permutation of values for stray delays that will produce a spurious pulse or cause the system to enter an incorrect stable state for an input transition (Friedman, 1986). A hazard represents only a possibility of malfunction in a worst case. A physical system may not malfunction even though a hazard exists. Hazards are associated with the configuration or structure of a system, not with physical systems. There are two types of hazard, *transient hazards* and *steady-state hazards* (Friedman, 1986). A transient hazard is the possibility of the occurrence of a momentary value on the output opposite to that expected for an input transition. A logic system is said to contain a steady state hazard if there exists some distribution of stray delays such that the system may reach an incorrect stable state for an input transition. In other words, a steady state hazard is the possibility of more than one stable state for a given input transition, so that the steady state actually reached will be a function of the delays.

A *race* is a condition where two or more signals are changing simultaneously in a circuit (Abramovici et al., 1990). The race may be caused by two or more simultaneous input signal changes, or it may be the result of a single signal change that traverses two or more signal paths upon arriving at a *fanout* point (a logic signal is said to have *fanout* if it has more than one destination). Races may or may not affect the behavior of a system. A *critical race* exists if the final stable state that the system reaches is a function of either the order in which the state variables change, or delays in the circuit. A *non-critical race* does not affect the behavior of the system. A critical race can be considered to be a steady state hazard.

2.4.2 Hazard and Race Detection

For a given input transition, a systematic procedure has been proposed to detect a steady state hazard including critical races (Eichelberger, 1965). This procedure can be applied to any asynchronous sequential logic system. Only system configuration is necessary to apply this procedure, and any feasible feedback cut set can be chosen for the purposes of simulation because the results of the detection procedure do not depend upon the location and number of the feedback paths. The system is analyzed for a steady state hazard including a critical race for an input transition $\mathbf{U}(t_k) \to \mathbf{U}(t_{k+1})$. In order to apply this procedure, the total state of the system at t_k, $\{\mathbf{U}(t_k), \mathbf{X}(t_k)\}$, must be known. Fundamental mode operation is assumed in this procedure. The procedure consists of two parts. Procedure A determines all the feedback signals that may be changing as a result of the input transition, and Procedure B determines whether or not these feedback signals will eventually stabilize in some predetermined state.

Procedure A: Transition from t_k to $t_{k+1/2}$

1. For $\forall i$, if $U_i(t_k) \neq U_i(t_{k+1})$, then $U_i(t_{k+1/2}) = u$ else $U_i(t_{k+1/2}) = U_i(t_k)$.
2. Evaluate $\mathbf{X}^+(t_{k+1/2})$, $\mathbf{X}(t_{k+1/2})$ given $\mathbf{U}(t_{k+1/2})$ and $\mathbf{X}(t_k)$ as the initial value of $\mathbf{X}(t_{k+1/2})$ until $\mathbf{X}^+(t_{k+1/2}) = \mathbf{X}(t_{k+1/2})$.

Procedure B: Transition from $t_{k+1/2}$ to t_{k+1}

1. Evaluate $\mathbf{X}^{+}\left(t_{k+1/2}\right)$, $\mathbf{X}\left(t_{k+1/2}\right)$ given $\mathbf{U}\left(t_{k+1}\right)$ and $\mathbf{X}\left(t_{k+1/2}\right)$ as the initial value of $\mathbf{X}\left(t_{k+1}\right)$ until $\mathbf{X}^{+}\left(t_{k+1}\right)=\mathbf{X}\left(t_{k+1}\right)$.

Theorem:

1. If $X_i\left(t_{k+1}\right)$=0 or 1 for $\forall i$, then the transition from $\left\{\mathbf{U}\left(t_k\right),\mathbf{X}\left(t_k\right)\right\}$ to $\mathbf{U}\left(t_{k+1}\right)$ determines $\mathbf{X}\left(t_{k+1}\right)$ uniquely regardless of the distribution of delays.
2. If $\exists i$ such that $X_i\left(t_{k+1}\right)=u$, then the transition from $\left\{\mathbf{U}\left(t_k\right),\mathbf{X}\left(t_k\right)\right\}$ to $\mathbf{U}\left(t_{k+1}\right)$ cannot determine $\mathbf{X}\left(t_{k+1}\right)$ uniquely. The state of the system or $\mathbf{X}\left(t_{k+1}\right)$ cannot be predicted from the configuration alone due to the existence of steady state hazards or critical races.
3. For a system with m state variables, at most only $2m$ simulation passes (m for each step) are required.

Ternary logic simulation (Abramovici et al., 1990) is used to evaluate state variables during each procedure. The unknown logic value u is processed together with the binary logic values by the truth tables shown in Table 2.2. Note that this procedure is a worst-case analysis. Even though the final value of $X_i\left(t_{k+1}\right)$ computed is u, a critical race or an oscillation may not occur for a physical circuit because it depends on the actual delay.

Even though the above procedure that detects a steady state hazard for a single input transition terminates in polynomial time, a complete test of a sequential logic system against hazards and races is a combinatorial task since a lower bound on the number of possible input transitions for a sequential logic system is given by $2^{N_U}(2^{N_U}-1)$ where N_U is the number of inputs. This lower bound admits the possibility that an initial set of inputs corresponds to a memory state, and hence the initial state variables may have multiple sets of values depending on the past history of the system. The transition from $\left(\mathbf{U}\left(t_k\right),\mathbf{X}\left(t_k\right)\right)$ to $\left(\mathbf{U}\left(t_{k+1}\right),\mathbf{X}\left(t_{k+1}\right)\right)$ can be excluded from consideration if $\mathbf{X}\left(t_{k+1}\right)$ can be uniquely determined by $\mathbf{U}\left(t_{k+1}\right)$ alone. It is only necessary to apply the hazard and race detection procedure to those transitions for which $\mathbf{X}(t_{k+1})$ cannot be determined uniquely by $\mathbf{U}(t_{k+1})$, i.e., memory states of the system. Further, in order to apply the detection procedure, the total state of the system at t_k, $\{\mathbf{U}(t_k), \mathbf{X}(t_k)\}$ must be known (i.e., the initial state). However, if the input $\mathbf{U}(t_k)$ cannot determine $\mathbf{X}(t_k)$ uniquely, then the total state at t_k is unknown. The transitions starting from these memory states are therefore excluded from consideration. The following complete test algorithm for a sequential logic system against hazards and races is devised.

Hazard and race detection algorithm:

1. Select an input $\mathbf{U}(t_k)$ as an initial state and evaluate $\mathbf{X}(t_k)$ using ternary logic simulation. If $\mathbf{X}(t_k)$ is uniquely determined, i.e., $\forall i,\ X_i(t_k)=0$ or 1, then go to step 2, otherwise select another input as an initial state until $\mathbf{X}(t_k)$ is uniquely determined.
2. Select an input $\mathbf{U}(t_{k+1})$ as a final state and evaluate $\mathbf{X}(t_{k+1})$ using ternary logic simulation. If $\mathbf{X}(t_{k+1})$ is uniquely determined, then the transition from $(\mathbf{U}(t_k), \mathbf{X}(t_k))$ to $(\mathbf{U}(t_{k+1}), \mathbf{X}(t_{k+1}))$ does not cause hazards and races, otherwise apply the detection procedure (Eichelberger, 1965) to this transition.
3. Repeat step 2 until all input transitions starting from $(\mathbf{U}(t_k), \mathbf{X}(t_k))$ are considered (i.e., all permutations for $\mathbf{U}(t_{k+1}) \neq \mathbf{U}(t_k)$).
4. Repeat step 1 through step 3 until all input permutations are considered as an initial state in step 1 (i.e., all permutations for $\mathbf{U}(t_k)$).

Since we exclude initial states that are memory states in step 1, it is important to note that this procedure can only be used to confirm the existence of a steady-state hazard. It cannot, for example, be used to prove that a hazard does not exist in a given configuration.

Table 2.2: Truth tables for ternary logic simulation

AND	0	1	u
0	0	0	0
1	0	1	u
u	0	u	u

OR	0	1	u
0	0	1	u
1	1	1	1
u	u	1	u

NOT	0	1	u
	1	0	u

2.4.3 Hazard and Race Detection Examples

The previous section has discussed the hazards and races in sequential logic systems. More specifically, the steady-state hazards are properties of the logic configuration alone, and admit the possibility of several different responses to a given input transition. Given this set of possible responses, the specific response exhibited by an implementation of a configuration will depend on the magnitude and position of delays in the implementation. In this section we present examples of two BLDs exhibiting steady-state hazards.

2.4.3.1 Set-Reset Flip-Flop

The purpose of this example is to illustrate that different Huffman models resulting from the same binary logic diagrams (i.e., configuration) can exhibit different functional behavior for the same input transition. Figure 2.10 shows a standard set-reset flip-flop (SRFF) constructed by OR and NOT blocks with feedback. The two Huffman models resulting from two minimal feedback cut sets are shown in Figure 2.11 (each model amounts to assuming that the largest delay in the system is located in a different position in the BLD). The two models predict different responses to the input transition $(S,R)\text{:}(1,1) \rightarrow (0,0)$ as shown in Table 2.3. This is because the configuration for the SRFF shown in Figure 2.10 contains a critical race condition. Given the configuration alone, there is no way to determine which of these two responses was the one intended by the functional specifications.

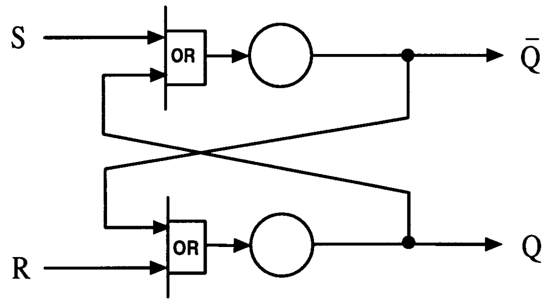


Figure 2.10: Set-Reset Flip-Flop

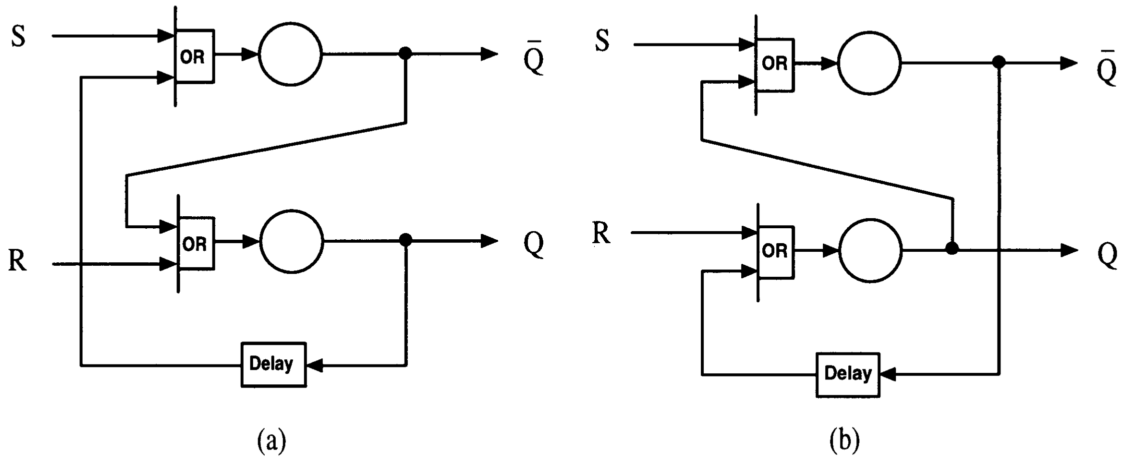


Figure 2.11: Two Huffman models of SRFF

Table 2.3: Critical race condition in SRFF

	S	R	Q	$\overline{Q}$
Initial State	1	1	0	0
Final State (Figure 2.11a)	0	0	0	1
Final State (Figure 2.11b)	0	0	1	0

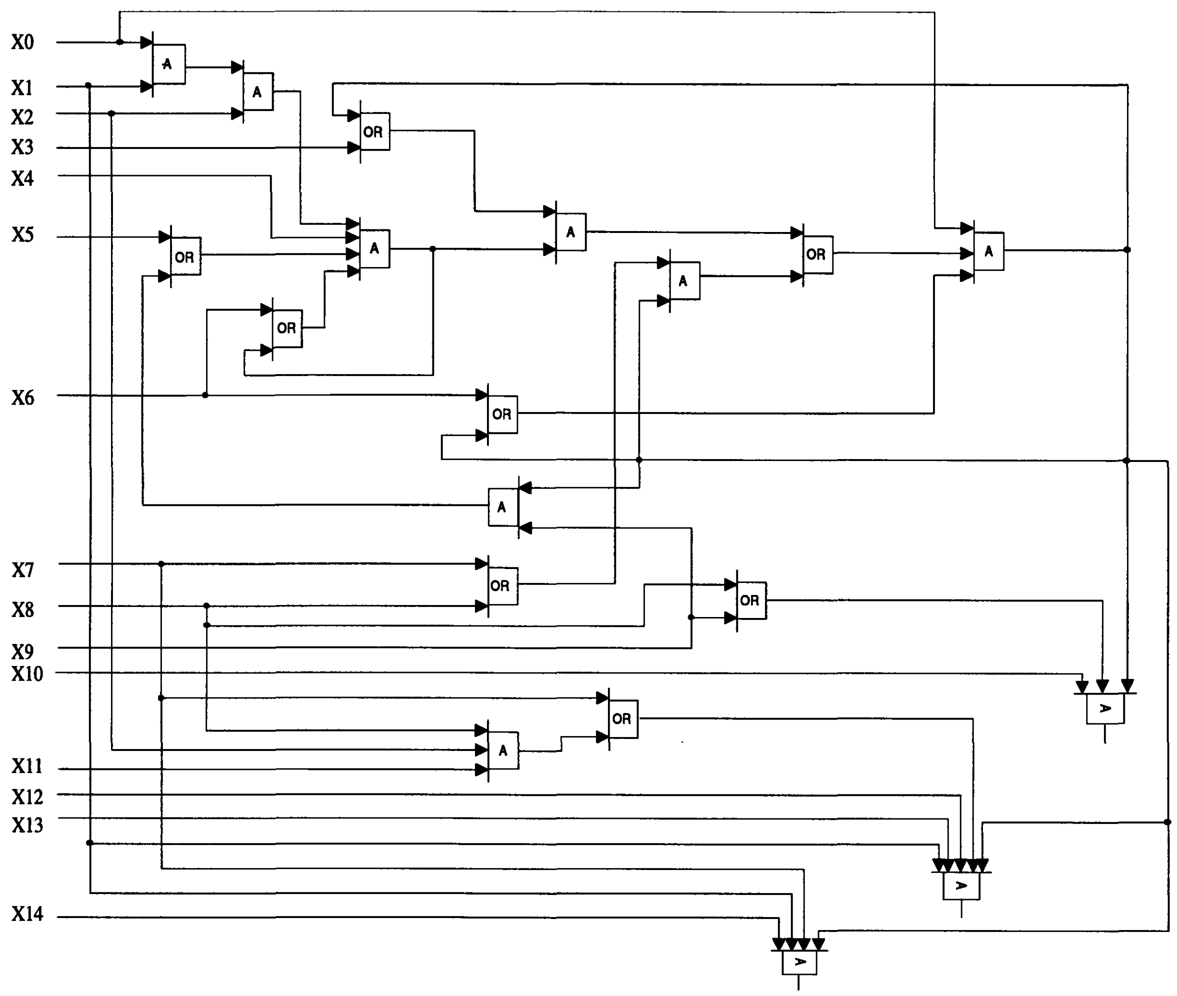


Figure 2.12: Burner control interlock system

Table 2.4: Hazard and race statistics for burner control interlock system

number of input signals that change simultaneously	lower bound on percentage of input transitions that cause steady state hazard and/or critical races
1	0.00 %
2	0.13 %
5	1.60 %
10	13.20 %
15	61.01 %

2.4.3.2 Burner Control Interlock System

The purpose of this example is to demonstrate that binary logic diagrams employed in industry can contain hazard and race conditions. The algorithm presented in section 2.4.2 is applied exhaustively to the burner control interlock configuration shown in Figure 2.12 (Watkins, 1993). Table 2.4 summarizes the results, which show clearly that the binary logic diagram for the burner interlock system embeds multiple responses for some input transitions due to the existence of steady state hazards and critical races. It is interesting to note that hazard and race conditions do not occur for input transitions that involve only one signal change, which is probably a consequence of exhaustive factory testing and refinement of this configuration.

2.5 Revised Binary Logic Diagrams

Above, the binary logic diagram is analyzed for the possibility of ambiguous statements of functional behavior by drawing analogies between BLDs and asynchronous sequential logic systems. Due to the existence of non-unique feedback cut sets or several possible Huffmann models, binary logic diagrams exhibits ambiguous responses when the transition corresponds to a steady-state hazard condition. The consequence of this observation is that certain BLD configurations represent a class of possible functionalities, rather than one specific functionality. When implementing a LCS from a BLD, a somewhat arbitrary choice of one functionality from this class is implicitly made. For example, given a configuration, the scanning sequence implemented by a specific brand of PLC will correspond to the choice of one particular feedback cut set. In particular, there are no guarantees that this arbitrarily selected functionality corresponds to the functionality originally intended. Clearly, this ambiguity is highly undesirable, especially in safety-related applications.

Even though the proposed hazard and race detection algorithm can indicate the existence of ambiguity in BLDs, it cannot be used to prove that no ambiguity exists. Therefore, additional specifications on BLDs are necessary in order to guarantee that it is unambiguous declaration of functionality.

Ambiguity in the BLD can be eliminated by forbidding the use of explicit logic feedback to symbolize memory, and instead requiring the use of memory blocks. However, the use of memory blocks only allows us to prevent hazard and race conditions through an *a priori* analysis of the memory block. For example, the critical race condition for the SRFF in Figure 2.10 can be excluded by adding the assertion $\neg(S \wedge R)$ to the BLD. In other words, in order to avoid the possibility of the transition $(S,R){:}(1,1) \rightarrow (0,0)$ ever occurring, the input signals are forbidden from both being in the ON position simultaneously. If both input signals to a SRFF block are also inputs to the overall BLD, this assertion is easily satisfied by the common practice of selector-type switches that forbid the start and stop positions from being selected simultaneously. However, if the input signals to a SRFF block are internal signals of the overall BLD, the above assertion must instead be enforced by design of the logic upstream of the SRFF.

The use of explicit logic feedback is also deprecated in the appendix of the BLD standard (ANSI/ISA, 1981), where the stated purpose of deprecation is to prevent memory loss in the event of loss of power supply. The analysis of hazards and races in BLDs provides strong new evidence to forbid the use of explicit logic feedback. In addition, any standard should explicitly state the assertion that must be satisfied by the input signals to each type of memory block.

Another option is to specify feedback paths as additional necessary information. Then, the BLD will exhibit a unique functional behavior because it contains a unique feedback cut set. In other words, no BLD is a complete representation of a logic-based control system without a specification of a feasible set of state branches corresponding to the intended functionality.

In this thesis, we will adopt the second option. The feedback cut set must be added to any BLD that contains any explicit feedbacks and that have multiple feedback cut sets. Then, the revised BLD will embed a unique representation of the functionality of the LCS.

Note that specifying feedback paths in binary logic diagrams is equivalent to specifying the evaluation order in ladder logic diagrams. Therefore, ladder logic diagrams with a specific evaluation order will embed a unique representation of the functionality of the LCS in a similar fashion.

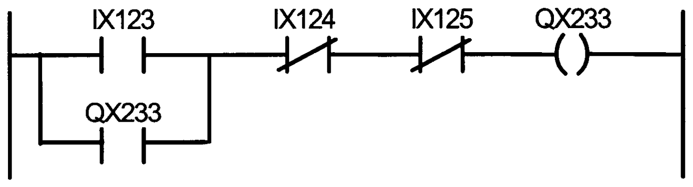
Instruction List

```
LD      IX123
OR      Qx233
ANDN    IX124
ANDN    IX125
ST      QX233
```

Structured Text

```
QX233 := (IX123 OR QX233) & NOT IX124 & NOT IX125
```

Ladder Diagram



Function Block

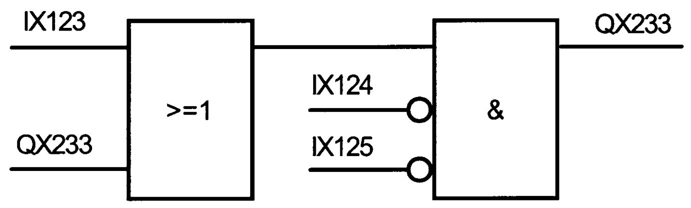


Figure 2.13: Examples of secondary languages in IEC Standard 1131

2.6 IEC Standard Programming Languages for Programmable Controllers

IEC Standard 1151 (IEC, 1993) specifies the syntax and semantics of a unified suite of programming languages for PLCs (Programmable Logic Controllers). The proposed language is a hierarchy of languages with the primary language called SFC (Sequential Function Chart) and four secondary languages. The secondary languages consist of two textual languages, IL (Instruction List) and ST (Structured Text), and two graphical languages, LD (Ladder Diagram) and FB (Function Block).

The SFC language provides a means of partitioning a program into a set of steps and transitions interconnected by direct links. The concept of a step is introduced to describe the various static behaviors of the system. With each step one or more actions (outputs) are associated. The concept of a transition is introduced to describe a possible evolution of the active state from one step to another routed by a directed link. With each transition a transition condition (input) is associated. The SFC can be combined with any of the four secondary languages. The secondary languages can be used to represent SFC elements such as transition conditions and actions.

Figure 2.13 shows the same PLC function represented in each of the four secondary languages. The instruction list is composed of a sequence of instructions, which is similar to assembly language used in computers. The structured text is derived from high-level programming languages. The ladder diagram is practically identical to that used by most PLCs. The function block is similar to the binary logic diagram. Each of the four secondary languages has its advantages and limitations.

The two graphical languages allow the use of feedback paths and associated feedback variables to represent the retentive nature of the system. Therefore, there is a possibility of non-unique interpretations without an additional specification even though the standard does not explain this explicitly. The order of evaluation is forced in the ladder diagram to prevent non-unique interpretations. However, in the function block, the standard allows the use of feedback paths with or without an explicit specification of feedback variables. The function block in Figure 2.13 corresponds to the function block with an explicit specification of feedback variables. The function block without an explicit specification of feedback variables can lead to the same non-unique interpretations as the BLD.

2.7 Conclusions

The desirable properties for LCS models are identified. Due to its safety critical application, the model for the LCS should be deterministic. Implicitness of the model is essential to capture all the functional behavior of the LCS without suffering from the state explosion problem. Declarativity is important to model a complex large-scale system in a systematic and hierarchical manner. Finally, the parallel interaction between the LCS and its underlying process requires concurrency in the model.

The BLD is selected to represent the model of the LCS, which has been widely used to document the functionality of the LCS. The BLD supports implicitness, declarativity, and concurrency. However, certain class of BLDs does not support determinism. Analyzing the BLDs as sequential logic systems, it is identified that the cause of the non-deterministic behavior is due to the existence of steady state hazards in the BLDs. An algorithm is developed to confirm the existence of steady state hazards in the BLDs. The use of additional specifications to eliminate the ambiguity in this class of BLDs is advocated. In this thesis, the explicit location of feedback paths is added to the standard notion of BLDs. The revised BLD is fully deterministic, and embeds a unique functionality of the LCS.

Chapter 3

Formal Verification of Logic-based Control Systems by Implicit Model Checking

3.1 Introduction

Formal verification is a process of determining rigorously whether the LCS design satisfies all the functional specifications or not. As an example, consider the tank interlock system (Victor, 1979) represented as a binary logic diagram shown in Figure 3.1. The filling operation is aborted by closing on-off valve `SV430` if there is a high pressure alarm (`PAH430`) or the `Stop` button is pressed, and can be resumed by pressing the `Reset` button. The signal `SV430` is identified as a state variable due to the existence of the feedback path, and the signals `PAH430`, `Stop`, and `Reset` are inputs. Figure 3.2 is a state transition graph for this system. A node represents a particular valuation of state variables, and the edge represents a transition from one state to another for a particular valuation of the inputs. One of the requirements for this interlock system design is that the inlet valve should be closed (`SV430` is `FALSE`) whenever there is a shutdown signal (`PAH430` or `Stop`) even if the operator presses the `Reset` button by mistake. Formal verification requires this specification to be checked for all possible states and transitions.

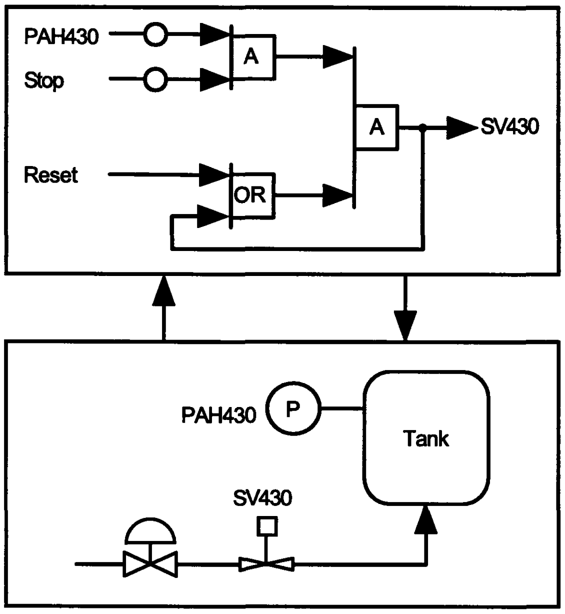
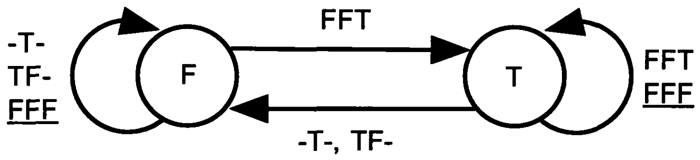


Figure 3.1: Tank interlock system with its underlying chemical process



Retentive transitions are underlined.

Figure 3.2: State transition graph for tank interlock system

As a class of discrete event dynamic systems, LCSs exhibit transient behavior between discrete states by processing a sequence of logic input signals to produce a series of logical outputs interfaced to control elements in the process. Thus, LCSs can be described as *sequential logic systems*, logic systems whose outputs depend upon both current inputs and the past history of inputs encapsulated in state variables. As discussed in section 2.1, the state space of sequential logic systems is finite but combinatorial, i.e., the maximum number of states is exponential in the number of state variables, which causes the state explosion problem. Formal verification of LCSs requires all reachable states to be checked (either explicitly or implicitly) against the specifications. Hence, state explosion creates severe problems for formal verification of large-scale systems.

Extensive simulation is the most widely used validation technique in industry. It simply simulates a LCS for all possible states and input transitions and checks whether the LCS is consistent with the specifications. As an example, consider the tank interlock system in Figure 3.1. There are 16 possible state transitions to consider because it has one state variable and three inputs. Therefore, it is necessary to simulate the binary logic diagram in Figure 3.1 for all these 16 transitions and to check whether the specification is satisfied or not. Even though simulation-based methods are relatively straightforward to implement, they are very costly and limited in the extent to which the complete state space can be explored. In particular, formal verification requires that a combinatorial number of states and transitions must be tested by simulation. For example, exhaustive simulation of a sequential logic system that contains 100 inputs and state variables together will take about 4×10^{12} years even if we assume that each simulation takes 10^{-10} seconds. Due to the combinatorial number of possible states and transitions, simulation cannot be considered as a viable formal verification technology for the large-scale systems currently being implemented in industry.

Until now, the most successful verification methods for sequential logic systems have been based on *model checking* (Clarke et al., 1986). Model checking is the problem of determining whether a model of a design satisfies a given abstract specification or not. It should be noted that it is the model, not the physical implementation of the design, that is verified; it is always critical to confirm the validity of the assumption upon which the model is based before applying any model checking technology. Within this framework, models are described as state transition systems (e.g., state transition graphs) and specifications are expressed in a propositional temporal logic. Most model checking algorithms are also based on enumeration of the complete state space for the system. Enumeration-based model checking is highly automatic because it can find the set of states where a given specification is true without any user intervention, and the complexity of the algorithm is linear in the size of the state transition graph and in the length of the specification formula. However, model checking suffers from an explosion in the size of the state transition graph for large-scale problems, since the number of nodes in this graph (states) is exponential in the number of state variables, and the number of edges (transitions) is exponential in the number of inputs and state variables. A more recent development that attempts to mitigate this problem is *symbolic model checking* (Bose and Fisher, 1989; Burch et al., 1990; Coudert et al., 1990). The basic idea is to manipulate sets of states and sets of transitions instead of individual states and individual transitions. The Boolean formulas that symbolize sets of states and sets of transitions are represented by ordered binary decision diagrams (OBDDs) (Bryant, 1986), which are often much more compact than explicit representations because they capture some of the regularity in the state space. While the idea of symbolic model checking and various refinements of the OBDD-based techniques have greatly increased the size of the problems that can be verified to over 400 state variables and 10^{120} states (Burch et al., 1991; Burch et al., 1994), many practical problems are still too large to be verified, and many much smaller problems lack this inherent regularity in the state space. It is therefore important to find techniques to extend the size of the problems that can be verified (Clarke et al., 1993). Both model checking and symbolic model checking have been applied with success to several small- to medium-scale verification problems in the chemical industries (Jeong et al., 1995; Moon, 1994; Moon et al., 1992; Probst and Powers, 1994; Probst et al., 1995; Probst et al., 1997), although these problems probably represent the size limit of current verification technology.

In this thesis, we present a novel model checking technique we have called *implicit model checking*. In particular, implicit model checking can be applied to systems of realistic sizes encountered in the chemical processing industries. Figure 3.3 shows an overview of the implicit model checking algorithm. Any formal verification technique raises the problems of model representation, specification language, and verification method. In our approach, the model for sequential logic systems is represented implicitly in terms of a system of Boolean equations, and specifications are expressed in a subset of propositional branching time temporal logic. Due to the implicit nature of the model representation, model formulation is not combinatorial, which increases the problem size that can be formulated dramatically. The formal verification problem is formulated as a Boolean satisfiability problem by combining the implicit Boolean state space model with specifications. The Boolean satisfiability problem is then transformed into its equivalent integer programming feasibility problem. An efficient verification method based on the implicit enumeration implemented in standard integer programming techniques is employed to determine automatically if the specifications are satisfied by the model. As a result, our method does not involve explicit enumeration of the full state space.

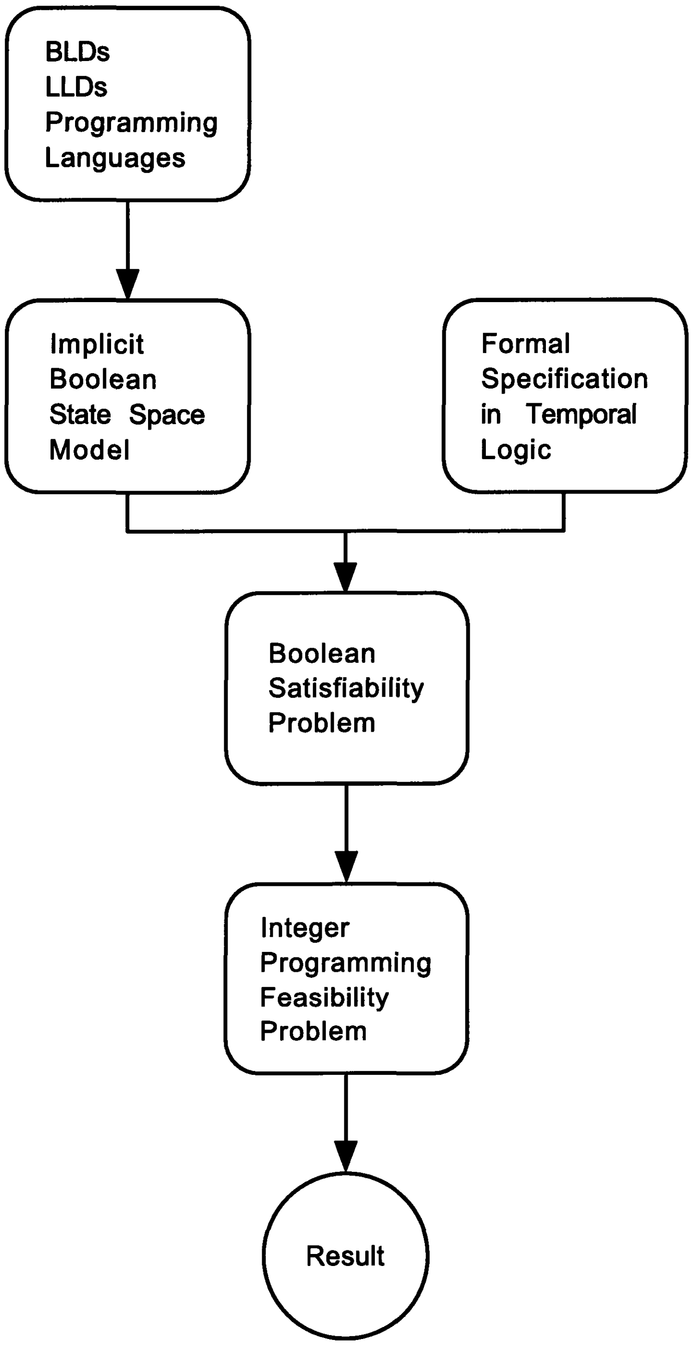

BLDs
LLDs
Programming Languages
Implicit Boolean State Space Model
Formal Specification in Temporal Logic
Boolean Satisfiability Problem
Integer Programming Feasibility Problem
Result

Figure 3.3: Overview of implicit model checking algorithm

Table 3.1: Conceptual difference in model checking methods

Category	Symbolic Model Checking	Implicit Model Checking
Model formulation	combinatorial with respect to the number of state variables	polynomial with respect to the number of state variables
Verification algorithm	polynomial with respect to the size of the model	combinatorial with respect to the size of the model
Worst case	run out of memory and time	run out of time

The fact is that formal verification of sequential logic systems is combinatorial, and any formal verification algorithm will eventually suffer from the state explosion problem. However, there are major differences between the implicit model checking and the current approach as shown in Table 3.1. The current approach will run out of memory and time in worst cases because it suffer from combinatorics in both model formulation and solution algorithm. However, we confine the combinatorial nature of the problem to the solution step. Therefore, the implicit model checking will run out of time only in worst cases.

3.2 Mathematical Formulation of Logical Propositions

3.2.1 Propositional Logic

Propositional logic is two-valued, TRUE or FALSE. A *proposition* is any propositional logic expression which contains a set of atomic propositions and a set of connectives. An *atomic* proposition or a *literal*, which corresponds to a statement and/or a variable that can assume either of two values, is the primitive operand of a propositional logic expression. The complete set of primitive Boolean operators or *connectives* is $\{\neg,\wedge,\vee\}$ meaning {NOT, AND, OR} respectively. Other connectives such as $\rightarrow$ (IMPLICATION), $\leftrightarrow$ (EQUIVALENCE), and $\otimes$ (EXCLUSIVE OR) can be expressed as a combination of primitive connectives. A proposition describes the relationship between a set of atomic propositions connected by Boolean operators.

3.2.2 Conjunctive Normal Form

There are many ways of writing down the same logical proposition. For example, it is not difficult to see that two logical propositions in Eqns (3.1a) and (3.1b) have the same valuation for all possible combinations.

$$(\neg p \wedge q) \vee (p \wedge \neg q) \tag{3.1a}$$

$$(p \vee q) \wedge (\neg p \vee \neg q) \tag{3.1b}$$

It is desirable to have a canonical representation for logical propositions. Such a representation must have the property that two logical propositions are equivalent if and only if they have isomorphic representations. This property simplifies tasks like checking equivalence of two proposition and deciding if a given proposition is satisfiable or not. *Conjunctive normal form* can represent any logical proposition in a canonical form. A logical proposition in conjunctive normal form is a conjunction of clauses where each *clause* is a disjunction of atomic propositions. A *conjunction* is a set of logical propositions connected by the AND operator while a *disjunction* is a set of logical propositions connected by the OR operator. An atomic proposition is a logical proposition that does not contain any Boolean connectives. For example, Eqn (3.1b) is already in its conjunctive normal form. Once Eqn (3.1a) is converted into its conjunctive normal form, it is straightforward to check the logical equivalence of Eqn (3.1a) and Eqn (3.1b).

An efficient algorithm exists that can convert any logical propositions into their conjunctive normal forms (Cavalier et al., 1990). The algorithm used in this thesis is listed as a PROLOG program in Appendix A.

The computational complexity of a naive algorithm that converts logical propositions into their conjunctive normal forms can be exponential in the worst case. In other words, the number of clauses in the conjunctive normal form can be exponential in the worst case. However, the number of clauses in the conjunctive normal form can be made linear in the number of atomic propositions or Boolean variables in the original proposition by introducing a polynomial number of intermediate variables, using an algorithm such as structure-preserving clause form translation (Boy de la Tour, 1990; Hähnle, 1993; Plaisted and Greenbaum, 1986). For example, consider the following logical proposition in Eqn (3.2):

$$\bigvee_{i=1}^{l}\left(Y_{r,i} \otimes Y_i\right) \tag{3.2}$$

which can be rewritten as $\bigvee_{i=1}^{l}\left(\neg\left(Y_{r,i} \leftrightarrow Y_i\right)\right)$, meaning that $Y_{r,i}$ and Y_i do not have the same valuations for all i. The number of clauses in the conjunctive normal form of the logical proposition in Eqn (3.2) is 2^l, i.e., increases exponentially in the number of terms l (Boy de la Tour, 1990). Table 3.2 illustrates this exponential growth for up to l =3. By introducing additional intermediate variables P_i, $i = 1..l$ and rewriting the proposition as in Eqn (3.3):

$$P_i \leftrightarrow \left(Y_{r,i} \otimes Y_i\right),\ i = 1..l \tag{3.3}$$

the number of clauses in the conjunctive normal form increases linearly in the number of terms l. In particular, the conjunctive normal form of the proposition $P_i \leftrightarrow \left(Y_{r,i} \otimes Y_i\right)$ contains 4 clauses. Therefore, the total number of clauses in the conjunctive normal form is $4l$.

Table 3.2: Conjunctive normal form of Eqn (3.2)

l	Conjunctive normal form
1	$\left(Y_{r,1} \vee Y_1\right) \wedge \left(\neg Y_{r,1} \vee \neg Y_1\right)$
2	$\left(Y_{r,1} \vee Y_1 \vee Y_{r,2} \vee Y_2\right) \wedge \left(Y_{r,1} \vee Y_1 \vee \neg Y_{r,1} \vee \neg Y_1\right) \wedge$ $\left(\neg Y_{r,1} \vee \neg Y_1 \vee Y_{r,2} \vee Y_2\right) \wedge \left(\neg Y_{r,1} \vee \neg Y_1 \vee \neg Y_{r,1} \vee \neg Y_1\right)$
3	$\left(Y_{r,1} \vee Y_1 \vee Y_{r,2} \vee Y_2 \vee Y_{r,3} \vee Y_3\right) \wedge \left(Y_{r,1} \vee Y_1 \vee \neg Y_{r,1} \vee \neg Y_1 \vee Y_{r,3} \vee Y_3\right) \wedge$ $\left(\neg Y_{r,1} \vee \neg Y_1 \vee Y_{r,2} \vee Y_2 \vee Y_{r,3} \vee Y_3\right) \wedge \left(\neg Y_{r,1} \vee \neg Y_1 \vee \neg Y_{r,1} \vee \neg Y_1 \vee Y_{r,3} \vee Y_3\right) \wedge$ $\left(Y_{r,1} \vee Y_1 \vee Y_{r,2} \vee Y_2 \vee \neg Y_{r,3} \vee \neg Y_3\right) \wedge \left(Y_{r,1} \vee Y_1 \vee \neg Y_{r,1} \vee \neg Y_1 \vee \neg Y_{r,3} \vee \neg Y_{33}\right) \wedge$ $\left(\neg Y_{r,1} \vee \neg Y_1 \vee Y_{r,2} \vee Y_2 \vee \neg Y_{r,3} \vee \neg Y_3\right) \wedge \left(\neg Y_{r,1} \vee \neg Y_1 \vee \neg Y_{r,1} \vee \neg Y_1 \vee \neg Y_{r,3} \vee \neg Y_3\right)$

3.2.3 Transformation of Logical Propositions into Inequalities

It is well known that any set of logical propositions can be transformed automatically to an equivalent set of linear inequalities in terms of binary variables (Cavalier et al., 1990). A set of logical propositions is first converted into their equivalent conjunctive normal forms. Having converted each logical proposition into its conjunctive normal form, it can be easily expressed as a set of linear inequalities. To each proposition P_i, a binary variable p_i is assigned, so that the negation of P_i is given by $1-p_i$. The logical value of TRUE corresponds to the binary value of 1 and FALSE corresponds to the binary value of 0. For the conjunctive normal form to be true, each clause must be true independent of the others and a clause ($P_1 \vee \neg P_2 \vee ... \vee P_N$) can be converted automatically into its equivalent inequalities ($p_1 + 1 - p_2 + ... + p_N \geq 1$). Any set of any logical propositions will give rise to the following set of inequalities by employing this procedure:

$$\mathbf{Ap} \geq \mathbf{a} \tag{3.4}$$

where $\mathbf{p} \in \{0,1\}^N$ is a set of binary variables corresponding to the propositions, and the data $\mathbf{A}, \mathbf{a}$ are respectively an integer matrix and an integer column vector of coefficients, which correspond to the conjunctive normal form of the set of propositions.

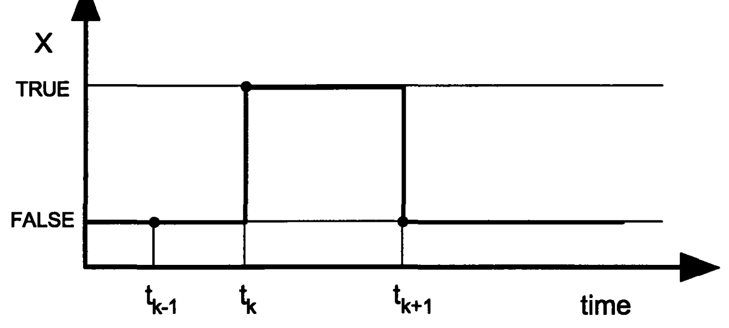


Figure 3.4: Mapping between discrete and continuous time domains

3.3 Model

3.3.1 Modeling Time

With reference to Figure 1.6, LCSs are passive in the sense that they do not manipulate control elements in the process unless an external *event* occurs, which can be a transition in explicit control signals (e.g., `Start_Filling_Tank_A` is pressed) or a predefined threshold crossing by the continuous state of the process (e.g., `Tank_A_Level_High` becomes TRUE). The system will then activate, calculate a transition in the output signals to the control elements (e.g., `Operate_Pump` becomes FALSE), and then become passive until activated again in a similar manner. Subject to this observation, in our model of the LCS the notion of physical time is replaced by the simpler notion of order among external events.

Furthermore, comparing the relative input frequencies and the input-output response times of LCSs to those of chemical processes, it is valid to assume that any LCS will finish its dynamic response to any external event before any further external event occurs. More formally, we assume that the LCS operates in a fundamental mode, which means that input signals are never changed unless the system is in a stable state. The system has reached a stable (or steady) state if none of the signals are changing given a fixed set of values for the input signals. Based on this assumption, we only consider stable states.

Based on these observations, the physical continuous time domain is discretized at the points of external events or, equivalently, transitions in the input signals. Therefore, time intervals between events will not necessarily be uniform. Note that this timing convention will result in the minimum number of discretization points while capturing all the relevant details, and timing is determined by the operation being performed by the overall system rather than an irrelevant uniform clock signal. Finally, the timing diagram of Figure 3.4 represents the convention for mapping between discrete and continuous time domains. For example, if the variable X experiences a transition from FALSE to TRUE at time t_k, the value of X at t_k is taken to be TRUE rather than FALSE.

3.3.2 Implicit Boolean State Space Model

In chapter 2, revised binary logic diagrams that exhibit fully deterministic behavior are adopted to represent the functionality of the LCS. The mathematical model to be used in formal verification will be derived from the revised BLD. Note that the mathematical model can be also derived from any standard representation for the LCS (IEC, 1993) as long as it conveys the unique functionality of the LCS. It is therefore necessary to guarantee that any logic design has had any potential ambiguities resolved before applying implicit model checking.

The key property that distinguishes our modeling framework from previous efforts in formal verification is the novel notion of ***implicitness***, in the sense that the model encapsulates only the relevant set of time invariant relationships between state variables, rather than a partial or full enumeration of the state space. Hence, our model is analogous to the state space model of a continuous dynamic system. This implicitness leads to a compact representation of the system dynamics and avoids the problem of state explosion during model formulation.

As mentioned in chapter 2, LCSs can be described as sequential logic systems whose current states depend upon both current inputs and the previous state. The sequential logic system to be verified is modeled mathematically as a deterministic finite state machine (FSM). The behavior of the FSM is described by a vector of n Boolean state variables $\mathbf{X} \in \{\mathrm{F},\mathrm{T}\}^n$, a vector of m Boolean inputs $\mathbf{U} \in \{\mathrm{F},\mathrm{T}\}^m$, and a vector of l Boolean outputs $\mathbf{Y} \in \{\mathrm{F},\mathrm{T}\}^l$. Each discrete point in state space is encoded by an assignment of Boolean values to the vector of state variables. The *state transition equation* (Eqn (3.5a)) and *output equation* (Eqn (3.5b)) for the FSM are given as a system of Boolean equations:

$$\mathbf{X}_k \leftrightarrow \mathbf{f}(\mathbf{X}_{k-1},\mathbf{U}_k) \tag{3.5a}$$

$$\mathbf{Y}_k \leftrightarrow \mathbf{g}(\mathbf{X}_k,\mathbf{U}_k) \tag{3.5b}$$

where $\mathbf{X}_k$ and $\mathbf{X}_{k-1}$ encode the current state and the previous state respectively, $\mathbf{U}_k$ and $\mathbf{Y}_k$ encode the current input and output respectively, and $\mathbf{f}:\{\mathrm{F},\mathrm{T}\}^n \times \{\mathrm{F},\mathrm{T}\}^m \mapsto \{\mathrm{F},\mathrm{T}\}^n$

and $\mathbf{g}:\{\mathrm{F},\mathrm{T}\}^n \times \{\mathrm{F},\mathrm{T}\}^m \mapsto \{\mathrm{F},\mathrm{T}\}^l$ are vectors of logical propositions. The state transition equation characterizes a transition from one state to another for a particular valuation of the inputs, and the output equation calculates outputs for the transition. Note that the FSM described by Eqn (3.5) is completely specified. In other words, the current state $\mathbf{X}_k$ and output $\mathbf{Y}_k$ are uniquely determined for any pair of $(\mathbf{X}_{k-1}, \mathbf{U}_k)$. The analogy between Eqn (3.5) and the familiar differential or difference equation models is evident.

Definition 1 : The transition from $\mathbf{X}_{k-1}$ to $\mathbf{X}_k$ is *legal* if the transition satisfies the state transition equation of Eqn (3.5a).

Definition 2 : The state $\mathbf{X}$ is a *valid* state if it is reachable through legal transitions.

Definition 3 : The state $\mathbf{X}$ is an *initial* state if it is unreachable from any other states but there exist legal transitions from this state.

The initial states represent states that the system enters at the beginning of operation such as power-up states. Note that initial states *must* be specified explicitly in order to include them in the verification because they cannot be reached via any transition.

By definition, any feasible solution that satisfies Eqn (3.5) represents a legal transition between valid states or from an initial state. Therefore, the Boolean state space model of Eqn (3.5) contains *implicitly* all possible legal transitions between valid states or from initial states. Note that the model is compact because it includes only the relevant set of invariant relationships between variables. However, an additional algorithmic procedure is necessary to draw out explicit information such as a particular sequence of transitions, just as numerical solution of ODEs is required to extract a particular trajectory.

As an example, consider the tank interlock system in Figure 3.1. After identifying the signal SV430 as a state variable due to the existence of the feedback path, the state transition equation is obtained directly from the BLD as:

$$X_{1,k} \leftrightarrow \left(\neg U_{1,k} \wedge \neg U_{2,k}\right) \wedge \left(U_{3,k} \vee X_{1,k-1}\right) \tag{3.6}$$

where $\mathbf{X} = [\texttt{SV430}]$ and $\mathbf{U} = [\texttt{PAH430,Stop,Reset}]$. Note that there is no output equation in this example. A feasible solution of Eqn (3.6) corresponds to an edge in the state transition graph in Figure 3.2. For example, the edge $\texttt{F} \xrightarrow{\texttt{FFT}} \texttt{T}$ represents the feasible solution $\mathbf{X}_{k-1} = [\texttt{F}]$, $\mathbf{X}_k = [\texttt{T}]$, $\mathbf{U}_k = [\texttt{F,F,T}]$ of Eqn (3.6). Substituting this feasible solution into Eqn (3.6), it is not difficult to see that the current state is uniquely determined by the current input alone. On the other hand, this is not the case for the transition $\texttt{T} \xrightarrow{\texttt{FFF}} \texttt{T}$ or $\texttt{F} \xrightarrow{\texttt{FFF}} \texttt{F}$. In this case, the current input cannot determine the current state uniquely. Substituting $\mathbf{U}_k = [\texttt{F,F,F}]$ into Eqn (3.6) yields $X_{1,k} \leftrightarrow X_{1,k-1}$. Therefore, the current state keeps the value of the previous state.

Subject to this observation, the transitions can be categorized into ***retentive*** and ***non-retentive*** transitions:

Definition 4 : The transition $\mathbf{X}_{k-1} \xrightarrow{\mathbf{U}_k} \mathbf{X}_k$ is *non-retentive* if $\mathbf{X}_k$ is uniquely determined by $\mathbf{U}_k$ regardless of $\mathbf{X}_{k-1}$. Otherwise, it is *retentive*. If $\mathbf{U}_k$ represents a retentive transition, at least one element of the state transition equation (Eqn (3.5a)) becomes $X_{i,k} \leftrightarrow X_{i,k-1}$ where $i \in \{1,..,n\}$.

The term *retentive* is adopted because these transitions retain partial or complete memory of the previous state of the system.

The objective of our modeling effort is to construct a model that contains implicitly all possible legal transitions between valid states or from initial states. The Boolean state space model of Eqn (3.5) satisfies this objective. However, as discussed in section 3.3.1, we are further only interested in stable states. The original model of Eqn (3.5) does not satisfy this requirement because it includes both stable and unstable states, which are defined as follows:

Definition 5 : The state $\tilde{\mathbf{X}}$ is a *stable* state if it is a fixed point of Eqn (3.5a) for given unchanging inputs $\mathbf{U}$, i.e., $\tilde{\mathbf{X}} \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}},\mathbf{U})$. Otherwise, it is an *unstable* state. The stable states correspond to the nodes with self-edges in the state transition graphs.

As an example, consider an alarm acknowledge system (Moon et al., 1992) represented as a *ladder logic diagram* (Otter, 1988) in Figure 3.5. The signals `Ack` and `Horn` are identified as state variables because of the latching of the signals in rung 3 and 4 respectively, and the signals `HiL`, `HiT`, `Reset`, and `PB` are inputs. The Boolean state space model can be obtained directly from Figure 3.5 as:

$$\begin{aligned} X_{1,k} &\leftrightarrow \neg U_{3,k} \wedge \left(U_{4,k} \vee X_{1,k-1}\right) \\ X_{2,k} &\leftrightarrow \neg X_{1,k-1} \wedge \left(U_{1,k} \vee U_{2,k} \vee X_{2,k-1}\right) \end{aligned} \tag{3.7}$$

where $\mathbf{X} = [\texttt{Ack,Horn}]$ and $\mathbf{U} = [\texttt{HiL,HiT,Reset,PB}]$. Figure 3.6 is a state transition graph representing all the feasible solutions of Eqn (3.7). The state $\mathbf{X} = [\texttt{T,T}]$ is unstable because there is no valuation of the inputs that makes this state a fixed point of Eqn (3.7) (i.e., there is no self-edge in the STG). It is therefore necessary to revise the original Boolean state space model to exclude any unstable states.

The revised implicit Boolean state space model is:

$$\tilde{\mathbf{X}}_k \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}}_k,\mathbf{U}_k) \tag{3.8a}$$

$$\tilde{\mathbf{X}}_{k-1} \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}}_{k-1},\mathbf{U}_{k-1}) \tag{3.8b}$$

$$\mathbf{h}(\tilde{\mathbf{X}}_k,\mathbf{U}_k) \rightarrow (\tilde{\mathbf{X}}_k \leftrightarrow \tilde{\mathbf{X}}_{k-1}) \tag{3.8c}$$

$$\mathbf{Y}_k \leftrightarrow \mathbf{g}(\tilde{\mathbf{X}}_k,\mathbf{U}_k) \tag{3.8d}$$

where Eqns (3.8a)-(3.8c) are state transition equations, and the output equations, Eqn (3.8d), are unchanged. Eqn (3.8a) (or Eqn (3.8b)) restrict the valuations of $\tilde{\mathbf{X}}_k$ (or $\tilde{\mathbf{X}}_{k-1}$) to stable states by definition. Note that the value of $\mathbf{U}_{k-1}$ is not limited to any particular valuation since we want $\tilde{\mathbf{X}}_{k-1}$ to represent all possible stable states. Note that non-retentive transitions are uniquely defined by Eqns (3.8a) and (3.8b). If $\mathbf{U}_k$ corresponds to a retentive transition, then at least one element of Eqn (3.8a) will be $\tilde{X}_{i,k} \leftrightarrow \tilde{X}_{i,k}$ where $i \in \{1,..,n\}$, which can be satisfied by any valuation of $\tilde{X}_{i,k}$. Therefore, $\tilde{\mathbf{X}}_k$ is not uniquely determined by Eqn (3.8a) and Eqn (3.8b) in this case. An additional constraint is necessary to determine $\tilde{\mathbf{X}}_k$ uniquely for retentive transitions. Eqn (3.8c) determines all retentive transitions uniquely. The Boolean function $\mathbf{h}\colon\{\mathrm{F},\mathrm{T}\}^n \times \{\mathrm{F},\mathrm{T}\}^m \mapsto \{\mathrm{F},\mathrm{T}\}^n$ is a vector of logical propositions called ***retentive functions*** that define conditions for retentive transitions. The retentive functions $\mathbf{h}$ can be derived from $\mathbf{f}$ automatically, and will be discussed later.

As an example, consider again the alarm acknowledge system in Figure 3.5. The revised implicit Boolean state space model can be derived from Eqn (3.7) as follows:

$$\tilde{X}_{1,k} \leftrightarrow \neg U_{3,k} \wedge \left(U_{4,k} \vee \tilde{X}_{1,k}\right) \tag{3.9a}$$

$$\tilde{X}_{2,k} \leftrightarrow \neg \tilde{X}_{1,k} \wedge \left(U_{1,k} \vee U_{2,k} \vee \tilde{X}_{2,k}\right) \tag{3.9b}$$

$$\tilde{X}_{1,k-1} \leftrightarrow \neg U_{3,k-1} \wedge \left(U_{4,k-1} \vee \tilde{X}_{1,k-1}\right) \tag{3.9c}$$

$$\tilde{X}_{2,k-1} \leftrightarrow \neg \tilde{X}_{1,k-1} \wedge \left(U_{1,k-1} \vee U_{2,k-1} \vee \tilde{X}_{2,k-1}\right) \tag{3.9d}$$

$$\left(\neg U_{3,k} \wedge \neg U_{4,k}\right) \rightarrow \left(\tilde{X}_{1,k} \leftrightarrow \tilde{X}_{1,k-1}\right) \tag{3.9e}$$

$$\left(\neg \tilde{X}_{1,k} \wedge \neg\left(U_{1,k} \vee U_{2,k}\right)\right) \rightarrow \left(\tilde{X}_{2,k} \leftrightarrow \tilde{X}_{2,k-1}\right) \tag{3.9f}$$

Eqns (3.9a)-(3.9d) defines all non-retentive transitions, and Eqns (3.9e) and (3.9f) defines all retentive transitions. Figure 3.7 is a state transition graph representing all feasible solutions of Eqns (3.9a)-(3.9d). Note that the unstable state $\mathbf{X} = [\mathrm{T},\mathrm{T}]$ is excluded. However, retentive transitions are not uniquely determined. For example, the transition from $\tilde{\mathbf{X}}_{k-1} = [\mathrm{T},\mathrm{F}]$ forced by $\mathbf{U}_k = [\mathrm{F},\mathrm{F},\mathrm{T},\mathrm{T}]$ is not unique. Instead, the transition can be $\mathrm{TF} \xrightarrow{\mathrm{FFTT}} \mathrm{FF}$ or $\mathrm{TF} \xrightarrow{\mathrm{FFTT}} \mathrm{FT}$. However, we know that the correct transition is $\mathrm{TF} \xrightarrow{\mathrm{FFTT}} \mathrm{FF}$ because $\mathbf{X}_k = \left[\mathrm{F}, X_{2,k-1}\right]$ by substituting $\mathbf{U}_k = [\mathrm{F},\mathrm{F},\mathrm{T},\mathrm{T}]$ into Eqn (3.7). By adding Eqns (3.9e) and (3.9f), all retentive transitions are determined uniquely. Figure 3.8 shows a state transition graph defined by the complete state space model of Eqn (3.9).

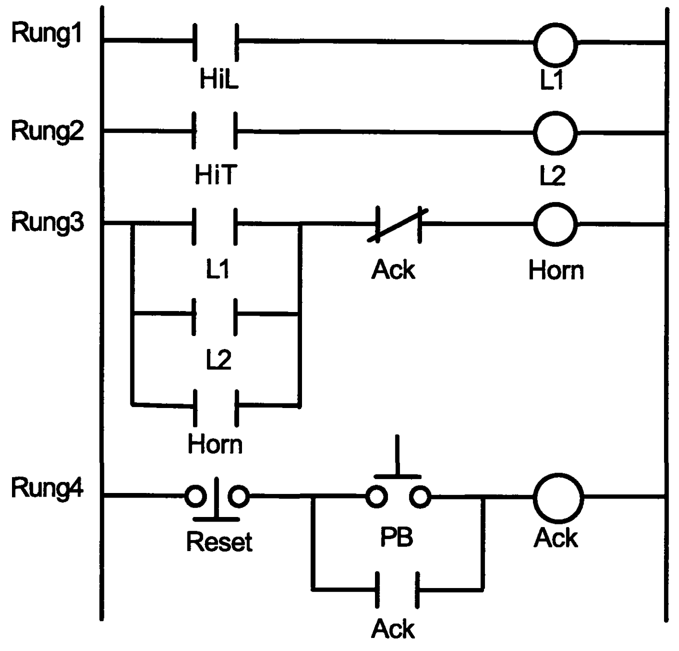


Figure 3.5: Alarm acknowledge system

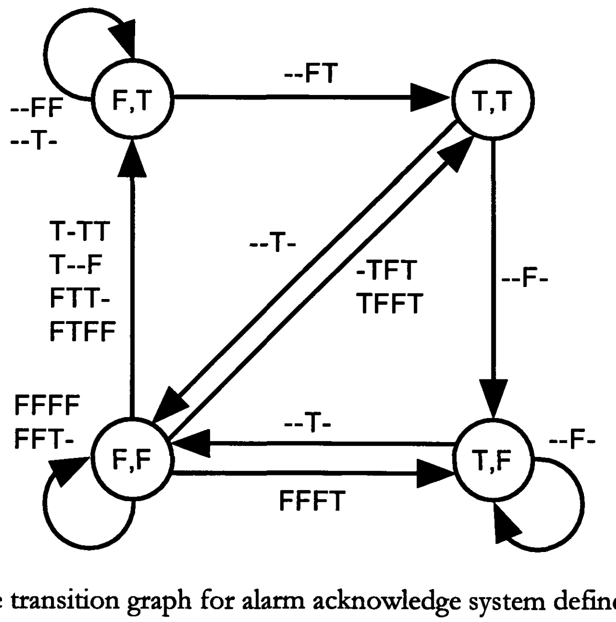


Figure 3.6: State transition graph for alarm acknowledge system defined by Eqn (3.7)

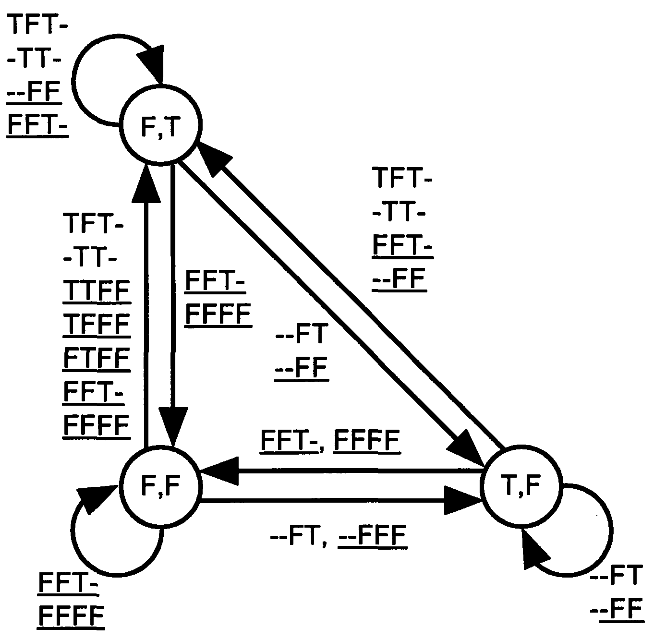


Figure 3.7: State transition graph for alarm acknowledge system defined by Eqns (3.9a)-(3.9d)

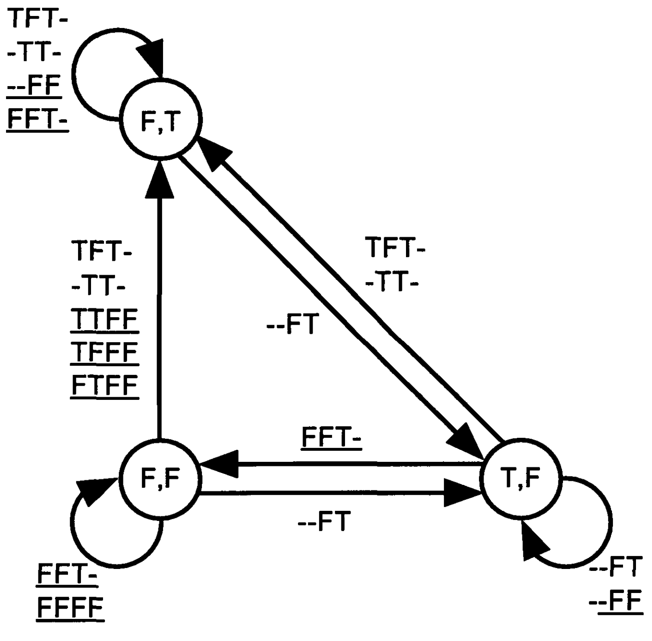


Figure 3.8: State transition graph for alarm acknowledge system defined by Eqn (3.9)

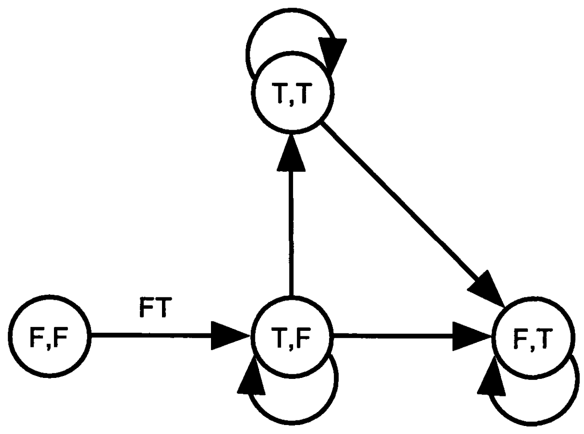


Figure 3.9: Finite state machine containing unstable initial states

The Boolean state space model of Eqn (3.8) now embeds implicitly all possible legal transitions between stable states. However, it does not embed transitions from initial states if these initial states are not stable. Figure 3.9 illustrates this situation where the state $\mathbf{X} = [\mathrm{F}, \mathrm{F}]$ corresponds to an unstable initial state. The model of Eqn (3.8) does not include the transition $\mathrm{FF} \xrightarrow{\mathrm{FT}} \mathrm{TF}$ because $\tilde{\mathbf{X}}_{k-1}$ in Eqn (3.8) does not include this unstable state. However, this is not a limitation because the initial states must be specified explicitly as mentioned earlier.

Coupled with an explicit specification of the initial states, the Boolean state space model of Eqn (3.8) or $\mathbf{v}(\mathbf{U}_{k-1}, \mathbf{U}_k, \tilde{\mathbf{X}}_{k-1}, \tilde{\mathbf{X}}_k, \mathbf{Y}_k)$ embeds implicitly all possible legal transitions between stable states or from initial states. Hence, any feasible solution of the model represents a particular legal transition between stable states or from initial states. This model forms the basis for our large-scale verification technology. Note that the size of the model in terms of the number of logical propositions is linear in the number of state variables and the number of outputs.

3.3.3 Derivation of Memory-retaining Propositions

The retentive functions in Eqn (3.8c) can be derived by examining the conditions under which the inputs do not determine the state vector uniquely. The transition equation for *i*-th state variable $\tilde{X}_{i,k}$ is:

$$\tilde{X}_{i,k} \leftrightarrow \mathrm{f}_i\left(\tilde{\mathbf{X}}_k, \mathbf{U}_k\right) \tag{3.10}$$

First, the transition function f_i can be transformed into the form:

$$\tilde{X}_{i,k} \leftrightarrow \mathrm{f}_{i,c1}\left(\tilde{\mathbf{X}}'_k, \mathbf{U}_k\right) \wedge \left(\mathrm{f}_{i,c2}\left(\tilde{\mathbf{X}}'_k, \mathbf{U}_k\right) \vee \tilde{X}_{i,k}\right) \tag{3.11}$$

where $\tilde{\mathbf{X}}'_k$ is a vector of state variables excluding $\tilde{X}_{i,k}$, and the Boolean formulas $\mathrm{f}_{i,c1}$, $\mathrm{f}_{i,c2}$ are in conjunctive normal form. An algorithm to convert Boolean formulas into conjunctive normal forms can be used with a distributive law to derive Eqn (3.11) automatically from Eqn (3.10). It is clear from Eqn (3.11) that $\tilde{X}_{i,k}$ cannot be determined uniquely if $\mathrm{f}_{i,c1}\left(\mathbf{U}_k, \tilde{\mathbf{X}}'_k\right)$ and $\neg\mathrm{f}_{i,c2}\left(\mathbf{U}_k, \tilde{\mathbf{X}}'_k\right)$. Therefore, the memory-retaining constraint for *i*-th state variable $\tilde{X}_{i,k}$ is:

$$\left(\mathrm{f}_{i,c1}\left(\mathbf{U}_k, \tilde{\mathbf{X}}'_k\right) \wedge \neg\mathrm{f}_{i,c2}\left(\mathbf{U}_k, \tilde{\mathbf{X}}'_k\right)\right) \rightarrow \left(\tilde{X}_{i,k} \leftrightarrow \tilde{X}_{i,k-1}\right) \tag{3.12}$$

For example, consider the following transition equation for $\tilde{X}_1$:

$$\tilde{X}_{1,k} \leftrightarrow \left(U_{1,k} \wedge U_{2,k}\right) \vee \left(\neg U_{3,k} \wedge \tilde{X}_{1,k}\right) \tag{3.13}$$

First, the transition function for $\tilde{X}_1$ can be converted into a conjunctive normal form:

$$\tilde{X}_{1,k} \leftrightarrow \left(U_{1,k} \vee \neg U_{3,k}\right) \wedge \left(U_{2,k} \vee \neg U_{3,k}\right) \wedge \left(U_{1,k} \vee \tilde{X}_{1,k}\right) \wedge \left(U_{2,k} \vee \tilde{X}_{1,k}\right) \tag{3.14}$$

Factoring the term $\tilde{X}_{1,k}$ on the right hand side of Eqn (3.14), the expression corresponding to Eqn (3.11) is obtained as:

$$\tilde{X}_{1,k} \leftrightarrow \left(U_{1,k} \vee \neg U_{3,k}\right) \wedge \left(U_{2,k} \vee \neg U_{3,k}\right) \wedge \left(\left(U_{1,k} \wedge U_{2,k}\right) \vee \tilde{X}_{1,k}\right) \tag{3.15}$$

Finally, the memory-retaining proposition is obtained as:

$$\left(U_{1,k} \vee \neg U_{3,k}\right) \wedge \left(U_{2,k} \vee \neg U_{3,k}\right) \wedge \neg\left(U_{1,k} \wedge U_{2,k}\right) \rightarrow \left(\tilde{X}_{1,k} \leftrightarrow \tilde{X}_{1,k-1}\right) \tag{3.16}$$

Note that derivation of memory-retaining propositions has the same computational complexity as converting Boolean formulas into conjunctive normal form. Even though the number of clauses in the conjunctive normal form can be exponential in the worst case, the number of clauses in the conjunctive normal form can be made linear in the number of Boolean variables in the original proposition as discussed in section 3.2.2.

The complete Boolean state space model for the tank interlock system in Figure 3.1 is obtained from Eqn (3.6) by applying this procedure as:

$$\begin{gathered}\tilde{X}_{1,k} \leftrightarrow \left(\neg U_{1,k} \wedge \neg U_{2,k}\right) \wedge \left(U_{3,k} \vee \tilde{X}_{1,k}\right) \\ \tilde{X}_{1,k-1} \leftrightarrow \left(\neg U_{1,k-1} \wedge \neg U_{2,k-1}\right) \wedge \left(U_{3,k-1} \vee \tilde{X}_{1,k-1}\right) \\ \left(\neg U_{1,k} \wedge \neg U_{2,k}\right) \wedge \neg U_{3,k} \rightarrow \left(\tilde{X}_{1,k} \leftrightarrow \tilde{X}_{1,k-1}\right)\end{gathered} \tag{3.17}$$

3.3.4 Modeling Unit Delay Timers

All delay timers are abstracted and modeled as unit delay timers. Abstract logical relationships between input and output signals are established by stating the minimum set of relevant properties of delay timers. For example, Figure 3.10 represents the timing diagram of unit delay-on timers. The abstract logical proposition that represents the functionality of the unit delay-on timer is:

$$\left(\neg X_{1,k} \wedge \neg X_{2,k}\right) \vee \left(\neg X_{1,k-1} \wedge X_{1,k} \wedge \neg X_{2,k}\right) \vee \left(X_{1,k-1} \wedge X_{1,k} \wedge X_{2,k}\right) \tag{3.18}$$

Similarly, Figure 3.11 represents the timing diagram of unit delay-off timers, and the abstract logical relationship is given by:

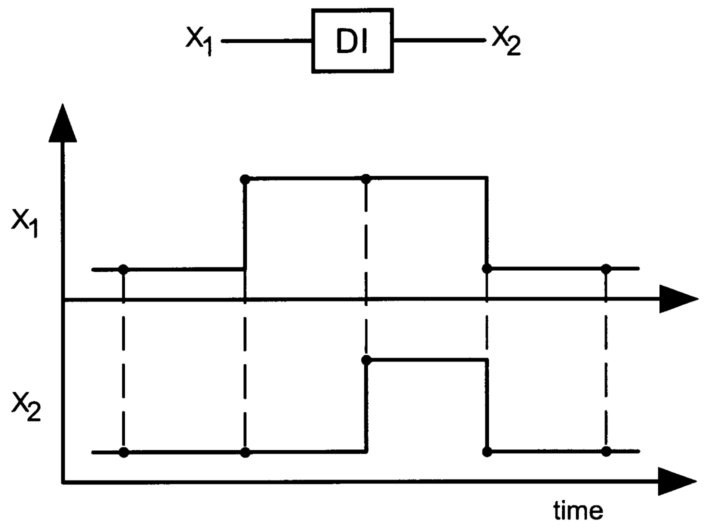


Figure 3.10: Timing diagram of delay-on timers

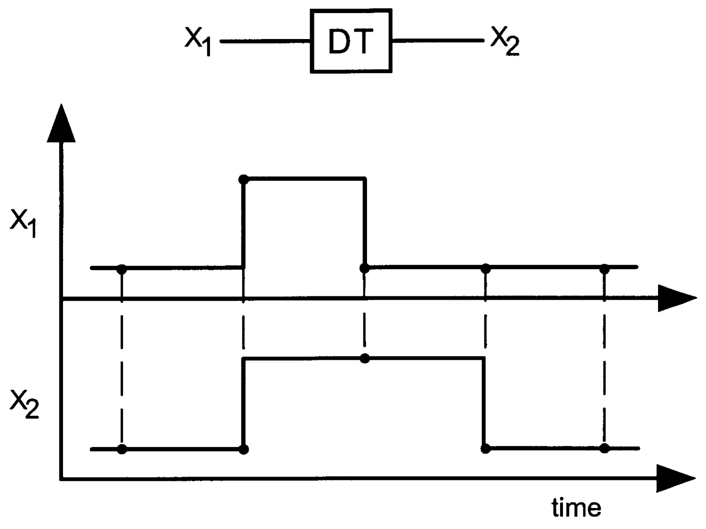


Figure 3.11: Timing diagram of delay-off timers

$$\left(X_{1,k} \wedge X_{2,k}\right) \vee \left(X_{1,k-1} \wedge \neg X_{1,k} \wedge X_{2,k}\right) \vee \left(\neg X_{1,k-1} \wedge \neg X_{1,k} \wedge \neg X_{2,k}\right) \quad (3.19)$$

From the simulation point of view, this abstraction is obviously a loss of information. However, from the verification point of view, the modeling of delay timers as unit delay timers will be adequate because the model checking algorithm explores the entire state space including the variables representing the abstract behavior of delay timers. If the specification is satisfied by the model, the model is correct with respect to the specification regardless of the magnitude of delays in timers because we consider all possible combinations of the inputs and outputs for the timers. Thus, the model is verified. If the specification is violated, it is necessary to analyze the problem with more rigorous delay timer models to reach a conclusion concerning correctness of the logic.

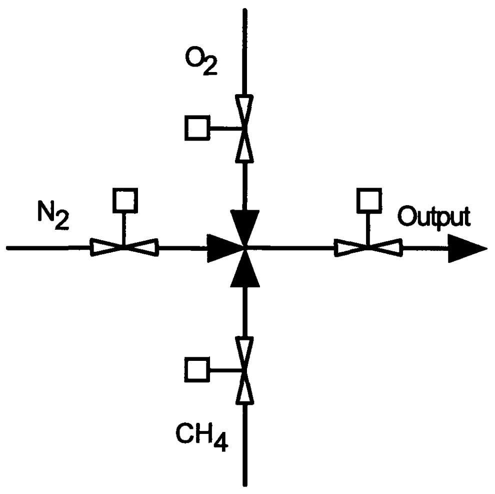


Figure 3.12: Network of pipes and valves

3.4 Formal Specification

A typical LCS embeds multiple functionalities coupled with each other, and thus there are large numbers of *correctness properties* that must be satisfied by the LCS model. These properties can be categorized into three classes: *shutdown logic*, *permissive logic*, and *sequences*. For example, consider the tank filling process of Figure 1.6. The LCS should always issue an `Stop` signal to the pump if the level of either tank A or B is higher than its threshold limit (shutdown logic). Similarly, the pump should not operate if an operator tries to fill tank A and B simultaneously (permissive logic).

Figure 3.12 shows a network of pipes and valves, which provides a sequence of an O_2 stream and then a CH_4 stream to the downstream process. Mixing of O_2 and CH_4 is not allowed to prevent an explosion. Therefore, the LCS for this process must ensure that the O_2 valve and CH_4 valve should not be open at the same time (permissive logic). Furthermore, the LCS must ensure that the N_2 valve should be open just after the O_2 valve or CH_4 valve is closed (sequence).

Finally, consider the typical LCS implemented for a furnace. If unsafe conditions occur or a stop signals exist, then the LCS should issue a shutdown signal immediately regardless of the current state (shutdown logic). If the furnace is in a shutdown state, then the furnace should stay in the shutdown state or the furnace can move only into the purge state (sequence). The furnace cannot be ignited unless the furnace has been purged (permissive logic).

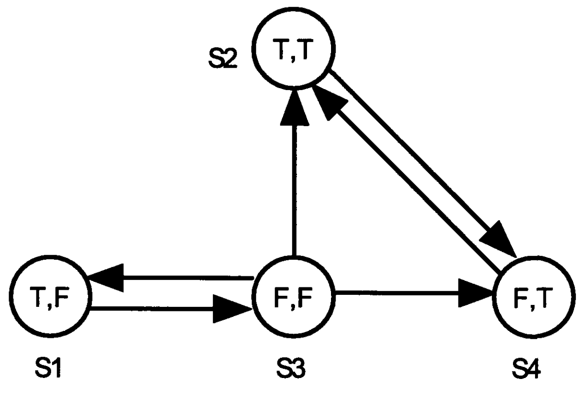
In order to verify the LCS model, it is necessary to represent all of these diverse properties in a formal manner, which is a *formal specification.* Note that every individual run or *computation* of the LCS yields a sequence of states and associated transitions, and specifications for the LCS are in general satisfied by some computations, and not satisfied by some other computations. *Temporal logic*, a logic of propositions whose truth and falsity may depend on time, is used to represent specifications because the language of temporal logic provides operators for reasoning about computations.

3.4.1 Syntax and Semantics

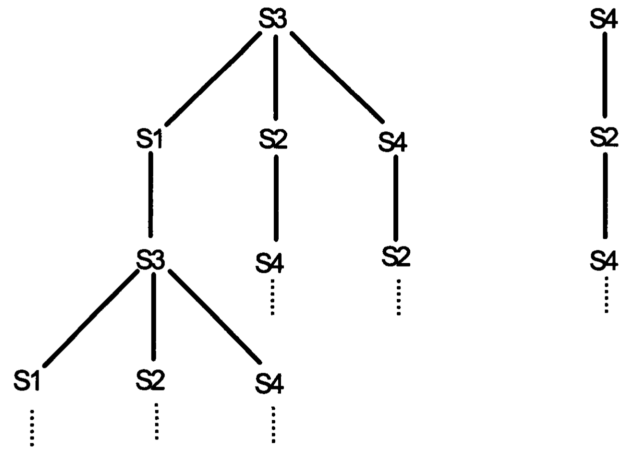
Among different classes of temporal logic, the subset of *Computation Tree Logic* (CTL) (Clarke and Emerson, 1981), which is a class of *propositional branching-time temporal logic* (Emerson, 1990), will be modified and used for implicit model checking. The computation tree is constructed by unwinding the state transition graph starting from the starting state, and the semantics of the formulas in CTL are defined with respect to this computation tree. Therefore, the valuation of a particular specification formula in CTL depends upon a model as well as a *starting* state. For example, Figure 3.13 (a) shows the FSM, and Figure 3.13 (b) and (c) represent the computation trees starting from the state S_3 and S_4

respectively. The specification checking whether the state $[T,F]$ exists or not is satisfied in the computation tree in Figure 3.13 (b). However, it is not satisfied in the computation tree in Figure 3.13 (c).

The dependency of valuation of a specification formula in CTL on a starting state will be removed in the semantics of the specification formula used in implicit model checking. Therefore, the semantics of the temporal formulas in the framework of implicit model checking is defined with respect to the model, and the valuation of a particular specification formula depends upon a model only, and the verification algorithm will check the formula for all possible starting states.



(a) FSM in state transition graph



(b) Computation tree starting from the state S3 (c) Computation tree starting from the state S4

Figure 3.13: FSM and its computation trees

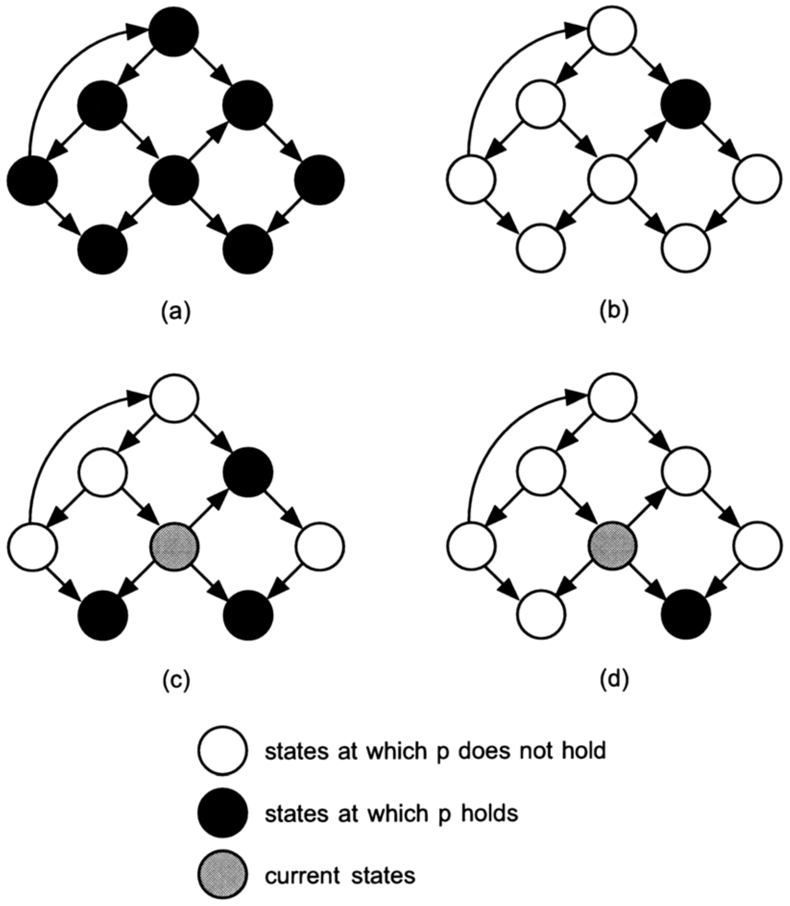


Figure 3.14: Semantics of temporal operators (a) $\forall\mathbf{G}(p)$, (b) $\exists\mathbf{F}(p)$, (c) $\forall\mathbf{N}(p)$, (d) $\exists\mathbf{N}(p)$

The temporal formulas are built up from atomic propositions (encoded by inputs, outputs, or state variables), Boolean connectives $\{\neg, \wedge, \vee\}$, and temporal operators $\{\mathbf{G}\text{: always}, \mathbf{F}\text{: sometimes}, \mathbf{N}\text{: next}\}$ restricted by quantifiers $\{\forall, \exists\}$ where $\forall$ (for all) and $\exists$ (there exists) are universal and existential quantifiers respectively. The syntax of the temporal logic is formally defined as:

- Any atomic proposition is a formula.
- If p, q are formulas, then so are the formulas $\neg p$, $p \wedge q$, and $p \vee q$.
- If p is a formula, then so are the formulas $\forall\mathbf{G}(p)$, $\exists\mathbf{F}(p)$, $\forall\mathbf{N}(p)$, and $\exists\mathbf{N}(p)$.

The semantics of the temporal formulas can be formally defined with respect to the model $\mathbf{v}(\mathbf{U}_{k-1}, \mathbf{U}_k, \tilde{\mathbf{X}}_{k-1}, \tilde{\mathbf{X}}_k, \mathbf{Y}_k)$. The notation $\mathbf{X} \models p$ means that the formula p is TRUE in the model at state $\mathbf{X}$.

- $\tilde{\mathbf{X}}_k \models \forall\mathbf{G}(p)$ iff p is TRUE at every state including $\tilde{\mathbf{X}}_k$.
- $\tilde{\mathbf{X}}_k \models \exists\mathbf{F}(p)$ iff p is TRUE at some $\tilde{\mathbf{X}}_k$.
- $\tilde{\mathbf{X}}_{k-1} \models \forall\mathbf{N}(p)$ iff p is TRUE at all $\tilde{\mathbf{X}}_k$ such that $\mathbf{v}(\mathbf{U}_{k-1}, \mathbf{U}_k, \tilde{\mathbf{X}}_{k-1}, \tilde{\mathbf{X}}_k, \mathbf{Y}_k)$.
- $\tilde{\mathbf{X}}_{k-1} \models \exists\mathbf{N}(p)$ iff p is TRUE at some $\tilde{\mathbf{X}}_k$ such that $\mathbf{v}(\mathbf{U}_{k-1}, \mathbf{U}_k, \tilde{\mathbf{X}}_{k-1}, \tilde{\mathbf{X}}_k, \mathbf{Y}_k)$.

Figure 3.14 illustrates the semantics of each temporal operators using an example. The formula $\forall\mathbf{G}(p)$ $(\text{always}\, p)$ means that the proposition p holds for all states and their associated transitions in the computation. As an example, consider the tank interlock system of Figure 3.1. One of the requirements for this interlock system is that the inlet valve should be closed (SV430 is FALSE) whenever there is a shutdown signal (PAH430 or Stop) even if the operator presses the Reset button by mistake. Since this property should be always satisfied regardless of the current state of the system, it can be specified formally as $\forall\mathbf{G}(U_{1,k} \vee U_{2,k} \rightarrow \neg\tilde{X}_{1,k})$. It is not difficult to see from Figure 3.2 that any transition from any state satisfies the proposition $U_{1,k} \vee U_{2,k} \rightarrow \neg\tilde{X}_{1,k}$. Therefore, the formula $\forall\mathbf{G}(U_{1,k} \vee U_{2,k} \rightarrow \neg\tilde{X}_{1,k})$ is TRUE with respect to the model of Eqn (3.17).

The formula $\exists\mathbf{F}(p)$ $(\text{sometimes}\, p)$ means that there is some state and its associated transition in the computation at which p holds. Consider the alarm acknowledge system of Figure 3.5 as an example. If there is an alarm (Horn is TRUE), then the operator can acknowledge the alarm (Ack is TRUE) and turn it off (Horn is FALSE) by pressing the PB button. Therefore, we can expect that the signals Ack and Horn cannot be TRUE at the same time, which can be formally specified as $\neg\exists\mathbf{F}(\tilde{X}_1 \wedge \tilde{X}_2)$. It is easy to see from Figure 3.8 that the proposition $\tilde{X}_1 \wedge \tilde{X}_2$ is not satisfied, therefore the formula $\exists\mathbf{F}(\tilde{X}_1 \wedge \tilde{X}_2)$ (or $\neg\exists\mathbf{F}(\tilde{X}_1 \wedge \tilde{X}_2)$) is FALSE (or TRUE). Note that checking the formula $\neg\exists\mathbf{F}(\tilde{X}_1 \wedge \tilde{X}_2)$ is equivalent to checking the formula $\forall\mathbf{G}(\neg(\tilde{X}_1 \wedge \tilde{X}_2))$. In general, the operators $\forall\mathbf{G}$ and $\exists\mathbf{F}$ are dual: $\forall\mathbf{G}(p) = \neg\exists\mathbf{F}(\neg p)$.

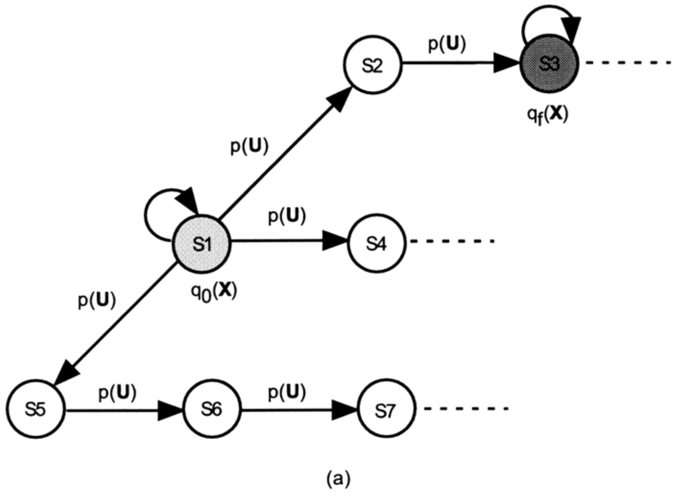
The formula $\forall\mathbf{N}(p)$ $(p \text{ at all next time})$ means that the proposition p holds in every immediate successor of the current state while $\exists\mathbf{N}(p)$ $(p \text{ at some next time})$ means that the proposition p holds in some immediate successor of the current state. For example, consider again the alarm acknowledge system. Note that once there is an alarm (Horn is TRUE), then either it should stay on or it is turned off (Horn is FALSE) only by pressing the PB button. This requirement can be formally specified as $\forall\mathbf{G}\left(\tilde{X}_{2,k} \rightarrow \forall\mathbf{N}\left(\tilde{X}_{2,k} \vee \left(\neg\tilde{X}_{2,k} \wedge U_{4,k}\right)\right)\right)$. It is not difficult to see from Figure 3.8 that this formula is TRUE by checking the transitions from the state $\mathbf{X} = [\mathrm{F}, \mathrm{T}]$. Note that the operators $\forall\mathbf{N}$ and $\exists\mathbf{N}$ are dual: $\forall\mathbf{N}(p) = \neg\exists\mathbf{N}(\neg p)$.

Note that the range of properties that can be specified is quite broad and will be sufficient to specify most intended properties of LCSs (Halbwachs et al., 1989; Jagadeesan et al., 1995). As an extension, the properties of eventuality and sequences can be specified in a direct manner.

3.4.2 Eventuality Properties

Eventuality (or liveness) properties state a temporal relationship for which the length of computation required to resolve a particular specification is not known in advance. Our implicit Boolean state space model is first-order, and the direct questions that can be asked are limited to those involving only one time step. However, it is in fact straightforward to specify eventuality properties because our model includes only stable states and transitions between them.

For example, consider the FSM in Figure 3.15 (a), where the propositions $q_0(\tilde{\mathbf{X}})$ and $q_f(\tilde{\mathbf{X}})$ are satisfied at states S_1 and S_3 respectively. Note that the propositions $q_0(\tilde{\mathbf{X}})$ (or $q_f(\tilde{\mathbf{X}})$) can represent a set of states symbolically. We want to verify whether it is possible for the system to reach the states $q_f(\tilde{\mathbf{X}})$ *eventually* in *some* computations from the states $q_0(\tilde{\mathbf{X}})$ by applying the input $p(\mathbf{U})$. In general, it is unknown a priori the length of computations or the number of transitions to reach the state $q_f(\tilde{\mathbf{X}})$ from the state $q_0(\tilde{\mathbf{X}})$. However, our model includes only stable states and transitions between them, and the number of transitions to reach the state $q_f(\tilde{\mathbf{X}})$ is always one if the state $q_f(\tilde{\mathbf{X}})$ does exist. Therefore, the specification can be formulated as $\exists\mathbf{F}\left(q_0(\tilde{\mathbf{X}}) \wedge p(\mathbf{U}) \rightarrow \exists\mathbf{N}\left(q_f(\tilde{\mathbf{X}})\right)\right)$. This specification is TRUE in the FSM of Figure 3.15 (a).



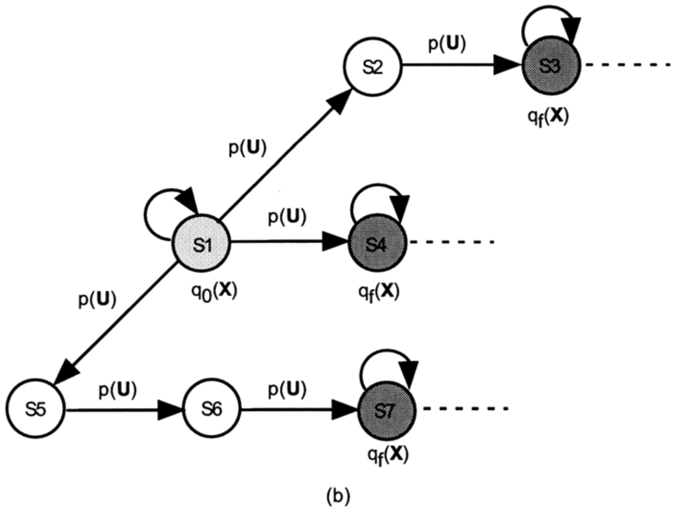


Figure 3.15: FSM showing eventuality properties

Similarly, we want to verify whether the system can reach the state $q_f(\tilde{\mathbf{X}})$ *eventually* in *all* computations from the state $q_0(\tilde{\mathbf{X}})$ by applying the input $p(\mathbf{U})$. The FSM in Figure 3.15b shows the case in which this specification is TRUE. The formal specification can be formulated as $\forall\mathbf{G}\left(q_0(\tilde{\mathbf{X}}) \wedge p(\mathbf{U}) \rightarrow \forall\mathbf{N}\left(q_f(\tilde{\mathbf{X}})\right)\right)$.

Note that the starting states are specified in the previous two specifications. Therefore, the previous two specifications check the eventual reachability from one state to another state forced by the particular input. More general formula will be one without starting state specifications $q_0(\tilde{\mathbf{X}})$. For example, the formula $\exists\mathbf{F}\left(p(\mathbf{U}) \rightarrow q_f(\tilde{\mathbf{X}})\right)$ asks whether it is possible for the system to reach the state $q_f(\tilde{\mathbf{X}})$ *eventually* in *some* computations by applying the input $p(\mathbf{U})$ starting from *some* states.

Similarly, the formula $\forall\mathbf{G}\left(p(\mathbf{U}) \rightarrow q_f(\tilde{\mathbf{X}})\right)$ means that the system can reach the state $q_f(\tilde{\mathbf{X}})$ *eventually* in *all* computations by applying the input $p(\mathbf{U})$ starting from *all* states. Note that these formulas should be checked for all possible starting states since no starting state $q_0(\tilde{\mathbf{X}})$ is specified.

3.4.3 Sequences

Sequence controls are extensively used in chemical industries. Sequence controls are necessary in continuous processes, especially during start-up, shutdown, and changeover operations, and are mandatory in batch processes even for routine operations. Sequence controls specify how and in what sequence plant operations are to be carried out, and thus time-dependent progression of the process in the state space. In modern plants, sequence control systems do not exist in isolation but instead are closely coupled to safety interlock systems, and sequence control systems can be regarded as logic-based control systems because it is common to implement them at the logic level.

State diagrams such as step diagrams, Petri-nets, and Grafcet are commonly used to specify the functionality of sequence control systems. A process to be controlled is divided into a number of well-defined successive steps separated by transitions, which are activated by satisfaction of permissives or transition conditions. During each step, sequence control turns on or off one or more outputs, waits for the response, then turns on or off another set of outputs, and so on. State diagrams essentially specify a sequence of steps and transitions. A *sequence* is formally defined as a series of state transitions forced by a series of transition conditions. The general form of sequences of length N is:

$$q_0(\tilde{\mathbf{X}}) \xrightarrow{p_1(\mathbf{U})} q_1(\tilde{\mathbf{X}}) \xrightarrow{p_2(\mathbf{U})} \dots q_{N-1}(\tilde{\mathbf{X}}) \xrightarrow{p_N(\mathbf{U})} q_N(\tilde{\mathbf{X}}) \tag{3.20}$$

which means that the system will move to the states $q_i(\tilde{\mathbf{X}})$ by applying the inputs $p_i(\mathbf{U})$ at the states $q_{i-1}(\tilde{\mathbf{X}})$ for $i = 1..N$. Recall that the proposition $q_i(\tilde{\mathbf{X}})$ can represent a set of states symbolically. The role of sequence control is to ensure that the system follow the sequence specified by state diagrams. Therefore, the sequence of Eqn (3.20) should be always satisfied. The sequence in Eqn (3.20) can be formally represented as:

$$\forall \mathbf{G}\left(\bigwedge_{i=1}^{N}\left(q_{i-1}(\tilde{\mathbf{X}}) \wedge p_i(\mathbf{U}) \rightarrow \forall \mathbf{N}\left(q_i(\tilde{\mathbf{X}})\right)\right)\right) \tag{3.21}$$

which means the sequence in Eqn (3.21) is TRUE in *all* computations. Alternatively, it may be necessary to find out whether a particular sequence is feasible or not. This requirement can be formulated as:

$$\exists \mathbf{F}\left(\bigwedge_{i=1}^{N}\left(q_{i-1}(\tilde{\mathbf{X}}) \wedge p_i(\mathbf{U}) \rightarrow \exists \mathbf{N}\left(q_i(\tilde{\mathbf{X}})\right)\right)\right) \tag{3.22}$$

which means that the sequence in Eqn (3.22) is TRUE in *some* computations.

As an example, consider the alarm acknowledge system in Figure 3.5. Consider the following operating sequence. The system is initially in the state $[F, F]$ where there is no horn and thus no acknowledgment signal from the operator. Then, the high level and/or temperature alarm is detected, and the horn is turned on. Once the horn sounds, the operator presses the push button to turn off the horn and acknowledge the alarm. It is assumed that the operator does not press the push button and reset button at the same time. This sequence that should be always satisfied can be formally specified as:

$$\forall\mathbf{G}\begin{pmatrix}\left(\left(\neg X_{1,1}\wedge\neg X_{2,1}\right)\wedge\left(U_{1,1}\vee U_{2,1}\right)\rightarrow\forall\mathbf{N}\left(\neg X_{1,2}\wedge X_{2,2}\right)\right)\wedge\\\left(\left(\neg X_{1,2}\wedge X_{2,2}\right)\wedge\left(\neg U_{3,2}\vee U_{4,2}\right)\rightarrow\forall\mathbf{N}\left(X_{1,3}\wedge\neg X_{2,3}\right)\right)\end{pmatrix} \tag{3.23}$$

3.5 Verification Algorithm

The verification problem is formulated as a Boolean satisfiability problem by combining the implicit Boolean state space model with the specification. Instead of solving the Boolean satisfiability problem, it is transformed into its equivalent integer programming feasibility problem. Depending upon the feasibility of the problem, the specification is proved or disproved with respect to the model.

3.5.1 Boolean Satisfiability Problem

The verification algorithm takes as inputs the model and a specification formula to be verified. The specification formula is transformed into a form that involves only existential quantifiers by removing universal quantifiers using $\forall\mathbf{G}(p)=\neg\exists\mathbf{F}(\neg p)$ and $\forall\mathbf{N}(p)=\neg\exists\mathbf{N}(\neg p)$, and then coupled with the model $\mathbf{v}(\mathbf{U}_{k-1},\mathbf{U}_k,\tilde{\mathbf{X}}_{k-1},\tilde{\mathbf{X}}_k,\mathbf{Y}_k)$ to yield a *Boolean satisfiability problem* (Nemhauser and Wolsey, 1988):

$$\begin{gathered}\exists\left[\mathbf{U}_{k-1},\mathbf{U}_k,\tilde{\mathbf{X}}_{k-1},\tilde{\mathbf{X}}_k,\mathbf{Y}_k\right]\\ s.t.\\ \mathbf{v}\left(\mathbf{U}_{k-1},\mathbf{U}_k,\tilde{\mathbf{X}}_{k-1},\tilde{\mathbf{X}}_k,\mathbf{Y}_k\right)\\ w\left(\mathbf{U}_k,\tilde{\mathbf{X}}_{k-1},\tilde{\mathbf{X}}_k,\mathbf{Y}_k\right)\end{gathered} \tag{3.24}$$

where $w\left(\mathbf{U}_k,\tilde{\mathbf{X}}_{k-1},\tilde{\mathbf{X}}_k,\mathbf{Y}_k\right)$ are specification constraints that can be derived from the specification formula by removing all universal quantifiers. Table 3.3 shows a list of specification formulas and corresponding specification constraints. The Boolean satisfiability problem will be feasible if there exists any solutions satisfying all the constraints. Otherwise, it will be infeasible. The original specification formulas can be formally proved or disproved by determining the satisfiability of the problem. The only remaining problem is to find out whether the problem in Eqn (3.24) is satisfiable or not, which will be discussed in section 3.5.5.

Table 3.3: List of specification formulas and constraints

Specification formula	$w(\mathbf{U}_k, \tilde{\mathbf{X}}_{k-1}, \tilde{\mathbf{X}}_k, \mathbf{Y}_k)$	Interpretation
$\forall\mathrm{N}(p)$, $\tilde{\mathbf{X}}_{k-1} = \mathbf{X}_0$	$\neg p(\tilde{\mathbf{X}}_k)$, $\tilde{\mathbf{X}}_{k-1} = \mathbf{X}_0$	The p holds at every next state of $\mathbf{X}_0$
$\exists\mathrm{N}(p)$, $\tilde{\mathbf{X}}_{k-1} = \mathbf{X}_0$	$p(\tilde{\mathbf{X}}_k)$, $\tilde{\mathbf{X}}_{k-1} = \mathbf{X}_0$	The p holds at some next state of $\mathbf{X}_0$
$\forall\mathbf{G}(p)$	$\neg p(\tilde{\mathbf{X}}_k)$	The p holds always.
$\exists\mathbf{F}(p)$	$p(\tilde{\mathbf{X}}_k)$	It is possible that p holds at some states.

3.5.2 Counter-examples and Witnesses

The results of the satisfiability problem should be analyzed differently depending upon the types of quantifiers by which the formula is preceded. For a formula with a universal quantifier $\forall$, satisfiability means that the formula is FALSE and the valuations that make the problem satisfiable represent *counter-examples*, whereas unsatisfiability means that the formula is TRUE. For a formula with a existential quantifier $\exists$, satisfiability means that the formula is TRUE and the valuations that make the problem satisfiable represent *witnesses*, whereas unsatisfiability means the formula is FALSE. Note that it is not necessary to have an additional algorithm to construct counter-examples or witnesses since the feasible solutions of the Boolean satisfiability problem are counter-examples or witnesses depending upon the formula verified. Furthermore, the solution algorithm presented in the next section can find all feasible solutions systematically.

For example, the formula $\forall\mathbf{G}\left(\neg U_{1,k} \wedge \neg U_{2,k} \rightarrow \tilde{X}_{1,k}\right)$ checks whether the valve will be open if there are no pressure alarm and no Stop signal for the tank interlock system of Figure 3.1. It is not difficult to see from Figure 3.2 that the formula is FALSE because the valve will remain closed if it was closed at the previous time step. Therefore, the solution of Boolean satisfiability problem will be feasible, and the feasible solution

$\mathbf{X}_{k-1} = [\mathrm{F}]$, $\mathbf{X}_k = [\mathrm{F}]$, $\mathbf{U}_k = [\mathrm{F,F,F}]$ or transition $\mathrm{F} \xrightarrow{\mathrm{FFF}} \mathrm{F}$ will serve as a counter-example to disprove the original specification formula.

The formula $\exists\mathbf{F}\left(\tilde{X}_{1,k} \wedge \tilde{X}_{2,k}\right)$ checks whether the horn can sound when the alarm is acknowledged by an operator for the alarm acknowledge system of Figure 3.5. The formula is FALSE because the state $\mathbf{X} = [\mathrm{T,T}]$ does not exist in the state transition graph of Figure 3.8. Therefore, the solution of Boolean satisfiability problem will be infeasible, hence, there is no witness.

3.5.3 Algorithm for Eventuality Properties

No particular algorithms are required to resolve eventuality properties. The temporal formulas representing eventuality properties can be added to the model, and the resulting Boolean satisfiability problem can be solved to resolve the formulas. Table 3.4 shows a list of specification formulas and corresponding specification constraints for eventuality properties.

Table 3.4: List of specification formulas and constraints for eventuality properties

Specification formula	$w(\mathbf{U}_k, \tilde{\mathbf{X}}_{k-1}, \tilde{\mathbf{X}}_k, \mathbf{Y}_k)$	Interpretation
$\exists\mathbf{F}\left(q_0(\mathbf{X}) \wedge p(\mathbf{U}) \rightarrow \exists\mathbf{N}\left(q_f(\tilde{\mathbf{X}})\right)\right)$	$\left(q_0(\tilde{\mathbf{X}}_{k-1}) \wedge p(\mathbf{U}_k)\right) \wedge q_f(\tilde{\mathbf{X}}_k)$	The state $q_f(\tilde{\mathbf{X}})$ can be reached eventually in some computations from the state $q_0(\tilde{\mathbf{X}})$ by applying the input $p(\mathbf{U})$.
$\forall\mathbf{G}\left(q_0(\mathbf{X}) \wedge p(\mathbf{U}) \rightarrow \forall\mathbf{N}\left(q_f(\tilde{\mathbf{X}})\right)\right)$	$\left(q_0(\tilde{\mathbf{X}}_{k-1}) \wedge p(\mathbf{U}_k)\right) \wedge \neg q_f(\tilde{\mathbf{X}}_k)$	The state $q_f(\tilde{\mathbf{X}})$ can be reached eventually in all computations from the state $q_0(\tilde{\mathbf{X}})$ by applying the input $p(\mathbf{U})$.
$\exists\mathbf{F}\left(p(\mathbf{U}) \rightarrow q_f(\tilde{\mathbf{X}})\right)$	$p(\mathbf{U}_k) \wedge q_f(\tilde{\mathbf{X}}_k)$	The state $q_f(\tilde{\mathbf{X}})$ can be reached eventually in some computations by applying the input $p(\mathbf{U})$ starting from some states
$\forall\mathbf{G}\left(p(\mathbf{U}) \rightarrow q_f(\tilde{\mathbf{X}})\right)$	$p(\mathbf{U}_k) \wedge \neg q_f(\tilde{\mathbf{X}}_k)$	The state $q_f(\tilde{\mathbf{X}})$ can be reached eventually in all computations by applying the input $p(\mathbf{U})$ starting from all states

3.5.4 Algorithm for Sequences

A natural decomposition of the sequence formulas in Eqn (3.21) and Eqn (3.22) can be exploited to verify the sequence formulas. First, the specification in Eqn (3.21) can be rewritten as:

$$\forall\mathbf{G}\left(\bigwedge_{i=1}^{N}\left(q_{i-1}\left(\tilde{\mathbf{X}}(t_{i-1})\right)\wedge p_i\left(\mathbf{U}(t_i)\right)\rightarrow q_i\left(\tilde{\mathbf{X}}(t_i)\right)\right)\right) \tag{3.25}$$

Using $\forall\mathbf{G}(p)=\neg\exists\mathbf{F}(\neg p)$, Eqn (3.25) can be transformed into:

$$\neg\exists\mathbf{F}\left(\bigvee_{i=1}^{N}\neg\left(q_{i-1}\left(\tilde{\mathbf{X}}(t_{i-1})\right)\wedge p_i\left(\mathbf{U}(t_i)\right)\rightarrow q_i\left(\tilde{\mathbf{X}}(t_i)\right)\right)\right) \tag{3.26}$$

which can be checked by the algorithm:

Verification Algorithm for Sequence in Eqn (3.25):

1. $j:=1$.
2. Solve the satisfiability problem:

   $r^j\left(\tilde{\mathbf{X}}_k\right):=\left\{\tilde{\mathbf{X}}_k:\mathbf{v}\left(\mathbf{U}_{k-1},\mathbf{U}_k,\tilde{\mathbf{X}}_k,\tilde{\mathbf{X}}_{k-1},\mathbf{Y}_k\right),\neg\left(q_{j-1}\left(\tilde{\mathbf{X}}_{k-1}\right)\wedge p_j\left(\mathbf{U}_k\right)\rightarrow q_j\left(\tilde{\mathbf{X}}_k\right)\right)\right\}$

   If $r^j\left(\tilde{\mathbf{X}}_k\right)\neq\varnothing$, then the original formula is not satisfied and the counter-examples will be a sequence of states $\left\{q_0\left(\tilde{\mathbf{X}}\right),q_1\left(\tilde{\mathbf{X}}\right),\ldots,q_{j-1}\left(\tilde{\mathbf{X}}\right),r^j\left(\tilde{\mathbf{X}}\right)\right\}$. Otherwise, go to step 3.
3. If $j=N$, then the original formula is satisfied. Otherwise, $j:=j+1$ and go to step 2.

Similarly, the specification in Eqn (3.22) can be rewritten as:

$$\exists\mathbf{F}\left(\bigwedge_{i=1}^{N}\left(q_{i-1}\left(\tilde{\mathbf{X}}(t_{i-1})\right)\wedge p_i\left(\mathbf{U}(t_i)\right)\rightarrow q_i\left(\tilde{\mathbf{X}}(t_i)\right)\right)\right) \tag{3.27}$$

and can be checked by the algorithm:

Verification Algorithm for Sequence in Eqn (3.27):

1. $j := 1$.
2. Solve the satisfiability problem:

   $r^j(\tilde{\mathbf{X}}_k) := \{\tilde{\mathbf{X}}_k : \mathbf{v}(\mathbf{U}_k, \tilde{\mathbf{X}}_k, \tilde{\mathbf{X}}_{k-1}, \mathbf{Y}_k), q_{j-1}(\tilde{\mathbf{X}}_{k-1}) \wedge p_j(\mathbf{U}_k) \rightarrow q_j(\tilde{\mathbf{X}}_k)\}$

   If $r^j(\tilde{\mathbf{X}}_k) = \varnothing$, then the original formula is not satisfied. Otherwise, go to step 3.
3. If $j = N$, then the original formula is satisfied, and the witnesses will be a sequence of states $\{q_0(\tilde{\mathbf{X}}), q_1(\tilde{\mathbf{X}}), \ldots, q_{N-1}(\tilde{\mathbf{X}}), q_N(\tilde{\mathbf{X}})\}$. Otherwise, $j := j+1$ and go to step 2.

Note that the above two algorithms will terminate in N or less steps, and the number of satisfiability problems to be solved is bounded by N.

3.5.5 Integer Programming Feasibility Problem

The final key step in our approach is that we exploit the equivalence between a Boolean satisfiability problem and a linear 0-1 integer programming (IP) feasibility problem (Cavalier et al., 1990) to perform the actual verification automatically and efficiently. Even though the two problems are known to be *NP-complete*, the advantages of solving the satisfiability problem as the *quantitative* IP feasibility problem have been demonstrated (Hooker, 1988). Most importantly, the IP feasibility problem derived from a satisfiability problem can be resolved by just solving the relaxed version of the problem in close to 90% of the cases (Hooker, 1988).

The Boolean satisfiability problem can be transformed automatically to an IP feasibility problem by transforming the set of propositions in the satisfiability problem into a set of linear inequalities in terms of binary variables using the procedure in section 3.2. The satisfiability problem (Eqn (3.24)) can be transformed into:

$$\begin{gathered} \exists[\mathbf{u}_{k-1}, \mathbf{u}_k, \tilde{\mathbf{x}}_{k-1}, \tilde{\mathbf{x}}_k, \mathbf{y}_k] \\ s.t. \\ \mathbf{A} \cdot [\mathbf{u}_{k-1}, \mathbf{u}_k, \tilde{\mathbf{x}}_{k-1}, \tilde{\mathbf{x}}_k, \mathbf{y}_k]^T \geq \mathbf{a} \end{gathered} \tag{3.28}$$

where $\mathbf{A}$, $\mathbf{a}$ are respectively an integer matrix and an integer column vector of coefficients, and $\mathbf{u}_{k-1}, \mathbf{u}_k \in \{0,1\}^m$, $\tilde{\mathbf{x}}_{k-1}, \tilde{\mathbf{x}}_k \in \{0,1\}^n$, $\mathbf{y}_k \in \{0,1\}^l$ are vectors of binary variables corresponding to $\mathbf{U}_{k-1}$, $\mathbf{U}_k$, $\tilde{\mathbf{X}}_{k-1}$, $\tilde{\mathbf{X}}_k$, and $\mathbf{Y}_k$ respectively. Standard algorithms such as branch and bound can be used to solve the IP feasibility problem.

The IP feasibility problem in Eqn (3.28) is actually formulated and solved as an IP optimization problem. By introducing artificial variables, the optimization problem can be obtained as:

$$\begin{gathered} \min \sum_{j=1}^{N_C} s_j \\ s.t. \\ \mathbf{A} \cdot \left[\mathbf{u}_{k-1}, \mathbf{u}_k, \tilde{\mathbf{x}}_{k-1}, \tilde{\mathbf{x}}_k, \mathbf{y}_k\right]^T + \mathbf{s} \geq \mathbf{a} \end{gathered} \tag{3.29}$$

where $\mathbf{s} \in \mathbf{N}^{N_c}$ is a vector of integer variables and N_C is the number of inequalities in the IP feasibility problem. The zero optimal objective function indicates that the IP feasibility problem is feasible. If the optimal objective function is greater than zero, then the problem is infeasible. This formulation is not practical for large problems because a large number of artificial variables should be introduced. In order to eliminate this potential problem, in this thesis, artificial variables are introduced only for the inequalities resulting from the specification formula. Note that the number of inequalities derived from the specification formula is very small. Therefore, the introduction of artificial variables will not cause any numerical inefficiency.

If the IP feasibility problem is feasible, then all the feasible solutions can be found automatically as follows:

1. Determine a feasible solution to the current IP feasibility problem.
2. If no feasible solution, terminate.
3. Add a cut set corresponding to the current feasible solution to the current IP feasibility problem.
4. Return to step 1.

In order to exclude a feasible solution, $\mathbf{z} := \left[\mathbf{u}_{k-1}, \mathbf{u}_k, \tilde{\mathbf{x}}_{k-1}, \tilde{\mathbf{x}}_k, \mathbf{y}_k\right]$, the integer cut set to be added is:

$$\sum_{z_j=1}\left(1-z_j\right)+\sum_{z_j=0} z_j \geq 1 \tag{3.30}$$

Note that transformation of the Boolean satisfiability problem into its equivalent IP feasibility problem has the same computational complexity as converting Boolean formulas into conjunctive normal forms, and that any propositions in the Boolean satisfiability problem can be converted efficiently into conjunctive normal forms containing a number of clauses polynomial in the number of Boolean variables as discussed in section 3.2.

3.5.6 Illustrative Example

Each step in the verification algorithm will be illustrated with the tank interlock system in Figure 3.1. The specification to be verified is that the inlet valve should be closed (`SV430` is `FALSE`) whenever there is a shutdown signal (`PAH430` or `Stop`) even if the operator presses the `Reset` button by mistake, which can be formally specified as:

$$\forall \mathbf{G}\left(U_{1,k} \vee U_{2,k} \rightarrow \neg \tilde{X}_{1,k}\right) \tag{3.31}$$

The Boolean satisfiability problem is formulated by combining the model (Eqn (3.17)) with the specification constraint from Table 3.3.

$$\begin{gathered}
\exists\left[U_{1,k-1}, U_{2,k-1}, U_{3,k-1}, U_{1,k}, U_{2,k}, U_{3,k}, \tilde{X}_{1,k-1}, \tilde{X}_{1,k}\right] \\
s.t. \\
\tilde{X}_{1,k} \leftrightarrow\left(\neg U_{1,k} \wedge \neg U_{2,k}\right) \wedge\left(U_{3,k} \vee \tilde{X}_{1,k}\right) \\
\tilde{X}_{1,k-1} \leftrightarrow\left(\neg U_{1,k-1} \wedge \neg U_{2,k-1}\right) \wedge\left(U_{3,k-1} \vee \tilde{X}_{1,k-1}\right) \\
\left(\neg U_{1,k} \wedge \neg U_{2,k}\right) \wedge \neg U_{3,k} \rightarrow\left(\tilde{X}_{1,k} \leftrightarrow \tilde{X}_{1,k-1}\right) \\
\neg\left(U_{1,k} \vee U_{2,k} \rightarrow \neg \tilde{X}_{1,k}\right)
\end{gathered} \tag{3.32}$$

which, if infeasible, verifies the original proposition (i.e., no state violating the original proposition exists), whereas if feasible, proves the violation of the original proposition and yields a set of feasible solutions corresponding to states and transitions that act as counter-examples.

Next, the Boolean satisfiability problem (Eqn (3.32)) needs to be transformed into the IP feasibility problem. The conjunctive normal form of each proposition in Eqn (3.32) is:

$$\begin{gathered}
\left(\neg U_{1,k} \vee \neg \tilde{X}_{1,k}\right) \wedge \left(\neg U_{2,k} \vee \neg \tilde{X}_{1,k}\right) \wedge \left(U_{1,k} \vee U_{2,k} \vee \neg U_{3,k} \vee \tilde{X}_{1,k}\right) \\
\left(\neg U_{1,k-1} \vee \neg \tilde{X}_{1,k-1}\right) \wedge \left(\neg U_{2,k-1} \vee \neg \tilde{X}_{1,k-1}\right) \wedge \left(U_{1,k-1} \vee U_{2,k-1} \vee \neg U_{3,k-1} \vee \tilde{X}_{1,k-1}\right) \\
\left(U_{1,k} \vee U_{2,k} \vee U_{3,k} \vee \neg \tilde{X}_{1,k} \vee \tilde{X}_{1,k-1}\right) \wedge \left(U_{1,k} \vee U_{2,k} \vee U_{3,k} \vee \tilde{X}_{1,k} \vee \neg \tilde{X}_{1,k-1}\right) \\
\left(U_{1,k} \vee U_{2,k}\right) \wedge \tilde{X}_{1,k}
\end{gathered} \tag{3.33}$$

which can be transformed into the IP feasibility problem:

$$\begin{gathered}
\exists\left[u_{1,k-1}, u_{2,k-1}, u_{3,k-1}, u_{1,k}, u_{2,k}, u_{3,k}, \tilde{x}_{1,k-1}, \tilde{x}_{1,k}\right] \\
s.t. \\
1-u_{1,k}+1-\tilde{x}_{1,k} \geq 1 \\
1-u_{2,k}+1-\tilde{x}_{1,k} \geq 1 \\
u_{1,k}+u_{2,k}+1-u_{3,k}+\tilde{x}_{1,k} \geq 1 \\
1-u_{1,k-1}+1-\tilde{x}_{1,k-1} \geq 1 \\
1-u_{2,k-1}+1-\tilde{x}_{1,k-1} \geq 1 \\
u_{1,k-1}+u_{2,k-1}+1-u_{3,k-1}+\tilde{x}_{1,k-1} \geq 1 \\
u_{1,k}+u_{2,k}+u_{3,k}+1-\tilde{x}_{1,k}+\tilde{x}_{1,k-1} \geq 1 \\
u_{1,k}+u_{2,k}+u_{3,k}+\tilde{x}_{1,k}+1-\tilde{x}_{1,k-1} \geq 1 \\
u_{1,k}+u_{2,k} \geq 1 \\
\tilde{x}_{1,k} \geq 1
\end{gathered} \tag{3.34}$$

It took 0.02 seconds to solve this IP feasibility problem using GAMS/OSL on an HP9000/735. The problem is infeasible, which means that the original specification (Eqn (3.31)) is satisfied by the model (Eqn (3.17)), and the tank interlock has been verified formally with respect to this constraint.

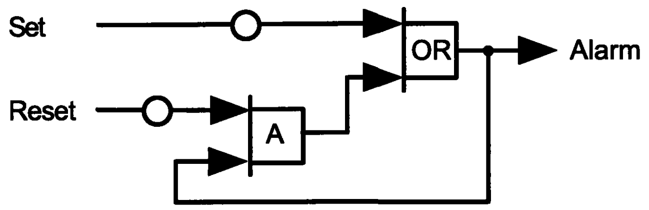


Figure 3.16: Simple memory interlock system

Table 3.5: Specifications for simple memory interlock system

No.	Formal specification	Meaning
1	$\forall G(\text{Set} \rightarrow \text{Alarm})$	`Set` signal should override `Reset` signal.
2	$\forall G(\neg\text{Set} \wedge \text{Reset} \rightarrow \neg\text{Alarm})$	`Alarm` signal is turned off by the `Reset` signal if there is no `Set` signal.
3	$\forall G(\neg\text{Set} \wedge \neg\text{Reset} \wedge \text{Alarm} \rightarrow \forall N(\text{Alarm}))$	`Alarm` signal is retained.

3.6 Examples

In this section, the capability of implicit model checking to identify an error in the LCS design is demonstrated using a small example. Then, results of small examples including literature problems are presented.

3.6.1 Design Modification Example

Figure 3.16 is a simple memory retaining interlock design represented as a binary logic diagram. Whenever there is a set signal, the alarm should be on and be retained thereafter until the reset signal turns off the alarm. From this informal specification, we can derive three formal specifications in Table 3.5. The implicit Boolean state space model of Eqn (3.34) is derived from the binary logic diagram in Figure 3.16.

$$\begin{aligned}\tilde{X}_{1,k} &\leftrightarrow \neg U_{1,k} \vee \left(\neg U_{2,k} \wedge \tilde{X}_{1,k}\right)\\ \tilde{X}_{1,k-1} &\leftrightarrow \neg U_{1,k-1} \vee \left(\neg U_{2,k-1} \wedge \tilde{X}_{1,k-1}\right)\\ U_{1,k} \wedge \neg U_{2,k} &\rightarrow \left(\tilde{X}_{1,k} \leftrightarrow \tilde{X}_{1,k-1}\right)\end{aligned} \tag{3.35}$$

where $\mathbf{U} = [\texttt{Set,Reset}]$ and $\mathbf{X} = [\texttt{Alarm}]$. The first specification is tested against the model in Eqn (3.35) by implicit model checking. The IP feasibility problem is formulated and then is solved by GAMS/OSL, which took 0.34 seconds on an HP9000/735. The IP feasibility problem is feasible, and the algorithm generates two counter-examples: $\mathbf{U} = [\texttt{T,F}]$, $\mathbf{X} = [\texttt{F}]$ and $\mathbf{U} = [\texttt{T,T}]$, $\mathbf{X} = [\texttt{F}]$. The counter-examples clearly show that the specification is not satisfied because the value of Alarm is FALSE even though the value of Set is TRUE. Substituting the values of variables from the counter-examples to the model or the binary logic diagram in Figure 3.16, it is not difficult to find out that the NOT element preceding the OR element should be removed. The model of Eqn (3.36) is derived from the modified design.

$$\begin{aligned}\tilde{X}_{1,k} &\leftrightarrow U_{1,k} \vee \left(\neg U_{2,k} \wedge \tilde{X}_{1,k}\right)\\ \tilde{X}_{1,k-1} &\leftrightarrow U_{1,k-1} \vee \left(\neg U_{2,k-1} \wedge \tilde{X}_{1,k-1}\right)\\ \neg U_{1,k} \wedge \neg U_{2,k} &\rightarrow \left(\tilde{X}_{1,k} \leftrightarrow \tilde{X}_{1,k-1}\right)\end{aligned} \tag{3.36}$$

All the three specifications in Table 3.5 are verified against the revised model, and all the IP problems are infeasible, meaning that all specifications are satisfied by the model.

3.6.2 Small Example Problems

The results of applying implicit model checking to small examples including literature problems are summarized in Tables 3.6-3.8. All the specifications are satisfied by the model as in the literature.

Table 3.6: Tank interlock system (Victor, 1979)

Formal Specification	Informal Specification	CPU time[1]
$\forall\mathbf{G}\left(\left(U_1 \vee U_2\right) \rightarrow \neg X_1\right)$	Shutdown signals override reset signal.	0.020
$\forall\mathbf{G}\left(\left(\neg U_1 \wedge \neg U_2 \wedge U_3\right) \rightarrow X_1\right)$	Valve is open by pressing Reset button if there are no shutdown signals.	0.010

[1]CPU time in seconds using GAMS/OSL on HP9000/735

Table 3.7: Alarm acknowledge system (Moon et al., 1992)

Formal Specification	Informal Specification	CPU time[1]
$\forall\mathbf{G}\left(\left(U_1 \vee U_2\right) \wedge \neg X_1 \rightarrow X_2\right)$	The horn sounds whenever high level or high temperature is detected and the system is not acknowledged.	0.010
$\forall\mathbf{G}\left(X_2 \rightarrow \forall\mathbf{N}\left(X_2 \vee \left(\neg X_2 \wedge U_4\right)\right)\right)$	After the horn sounds, either it stays on or it is turned off only if the push button is pressed.	0.010
$\forall\mathbf{G}\left(U_4 \rightarrow \left(X_1 \wedge \neg X_2\right)\right)$	Once the operator presses the push button, the system is acknowledged and the horn goes off.	0.020

[1]CPU time in seconds using GAMS/OSL on HP9000/735

Table 3.8: Furnace system (Probst et al., 1995)

Formal Specification	Informal Specification	CPU time[1]
$\forall\mathbf{G}\left(\text{estop} \rightarrow \begin{pmatrix} \neg\text{mgssov1\_z} \wedge \neg\text{mgssov2\_z} \\ \wedge \neg\text{pgssov1\_z} \wedge \neg\text{pgssov2\_z} \end{pmatrix}\right)$	All gas valves should be closed when estop is pressed.	0.25
$\forall\mathbf{G}\left(\neg\text{otr} \rightarrow \begin{pmatrix} \neg\text{mgssov1\_z} \wedge \neg\text{mgssov2\_z} \\ \wedge \neg\text{pgssov1\_z} \wedge \neg\text{pgssov2\_z} \end{pmatrix}\right)$	All gas valves should be closed if there is an over-temperature condition.	0.22
$\forall\mathbf{G}\left(\neg\text{wdr} \rightarrow \begin{pmatrix} \neg\text{mgssov1\_z} \wedge \neg\text{mgssov2\_z} \\ \wedge \neg\text{pgssov1\_z} \wedge \neg\text{pgssov2\_z} \end{pmatrix}\right)$	All gas valves should be closed if there is a PLC malfunction.	0.17

[1]CPU time in seconds using GAMS/OSL on HP9000/735

3.7 Conclusions

A formal verification methodology termed as implicit model checking has been developed that can be applied to large-scale sequential logic systems without encountering the state explosion problem. The key new result that facilitates this is the development of a novel implicit representation for sequential logic systems. In addition, significant computational benefits arise from solving the verification problem in the domain of binary variables rather than in the domain of Boolean variables.

Sequential logic systems are represented implicitly by a Boolean state space model that embeds all possible states and transitions in a compact closed form. Specifications are represented in a subset of temporal logic, which can formulate properties of interest for the LCS including shutdown logic, permissive logic, and sequences. Combining the model with specifications, the verification problem is formulated as Boolean satisfiability problem, which in turn is transformed into its equivalent integer programming (IP) feasibility problem. The IP feasibility problem is solved using a standard branch and bound algorithm, whose solution determines whether the specification is satisfied or not, yielding counter-examples or witnesses as necessary.

In principle, the verification problem is combinatorial. The key feature of our approach is to confine the combinatorial nature of the problem to the solution of the IP feasibility problem, which implies that 1) model formulation is not combinatorial at all, and 2) the whole verification problem can be solved efficiently if we can solve the resulting IP feasibility problems efficiently because all the intermediate steps involved can be automated and performed efficiently in polynomial time. Empirical studies presented in the next chapter indicate that the IP feasibility problem can be solved very efficiently in polynomial time using a standard branch and bound algorithm even though the IP feasibility problem is combinatorial in the worst case. This high efficiency in terms of computational cost is due to the fact that the implicit enumeration conducted by a branch and bound algorithm based an information from partial relaxations of the IP feasibility problem is dramatically more efficient and tractable than enumeration-based verification techniques.

Chapter 4

Burner Management System Case Studies

The implicit model checking technology developed in chapter 3 is applied to two industrial-scale burner management systems in order to verify their functionalities formally. In order to test the performance of our algorithm, empirical studies are conducted by applying the algorithm to a series of problems of increasing size.

4.1 Mod5 Burner Management System

Fired heating equipment (e.g., boilers, furnaces, superheaters, etc.) is strictly regulated by governmental regulations because even minor incidents involving this type of equipment can have significant consequences in terms of safety. In order to operate this type of equipment while maintaining safety, very complex *burner management systems* are installed. For example, *North American Guidelines for Application of MOD5™ Burner Management* (Dow Chemical Company, 1994) provides a basic burner management system for single-burner fuel-gas fired equipment as shown in Figure 4.1. The heater generates steam or heats fluids by burning a mixture of combustion air and fuel gas. In general, burner management systems include regulatory control systems (e.g., combustion control) in addition to logic-based control systems. The combustion control part of the guidelines will be excluded in the case study. The logic-based control systems in the guidelines are verified employing the implicit model checking technology.

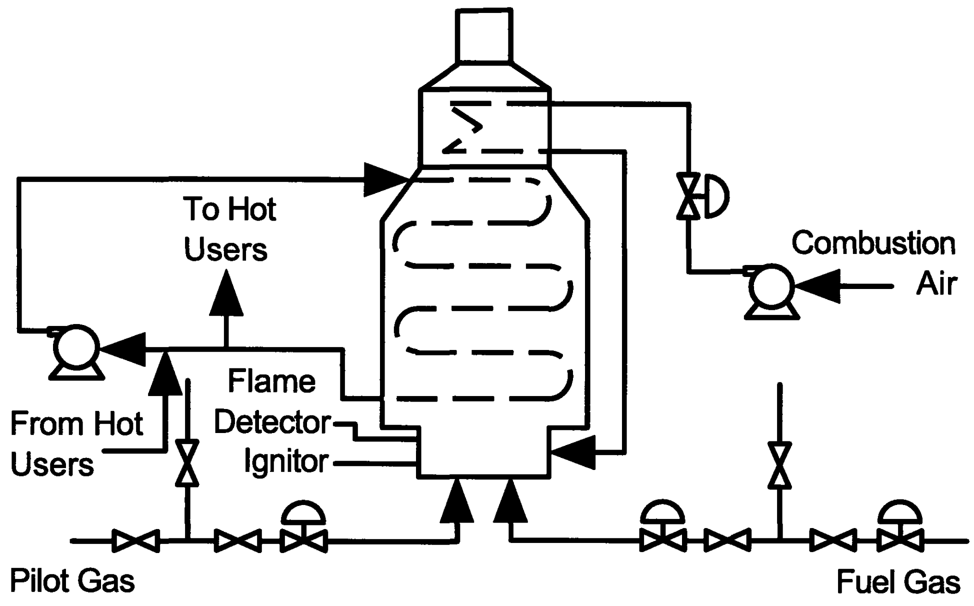


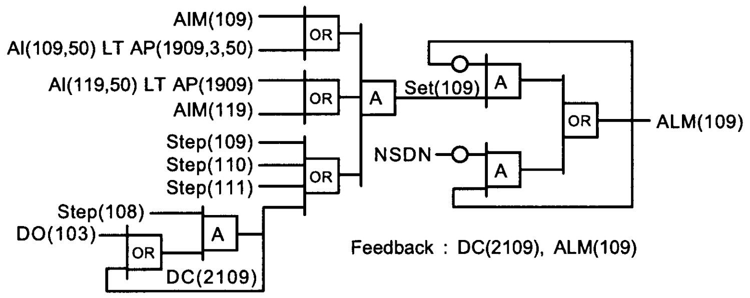
Figure 4.1: Single-burner fuel-gas fired equipment

4.1.1 Implicit Boolean State Space Model

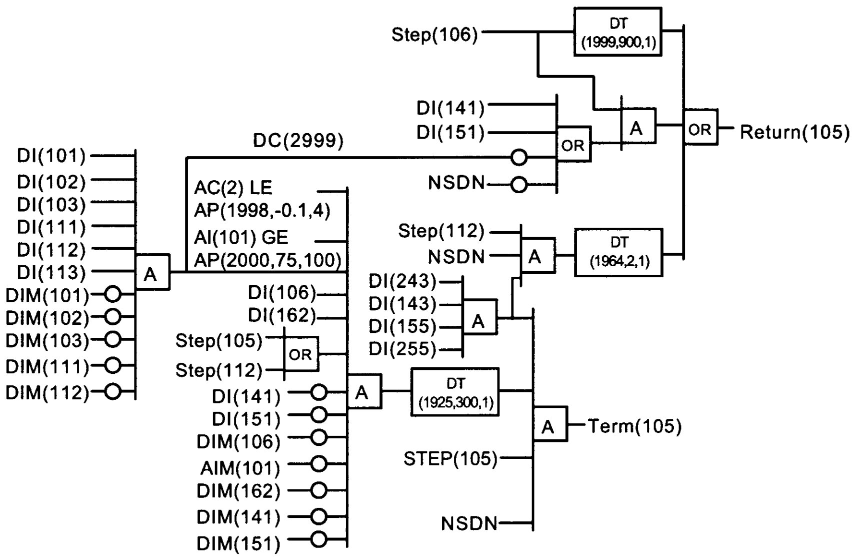
The guidelines contain 9 shutdown interlocks, an operation sequence involving 10 steps, and 7 related output and abort logics coupled together. Examples of each are shown in the binary logic diagrams of Figure 4.2. Note that they are not independent since they are coupled by sharing variables together. The original LCS logic in the guidelines is coded in DOWTRAN™ (a proprietary procedural control language). The set of Boolean equations are extracted from this code. Table 4.1 shows the problem size. The implicit Boolean state space model derived and variables used in the model are listed in Appendix B.

Table 4.1: Statistics on problem size for MOD5™

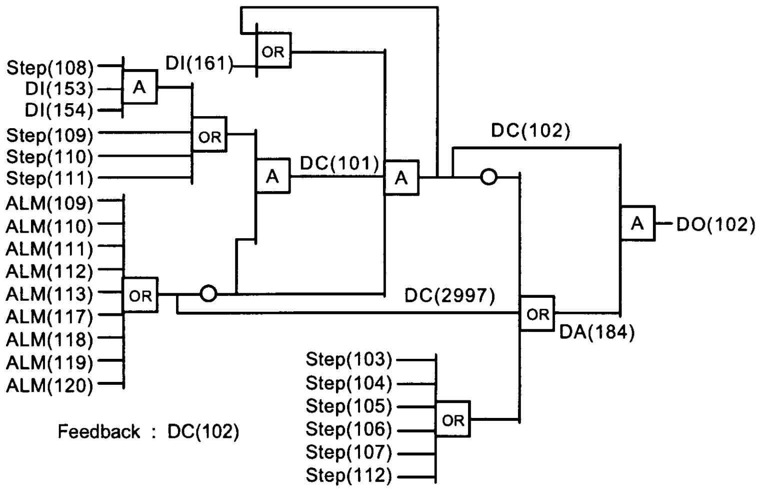
Category	Number
No. of Propositions	271
No. of Inputs	65
No. of State Variables	44
No. of Outputs	20
No. of Inequalities in IP feasibility problem	1100



SHDN(109) : Low Low Fuel Gas Pressure Alarm



STEP(105) : Purge Step



DO(102) : F1 Main Upstream EBV

Figure 4.2: Sample logic of MOD5™ burner management system

4.1.2 Specifications

The requirements or specifications for the MOD5™ burner management system are listed informally as comments in the DOWTRAN™ code. The following representative specifications are extracted from the comments and are represented formally in terms of temporal logic.

Specification No. 1

- Formal specification : $\forall \mathbf{G}(SET(i) \wedge u47 \rightarrow ALM(i))$ for $\forall i$
- Description : Shutdown signals should override set signals.

Specification No. 2

- Formal specification : $\forall \mathbf{G}(\neg u47 \wedge \neg SET(i) \wedge ALM(i) \rightarrow \forall \mathbf{N}(ALM(i)))$ for $\forall i$
- Description : Once set signals become ON, shutdown alarm signals should remain ON even though set signals become OFF afterward until the alarm is acknowledged by the operator.

Specification No. 3

- Formal specification : $\forall \mathbf{G}(\neg(xs1 \vee xs2) \wedge u53 \wedge u55 \rightarrow x3)$
- Description : The process is not in step 103 or 104. If air flow measurement is below AP(1911) and blower motor amps are below AP(1910), then the shutdown ALM(111) should sound.

Specification No. 4

- Formal specification : $\forall \mathbf{G}(((\neg u19 \wedge \neg u28) \vee (\neg u24 \wedge \neg u29)) \wedge (xs5 \vee \ldots \vee xs10) \rightarrow x5)$
- Description : The process is in step 107, 108, 109, 110, 111, or 112. If either MOD5™ emergency stop switches or field emergency stop switches are activated, then shutdown ALM(113) should sound.

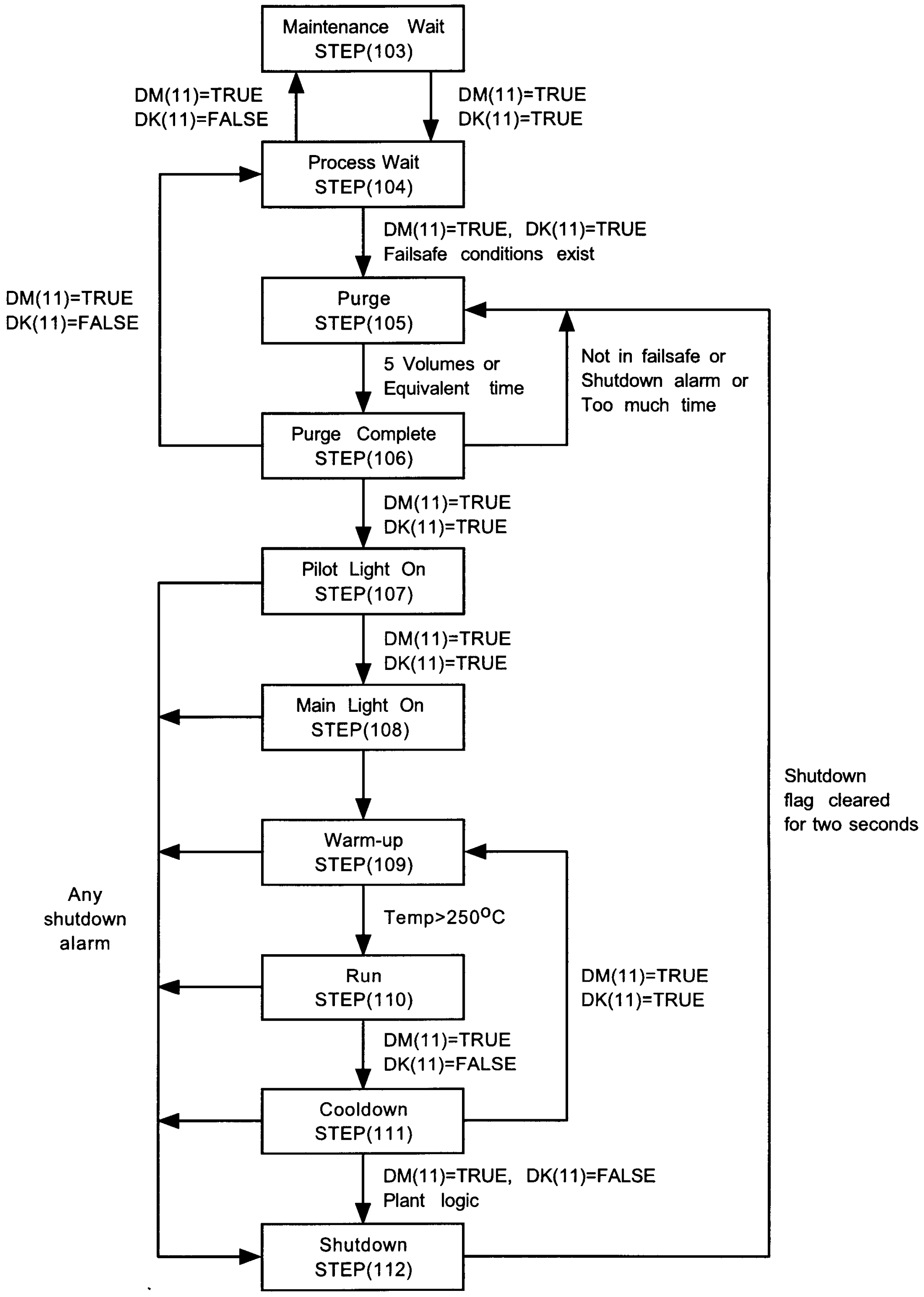


Figure 4.3: Step diagram of operating sequences in MOD5™ burner management system

Figure 4.3 shows the step diagram for the operating sequences in MOD5™ burner management system. The step diagram embeds three categories of operating sequences: normal operating sequences, shutdown operating sequences, and secondary operating sequences. These sequences in this diagram can be formally specified in temporal logic as follows.

Normal operating sequences

$$\forall\mathbf{G}\begin{pmatrix}(xs1 \wedge p_1 \rightarrow \forall\mathbf{N}(xs2)) \wedge (xs2 \wedge p_2 \rightarrow \forall\mathbf{N}(xs3)) \wedge (xs3 \wedge p_3 \rightarrow \forall\mathbf{N}(xs4)) \\ \wedge (xs4 \wedge p_4 \rightarrow \forall\mathbf{N}(xs5)) \wedge (xs5 \wedge p_5 \rightarrow \forall\mathbf{N}(xs6)) \wedge (xs6 \wedge p_6 \rightarrow \forall\mathbf{N}(xs7)) \\ \wedge (xs7 \wedge p_7 \rightarrow \forall\mathbf{N}(xs8)) \wedge (xs8 \wedge p_8 \rightarrow \forall\mathbf{N}(xs9)) \wedge (xs9 \wedge p_9 \rightarrow \forall\mathbf{N}(xs10))\end{pmatrix}$$

where $p_1 \leftrightarrow u45 \wedge u46$

$p_2 \leftrightarrow z22 \wedge \neg u17 \wedge \neg u20 \wedge u45 \wedge u46 \wedge u47 \wedge z20$

$p_3 \leftrightarrow to1 \wedge u47 \wedge z20$

$p_4 \leftrightarrow to6 \wedge u45 \wedge u46 \wedge u47 \wedge z20$

$p_5 \leftrightarrow to8 \wedge u22 \wedge u23 \wedge u45 \wedge u46$

$p_6 \leftrightarrow to8$

$p_7 \leftrightarrow to7$

$p_8 \leftrightarrow \neg u45 \wedge u46$

$p_9 \leftrightarrow \mathrm{u48} \wedge \neg u45 \wedge u46$

Shutdown operating sequences

$$\forall\mathbf{G}((xs5 \vee xs6 \vee xs7 \vee xs8 \vee xs9) \wedge z21 \rightarrow \forall\mathbf{N}(xs10))$$

Secondary operating sequences

$$\forall\mathbf{G}(xs2 \wedge \neg u45 \wedge u46 \rightarrow \forall\mathbf{N}(xs1))$$

$$\forall\mathbf{G}(xs4 \wedge to4 \wedge \neg u45 \wedge u46 \rightarrow \forall\mathbf{N}(xs2))$$

$$\forall\mathbf{G}(xs4 \wedge (u17 \vee u20 \vee \neg z22 \vee \neg u47) \rightarrow \forall\mathbf{N}(xs3))$$

$$\forall\mathbf{G}(xs10 \wedge to5 \rightarrow \forall\mathbf{N}(xs3))$$

$$\forall\mathbf{G}(xs9 \wedge u45 \wedge u46 \rightarrow \forall\mathbf{N}(xs7))$$

Table 4.2: Verification results and computational statistics for MOD5™

Specification	Feasibility	No. of nodes explored	CPU time (s)
No. 1	infeasible	50	3.74[1]
No. 2	infeasible	35	2.55[1]
No. 3	infeasible	66	4.16[1]
No. 4	infeasible	0	0.92[1]
Normal sequences	infeasible	82	3.26[2]
Shutdown sequences	infeasible	34	0.66[2]
Secondary sequences	infeasible	38	1.76[2]

[1]CPU time using GAMS/OSL on HP9000/735

[2]CPU time using GAMS/CPLEX on HP9000/J200

Table 4.3: Statistics on problem size for Throx sequence

Category	Number
No. of Propositions	775
No. of Inputs	392
No. of State Variables	97
No. of Outputs	85
No. of Inequalities in the IP feasibility problem	4459

4.1.3 Results

The verification problem is formulated as a Boolean satisfiability problem by combining the implicit Boolean state space model with each specification. All the logical propositions in the Boolean satisfiability problem are automatically transformed to their equivalent inequality constraints by the algorithm in Appendix A. The resulting integer programming feasibility problem is solved by standard branch and bound codes in GAMS (Brooke et al., 1992). The linear programming sub-problems during branch and bound search are solved by the primal simplex method. Table 4.2 shows verification results with computational statistics. Note that the IP feasibility problems were resolved without much computational effort, especially, the number of nodes visited during branch and bound search is extremely small. All IP feasibility problems are infeasible, which means that all the specifications are satisfied by the model. Therefore, the logic design is formally verified with respect to the specifications tested.

4.2 Throx Sequence in MDI Plant

The Throx sequence is a burner management system implemented in an MDI plant. It consists of five sequences, `SEQ(30)`, `SEQ(31)`, `SEQ(32)`, `SEQ(34)`, and `SEQ(36)` programmed in DOWTRAN™. Each sequence contains shutdown interlocks, an operation sequence involving multiple steps, and related output and abort logics. The sequence `SEQ(36)` is only considered in this case study because it is the largest system with respect to the number of logical propositions and the number of variables.

4.2.1 Implicit Boolean State Space Model

The sequence `SEQ(36)` contains 56 shutdown interlocks, an operation sequence involving 13 steps, and related output and abort logics coupled together. Table 4.3 shows the problem size. The implicit Boolean state space model derived from the DOWTRAN™ code and the variables used in the model are listed in Appendix C.

4.2.2 Specifications

The requirements for the sequence SEQ(36) are partially listed as comments in the DOWTRAN™ codes. Although it is very difficult to derive or formulate specifications to be tested due to insufficient explanation of the codes, 22 specifications are derived from the comments in the codes. Note that we do not try to derive specifications based on *incorrect* assumptions.

4.2.3 Results

Several possible implementation or coding errors were found while deriving logical propositions from the codes, and are listed in Table 4.5 (i.e., the code is clearly inconsistent with the comments). These errors would have been found by the implicit model checking algorithm as well as by a careful study of the code, so it is unfair to attribute them to our method.

Table 4.4 : Specification list for Throx sequence

No.	Formal specification	Description in the comments
1	$\forall\mathbf{G}(\mathit{SHDN}(333) \rightarrow \mathit{STEP}(363) \wedge \neg \mathit{DO}(372))$	ALM(333) jumps into STEP(363) and takes DO(372) away.
2	$\forall\mathbf{G}(\mathit{WARN}(336) \rightarrow \mathit{DA}(338))$	This alarm prevents the system to ignite the main burner.
3	$\forall\mathbf{G}(\mathit{SHDN}(337) \rightarrow \mathit{STEP}(363))$	Program jumps to the shutdown step.
4	$\forall\mathbf{G}(\mathit{WARN}(339) \rightarrow \mathit{DA}(338))$	Process of ignition will be aborted
5	$\forall\mathbf{G}(\mathit{STEP}(354) \wedge \mathit{WARN}(344) \rightarrow \forall\mathbf{N}(\mathit{STEP}(354)))$	STEP(354) cannot be left when WARN(344) goes on.
6	$\forall\mathbf{G}(\mathit{WARN}(346) \rightarrow \mathit{STEP}(354))$	Program should jump back to STEP(354).
7	$\forall\mathbf{G}(\mathit{WARN}(347) \rightarrow \mathit{DA}(338))$	Avoids further ignitions.
8	$\forall\mathbf{G}(\mathit{STEP}(355) \wedge \mathit{WARN}(348) \rightarrow \forall\mathbf{N}(\mathit{STEP}(354)))$	In STEP(355), jumps back to STEP(354).
9	$\forall\mathbf{G}(\mathit{STEP}(355) \wedge \mathit{WARN}(364) \rightarrow \forall\mathbf{N}(\mathit{STEP}(354)))$	In STEP(355), jumps back to STEP(354).
10	$\forall\mathbf{G}(\mathit{WARN}(377) \rightarrow \mathit{STEP}(363))$	Jump into the shutdown step.
11	$\forall\mathbf{G}(\mathit{STEP}(358) \wedge \mathit{ALM}(378) \rightarrow \forall\mathbf{X}(\mathit{STEP}(363)))$	There is a shutdown if in STEP(358) ALM(378) is activated.
12	$\forall\mathbf{G}(\mathit{WARN}(380) \rightarrow \mathit{ALM}(392) \wedge \mathit{STEP}(363))$	WARN(380) causes ALM(392).
13	$\forall\mathbf{G}(\mathit{WARN}(381) \rightarrow \mathit{STEP}(357))$	If WARN(381), then jump into STEP(357).
14	$\forall\mathbf{G}(\mathit{WARN}(383) \rightarrow \mathit{STEP}(357))$	If WARN(383), then jump into STEP(357).
15	$\forall\mathbf{G}(\mathit{WARN}(384) \rightarrow \mathit{STEP}(357) \wedge \mathit{ALM}(336))$	If WARN(384), then jump into STEP(357) and release of ALM(336).
16	$\forall\mathbf{G}(\mathit{WARN}(387) \rightarrow \mathit{ALM}(338))$	WARN(387) activates the residue shutdown ALM(338).
17	$\forall\mathbf{G}(\mathit{WARN}(388) \rightarrow \mathit{ALM}(338))$	WARN(388) activates the residue shutdown ALM(338).
18	$\forall\mathbf{G}(\mathit{WARN}(389) \rightarrow \mathit{STEP}(357) \wedge \mathit{ALM}(336))$	WARN(389) activates main burner shutdown ALM(336) and jumps into STEP(357).
19	$\forall\mathbf{G}(\mathit{WARN}(390) \rightarrow \mathit{ALM}(338))$	ALM(390) activates the residue shutdown.
20	$\forall\mathbf{G}(\mathit{STEP}(356) \vee \mathit{STEP}(357) \rightarrow \mathit{ALM}(334))$	If STEP(356) or STEP(357), then ALM(334).
21	$\forall\mathbf{G}(\mathit{ALM}(393) \rightarrow \mathit{ALM}(389))$	If ALM(393), then ALM(389).
22	$\forall\mathbf{G}(\mathit{ALM}(394) \rightarrow \mathit{STEP}(357))$	If ALM(394), then jump into STEP(357).

Table 4.5: Possible implementation errors in Throx sequence

Place	Original Code	Corrected Code
WARN(336)	`WARN(336) IF [AC(1777) GT AC(1776) AND AC(1777) LE AC(1776)] …`	`WARN(336) IF [AC(1777) GT ZERO AND AC(1777) LE AC(1776)] …`
WARN(367)	`DC(1286) IF AC(343) LT AP(1014) OR [DC(1003) AND AC(343) LT AC(1015)] …`	`DC(1286) IF AC(343) LT AP(1014) OR [DC(1286) AND AC(343) LT AC(1015)] …`
WARN(367)	`DC(1004) IF AC(334) LT AP(1177) OR [DC(1003) AND AC(334) LT AC(1196)] …`	`DC(1004) IF AC(334) LT AP(1177) OR [DC(1004) AND AC(334) LT AC(1196)] …`
WARN(367)	`DC(1287) IF AC(344) LT AP(1177) OR [DC(1003) AND AC(344) LT AC(1196)] …`	`DC(1287) IF AC(344) LT AP(1177) OR [DC(1287) AND AC(344) LT AC(1196)] …`
WARN(387)	`DC(1303) IF [AC(333) GT AP(1366,860,2000) OR AC(343) GT AP(1366)] AND [AC(334) GT AP(1367,860,2000) OR AC(344) GT AP(1367)] AND STEP(359) OR [STEP(360) OR STEP(361) OR STEP(362)]`	`DC(1303) IF [AC(333) GT AP(1366,860,2000) OR AC(343) GT AP(1366)] AND [AC(334) GT AP(1367,860,2000) OR AC(344) GT AP(1367)] AND [STEP(359) OR STEP(360) OR STEP(361) OR STEP(362)]`
DO(335) and DO(336)	`DO(335) IF STEP(358) OR STEP(359) OR STEP(360) OR STEP(361) OR STEP(362) AND DC(2453) AND #[ALM(378) OR ALM(336)] AND DI(321)` `DO(336) IF STEP(358) OR STEP(359) OR STEP(360) OR STEP(361) OR STEP(362) AND DC(2453) AND #[ALM(378) OR ALM(336)] AND DI(321)`	Since DO(335) and DO(336) are digital outputs for main-gas block valve 1 and main-gas block valve 2 respectably, logic representing DO(335) and DO(336) should be different. However, they are identical in the codes.

Table 4.6: Verification results and computational statistics for Throx sequence

Spec. No.	Feasibility	No. of nodes explored	CPU time[1] (s)
1	feasible	11	3.23
2	infeasible	10	3.47
3	feasible	155	19.41
4	infeasible	34	5.54
5	infeasible	6	3.01
6	feasible	33	6.55
7	infeasible	23	5.58
8	infeasible	9	3.15
9	infeasible	29	5.28
10	feasible	370	30.4
11	infeasible	6	2.69
12	infeasible	10	4.22
13	feasible	21	4.08
14	feasible	10	2.76
15	feasible	14	3.29
16	feasible	2	2.60
17	feasible	26	3.91
18	feasible	3	2.04
19	feasible	2	2.43
20	infeasible	3	2.50
21	feasible	19	3.54
22	feasible	4	2.40

[1]CPU time using GAMS/CPLEX on HP9000/J200

The verification problem is formulated as a Boolean satisfiability problem by combining the implicit Boolean state space model with each specification. All the logical propositions in the Boolean satisfiability problem are automatically transformed to their equivalent inequality constraints by the algorithm in Appendix A. The resulting integer programming feasibility problem is solved by standard branch and bound codes in GAMS (Brooke et al., 1992). The linear programming sub-problems during the branch and bound search are solved by the primal simplex method. Table 4.6 shows verification results with computational statistics. Note that all the IP feasibility problems were resolved in less than a minute in terms of CPU time, and the number of nodes visited during the branch and bound search is very small.

Some IP problems are infeasible, which means that the specifications are satisfied by the model. However, some IP problems are feasible, which means that the model is inconsistent with the specifications. There can be two explanations in this case. First, the specification may be formulated incorrectly. This is possible because we do not have complete knowledge about this Throx sequence. However, note that all the specifications in Table 4.4 are almost direct translations of comments in the codes. Second, the specification is not satisfied by the model. In other words, the comments in the codes are not consistent with the functionality implemented by the code. Again, this may indicate an error with the code or the comment. At this point, we do not have enough information to determine the causes of feasibility of some IP problems. If the IP problem is feasible, each feasible solution can be used as a counter-example to find errors in the design. Literally, a feasible solution of IP problem is a set of variables and their values violating the original specification tested. This information combined with the model should help users to find causes of feasibility of the IP problem.

4.3 Empirical Complexity of the Algorithm

Any formal verification method will ultimately be limited by the combinatorial nature of the problem. Therefore, the best way to prove the value of any approach is to apply it to a series of problems of increasing size. In order to obtain empirical results on the performance of our approach, we formulate a global burner management system by combining a number of single MOD5™ burner management systems in parallel and by adding a global shutdown logic as shown in Figure 4.4. Then, the global model is tested against five specifications listed in Table 4.7 by increasing the number of single burners from 1 to 10.

All the integer programming feasibility problems are solved by standard branch and bound codes in GAMS/OSL (Brooke et al., 1992). The linear programming subproblems during branch and bound search are solved by the primal simplex method. All the IP problems tested are infeasible, meaning that all the specifications tested are satisfied by the model.

Table 4.8 and Figure 4.5 show model statistics. Note that the problem size in terms of the number of constraints and total number of variables is increasing linearly with the number of state variables, which demonstrates that the size of the model that can be constructed is not limited by the amount of computer memory available due to its implicitness. Therefore, very large-scale verification problems can be formulated in our approach, which is impossible in explicit enumeration-based model checking.

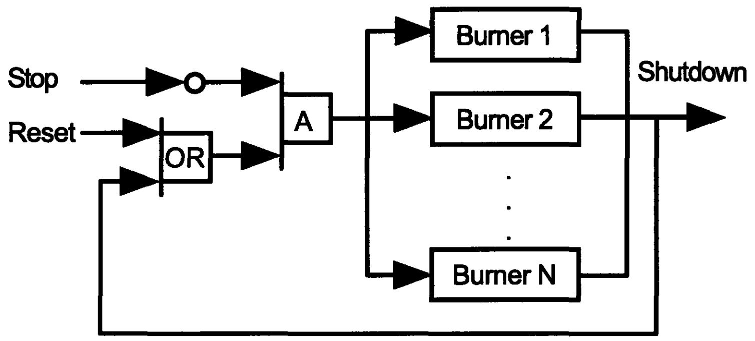


Figure 4.4: Global burner management system

Table 4.7: Specification list for global burner management system

No.	Informal specification	Formal specification
1	STOP signal should bring all burners in safe shutdown states, and should overrides RESET signal.	$\forall\mathbf{G}(\mathrm{STOP} \wedge \mathrm{RESET} \rightarrow \mathrm{STEP}_1(112) \wedge \ldots \wedge \mathrm{STEP}_N(112))$
2	Once STOP signal becomes ON, shutdown signal should remain ON even though STOP signal becomes OFF.	$\forall\mathbf{G}(\neg\mathrm{STOP} \wedge \mathrm{SHUTDOWN} \rightarrow \forall\mathbf{N}(\mathrm{SHUTDOWN}))$
3	Upon receiving STOP signal, outputs of all main fuel downstream block valves should be aborted.	$\forall\mathbf{G}(\mathrm{STOP} \rightarrow \neg(\mathrm{DO}_1(103) \vee \ldots \vee \mathrm{DO}_N(103)))$
4	Upon receiving STOP signal, outputs of all pilot gas upstream block valves should be aborted.	$\forall\mathbf{G}(\mathrm{STOP} \rightarrow \neg(\mathrm{DO}_1(112) \vee \ldots \vee \mathrm{DO}_N(112)))$
5	Upon receiving STOP signal, outputs of all burner electronic ignitors should be aborted.	$\forall\mathbf{G}(\mathrm{STOP} \rightarrow \neg(\mathrm{DO}_1(114) \vee \ldots \vee \mathrm{DO}_N(114)))$

Table 4.8: Model statistics for global burner management system

Unit	n	m	Total	Constraints	States	Transitions
1	45	67	169	1116	3.5×10^{13}	5.2×10^{33}
2	89	132	335	2220	6.2×10^{26}	3.4×10^{66}
4	177	262	667	4428	1.9×10^{53}	1.4×10^{132}
6	265	392	999	6636	5.9×10^{79}	6.0×10^{197}
8	353	522	1331	8844	1.8×10^{106}	2.5×10^{263}
10	441	652	1663	11052	5.7×10^{132}	4.8×10^{474}

Unit : the number of single burners in the global burner management system
n : the number of state variables
m : the number of inputs
Total : the total number of variables (inputs, outputs, state variables, internal variables)
Constraints : the number of inequality constraints generated
States : the upper bound on the number of possible states
Transitions : the upper bound on the number of possible transitions

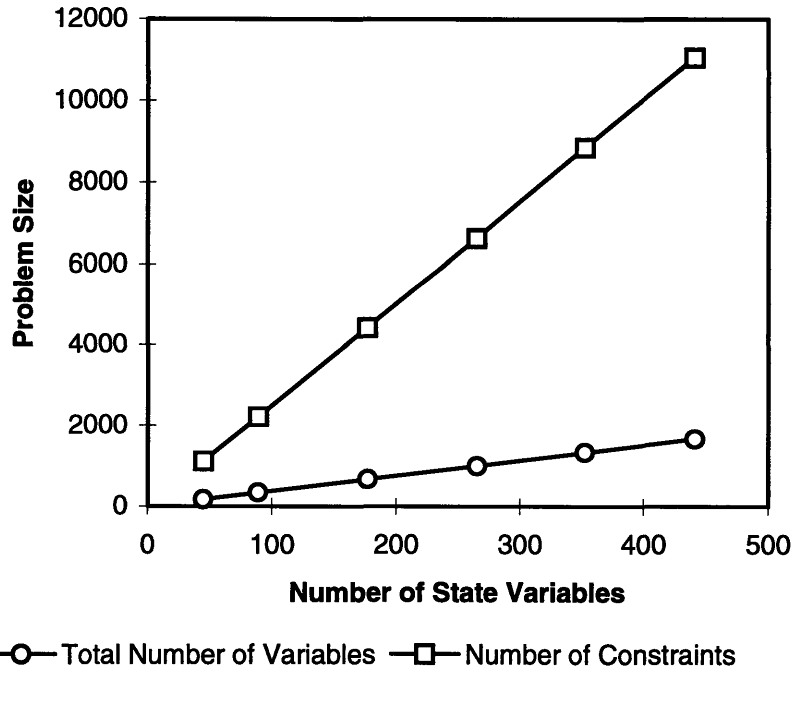


Figure 4.5: Problem size with respect to the number of state variables.

Table 4.9: Verification results for global burner management system

	Specification 1		Specification 2		Specification 3		Specification 4		Specification 5	
n	Nodes	CPU	Nodes	CPU	Nodes	CPU	Nodes	CPU	Nodes	CPU
45	0	1.56	10	2.46	0	2.45	0	2.67	0	1.64
89	0	7.72	6	5.25	21	11.46	0	8.22	21	8.63
177	41	38.48	21	32.45	41	43.21	41	35.60	21	24.47
265	61	67.81	41	66.06	61	62.19	41	73.38	41	70.46
353	61	122.34	61	146.31	61	138.92	81	101.73	81	163.58
441	141	296.44	84	211.51	241	447.40	81	211.83	81	235.60

n : the number of state variables
Nodes : the number of binary nodes visited during branch and bound search
CPU : CPU times in seconds using GAMS/OSL on HP9000/735

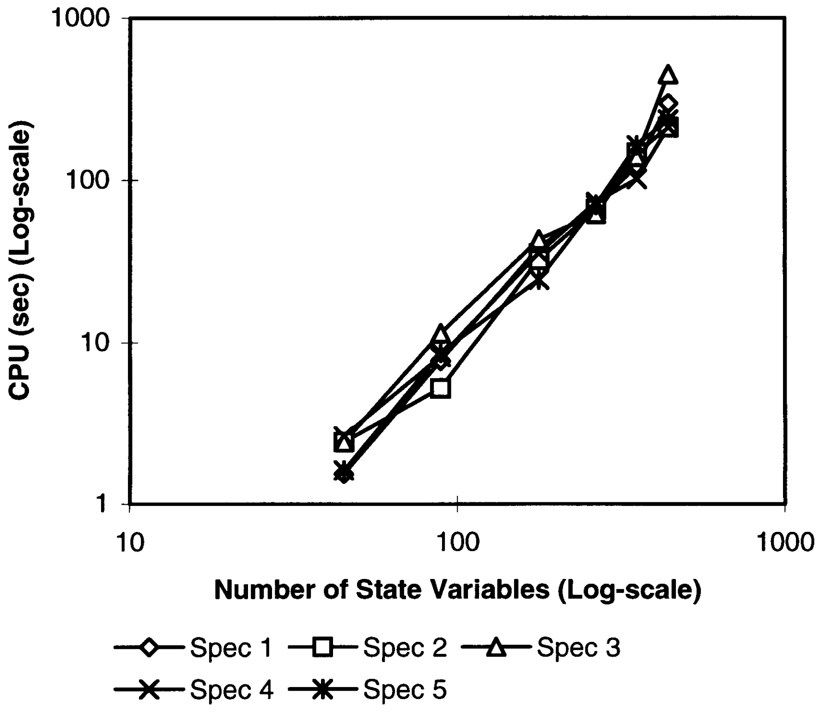


Figure 4.6: Verification time with respect to problem size

Table 4.9 shows the verification performance in terms of the number of state variables. First of all, the largest verification problem has 441 state variables and 652 inputs, resulting in the possibility of more than 10^{132} states and 10^{474} transitions. Verification took from 3.5 to 7.5 minutes of CPU time depending upon the specifications tested. Note that the IP feasibility problem was resolved by just solving the relaxed linear programming problem in several cases, and in other cases, the number of nodes visited during branch and bound search for IP solution is very small. Figure 4.6 shows how the verification time depends on the problem size in terms of the number of state variables on log-log scale. On a log-log scale plot, the polynomially growing function $y = x^n$ appears as a straight line with a slope n. Note that in all cases, the verification time is growing roughly quadratically (i.e., $n \approx 2$) with respect to the number of state variables.

4.4 Identification of Defects

The purpose of this section is to demonstrate that the implicit model checking can be used to identify design defects in the LCS logic for large-scale problems. The single MOD5™ burner management system is used as an example. First, a known bug is deliberately introduced into the burner system, and a relevant specification is tested to identify the bug inserted. Second, a set of random bugs are introduced into the burner system, and the implicit model checking is applied to detect these bugs.

The MOD5™ burner management system contains 7 output and abort logics coupled together. One of the specifications that should be satisfied is:

$$\forall \mathbf{G}\left(\neg z_{15} \wedge \neg z_{17} \rightarrow \neg x_{13} \wedge \neg y_6\right) \tag{4.1}$$

which means that the abort outputs `DO(103)` and `DO(114)` must be turned off if `DC(101)` (main vent ebv) and `DC(111)` (pilot vent ebv) are both open. Note that in order to verify this specification, all abort logics are required together with related shutdown interlocks and sequence logics. The specification is formally proved against the original model. It takes 0.33 seconds using `GAMS/CPLEX` on `HP9000/J200` to solve the

IP feasibility problem. The original model is then corrupted by introducing a bug into the following two propositions in the abort logics:

```
(x13 <-> (z16 & ~z3)).
(y6 <-> (z19 & ~z13)).
```

Each negation is removed and each conjunction is replaced by disjunction. The specification in Eqn (4.1) is tested against the corrupted model. The IP feasibility problem is feasible, meaning that the specification is violated in the corrupted model. It takes 0.43 seconds using `GAMS/CPLEX` on `HP9000/J200`. The counter-example generated is `[x13,y6,z3,z13,z15,z16,z17,z19]=[T,T,T,T,F,F,F,F]`, which clearly shows that the abort outputs `DO(103)` and `DO(114)` are turned on even though `DC(101)` and `DC(111)` are both open.

As a second example, a set of random bugs are introduced into the burner system, and the implicit model checking is applied to this randomly corrupted model. The 10 propositions are randomly chosen from the model in Appendix B:

```
(z23 <-> ((u4 # u60) & (u6 # u61) & (x12 # z32))).
(z25 <-> ((u1#u53)&(u5#u55)& ~xs1 & ~xs2)).
(x5 <-> ((z27 & ~x5p) # (~u47 & x5p))).
(z35 <-> (xs6#xs7#xs8#xs9)).
(z30 <-> ((u7#u64)&(u9#u65))).
(z1 <-> (~z15 # z21 # z36)).
(z16 <-> (x10 & ~z21)).
(z17 <-> ((xs5#(xs6&u52)) & ~z21)).
(y5 <-> (x11 & ~z9)).
(z11 <-> (~z18 # z21 # z37)).
```

which are then corrupted as:

```
(z23 <-> ((u4 & u60) & (u6 # u61) & (x12 # z32))).
(z25 <-> ((u1#u53)#(u5#u55)# ~xs1 # ~xs2)).
(x5 <-> ((z27 & x5p) # (~u47 & x5p))).
(z35 <-> (xs6&xs7&xs8&xs9)).
(z30 <-> ((u7&u64)&(u9#u65))).
(z1 <-> (~z15 & z21 & z36)).
(z16 <-> (x10 # ~z21)).
(z17 <-> ((xs5#(xs6&u52)) & z21)).
(y5 <-> (x11 & z9)).
(z11 <-> (z18 # z21 # z37)).
```

Note that in order to find a particular bug in a LCS design, the model should be tested against a relevant set of specifications, which are not known in advance. Therefore, all the specifications in section 4.1.2 including Eqn (4.1) are tested against the corrupted model. The IP feasibility problem is solved for each specification. However, only one specification (the specification No. 4 in section 4.1.2) was able to identify a bug introduced in the 3rd proposition. The IP feasibility problem was feasible, and the counter-example is `[u19,u24,u28,u29,z27,x5]=[F,F,F,F,T,F]`, which shows that the alarm `ALM(115)` (or `x5`) does not sound even though the set signal `SET(115)` (or `z27`) is turned on by the activation of emergency stop switches. It takes 0.49 seconds using `GAMS/CPLEX` on `HP9000/J200`. All the other specifications are satisfied by the corrupted model, thus fail to identify any bug. This example clearly illustrates that a complete set of specifications should be tested against the model in order to identify all defects in the LCS design.

4.5 Conclusions

It has been demonstrated that formal verification of large-scale industrial logic-based control systems can be conducted efficiently without encountering the state explosion problem. The implicit model checking technology is applied to two burner management systems, MOD5™ and the Throx sequence. Properties of interest are formally specified in temporal logic, and are formally checked against the Boolean state space model. For the Throx sequence, some of the specifications are found to be inconsistent with the model.

A series of burner management systems that combine a number of a single burner management system in parallel is solved by the implicit model checking algorithm to explore the empirical complexity of the algorithm. The problem size in terms of the number of inequalities increases linearly with respect to the number of state variables. It has been shown empirically that the solution time increases quadratically with respect to the number of state variables. This high efficiency in terms of computational cost is because the IP feasibility problem was solved by just solving the relaxed linear programming problem in many cases, and in other cases, the number of nodes visited during branch and bound search for IP solution is very small due to the very efficient implicit enumeration conducted by branch and bound algorithms.

Chapter 5

Dynamic Validation of Logic-based Control Systems using Hybrid Discrete/Continuous Simulation

5.1 Introduction

The functionality of a LCS can be verified against formal specifications by employing the implicit model checking technology introduced in chapter 3. Even if the functionality of a LCS is fully consistent with a complete set of specifications, the LCS may fail to provide an appropriate action when called upon. This situation arises because the functionality of the LCS considering its dynamic coupling with the underlying chemical process has not been verified. In other words, formal verification of the LCS in isolation is a necessary but not sufficient condition for verification of the coupled system (plant and LCS).

The notion of *hybrid discrete/continuous dynamic systems* (that will be defined in the next section) can be used to formulate the problem of verifying the coupled plant and LCS. However, only a very limited class of hybrid dynamic systems (with respect to model complexity) can be verified in a formal manner. A typical sub-problem in the verification problem is *reachability analysis* - given a final state, determine whether or not the hybrid system has a sequence of computation terminating at that final state from a given initial state. This reachability problem has already been shown to be undecidable for very restricted classes of hybrid systems, e.g., constant slope hybrid systems, in which the right hand side of all differential equations is an integer constant (Alur et al., 1993; Kestne et al., 1993).

The hybrid system composed of the LCS and the underlying nonlinear chemical process cannot be verified because no formal analytical tools will become available. However, it is essential at least to validate the performance of the hybrid system via simulation even though it does not amount to formal verification (Lygeros et al., 1994). In particular, hybrid discrete/continuous simulation technology can be used to study the overall system response in a set of critical abnormal scenarios identified by a process hazard analysis, and allows extensive system checkout to be done prior to plant start-up. The results of this validation may be used to amend the LCS design by identifying problems with the original specifications. For example, in the pre-commissioning of the THORP nuclear fuel reprocessing plant in Cumbria, UK, a dynamic plant model including details of control systems was built in order to facilitate the early testing of the control systems, which were critical for the safety of the plant, consequently eliminating hundreds of bugs even before the software was implemented on plant (Evans and Wylie, 1990).

Even though it has been recognized that the role of dynamic validation is critical to identify any potential defects in the LCS, it is often left out in practice because current dynamic simulation tools are not suitable for dynamic validation of the LCS, and it is a very costly and time consuming task to develop plant models that include details of the LCS from scratch using a programming language. For example, the general-purpose dynamic simulation packages (e.g., DIVA (Holl et al., 1988), DYNSIM (Gani et al., 1992), POLYRED (Ray, 1993), SpeedUp (AspenTech, 1993), etc.) are mainly concerned with modeling physico-chemical systems, and do not provide facilities for effective modeling of complex discrete systems (Pantelides and Barton, 1993). Recent work on general-purpose dynamic simulation technology (gPROMS (Barton and Pantelides, 1993) and ABACUSS (Allgor et al., 1996)) combines models of physico-chemical phenomena (possibly discontinuous) with the systematic modeling of sequences of control actions (Barton and Pantelides, 1994), although it does not yet include models of the LCS. Some simulation packages have the capability to model logic systems in addition to the physico-chemical system (CADAS (Eikaas, 1990), PROTISS (Goldfarb et al., 1993), TRAINER (Morton, 1992)). However, these tools appear to have drawbacks. Models of the LCS are usually solved in an ad hoc manner such as the sequential modular approach (Goldfarb et al., 1993; Morton, 1992).

In this chapter, dynamic validation of the LCS is formulated as a hybrid discrete/continuous simulation problem, and an efficient solution algorithm for simultaneous dynamic simulation of the LCS and the physico-chemical phenomena will be developed. This technology will provide significant advances in the efficiency of model construction efforts and accuracy of model solution.

5.2 Mixed-mode Simulation of Integrated Circuits

Circuit simulation is used routinely in the design of integrated circuits (ICs) to validate circuit correctness. Circuits can be simulated at the electrical level or at the logic level depending upon the level of abstraction required. In electrical simulation, a circuit is represented as a system of nonlinear, first-order, ordinary differential equations, which model the dynamic characteristics of the circuit for a set of applied input signals and initial conditions. In logic simulation, the circuit is abstracted as a network of logic gates, and the functionality of the circuit is represented by discrete logic states determined by simple Boolean operations. Logic simulation is typically 100 to 1000 times faster than the most efficient forms of electrical simulation.

There are many cases for which one level of simulation is not sufficient for the analysis of a circuit design. For example, complex VLSI circuits usually include both analog and digital components in a single chip, and the integration of analog/digital systems on a single chip is ever increasing. Typical examples include voltage regulators, speech recognition circuits, and power up/down sense circuits. Logic simulators cannot model analog circuits, and it is too expensive to simulate the entire mixed analog/digital circuit at the electrical level. In this case, a new modeling and simulation technique is required that allows different portions of the circuit to be described and simulated at different levels of abstraction. The problem of modeling and simulation of a mixed analog/digital circuit is known as *mixed-mode simulation* (Saleh et al., 1994). Mixed-mode simulation has been popular, and a large number of simulators (e.g., Attsim, iSPLICE3, Saber, VHDeLDO) have been developed over the last few years.

There are a number of issues inherent in the mixed-mode simulation problem. For example, accurate mixed-mode simulation requires a consistent representation of the circuit signals over two different simulation levels. In the logic level, the signal is represented as bits or logic values. On the other hand, electrical simulation uses real numbers to represent values of voltage and current. These different signal types should be mapped to each other correctly at the interface. Second, time is usually represented as a real number in electrical simulation and as an integer in logic simulation. This discrepancy between different representations of time must be resolved. Furthermore, time-step synchronization between the electrical and logic simulators is important in determining the accuracy and speed of mixed-mode simulation.

Mixed-mode simulation of circuit is mathematically similar to hybrid discrete/continuous simulation of a LCS and its chemical process. Two physically different systems give rise to similar mathematical problems: a combined simulation problem involving simultaneous solution of a differential equation based model and a logic based model. However, there are important differences between the two problems. First, the differential equation models are different: dynamic models of typical chemical processes require the general nonlinear form of differential-algebraic equations, whereas ordinary differential equations are typically used to model dynamic behavior of circuits (Saleh et al., 1994). Second, circuit simulation even at the logic level requires detailed timing information because the two circuit models at the electrical and logic level effectively describe the same physical system at different levels of abstraction and the time delays of the model at the logic level are of similar order to the time constants of the analog dynamics. However, as we argue in this thesis, a model of the LCS does not require detailed timing information. Finally, the nature of mixed-mode simulation demands an interface between the two different types of model. Again, the difference between the physical systems alters the nature of this interface.

5.3 Hybrid Discrete/Continuous Systems

5.3.1 Definitions

There are many examples in the chemical processing industries in which discrete phenomena superimposed on the more familiar continuous system dynamics make a significant contribution to the overall system response. In these situations, the behavior of a typical chemical plant is probably more correctly viewed as that arising from a series of coupled and interacting discrete and continuous subsystems. In particular, the strong coupling of these two facets of process behavior in many problems of practical interest demands the development of analytical technologies that can address discrete and continuous behavior appropriately and simultaneously.

The process transients in the chemical processing industries are fundamentally of hybrid nature. While continuous behavior arises in a familiar manner from phenomena such as mass, energy, and momentum conservation, it is useful to classify discrete behavior into two broad categories (Barton and Pantelides, 1994): physico-chemical or autonomous discontinuities, and discrete controls and/or disturbances. Physico-chemical discontinuities arise as an integral part of the physical behavior of most systems, especially if system behavior is studied over a large enough region of state space. Numerous examples include phase changes, flow reversals, shocks and transitions (e.g., laminar to turbulent), discontinuities in equipment geometry, internal features of vessels (e.g., weirs), etc.. In all of these cases, discrete phenomena occur purely as a consequence of the system moving through state space. On the other hand, discrete controls and/or disturbances are most frequently encountered in the study of process operations. Examples include digital regulatory control, the use of hybrid controllers to regulate highly nonlinear continuous dynamics (Friedrich and Perne, 1995; Rovaglio et al., 1995), process upsets and the action of automatic protective devices, and planned operational changes such as start-up, shut-down and feedstock changeovers. Further, there are whole classes of processes such as batch, semi-continuous and periodic processes that rely on discrete control actions to implement normal operation.

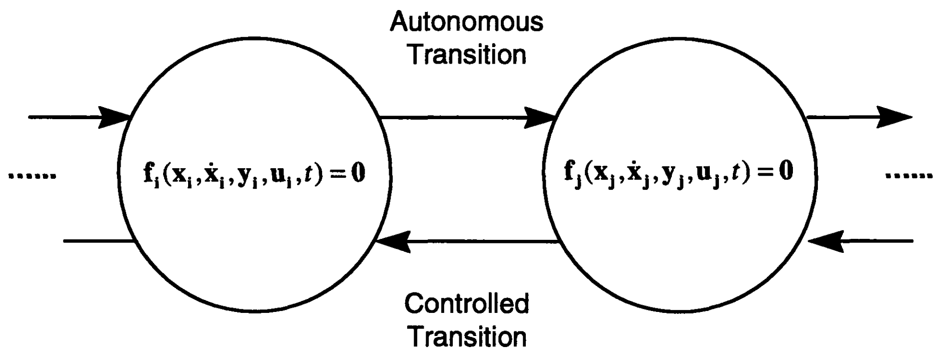

Autonomous Transition
$f_i(x_i,\dot{x}_i,y_i,u_i,t)=0$
$f_j(x_j,\dot{x}_j,y_j,u_j,t)=0$
Controlled Transition
......
......

Figure 5.1: Autonomous/Controlled transitions in hybrid systems

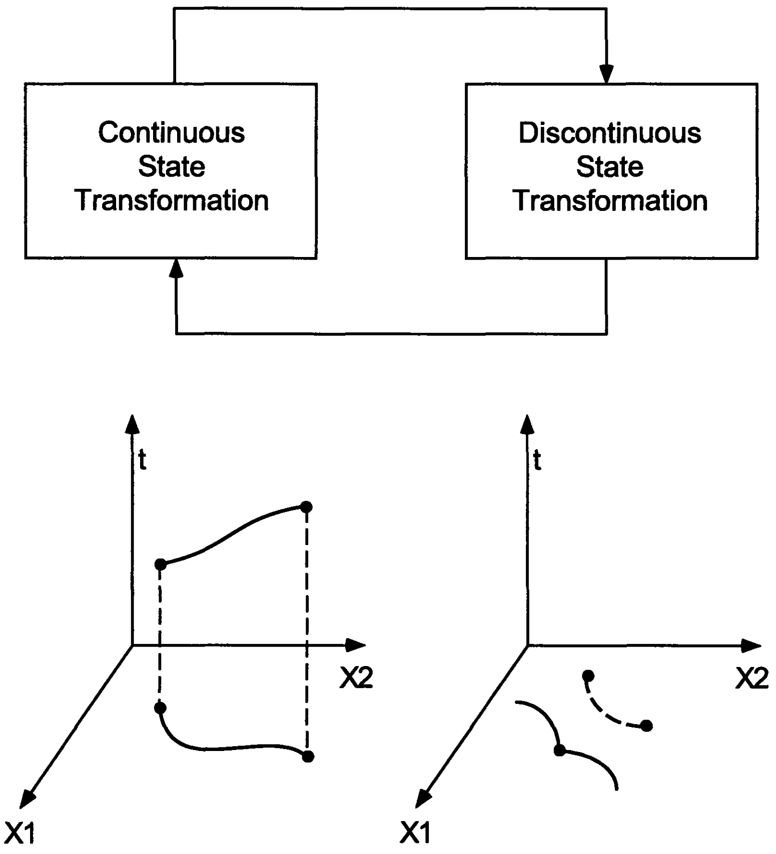

Continuous State Transformation
Discontinuous State Transformation
t
X2
X1
t
X2
X1

Figure 5.2: Continuous and discontinuous state transformation

Depending upon a particular application, various classes of hybrid systems can arise with respect to the configuration and types of interaction between continuous and discrete sub-systems. However, at the lowest level *hybrid systems* are defined as discrete and continuous sub-systems inherently coupled and interacting with each other (Grossman et al., 1993; Nerode and Kohn, 1994). For the purpose of this thesis, we employ the notion of hybrid systems focused on transitions between continuous sub-systems. Figure 5.1 depicts this notion of hybrid systems, where each node represents a continuous sub-system and each arc represents a transition between continuous sub-systems. Transitions can be *controlled* by discrete sub-systems or *autonomous*. It is usually assumed that these transitions are instantaneous, and the instant at which these transitions occur is called the *transition time*, which is in general not known *a priori* and is defined implicitly by *transition conditions*. The satisfaction of transition conditions is known as *events* in the dynamic simulation problem. The time or location of events can be defined explicitly (*time events*) or implicitly (*state events*).

With this notion of hybrid systems, hybrid systems experience a repeated sequence of two steps: a *continuous state transformation* and a *discontinuous state transformation* where a state is a point on the trajectory determined by the current continuous sub-system as shown in Figure 5.2. During a continuous state transformation, the current continuous sub-system is evolving over time continuously according to the set of governing equations. At transition times, a discontinuous state transformation occurs autonomously or by control actions exerted by discrete sub-systems. Discontinuous state transformations can be simple parameter settings or changes in the functional form of the continuous system, including dimensional change of the state space. Discontinuous state transformations result in a new current continuous sub-system, and continuous state transformation is resumed. In summary, a discontinuous state transformation changes the *mode* of hybrid system (active continuous sub-system at a given time).

Discontinuous state transformations or mode changes are hybrid phenomena that can occur in the class of hybrid systems we have defined. The hybrid system changes its mode based on the partition of the state space (not necessarily disjoint), which is determined by the inherent dynamics of the embedded continuous systems (for autonomous transitions) and/or by the interface between the continuous sub-systems and input symbols of discrete sub-systems (for controlled transitions). In either case, we know that events occur, which is the satisfaction of the corresponding transition conditions at the boundaries of the partition. For controlled transitions, output symbols of the discrete sub-systems are associated with modes of the hybrid system.

5.3.2 Hybrid Simulation Example - Tank Filling Operation

In order to illustrate hybrid system simulation of the LCS and the chemical process, consider the tank filling operation in chapter 1. Figure 1.6 is a flowsheet that consists of two tanks and a feed pump, and Figure 1.7 is a logic-based control system for this tank filling operation represented as a binary logic diagram. Fluid is pumped into either tank A or B. HS1 and HS2 are switches to start or stop the filling operation for tanks A and B respectively. The START (or STOP) position of switches HS1 and HS2 causes the control valves HV1 and HV2 to open (or close) respectively. In order to operate the pump, all the following conditions must be satisfied:

- Only one of two switches HS1 and HS2 is in the START position.
- The tank levels must be below a given value, L_{High}, as indicated by level switches LSH3 and LSH4.
- The pump suction pressure must be above a given value, as indicated by pressure switch PSL5.
- The pump is in the ON position.

The operation is stopped if any of the following conditions occur:

- While pumping into a tank, its control valve leaves the fully-open position, or the valve of the other tank leaves its fully-closed position.
- The tank selected for filling becomes full, i.e., $L_A \geq L_{High}$ (or $L_B \geq L_{High}$) where L_A (or L_B) is the level in tank A (or B).
- The pump suction pressure is continuously low for 5 seconds.
- The operation is stopped manually by HS1, HS2, or HS7.

Table 5.1: Events during simulation of tank filling operation

Time	Event
0	Tank A and B are partially filled initially.
t_1	`Start_filling_tank_A` is pressed.
t_2	`Tank_A_level_high` becomes TRUE.
t_3	`Start_filling_tank_B` is pressed.
t_4	`Stop_filling_tank_B` is pressed.

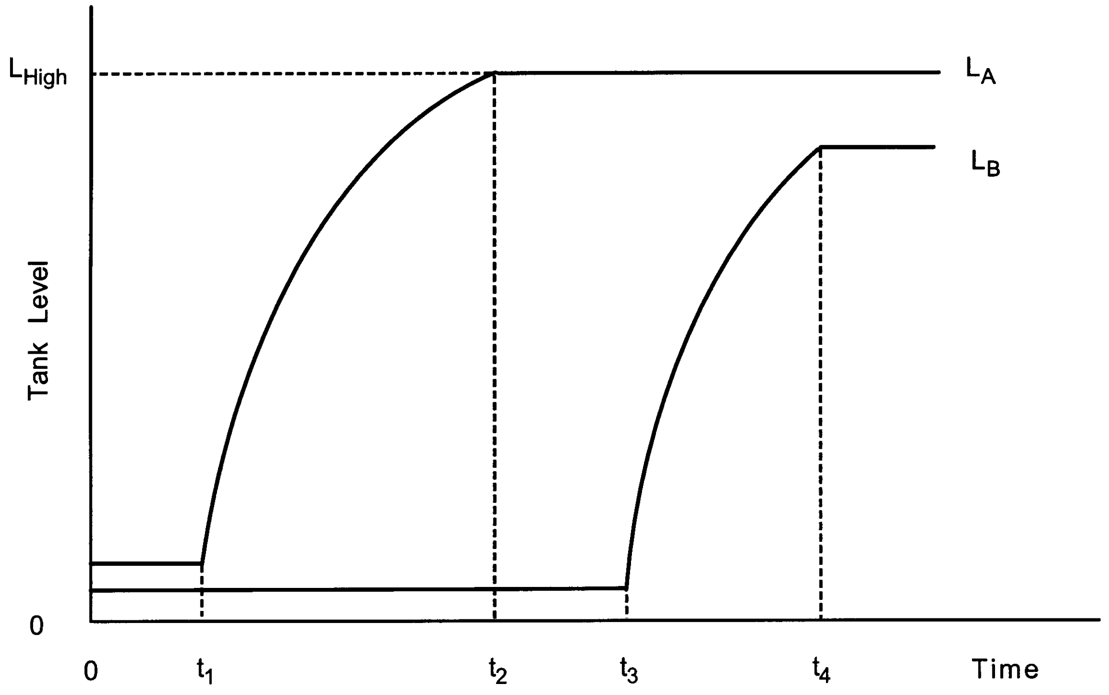


Figure 5.3: Simulation of tank filling operation

Consider the filling operation according to a particular scenario in Table 5.1. Initially, tank A and B are partially filled. At time t_1, the signal `Start_filling_tank_A` becomes `TRUE` by turning on the switch `HS1`, and all the other conditions to operate the pump are satisfied. Then, the inlet valve for tank A will be open and the signal `Operate_pump` is sent to the pump from the LCS by solving the permissive LCS logic. Once the pump is running, a continuous state transformation will immediately follow, and time will advance as the level rises. At time t_2, the level of tank A will exceed the limit, L_{High}, which sends a signal `Tank_A_level_high` to the LCS. This signal activates a shutdown LCS logic, which in turn sends a signal Stop_pump to the pump, and the filling operation for tank A is stopped. This marks a transition time. Note that this transition is defined implicitly by a condition $L_A \geq L_{High}$ (*state events*), and the actual time can only be determined by playing out a particular scenario. This triggers a discontinuous state transformation: the current continuous sub-system is switched to a mode in which the pressure drop across the pump is set to zero. Following this transition, the system experiences a continuous state transformation in this new mode (steady state) until the switch HS2 is turned on by the operator at time t_3. Then, the inlet valve for tank B will be open and the signal Operate_pump is sent to the pump again. This marks another transition time. Note that this transition is explicitly defined (*time events*). Following this discontinuous state transformation, the system reverts to another continuous state transformation and the level of tank B rises until the operator turns off the switch HS2 at time t_4. Figure 5.3 depicts the simulation result according to this particular filling operation. It is assumed in this plot that the opening and closing of valves are instantaneous. It should be noted that this sequence of continuous and discontinuous state transformations experienced by a hybrid system is a function of the specific scenario. In general, there may be a very large number of possible sequences a system may experience.

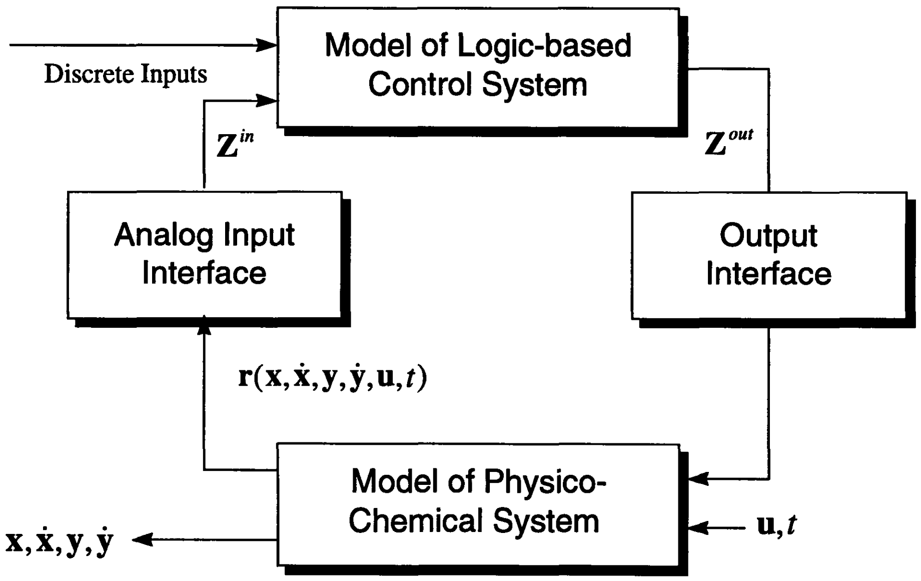

Discrete Inputs
Model of Logic-based Control System
$\mathbf{Z}^{in}$
$\mathbf{Z}^{out}$
Analog Input Interface
Output Interface
$\mathbf{r}(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\dot{\mathbf{y}},\mathbf{u},t)$
Model of Physico-Chemical System
$\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\dot{\mathbf{y}}$
$\mathbf{u},t$

Figure 5.4: Hybrid models of the LCS and the process

5.4 Mathematical Formulation

There are two main tasks in the study of hybrid simulation problem: model construction and solution algorithm. The model construction or mathematical formulation will be addressed in this section, and the solution algorithm will be developed in the next section.

The task of hybrid model construction requires three sub-tasks: modeling of continuous sub-system, modeling of discrete sub-system, and interface between two distinct models. Figure 5.4 shows the abstract relationship between models of the LCS and the physico-chemical system with input/output interfaces.

5.4.1 Model for Physico-Chemical Systems

The time dependent behavior of the lumped parameter physico-chemical system can be effectively modeled in terms of differential-algebraic equations (DAEs). Continuous state transformation or evolution of each mode can be described by an initial value problem, and the series of continuous state transformations can be modeled as a sequence of initial value problems interspersed by events that may cause some form of discrete change to the current initial value problem (Barton and Pantelides, 1994). As a consequence, the time interval of interest $\left[t^{(0)}, t^{(f)}\right]$ is partitioned into NCD continuous sub-intervals $\left[t^{(k-1)}, t^{(k)}\right)$, $\forall k = 1..NCD$. The initial time $t^{(0)}$ is given, whereas the sub-interval boundaries, $t^{(k)}$, $\forall k = 1..NCD$, may be specified explicitly or determined implicitly during the course of a simulation. The hybrid simulation problem can therefore be defined as:

$$\left.\begin{aligned} \mathbf{f}^{(k)}\left(\mathbf{x}^{(k)}, \dot{\mathbf{x}}^{(k)}, \mathbf{y}^{(k)}, \mathbf{u}^{(k)}, t\right) = \mathbf{0} \\ \mathbf{u}^{(k)} = \mathbf{u}^{(k)}(t) \end{aligned}\right\} \quad t \in \left[t^{(k-1)}, t^{(k)}\right), \ \forall k = 1..NCD \tag{5.1}$$

where $\mathbf{f}^{(k)} : \mathbf{R}^{n^{(k)}} \times \mathbf{R}^{n^{(k)}} \times \mathbf{R}^{m^{(k)}} \times \mathbf{R}^{l^{(k)}} \times \mathbf{R} \mapsto \mathbf{R}^{n^{(k)}+m^{(k)}}$, and $\mathbf{x}^{(k)} \in \mathbf{R}^{n^{(k)}}$, $\dot{\mathbf{x}}^{(k)} \in \mathbf{R}^{n^{(k)}}$, $\mathbf{y}^{(k)} \in \mathbf{R}^{m^{(k)}}$, $\mathbf{u}^{(k)} \in \mathbf{R}^{l^{(k)}}$. The unknowns $\mathbf{x}$ and $\mathbf{y}$ are usually referred to as the differential variables and algebraic variables respectively, and $\mathbf{u}$ are the known system inputs. The following condition is assumed:

$$\mathrm{Rank}\left[\mathbf{f}^{(k)}_{\dot{\mathbf{x}}^{(k)}}\,\mathbf{f}^{(k)}_{\mathbf{y}^{(k)}}\right] = n^{(k)} + m^{(k)},\ t \in \left[t^{(k-1)}, t^{(k)}\right),\ \forall k = 1..NCD \tag{5.2}$$

This is a sufficient condition for Eqn (5.1) to have a *differential index* equal to or less than unity. The differential index of Eqn (5.1) is defined as the minimum number of times that all or part of it must be differentiated with respect to time in order to define $\dot{\mathbf{x}}$ and $\dot{\mathbf{y}}$ uniquely as functions of $\mathbf{x}$, $\mathbf{y}$, $\mathbf{u}$ (and its time derivatives), and t (Brenan et al., 1989).

5.4.2 Model for Logic-based Control Systems

The revised BLD in chapter 2 will be used as a model for the LCS. As discussed in chapter 2, the revised BLD is deterministic, i.e., it embeds a unique functionality for the LCS. Furthermore, it supports implicitness, declarativity, and concurrency, which are essential for modeling of large-scale systems.

Alternatively, the Boolean state space model of Eqn (3.5) can be derived from the revised BLD. Furthermore, the set of logical propositions in the Boolean state space model can be transformed into their inequalities in terms of binary variables as shown in section 3.2.

5.4.3 Input and Output Interfaces

5.4.3.1 Input Interface

The LCS is only activated by a change in status of discrete input signals (e.g., `Start_filling_tank_A`) and/or by state events, which are triggered by transition conditions or *state conditions*. Events usually occur when the value of analog input signals (e.g., the level of tank) passes through a pre-determined threshold. No special interface between a discrete input signal and the corresponding logic signal is necessary because the discrete input is usually a two-valued variable (e.g., `Motor_is_running` or

`Motor_is_stopped`). On the other hand, an interface between the analog input signals and the LCS is essential to establish the mathematical relationship between two different models, specifically, a mathematical linkage between a state condition in the physico-chemical system and its corresponding Boolean input signal in the LCS.

Although a simple high-low comparator is a commonly used input interface, the more general input interface that can deal with a combination of input variables or multivariable equations is necessary (Bradbury et al., 1989; Smith, 1991). A general form for the state condition is a logical proposition P_i, which asserts a relationship between a set of atomic propositions Q_{ij}, $j = 1..N_i$ where N_i is the number of atomic propositions (or relational expressions) in P_i. Each atomic proposition Q_{ij} represents a corresponding relational expression r_{ij} of the general form:

$$r_{ij}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \dot{\mathbf{y}}, \mathbf{u}, t) \geq 0, \ \forall j = 1..N_i \tag{5.3}$$

which asserts the following proposition:

$$Q_{ij} \leftrightarrow (r_{ij} \geq 0), \ \forall j = 1..N_i \tag{5.4}$$

The input interface between a state condition P_i and the corresponding logic signal Z_i^{in} asserts the following logical proposition:

$$Z_i^{in} \leftrightarrow P_i, \ \forall i = 1..N_{in} \tag{5.5}$$

where N_{in} is the number of state conditions. In summary, the input interface asserts the following set of logical propositions:

$$\left.\begin{array}{l} Z_i^{in} \leftrightarrow P_i \\ P_i \leftrightarrow P_i(Q_{i1}, Q_{i2}, \ldots, Q_{iN_i}) \\ Q_{ij} \leftrightarrow (r_{ij} \geq 0), \ \forall j = 1..N_i \end{array}\right\} \forall i = 1..N_{in} \tag{5.6}$$

By employing the formal procedure described in section 3.2, Eqn (5.6) can be converted into a set of inequalities:

$$\mathbf{D}_i\mathbf{q}_i + \mathbf{b}_i z_i^{in} \geq \mathbf{d}_i,\ \forall i = 1..N_{in} \tag{5.7}$$

where $\mathbf{q}_i \in \{0,1\}^{N_i}$ is a vector of binary variables q_{ij} corresponding to each atomic proposition Q_{ij}, and $z_i^{in} \in \{0,1\}$ is a binary variable corresponding to a logic signal Z_i^{in}, and $\mathbf{D}_i$ is a matrix of integer coefficients and $\mathbf{b}_i$, $\mathbf{d}_i$ are column vectors of integer coefficients. The remaining task is to establish a relationship between each relational expression r_{ij} and its corresponding atomic proposition Q_{ij}. The set of propositions (Eqn (5.4)) can be transformed into equivalent inequality constraints as follows:

$$L_{ij}\left(1-q_{ij}\right) \leq r_{ij} \leq U_{ij}q_{ij} - e_{ij},\ \forall j = 1..N_i \tag{5.8}$$

where L_{ij}, U_{ij} are the lower and upper bounds on the value of r_{ij}, and e_{ij} is a small positive tolerance. In general, this transformation is not unique, and it is difficult to derive rigorous upper and lower bounds on the value of a general nonlinear function in many variables. However, it is observed that most relational expressions in state conditions are linear and do not involve many variables. Furthermore, upper and lower bounds of a nonlinear function can be obtained relatively efficiently by evaluating an enclosure of the function using interval arithmetic as discussed in section 6.4.2.2. In summary, the state condition and its corresponding input logic signal establishes the following mathematical relationship:

$$\left.\begin{array}{r} \mathbf{D}_i\mathbf{q}_i + \mathbf{b}_i z_i^{in} \geq \mathbf{d}_i \\ \mathbf{L}_i\left(\mathbf{1}-\mathbf{q}_i\right) \leq \mathbf{r}_i \leq \mathbf{U}_i\mathbf{q}_i - \mathbf{e}_i \end{array}\right\} \forall i = 1..N_{in} \tag{5.9}$$

where $\mathbf{r}_i = \left[r_{i1}, r_{i2}, \ldots, r_{iN_i}\right]^T$, $\mathbf{e}_i = \left[e_{i1}, e_{i2}, \ldots, e_{iN_i}\right]^T$, $\mathbf{1} = [1,1,\ldots,1]^T$, and $\mathbf{L}_i$, $\mathbf{U}_i$ are diagonal matrices whose diagonal components are L_{ij} and U_{ij}, $\forall i = 1..N_i$ respectively.

In the tank filling operation example, an analog interface is required between a state condition $L_A \geq L_{High}$ (or $L_B \geq L_{High}$) and a logic input signal `Tank_A_level_high` (or `Tank_B_level_high`). Eqn (5.10) shows this mathematical linkage:

$$\begin{gathered}\text{Tank\_A\_level\_high} \leftrightarrow \left(L_A - L_{High} \geq 0\right) \\ L\left(1 - z^{in}\right) \leq L_A - L_{High} \leq Uz^{in} - e\end{gathered} \tag{5.10}$$

where z^{in} is a binary variable corresponding to `Tank_A_level_high`.

5.4.3.2 Output Interface

The output signals from the LCS can be linked to the physico-chemical system in a similar manner. It is also assumed that output signals are two-valued variables. The output signals from the LCS impose discrete actions on the physico-chemical system, which is realized mathematically by inserting an equation according to the value of the logic signal. In general, the conditional relationship can be established between the logic output signal Z_i^{out} and its corresponding equality h_i^T or h_i^F as follows:

$$\begin{cases} h_i^T(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{u}, t) = 0 \ : & Z_i^{out} \\ h_i^F(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{u}, t) = 0 \ : & \neg Z_i^{out} \end{cases} \tag{5.11}$$

This conditional relationship can be transformed into one equality that involves integer and continuous variables as follows:

$$h_i^T(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{u}, t) z_i^{out} + h_i^F(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{u}, t)\left(1 - z_i^{out}\right) = 0 \tag{5.12}$$

where z_i^{out} is a binary variable corresponding to the logic signal Z_i^{out}. This transformation is also not unique, and the binary variables do not appear linearly in this formulation. An alternative inequality formulation is:

$$
\begin{aligned}
L_i^T\left(1-z_i^{out}\right) \le h_i^T(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\mathbf{u},t) \le U_i^T\left(1-z_i^{out}\right) \\
L_i^F z_i^{out} \le h_i^F(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\mathbf{u},t) \le U_i^F z_i^{out}
\end{aligned}
\tag{5.13}
$$

where L_i^T, L_i^F, U_i^T, and U_i^F are lower and upper bounds on the corresponding function. The N_{out} logic output signals establish a set of inequalities as follows:

$$
\begin{aligned}
\mathbf{L}^T\left(\mathbf{1}-\mathbf{z}^{out}\right) \le \mathbf{h}^T(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\mathbf{u},t) \le \mathbf{U}^T\left(\mathbf{1}-\mathbf{z}^{out}\right) \\
\mathbf{L}^F \mathbf{z}^{out} \le \mathbf{h}^F(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\mathbf{u},t) \le \mathbf{U}^F \mathbf{z}^{out}
\end{aligned}
\tag{5.14}
$$

where $\mathbf{L}^T$, $\mathbf{L}^F$, $\mathbf{U}^T$, $\mathbf{U}^F$ are diagonal matrices whose diagonal components are $L_i^T, L_i^F, U_i^T, U_i^F$, $\forall i = 1..N_{out}$ respectively.

In the tank filling operation example, a logic output signal `Operate_pump` (or `Stop_pump`) selects an appropriate equation according to its value as follows:

$$
\begin{cases}
P_{pump} = P_0 + \Delta P & : \text{Operate\_pump} \\
P_{pump} = P_0 & : \neg\text{Operate\_pump}\left(\leftrightarrow \text{Stop\_pump}\right)
\end{cases}
\tag{5.15}
$$

which can be transformed into an equality (Eqn (5.16)) or inequalities (Eqn (5.17)).

$$
\left(P_{pump} - P_0 - \Delta P\right)z^{out} + \left(P_{pump} - P_0\right)\left(1 - z^{out}\right) = 0 \tag{5.16}
$$

$$
\begin{aligned}
L^T\left(1-z^{out}\right) \le P_{pump} - P_0 - \Delta P \le U^T\left(1-z^{out}\right) \\
L^F z^{out} \le P_{pump} - P_0 \le U^F z^{out}
\end{aligned}
\tag{5.17}
$$

where z^{out} is a binary variable corresponding to `Operate_pump`.

5.5 Solution Algorithms

5.5.1 Solution of Hybrid Simulation Problem without LCSs

As shown in section 5.4.1, the hybrid simulation problem without LCSs is formulated as a sequence of initial value problems described by DAEs. Each initial value problem is solved in its corresponding continuous sub-interval of time (whose boundaries are determined by events). There are three-sub-problems for the solution of Eqn (5.1). First, the condition of the system at the beginning of each sub-interval must be determined (*initialization* and *reinitialization*), secondly the system behavior over the sub-interval must be calculated (*numerical integration of DAEs*), and finally the precise end point of the sub-interval must be located (*state event location).*

5.5.1.1 Initialization and Reinitialization

Before simulation can begin, a set of consistent initial values for the describing variables at the beginning of the first sub-interval must be determined (*initialization*). This problem is extremely difficult because a consistent set of equations $\mathbf{f}^{(1)} = \mathbf{0}$ and initial values for the unknowns $\mathbf{x}^{(1)}$, $\dot{\mathbf{x}}^{(1)}$, and $\mathbf{y}^{(1)}$ at $t^{(0)}$ must be determined simultaneously. In order to calculate initial values for the unknowns, $n^{(1)}$ additional relationships, known as the initial condition specification, are necessary, which can be expressed in general as a set of equations of the form:

$$\mathbf{c}^{(1)}\left(\mathbf{x}^{(1)}\left(t^{(0)}\right),\dot{\mathbf{x}}^{(1)}\left(t^{(0)}\right),\mathbf{y}^{(1)}\left(t^{(0)}\right),\mathbf{u}^{(1)}\left(t^{(0)}\right),t^{(0)}\right)=\mathbf{0} \tag{5.18}$$

At present, the initialization problem can be solved in an ad hoc manner by using the current values of the unknowns to update the set of equations employed at the end of each Newton step (Barton, 1992), or by the more sophisticated algorithms that have been reported (Bullard and Biegler, 1993; Zaher and Westerberg, 1991).

The problem of determining consistent initial values for the variables at the beginning of sub-intervals $\left[t^{(k-1)}, t^{(k)}\right)$, $\forall k = 2..NCD$ (*reinitialization*) is slightly different from that of the first sub-interval. If the set of differential variables appearing in the describing equations remains unchanged across a sub-interval boundary, it is normal practice to assume that the values of these variables are continuous across the boundary:

$$\mathbf{x}^{(k-1)}\left(t^{(k-1)}\right) = \mathbf{x}^{(k)}\left(t^{(k-1)}\right) \tag{5.19}$$

Brüll and Pallaske (1992) derive the sufficient condition for the satisfaction of Eqn (5.19) for a linear implicit DAEs. In general, a set of consistent initial values for the describing variables at time $t^{(k-1)}$ can be determined from the simultaneous solution of:

$$\begin{gathered}
\mathbf{f}^{(k)}\left(\mathbf{x}^{(k)}\left(t^{(k-1)}\right), \dot{\mathbf{x}}^{(k)}\left(t^{(k-1)}\right), \mathbf{y}^{(k)}\left(t^{(k-1)}\right), \mathbf{u}^{(k)}\left(t^{(k-1)}\right), t^{(k-1)}\right) = \mathbf{0} \\
\mathbf{u}^{(k)} = \mathbf{u}^{(k)}\left(t^{(k-1)}\right) \\
\mathbf{c}^{(k)}\left(\mathbf{x}^{(k-1)}\left(t^{(k-1)}\right), \dot{\mathbf{x}}^{(k-1)}\left(t^{(k-1)}\right), \mathbf{y}^{(k-1)}\left(t^{(k-1)}\right), \mathbf{u}^{(k-1)}\left(t^{(k-1)}\right), \right. \\
\left. \mathbf{x}^{(k)}\left(t^{(k-1)}\right), \dot{\mathbf{x}}^{(k)}\left(t^{(k-1)}\right), \mathbf{y}^{(k)}\left(t^{(k-1)}\right), \mathbf{u}^{(k)}\left(t^{(k-1)}\right), t^{(k-1)}\right) = \mathbf{0}
\end{gathered} \tag{5.20}$$

where $\mathbf{c}^{(k)} = \mathbf{0}$ is a general form of the $n^{(k)}$ additional initial condition specification for the reinitialization problem and $\mathbf{f}^{(k)}$ can change during this calculation.

5.5.1.2 Numerical Integration of DAEs

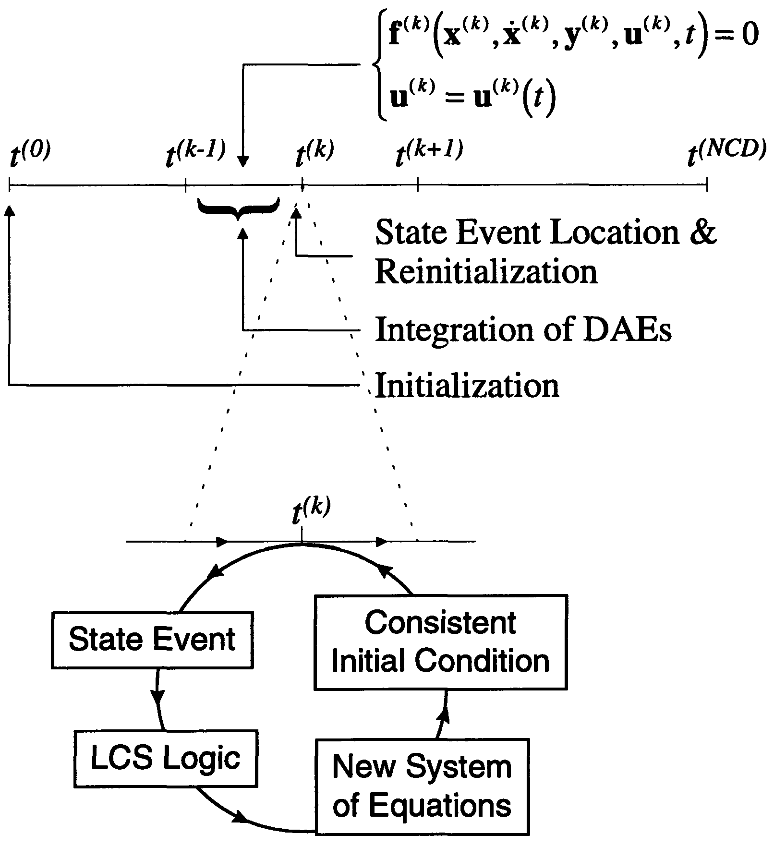
The majority of codes available for the solution of the initial value problems composed of DAEs are based on the multi-step Backward Differential Formula (BDF) method (Gear, 1971). Brenan et al. (1989) review some techniques based on the BDF method to solve DAEs.

5.5.1.3 State Event Location

The end of each continuous sub-interval is marked by the occurrence of a time event or state event. Since time events occur at a predetermined time, their detection and processing is relatively uncomplicated. On the other hand, the time of occurrence of state events is never known in advance because it is dependent on the system fulfilling transition conditions or state conditions. Instead, the numerical solution of the equations in a sub-interval must be advanced speculatively until the state condition becomes satisfied. The state event location problem will be discussed in detail in chapter 6.

5.5.2 Solution of Hybrid Simulation Problem with LCSs

The solution algorithm for hybrid simulation of the LCS and the physico-chemical system will be built on the mathematical formulation presented in section 5.5.1, which does not include the model of the LCS.



$\begin{cases} \mathbf{f}^{(k)}\left(\mathbf{x}^{(k)},\dot{\mathbf{x}}^{(k)},\mathbf{y}^{(k)},\mathbf{u}^{(k)},t\right)=0 \\ \mathbf{u}^{(k)}=\mathbf{u}^{(k)}(t) \end{cases}$
$t^{(0)}$
$t^{(k-1)}$
$t^{(k)}$
$t^{(k+1)}$
$t^{(NCD)}$
State Event Location & Reinitialization
Integration of DAEs
Initialization
$t^{(k)}$
State Event
Consistent Initial Condition
LCS Logic
New System of Equations

Figure 5.5: Solution of hybrid simulation problem

As mentioned in section 5.4.3.1, the LCS is only activated by a change in status of discrete input signals and/or by state events. In other words, the logic output signals of the LCS can only change as a consequence of a transition in the discrete inputs and/or state events. With the assumption of an instantaneous response of the LCS logic to changes in the status of its inputs (see section 3.3.1), the output from the LCS will impose control actions to the process immediately, which causes discrete changes to the functional form of physico-chemical model and mathematically gives rise to a new initial value problem. In summary, the LCS only changes its output as a consequence of an event (a discrete input signal transition and/or a state event), and only impacts the way in which the initial condition of the physico-chemical model is determined at the beginning of each sub-interval. Inside each sub-interval, the output of the LCS does not change and consequently does not affect the physico-chemical system. Figure 5.5 illustrates the solution of a hybrid simulation problem that is complicated by the addition of the model of the LCS. It is clear from this discussion that the initialization and reinitialization problems are changed by addition of models of the LCS, but the other sub-problems are not altered. Even though the nature of state event location problem is not altered, the accurate and efficient location of state events is essential to solve the hybrid simulation problem involving the model of the LCS because the LCS is activated by state events. The new state event location algorithm along with new initialization and reinitialization problems will be addressed in the following sections.

5.5.3 State Event Location Problem

An efficient state event location algorithm for initial value problems in differential-algebraic equations is developed in chapter 6. The algorithm supports flexible representation of state conditions in propositional logic, and guarantees the location of all state events in strict time order. The algorithm consists of two phases: 1) event detection and 2) consistent event location. In the event detection phase, the entire integration step is searched for the state event by solving the interpolation polynomials for the discontinuity functions generated by the BDF method. An efficient hierarchical polynomial root-finding procedure based upon interval arithmetic guarantees detection of the state event even if multiple state condition transitions exist in an integration step, in which case many existing algorithms may fail. As a second phase of the algorithm, a consistent event location calculation is developed that accurately locates the state event detected earlier while completely eliminating incorrect reactivation of the same state event immediately after the consistent initialization calculation that may follow. This numerical phenomenon has not been explained before and is termed discontinuity sticking.

5.5.4 Reinitialization Problem

The reinitialization calculation determines consistent initial values for the unknown variables at the beginning of each sub-interval except the first one. The inclusion of the LCS model affects the reinitialization problem in the following manner. As shown in Figure 5.5, the end of each sub-interval is marked by a state and/or time event, which may activate the LCS by a transition in its input signals, and the LCS determines new output signals by solving its logic, which then impose control actions by replacing a subset of equations according to the value of the logic outputs. The modified reinitialization problem requires the following calculation sequence:

Reinitialization Calculation Sequence:

1. Calculate new logic input signals by solving a set of logical propositions (Eqn (5.6)) or a set of inequality constraints (Eqn (5.9)).
2. Solve the LCS logic to determine the new logic outputs. The inputs to the physico-chemical system are now completely specified by the new logic outputs of the LCS that partially determine the set of describing equations for the next sub-interval.
3. Solve the set of equations determined in the previous step to find the consistent initial values for the describing variables.

Comparing the modified reinitialization problem with the original reinitialization problem for the physico-chemical system (step 3), the additional calculations required are steps 1 and 2. Even though step 1 can be formulated as a Boolean satisfiability problem (Eqn (5.6)) or an integer programming feasibility problem (Eqn (5.9)), the solution of step 1 will not be complicated. Because the values of all relational expressions r_{ij} are already available at the end of the previous sub-interval, the value of each Q_{ij} (or q_{ij}) is obtained by simple comparison. After having all values of Q_{ij}, it is straightforward to calculate the values of P_i or Z_i from the given logical propositions. The solution of step 2 is rather complicated, an algorithm for which will be discussed in detail next.

5.5.4.1 Solution of the LCS logic

The solution of the LCS logic is basically equivalent to sequential logic simulation. The ***sequential modular approach*** to process flowsheeting is currently adapted to solve LCS logic (Goldfarb et al., 1993; Morton, 1992), which converts a sequential logic system represented as binary logic diagrams into a pseudo-combinational logic system by tearing an arbitrary set of feedback paths, and simulates the corresponding pseudo-combinational logic system. In order to simulate the pseudo-combinational logic system, all input values must be known including the values for the torn feedback signals. The current approach is to guess the value of feedback signals and simulation is repeated until all feedback signals are converged to one of the two states. This approach is wrong with respect to two aspects. First, the consequences of choosing an arbitrary feedback tear set is not appreciated in practice. As discussed in chapter 2, choosing a feedback tear set corresponds to selecting a specific Huffman model and different Huffman models will behave differently due to the presence of hazards and races. This problem can be amended by using revised binary logic diagrams, which have a unique feedback tear set (see chapter 2). Second, the feedback signal can be indeterminate when input combinations represent retentive transitions (see section 3.3.2). In this case, the solution will depend on the initial guesses for the feedback signals in the sequential modular approach. Because of this, the previous values of feedback signals should be assigned rather than an arbitrary initial guess, since the current state of the sequential logic system is a function of the current input and the previous state of the system.

Given a unique Huffman model derived from the revised binary logic diagram, the sequential logic simulation can be performed correctly and extremely efficiently. This problem is equivalent to solving Eqn (3.5) or Eqn (5.21) using the notation of this chapter.

$$\mathbf{Z}_k^{state} \leftrightarrow \mathbf{f}\left(\mathbf{Z}_{k-1}^{state}, \mathbf{Z}_k^{in}\right)$$
$$\mathbf{Z}_k^{out} \leftrightarrow \mathbf{g}\left(\mathbf{Z}_k^{state}, \mathbf{Z}_k^{in}\right) \tag{5.21}$$

Remember that Boolean state space model of Eqn (5.21) can be derived from any standard representation as discussed in chapter 3. Even though Eqn (5.21) is formulated as Boolean satisfiability problem, its solution is straightforward because we need to calculate $\mathbf{Z}_k^{state}$ and $\mathbf{Z}_k^{out}$ given $\mathbf{Z}_k^{in}$ and $\mathbf{Z}_{k-1}^{state}$. Note that we are interested in the static behavior of the LCS as discussed in sections 3.3.2 and 5.4.2, therefore we need to repeat solving Eqn (5.21) until all state variables reach the stable state. The following algorithm calculates the current stable state $\tilde{\mathbf{Z}}_k^{state}$ and outputs $\mathbf{Z}_k^{out}$ given inputs $\mathbf{Z}_k^{in}$ and previous stable state $\tilde{\mathbf{Z}}_{k-1}^{state}$.

Sequential logic simulation algorithm:

1. Calculate $\mathbf{Z}_k^{state}$ by $\mathbf{Z}_k^{state} := \mathbf{f}\left(\mathbf{Z}_{k-1}^{state}, \mathbf{Z}_k^{in}\right)$ given $\tilde{\mathbf{Z}}_{k-1}^{state}$ and $\mathbf{Z}_k^{in}$.
2. If $\mathbf{Z}_k^{state} = \tilde{\mathbf{Z}}_{k-1}^{state}$, then $\tilde{\mathbf{Z}}_k^{state} := \mathbf{Z}_k^{state}$ and go to step 3. Otherwise, $\tilde{\mathbf{Z}}_{k-1}^{state} := \mathbf{Z}_k^{state}$ and go to step 1.
3. Calculate $\mathbf{Z}_k^{out}$ by $\mathbf{Z}_k^{out} := \mathbf{g}\left(\tilde{\mathbf{Z}}_k^{state}, \mathbf{Z}_k^{in}\right)$.

Note that the number of combinational logic simulations required (solving step 1) is bounded by the number of state variables as shown in section 2.4.2. Step 1 can be solved extremely efficiently using standard logic simulation algorithms such as *compiler driven simulation* (Abramovici et al., 1990). The first step in compiler drive simulation is *levelizing* (see chapter 7), which basically determines the computation sequence for each logic signal. For example, the value of level of $\mathbf{Z}_k^{in}$ and $\tilde{\mathbf{Z}}_{k-1}^{state}$ are zero, and the outputs $\mathbf{Z}_k^{out}$ have the highest value of level. If the logic elements are simulated in the order determined by levelizing, the output of each element can be determined properly. After levelizing, the compiler-drive simulation translates the description of the logic system into a programming code. Simulation of the logic system is performed by executing the compiler code.

Instead of solving the original problem, the topology of the binary logic diagram can be exploited to derive a calculation sequence in which a series of smaller sub-problems that contain feedback loops are solved rather than the larger original problem. The calculation sequence between sub-problems is unidirectional because there is no feedback loop between them, and can be easily obtained by applying well established algorithms that identify the strongly connected components of a directed graph (Tarjan, 1972). For example, consider the LCS of tank filling operation in Figure 1.7. Because there are no external feedback loops, the only sub-problem (sequential logic simulation problem) is to solve a SRFF.

Finally, timing elements in the binary logic diagrams can be handled directly by scheduling time events on the output of timing elements.

5.5.5 Initialization Problem

The initialization problem is slightly different from the reinitialization problem because a set of describing equations and a set of consistent initial values for the describing variables must be determined simultaneously. This problem for the physico-chemical system alone is extremely difficult (Bullard and Biegler, 1993; Zaher and Westerberg, 1991). The difficulty of the initialization calculation is further complicated by the addition of the LCS.

The solution of the initial set of equations determines the initial status of the logic inputs via the state conditions, and then the initial status of the logic outputs is calculated by solving the LCS logic, which in turn dictates the initial set of describing equations for the first sub-interval. Therefore, simultaneous solution is required. The fact that a set of logical propositions can be transformed into an equivalent set of inequality constraints can be exploited to formulate the initialization problem mathematically. A complete formulation for an initialization calculation at $t^{(0)}$ can be obtained by appending 1) the inequality constraints (or equality constraints) derived from the output interface (Eqn (5.22b)), 2) the inequality constraints derived from the input interface (Eqn (5.22c)), and 3) the inequality constraints derived from the LCS logic (Eqn (5.22d)), to the set of equality constraints that define the sufficient conditions for a set of consistent initial values in the physico-chemical system (Eqn (5.22a)):

$$\mathbf{f}_s^{(1)}\left(\mathbf{x}^{(1)}\left(t^{(0)}\right),\dot{\mathbf{x}}^{(1)}\left(t^{(0)}\right),\mathbf{y}^{(1)}\left(t^{(0)}\right),\mathbf{u}^{(1)}\left(t^{(0)}\right),t^{(0)}\right)=\mathbf{0}$$
$$\mathbf{u}^{(1)}=\mathbf{u}^{(1)}\left(t^{(0)}\right) \tag{5.22a}$$
$$\mathbf{c}^{(1)}(\mathbf{x}^{(1)}\left(t^{(0)}\right),\dot{\mathbf{x}}^{(1)}\left(t^{(0)}\right),\mathbf{y}^{(1)}\left(t^{(0)}\right),\mathbf{u}^{(1)}\left(t^{(0)}\right),t^{(0)})=\mathbf{0}$$

$$\begin{gathered}\mathbf{L}^T\left(\mathbf{1}-\mathbf{z}^{out}\right)\le\mathbf{h}^T\left(\mathbf{x}^{(1)}\left(t^{(0)}\right),\dot{\mathbf{x}}^{(1)}\left(t^{(0)}\right),\mathbf{y}^{(1)}\left(t^{(0)}\right),\mathbf{u}^{(1)}\left(t^{(0)}\right),t^{(0)}\right)\le\mathbf{U}^T\left(\mathbf{1}-\mathbf{z}^{out}\right)\\ \mathbf{L}^F\mathbf{z}^{out}\le\mathbf{h}^F\left(\mathbf{x}^{(1)}\left(t^{(0)}\right),\dot{\mathbf{x}}^{(1)}\left(t^{(0)}\right),\mathbf{y}^{(1)}\left(t^{(0)}\right),\mathbf{u}^{(1)}\left(t^{(0)}\right),t^{(0)}\right)\le\mathbf{U}^F\mathbf{z}^{out}\end{gathered} \tag{5.22b}$$

$$\mathbf{D}_i\mathbf{q}_i\left(t^{(0)}\right)+\mathbf{b}_i z_i^{in}\left(t^{(0)}\right)\ge\mathbf{d}_i$$
$$\mathbf{L}_i\left(\mathbf{1}-\mathbf{q}_i\left(t^{(0)}\right)\right)\le\mathbf{r}_i\left(\mathbf{x}^{(1)}\left(t^{(0)}\right),\dot{\mathbf{x}}^{(1)}\left(t^{(0)}\right),\mathbf{y}^{(1)}\left(t^{(0)}\right),\mathbf{u}^{(1)}\left(t^{(0)}\right),t^{(0)}\right)\le\mathbf{U}_i\mathbf{q}_i\left(t^{(0)}\right)-\mathbf{e}_i \tag{5.22c}$$
$$\forall i=1..N_{in}$$

$$\begin{gathered}\mathbf{A}\mathbf{z}\left(t^{(0)}\right)\ge\mathbf{a}\\ \mathbf{z}^{state}\left(t^{(0)}\right)=\mathbf{z}_0^{state}\end{gathered} \tag{5.22d}$$

where $\mathbf{z}^{in}$, $\mathbf{z}^{out}\subset\mathbf{z}$ are binary variables representing the input and output signals of the LCS, and $\mathbf{z}_0^{state}$ is the initial condition for the state variables. The $\mathbf{f}_s^{(1)}\subset\mathbf{f}^{(1)}$, $\mathbf{h}^T$, $\mathbf{h}^F$ are the subset of the original set of equations, $\mathbf{f}^{(1)}$, that describes the first sub-interval and a set of equations dictated by the output logic signals respectively. This formulation is a set of nonlinear equality and inequality constraints in terms of a mixed set of continuous and binary variables, and therefore requires solution of a MINLP (Mixed Integer Nonlinear Programming) feasibility problem, which calculates consistent initial values for the set of unknowns $\left\{\mathbf{x}\left(t^{(0)}\right),\dot{\mathbf{x}}\left(t^{(0)}\right),\mathbf{y}\left(t^{(0)}\right),\mathbf{z}\left(t^{(0)}\right)\right\}$.

At present, the initialization problem can be solved by an ad hoc decomposition strategy. This is based on the observation that given a complete (but not necessarily consistent) set of values for the binary variables, Eqn (5.22) reduces to a fully determined set of nonlinear equations in the continuous variables (*continuous sub-problem*). Similarly, given a complete set of values for the continuous variables, Eqn (5.22) reduces to a set of linear inequality constraints in the binary variables although the coefficients will no longer be entirely integer (*binary sub-problem*). The continuous sub-problem is equivalent to the initialization problem without the addition of the LCS model, and the binary sub-problem is an integer programming feasibility problem, which has been used to solve formal verification problem efficiently in chapter 3. Therefore, a suitable iteration scheme that employs these two sub-problems can be used to search for a feasible solution to the complete set of constraints.

A general purpose simulator must be able to execute a simulation experiment from arbitrary initial conditions although *steady state* or *cold and empty* are most frequently encountered in industrial applications. In the above formulation, arbitrary initial conditions for the physico-chemical system as well as the LCS may be specified for the purpose of simulation experiments. However, the two initial conditions must be consistent with each other in order to perform initialization successfully, i.e., the initial condition for the LCS, $\mathbf{z}^{state}\left(t^{(0)}\right)=\mathbf{z}_0^{state}$, must be consistent with the initial condition for the physico-chemical system, $\mathbf{c}^{(1)}(\mathbf{x}^{(1)}\left(t^{(0)}\right),\dot{\mathbf{x}}^{(1)}\left(t^{(0)}\right),\mathbf{y}^{(1)}\left(t^{(0)}\right),\mathbf{u}^{(1)}\left(t^{(0)}\right),t^{(0)})=\mathbf{0}$. The consistency between two initial conditions can be checked during the solution of Eqn (22). The values of LCS inputs are determined by the input interface from the values of the continuous variables determined from the initial condition for the physico-chemical system. Therefore, the inconsistency between the LCS inputs and the initial condition for the LCS will appear as infeasibility in the binary sub-problem.

5.6 Conclusions

Dynamic validation of the LCS is formulated as a hybrid discrete/continuous dynamic simulation problem. Two major tasks are accomplished to solve the problem. First, the hybrid model is constructed by combining the model of the LCS and the model of the physico-chemical process. The LCS is modeled as a revised binary logic diagram that supports determinism, implicitness, declarativity, and concurrency, while the physico-chemical process is modeled as well-established differential-algebraic equations. Due to the difference between the two models, rigorous input/output interfaces are developed.

Second, an efficient solution algorithm for hybrid dynamic simulation problem is developed. Due to the nature of interaction between the two sub-systems, initialization and reinitialization problems are further complicated compared to those for hybrid dynamic simulation problem without the LCS. Solution algorithms for both sub-problems are developed.

Chapter 6

State Event Location in Hybrid Discrete/Continuous Simulation

6.1 Introduction

In chapter 5, the notion of hybrid discrete/continuous systems is adopted to formulate the dynamic validation problem of the LCS with its underlying physico-chemical process. Hybrid systems are defined as discrete and continuous sub-systems inherently coupled and interacting with each other. Hybrid systems experience a series of continuous and discrete state transformations. Continuous state transformations are trajectories in the state space driven by the active continuous sub-system, and discrete state transformations are caused by transitions between continuous sub-systems. Transitions can be autonomous and/or controlled by discrete sub-systems. The occurrence of these transitions is known as an event, and its location in time is defined explicitly (time events) or implicitly by transition or state conditions (state events). The efficient location of events in strict time order is essential to the correct solution of the hybrid discrete/continuous simulation problem because numerous events occur in the course of a hybrid simulation, and missing or incorrect location of events leads to incorrect simulation results. In fact, failure to locate events correctly can cause drastic changes to the qualitative dynamic behavior. The detection and location of time events is straightforward because they occur at a pre-determined time. However, the location of state events is never known in advance because they are determined by state conditions whose value cannot be known in advance, but must be determined from numerical solution of the differential equations.

This chapter presents an efficient algorithm for detecting and locating state events tailored to the needs of hybrid discrete/continuous simulation with differential-algebraic equation (DAE) models. As described in section 5.4.1, the series of continuous state transformations can be modeled as a sequence of initial value problems interspersed by events. As a consequence, the time interval of interest $\left[t^{(0)}, t^{(f)}\right)$ is partitioned into NCD continuous sub-intervals $\left[t^{(k-1)}, t^{(k)}\right) \quad \forall k = 1..NCD$. The simulation problem can therefore be defined as:

$$\left.\begin{array}{c} \mathbf{f}^{(k)}\left(\mathbf{x}^{(k)}, \dot{\mathbf{x}}^{(k)}, \mathbf{y}^{(k)}, \mathbf{u}^{(k)}, t\right) = \mathbf{0} \\ \mathbf{u}^{(k)} = \mathbf{u}^{(k)}(t) \end{array}\right\} \quad t \in \left[t^{(k-1)}, t^{(k)}\right) \quad \forall k = 1..NCD \tag{6.1}$$

where $\mathbf{f}^{(k)}: \mathbf{R}^{n^{(k)}} \times \mathbf{R}^{n^{(k)}} \times \mathbf{R}^{m^{(k)}} \times \mathbf{R}^{l^{(k)}} \times \mathbf{R} \mapsto \mathbf{R}^{n^{(k)}+m^{(k)}}$, and $\mathbf{x}^{(k)} \in \mathbf{R}^{n^{(k)}}$, $\dot{\mathbf{x}}^{(k)} \in \mathbf{R}^{n^{(k)}}$, $\mathbf{y}^{(k)} \in \mathbf{R}^{m^{(k)}}$, $\mathbf{u}^{(k)} \in \mathbf{R}^{l^{(k)}}$. The unknowns $\mathbf{x}$ and $\mathbf{y}$ are usually referred to as the differential and algebraic variables respectively, and $\mathbf{u}$ are the known system inputs. The following condition is assumed in this chapter:

$$\text{Rank}\left[\mathbf{f}^{(k)}_{\dot{\mathbf{x}}^{(k)}} \; \mathbf{f}^{(k)}_{\mathbf{y}^{(k)}}\right] = n^{(k)} + m^{(k)}, \; t \in \left[t^{(k-1)}, t^{(k)}\right) \quad \forall k = 1..NCD \tag{6.2}$$

This is a sufficient condition for Eqn (6.1) to have a *differential index* (Brenan et al., 1989) equal to or less than unity. The initial time $t^{(0)}$ is given, and the end of each sub-interval is marked by the occurrence of an event. Since the time of occurrence of state events is never known in advance, the numerical solution of each initial value problem must be advanced speculatively until the state condition becomes satisfied. The state event location algorithm is implemented in conjunction with the BDF (Backward Difference Formula) method (Gear, 1971) because it is the most widely used approach for the solution of DAEs (Brenan et al., 1989). However, the basic idea applies to any linear multi-step integration method. The next section discusses the desired properties of a state event location algorithm employed by a general-purpose simulation package. This is followed by a review of existing algorithms, a description of the algorithm, and the chapter concludes with results and discussion.

6.2 Qualifications for State Event Location Algorithm

6.2.1 Correct Event Detection and Location

A fundamental requirement for state event location algorithms is to guarantee detection and location of all state events in strict time order because the discontinuities resulting from the occurrence of a state event can radically change the future evolution of the overall system behavior. A state event is defined as the *earliest* transition of a pending state condition in an integration step. If multiple state condition transitions exist in an integration step, the earliest transition must be detected and its time of occurrence must be located accurately.

6.2.2 Representation of State Conditions

Even though simple relational expressions are commonly used to express state conditions, a more general form of state conditions is preferred for more flexible problem formulation and wider application. For example, consider the rectifier circuit (Carver, 1978) shown in Figure 6.1. Depending on the values of voltage and current, diode D_1, diode D_2, or both are conductive. The condition for both D_1 and D_2 to be conductive requires the logical proposition:

$$\big((i_1 > 0) \vee (v_1 > v_3)\big) \wedge \big((i_2 > 0) \vee (v_2 > v_3)\big) \tag{6.3}$$

Consider also the steam trap shown in Figure 6.2 employed to allow only a liquid phase to flow. The steam trap valve is only open if the liquid level H is greater than the minimum level H_{MIN} and there is a positive pressure difference across the valve, which can be expressed as:

$$\left(H \geq H_{MIN}\right) \wedge (\Delta P \geq 0) \tag{6.4}$$

Finally, consider the temperature-composition diagram for a binary system of partially miscible liquids at constant pressure as shown in Figure 6.3. The system can exist as a single phase, two phases, or even three phases depending upon the temperature and/or total composition. The condition for the system to exist as a single vapor phase can be expressed as:

$$\left(\left(T \geq T_{L_1V}^{dew}(Z_A)\right) \wedge \left(Z_A \leq Z_A^*\right)\right) \vee \left(\left(T \geq T_{L_2V}^{dew}(Z_A)\right) \wedge \left(Z_A \geq Z_A^*\right)\right) \tag{6.5}$$

where $T_{L_1V}^{dew}(Z_A)$, $T_{L_2V}^{dew}(Z_A)$ are dew point temperatures for systems L_1V and L_2V respectively, Z_A is the total mole fraction of a species A, and Z_A^* is the total mole fraction of a species A at the eutectic point.

The three examples clearly illustrate the need for a more general formulation of state conditions. Classical propositional logic is well suited for the representation of state conditions because it can represent conjunctions and/or disjunctions of relational expressions effectively. Note that in general, the function on either side of a relational operator may be nonlinear (e.g., the dew point temperatures).

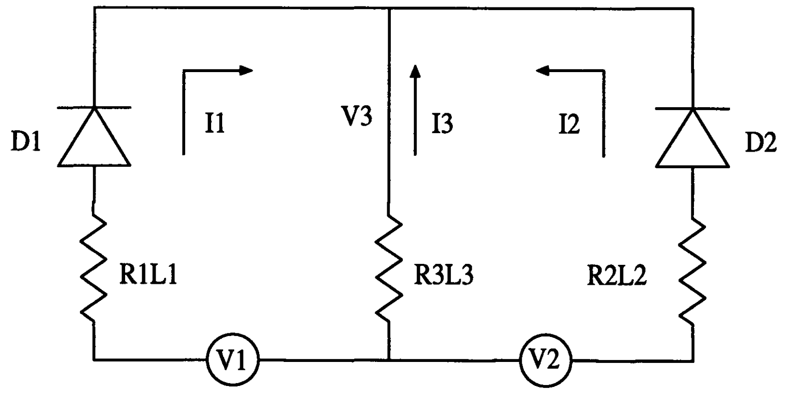


Figure 6.1: Rectifier circuit (Carver, 1978)

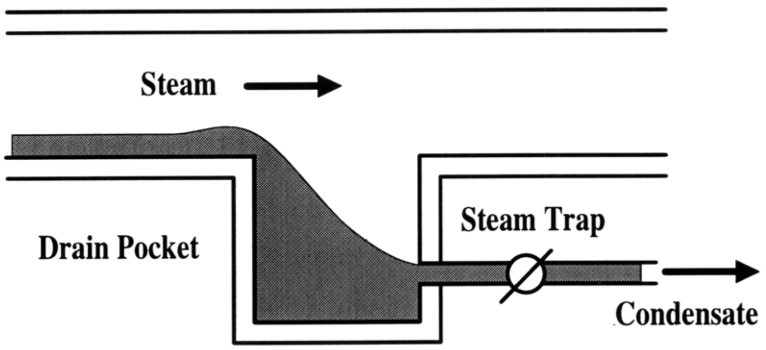


Figure 6.2: Steam trap

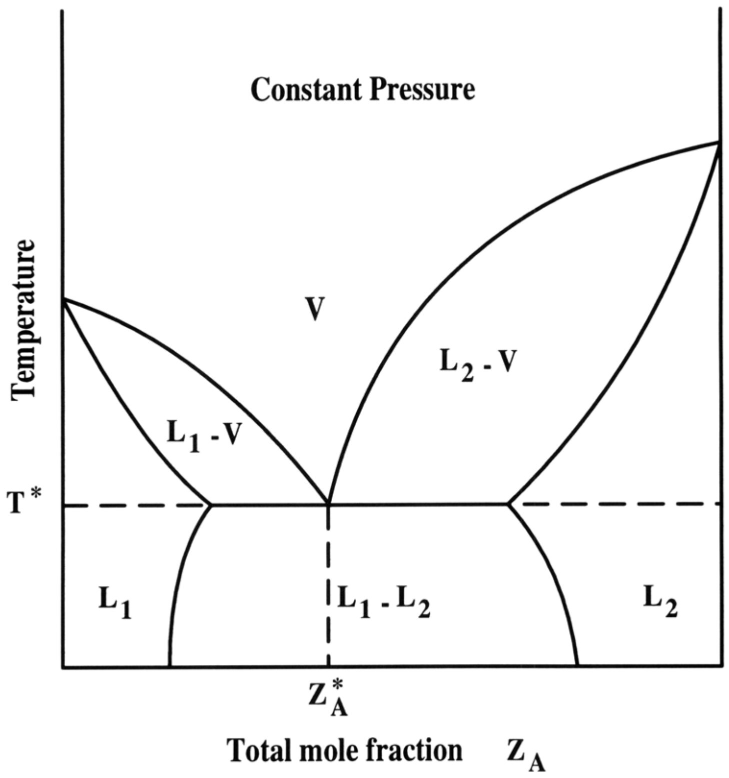


Figure 6.3: VLLE diagram of binary partially miscible system

6.2.3 Discontinuity Locking

The location of state events that lead to discontinuities is equivalent to integration over discontinuities. If an integration step is attempted across a discontinuity, this normally results in repeated step failures and a significant reduction of the integration step size (Cellier, 1979; Hay and Griffin, 1979). One way to avoid this problem is to *lock* the system of equations for each sub-interval. During the step, the system of equations cannot change even if one or more state conditions are satisfied. Instead, the state conditions are examined at the end of each step, and if any of them are satisfied, the exact time of occurrence is then located. This approach has been demonstrated to be both efficient and correct (Cellier, 1979) provided that the system of equations employed before the state event is mathematically well behaved in a small interval following the state event (even if the solution is not physically meaningful). The main advantage of this approach is that the solution trajectories are C^1 functions over the whole step. This eliminates the difficulties of integration over discontinuities, and also provides the means for efficient interpolation procedures for location of the state event. Discontinuity locking has already been implemented in many algorithms (Birta et al., 1985; Carver, 1978; Joglekar and Reklaitis., 1984; Pantelides, 1988).

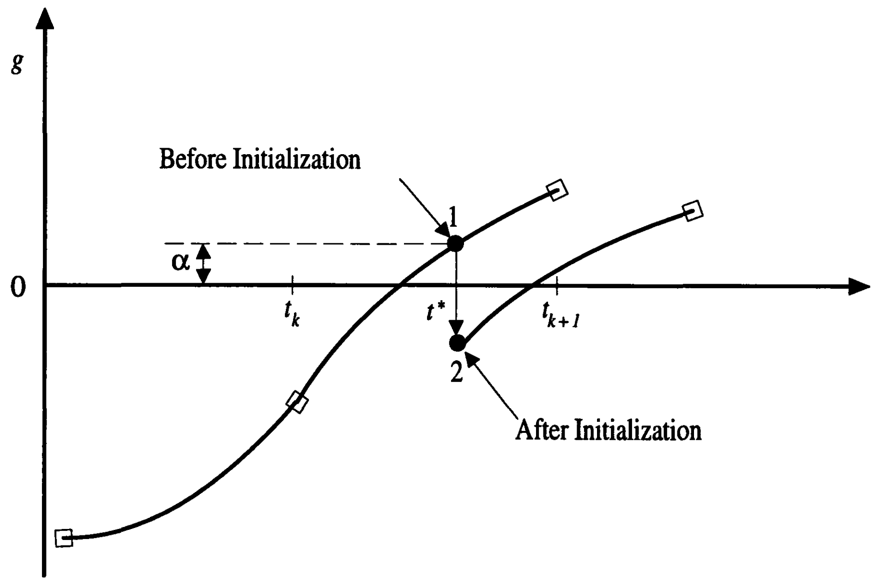


Figure 6.4: Illustration of discontinuity sticking

6.2.4 Discontinuity Sticking and Consistent Event Location

State event location requires the evaluation of state conditions over the entire integration interval. This is typically done by interpolating the unknowns (differential and algebraic variables) required to evaluate state conditions. After location of the state event, a consistent initialization calculation is required to restart the integration in the new sub-interval. This calculation is usually based on the assumption of continuity of the differential variables (Brüll and Pallaske, 1992). Given values for the differential variables, the new system of DAEs is solved to find consistent initial values for the algebraic variables and the time derivatives of the differential variables. If the state event does not occur at mesh points and it has been located by interpolation, the converged initialization calculation may indicate that the state event detected has actually not quite been triggered. This situation occurs because the BDF method (Gear, 1971) provides no guarantees for the consistency of differential and algebraic variables between mesh points although it ensures that the values of the algebraic variables, differential variables, and their time derivatives are consistent at mesh points (within the convergence tolerance of the corrector iteration).

Figure 6.4 illustrates this numerical phenomenon. The state event resulting from a state condition:

$$g(\mathbf{x},\mathbf{y},t)>0 \tag{6.6}$$

is detected in the integration step $[t_k, t_{k+1}]$. The state event time, t^*, can be determined by solving:

$$g\left(\mathbf{x}^p(t^*),\mathbf{y}^p(t^*),t^*\right)=\alpha \tag{6.7}$$

where $\mathbf{x}^p(t)$, $\mathbf{y}^p(t)$ are interpolation formulae (usually polynomials in *t*) for differential and algebraic variables respectively provided by the BDF method. The small positive constant α is an error band employed to ensure that $g\left(\mathbf{x}^p(t^*),\mathbf{y}^p(t^*),t^*\right)>0$ at t^* found by a root-finding procedure (Birta et al., 1985; Hay and Griffin, 1979). A consistent initialization calculation with the assumption of $\mathbf{x}(t^*) = \mathbf{x}^p(t^*)$ is then performed at t^* to find consistent values for algebraic variables $\mathbf{y}(t^*)$ which are different from $\mathbf{y}^p(t^*)$ if t^* is not a mesh point:

$$\mathbf{y}^p(t^*) \neq \mathbf{y}(t^*) \text{ if } t^* \neq t_k, t_{k+1} \tag{6.8}$$

Consequently, at the computed state event time, t^*:

$$g(\mathbf{x}^p(t^*), \mathbf{y}^p(t^*), t^*) \neq g(\mathbf{x}^p(t^*), \mathbf{y}(t^*), t^*) \tag{6.9}$$

Contrary to what one might expect, the value of the state condition can be less than zero after the initialization calculation $\left(g(\mathbf{x}(t^*), \mathbf{y}(t^*), t^*) < 0\right)$ as shown in Figure 6.4. If this is the case, then the same state event is detected again immediately in the new sub-interval. We term this numerical phenomenon ***discontinuity sticking***. Our experience with several DAE-based simulators indicates that discontinuity sticking is not a rare phenomenon; it frequently causes computational inefficiency and sometimes unnecessary simulation failures. However, this problem and its causes have not been recognized in the design of existing algorithms. The algorithm described in this paper eliminates discontinuity sticking completely.

Discontinuity sticking is an inherent problem in the simulation of DAE-based models. However, the same phenomenon may occur during the solution of initial value problems in ordinary differential equations (ODEs) even though it is not observed in general because a consistent initialization calculation is unnecessary after the location of a discontinuity in the simulation of ODE-based models. For example, consider the solution of the ODEs (Eqn (6.10a)) with a state condition (Eqn (6.10b)):

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t), \ \mathbf{x}(0) = \mathbf{x}_0 \tag{6.10a}$$
$$g(\mathbf{x}, t) > 0 \tag{6.10b}$$

The interpolation formula for $g(\mathbf{x}, t)$ is required to detect and locate the state event. One approach to construct this interpolation formula using an ODE solver is to integrate the following system of augmented ODEs (Carver, 1978):

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x},t),\ \mathbf{x}(0)=\mathbf{x}_0 \\ \dot{z} &= \dot{g}(\mathbf{x},\dot{\mathbf{x}},t),\ z_0 = g(\mathbf{x}_0,0)\end{aligned} \tag{6.11}$$

where z is an additional differentiable variable introduced for $g(\mathbf{x},t)$. Suppose that the state condition is satisfied in the integration step $[t_k, t_{k+1}]$, then the state event time t^* can be found by solving:

$$z^p(t^*) = \alpha \tag{6.12}$$

where $z^p(t)$ is an interpolation formula for $g(\mathbf{x},t)$ provided by the integration routine. An appropriate set of initial conditions for $\mathbf{x}$ and z to restart integration following the state event is $\{\mathbf{x}^p(t^*), g(\mathbf{x}^p(t^*),t^*)\}$ rather than $\{\mathbf{x}^p(t^*), z^p(t^*)\}$. However, $z^p(t^*) \neq g(\mathbf{x}^p(t^*),t^*)$ even though $z(0) = g(\mathbf{x}(0),0)$. Therefore, simulation of Eqn (6.11) can lead to discontinuity sticking problems.

6.3 Literature Review

Most modern approaches to state event location employ the discontinuity locking mechanism and *discontinuity functions*. The system of discontinuity functions (or variables) $\mathbf{z}(t) = \mathbf{g}(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\mathbf{u},t)$ is constructed from the relational expressions in the pending state conditions (expressed in general as logical propositions) so that one of their zeros correspond to the state event time. A major difference amongst the various algorithms reported in the literature is the manner in which the interpolation formulae for discontinuity functions $\mathbf{z}^p(t)$ are generated and the state event time is located in the integration interval $[t_k, t_{k+1}]$.

Carver (1978) constructs $\mathbf{z}^p(t)$ from an auxiliary system of differential equations (obtained from differentiation of the discontinuity functions with respect to time) appended to the original system of ODEs, and integrating the augmented system of ODEs. The signs of $\mathbf{z}^p(t_k)$, $\mathbf{z}^p(t_{k+1})$, $\dot{\mathbf{z}}^p(t_k)$, $\dot{\mathbf{z}}^p(t_{k+1})$ are employed to detect state events and, if detected, the state event time is located by solving one of the $\mathbf{z}^p(t) = \mathbf{0}$ detected to cross zero.

Hay and Griffin (1979), Ellison (1981), Joglekar and Reklaitis (1984), and Preston and Berzins (1991) detect the state event by checking the signs of $\mathbf{z}(t_k)$ and $\mathbf{z}(t_{k+1})$. Hay and Griffin (1979) locate the state event time by successive integration with reduced step size combined with linear and quadratic interpolation. Ellison (1981) proposes a location algorithm based on Hermite interpolation. Joglekar and Reklaitis (1984) restrict themselves to linear discontinuity functions, construct $\mathbf{z}^p(t)$ from $\mathbf{x}^p(t)$ and $\mathbf{y}^p(t)$, and locate the state event time by solving one of $\mathbf{z}^p(t) = \mathbf{0}$ using a Newton iteration scheme. Preston and Berzins (1991) propose a location algorithm specific to a particular class of problems, which basically marches up to the discontinuity with successive integration combined with several interpolation formulae.

Birta et al. (1985) approximate $\mathbf{z}(t)$ by cubic polynomials, tabulate the various possible configurations of a cubic polynomial to detect a state event, and locate the state event by Regula-Falsi and Newton iteration schemes.

Shampine et al. (1987) also consider linear discontinuity functions of the form, $g = x(t) + \beta$ or $\dot{x}(t)$ where β is a constant, and construct $z^p(t)$ algebraically from $x^p(t)$. A Sturm sequence is generated to determine the number of zeroes for $z^p(t) = 0$, and the state event time is located using a bisection algorithm and the Sturm sequence.

Rather than employing discontinuity functions, Pantelides (1988) uses the state conditions directly. The state event is detected by comparing the logical values of state conditions at t_k and t_{k+1}, and if detected, the time of occurrence is located by a bisection algorithm with interpolation formulae $\mathbf{x}^p(t)$ and $\mathbf{y}^p(t)$.

Although the use of discontinuity functions has been the most common method for handling state events, Gear and Østerby (1984) deal with the problem in a direct manner by examining the behavior of the local truncation error and its estimate in the region of the discontinuity.

None of the existing algorithms satisfy all the requirements for a state event location algorithm outlined in the previous section. First, only Shampine et al. (1987) can *guarantee* the location of the earliest state condition transition when there are multiple state condition transitions caused by the multiple zero crossings of a discontinuity function in an integration step. Second, state conditions in classical propositional logic are not supported in most algorithms except Pantelides (1988). Finally, there are no provisions for the discontinuity sticking problem in DAE-based algorithms (Joglekar and Reklaitis., 1984; Pantelides, 1988; Preston and Berzins, 1991).

6.4 State Event Location Algorithm

A new algorithm for the detection and location of state events in simulation of DAE models is presented. The algorithm supports flexible representation of state conditions in classical propositional logic, and all state events are *guaranteed* to be detected and processed in strict time order. The algorithm consists of two phases: 1) event detection, and 2) consistent event location. These are presented in the subsections.

6.4.1 Representation of State Conditions

As discussed in section 6.2.2, state conditions are represented by logical propositions that contain members of a complete set of connectives (e.g., $\{\mathtt{NOT}, \mathtt{OR}, \mathtt{AND}\}$) and a set of atomic propositions (relational expressions), which can be expressed in *Backus-Naur form* as:

$$\langle \mathrm{Rel\_Exp} \rangle ::= \langle \mathrm{Exp} \rangle \langle \mathrm{Rel\_Op} \rangle \langle \mathrm{Exp} \rangle \tag{6.13}$$

where a valid $\langle \mathrm{Rel\_Op} \rangle$ is one out of $\{>,<,\geq,\leq\}$ and $\langle \mathrm{Exp} \rangle$ is any valid real expression. Therefore, the state condition can be represented as:

$$L(t) = P(\mathbf{r}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \dot{\mathbf{y}}, \mathbf{u}, t)) \tag{6.14}$$

where P is a logical proposition between a set of relational expressions $\mathbf{r}$, and L is the corresponding Boolean function. A set of relational expressions can also be rearranged as:

$$\mathbf{g}(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\dot{\mathbf{y}},\mathbf{u},t) \geq 0 \tag{6.15a}$$

$$\mathbf{z}(t) = \mathbf{g}(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\dot{\mathbf{y}},\mathbf{u},t) \tag{6.15b}$$

where $\mathbf{g}$ is a set of discontinuity functions and $\mathbf{z}$ are the corresponding discontinuity variables.

```
DetectEvent($t_k$, $t_{k+1}$)
BEGIN
  Initialize $L^*$, $g^*$, $t_d^*$
  FOR i := 1 TO $N_{SC}$
     FOR j := 1 TO $N_i$
         FindRoot($z_{ij}^p(t)$,Root,$N_{root}$,$t_k$, $t_{k+1}$)
         FOR k := 1 TO $N_{root}$
             IF StateConditionChange($L_i$,Root[k]) AND Root[k]<$t_d^*$ THEN
                 $L^* := L_i$; $g^* := g_{ij}$; $t_d^*$:=Root[k]
             ENDIF
         END
     END
  END
END

FindRoot($z_{ij}^p(t)$,Root,$N_{root}$,$t_k$, $t_{k+1}$)
BEGIN
  Root := $\{t : z_{ij}^p(t)=0,\ t_k \leq t \leq t_{k+1}\}$
  $N_{root}$ := $|$Root$|$
END

StateConditionChange($L$, $t$ )
BEGIN
  RETURN $\gamma(L,t)=1$
END
```

Figure 6.5: Event detection algorithm

6.4.2 Event Detection Phase

The event detection phase determines whether or not any Boolean state condition $L_i, i = 1..N_{sc}$ changes its value within the current integration interval $[t_k, t_{k+1}]$ and, if it does, finds the *limiting* state condition L^* that changes its value first, and therefore triggers the state event. The limiting state condition L^* is defined along with the state event time t_d^* as follows:

$$L^* = \{L_i : \gamma(L_i, t_d^*) = 1,\ 1 \le i \le N_{sc}\} \tag{6.16a}$$

$$t_d^* = \min_t \{t : \gamma(L_i, t) = 1,\ 1 \le i \le N_{sc},\ t_k \le t \le t_{k+1}\} \tag{6.16b}$$

where $\gamma \in \{0,1\}$ is a function defined for a Boolean function L as:

$$\gamma(L,t) = \begin{cases} 1 & \text{if } L \text{ changes its value at } t \\ 0 & \text{otherwise} \end{cases} \tag{6.17}$$

The transition of a state condition L can be determined by monitoring values of its constituent relational expressions. Each relational expression changes its value whenever its corresponding discontinuity function crosses zero. Therefore, an event detection algorithm requires finding zero crossings of the discontinuity functions, and determination of t_d^* and L^* at these zero crossings. Figure 6.5 shows the event detection algorithm. The *limiting* discontinuity function g^* is one of the discontinuity functions in L^*, and one of its zero crossings actually determines t_d^* and L^*. Note that the state event time t_d^* found in this phase is only an initial guess for the consistent event location phase, in which the consistent state event time t_l^* will be calculated. The detection algorithm can be made more efficient by first finding the earliest zero crossing of each discontinuity function and checking whether the state condition changes at these zero crossings. However, it is very rare for a discontinuity function to have multiple zero crossings in an integration interval.

In general, there can be more than one L^* and g^* in an integration step. However, it is assumed that they will be isolated relative to machine accuracy. Furthermore, the root-finding procedure based on interval arithmetic is able to separate multiple instances of L^* and g^* to within machine accuracy.

The performance of the event detection algorithm depends on finding zero crossings of the discontinuity functions, which is based on the interpolation formulae for the discontinuity functions $\mathbf{z}^p(t)$. Therefore, the degree of approximation of $\mathbf{z}^p(t)$ to $\mathbf{z}(t)$ is critical for the performance of the event detection algorithm (as will be demonstrated later). Construction of $\mathbf{z}^p(t)$ will be discussed next followed by a rigorous and efficient algorithm to find zero crossings of $\mathbf{z}^p(t)$.

6.4.2.1 Interpolation Formulae for Discontinuity Functions

The BDF method provides interpolation polynomials $\mathbf{x}^p(t)$ and $\mathbf{y}^p(t)$ for differential and algebraic variables respectively for each integration interval $[t_k, t_{k+1}]$. Errors in $\mathbf{x}^p(t)$ and $\mathbf{y}^p(t)$ are controlled even though $\mathbf{x}^p(t)$ and $\mathbf{y}^p(t)$ do not automatically satisfy algebraic relations of the DAEs between mesh points. Interpolation polynomials $\mathbf{z}^p(t)$ for the discontinuity functions should be constructed that are as accurate as $\mathbf{x}^p(t)$ and $\mathbf{y}^p(t)$. Also, this should be computationally efficient because the construction of $\mathbf{z}^p(t)$ is required at every integration step. This can be achieved by appending a set of discontinuity functions (Eqn (6.15b)) to the original set of DAEs (Eqn (6.1)) and integrating an augmented system of equations of the form:

$$\begin{gathered}\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{u}, t) = \mathbf{0} \\ \mathbf{u} = \mathbf{u}(t) \\ \mathbf{z}(t) = \mathbf{g}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \dot{\mathbf{y}}, \mathbf{u}, t)\end{gathered} \tag{6.18}$$

Note that it is not necessary to differentiate the discontinuity functions (Carver, 1978). Then, the BDF method automatically provides $\mathbf{z}^p(t)$ that are as accurate as $\mathbf{x}^p(t)$ and $\mathbf{y}^p(t)$ at each integration step because the mesh points are chosen considering all the equations including the discontinuity functions. This approach increases the system of equations to be integrated by the number of discontinuity functions. However, the dominant cost of integrating a system of DAEs is that of computing and factoring an iteration matrix (Brenan et al., 1989). Figure 6.6 shows the incidence form of the iteration matrix for the augmented system of equations, and clearly the size of the matrix that should be factored does not change by adding the additional equations because $\mathbf{z}(t)$ only appear in $\mathbf{z}(t)=\mathbf{g}(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\dot{\mathbf{y}},\mathbf{u},t)$. Therefore, the computational cost increase per integration step related to appending the discontinuity functions is insignificant. However, integration of the augmented system (Eqn (6.18)) may require more integration steps than that of the original system (Eqn (6.1)) because the functional properties of the discontinuity functions (Eqn (6.15b)) *may* limit the step size of the augmented system.

If the discontinuity functions involve the time derivatives of the algebraic variables $\dot{\mathbf{y}}$, the index of the augmented system is raised to 2. Hence, in this case the augmented system should be integrated by an algorithm appropriate for an index 2 problem in the semi-explicit form (Brenan et al., 1989; Gritsis et al., 1988).

$$\begin{array}{r c}
 & \begin{array}{ccc} \mathbf{x} & \mathbf{y} & \mathbf{z} \end{array} \\
\begin{array}{r} \mathbf{f}()=\mathbf{0} \\ \mathbf{z}=\mathbf{g}() \end{array} & \left[\begin{array}{cc:c} \times & \times & \mathbf{0} \\ \hdashline \times & \times & \mathbf{I} \end{array}\right]
\end{array}$$

Figure 6.6: Iteration matrix of the augmented system of equations

6.4.2.2 Zero Crossings of Discontinuity Functions

Zero crossings of each discontinuity function can be found by solving an interpolation polynomial $\mathbf{z}^p(t)$ in $[t_k, t_{k+1}]$. The interpolation polynomial provided by the BDF method takes the form:

$$z_{ij}^{p}(t)=\sum_{n=0}^{q}\frac{h_{next}^{n}z_{ij}^{(n)}(t_{k+1})}{n!}\left(\frac{t-t_{k+1}}{h_{next}}\right)^{n} \qquad (6.19)$$

where h_{next} is the next suggested step length and q is the current order of integration. The introduction of scaled time s bounded by 0 and 1 in the current integration interval $[t_k, t_{k+1}]$ simplifies the numerical computation involved with the root-finding procedure. The polynomial (Eqn (6.19)) rescaled by the current step length h is:

$$z_{ij}^{p}(s) = \sum_{n=0}^{q} \frac{(-1)^n h^n z_{ij}^{(n)}(t_{k+1})}{n!} s^n = \sum_{n=0}^{q} C_n s^n \tag{6.20}$$

where $s = (t_{k+1} - t)/h$. Thus, we need to find roots of $z_{ij}^{p}(s) = 0$ in $[0,1]$.

Interval arithmetic (Neumaier, 1990) is applied for rigorous root-finding. The root-finding procedure consists of two steps: 1) a root exclusion test, and 2) Newton's method with recursive interval bisection (Moore, 1979). The exclusion test identifies most of the polynomials that have no roots in $[0,1]$. The exclusion test is based on the enclosure of $z_{ij}^{p}(s)$, which contains the range of values of $z_{ij}^{p}(s)$ in $[0,1]$. There are no roots of $z_{ij}^{p}(s) = 0$ in $[0,1]$ if zero is not contained in this enclosure. Even though there are a variety of ways to evaluate an enclosure of a real function with interval arithmetic, depending on the quality of the enclosure and computational costs (Neumaier, 1990), we employed the most efficient way to evaluate an enclosure of a polynomial as shown in Eqn (6.21):

$$\mathbf{z}_{ij}^{p}([0,1]) := \sum_{n=0}^{q} C_n [0,1]^n = \sum_{n=0}^{q} C_n [0,1] \tag{6.21}$$

where $\mathbf{z}_{ij}^{p}([0,1])$ is an enclosure of a polynomial $z_{ij}^{p}(s)$ in the range of $[0,1]$. Note that the evaluation of $\mathbf{z}_{ij}^{p}([0,1])$ requires only q additions. Furthermore, as will be demonstrated later, the exclusion test is extremely successful because it can identify most of the polynomials that have no roots in $[0,1]$. The use of this exclusion test greatly enhances the overall efficiency of the root-finding procedure because it is expected that there will be no roots at all in most integration steps.

Due to the nature of the exclusion test, a polynomial not eliminated by this test may or may not have roots in $[0,1]$. Newton's method with recursive interval bisection determines whether or not this polynomial has roots in $[0,1]$, and finds its roots if it does. The intervals that enclose the roots are searched by existence and nonexistence tests with recursive interval bisection. Once this interval is found, the root can be determined by Newton iteration method. Existence and nonexistence tests are based on the Krawcyk operator (Moore, 1977):

$$\mathbf{K}(\mathbf{s}) := m(\mathbf{s}) - \frac{z_{ij}^{p}(m(\mathbf{s}))}{m(\dot{\mathbf{z}}_{ij}^{p}(\mathbf{s}))} + \left(1 - \frac{\dot{\mathbf{z}}_{ij}^{p}(\mathbf{s})}{m(\dot{\mathbf{z}}_{ij}^{p}(\mathbf{s}))}\right)(\mathbf{s} - m(\mathbf{s})) \tag{6.22}$$

where $\mathbf{s}$ is an interval of scaled time s, $\dot{\mathbf{z}}_{ij}^{p}(\mathbf{s})$ is an interval extension of $\dot{z}_{ij}^{p}(s)$, and $m(\;)$ is a midpoint operator to an interval. If $\mathbf{K}(\mathbf{s}) \cap \mathbf{s} = \varnothing$, then $z_{ij}^{p}(s)$ contains no zero in $\mathbf{s}$ (*nonexistence test*). On the other hand, if $\mathbf{K}(\mathbf{s}) \subseteq \text{int}(\mathbf{s})$, then $z_{ij}^{p}(s)$ contains a unique zero in $\mathbf{s}$ (*existence test*), which can be found by a simplified Newton method (Eqn (6.23)) from any initial guess in $\mathbf{s}$:

$$s^{(k+1)} := s^{(k)} - z_{ij}^{p}\left(s^{(k)}\right) / m\left(\dot{\mathbf{z}}_{ij}^{p}(\mathbf{s})\right) \tag{6.23}$$

If $\mathbf{K}(\mathbf{s}) \cap \mathbf{s} \neq \varnothing$ and $\mathbf{K}(\mathbf{s}) \not\subset \text{int}(\mathbf{s})$ where $\text{int}(\mathbf{s})$ is interior of an interval $\mathbf{s}$, then the interval $\mathbf{s}$ is bisected and the existence and nonexistence tests are applied to the bisected intervals recursively. Note that the Krawcyk operator and the Hansen operator (Hansen and Sengupta., 1981) give the same existence and nonexistence tests for a single equation.

While the nonexistence test excludes an interval that has no zeros, the existence test identifies an interval enclosing a unique zero. There is a case in which the existence test fails. The existence test cannot identify an interval enclosing a zero of multiplicity greater than one because an interval extension of the Jacobian of the polynomial is not *regular* (Neumaier, 1990). In other words, the root-finding procedure can only find *simple* zeros of the polynomial. However, this limitation does not cause problems in this particular application because it is extremely rare that the polynomial produced by the BDF method has a zero of multiplicity greater than one.

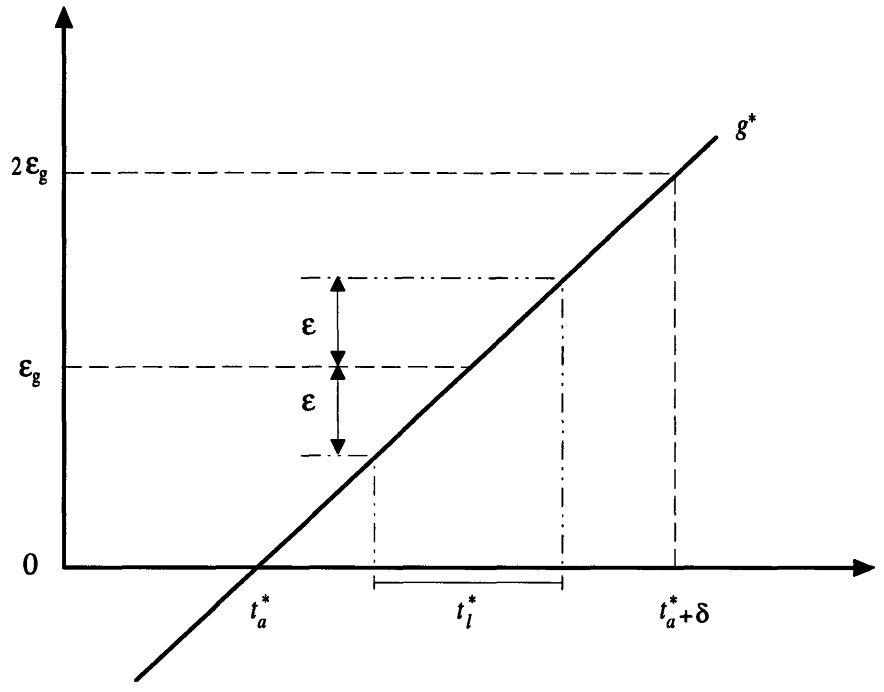


Figure 6.7: Selection of discontinuity tolerance ε_g

6.4.3 Consistent Event Location Phase

The second phase of the algorithm is consistent event location, which determines the consistent state event time t_l^* at which consistency between the differential and algebraic variables is retained, and consequently eliminates discontinuity sticking problems. The consistent event location problem can be formulated as a system of nonlinear equations:

$$\mathbf{f}\left(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\mathbf{u},t_l^*\right)=0 \tag{6.24a}$$

$$\mathbf{u}=\mathbf{u}\left(t_l^*\right) \tag{6.24b}$$

$$\mathbf{x}=\mathbf{x}^p\left(t_l^*\right) \tag{6.24c}$$

$$g^*\left(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\mathbf{u},t_l^*\right)=\pm\varepsilon_g \tag{6.24d}$$

where ε_g (which will be called the ***discontinuity tolerance***) is a small positive tolerance. The sign on ε_g is determined by the direction of the zero crossing of g^*. An extra Eqn (6.24d) is introduced to determine the additional unknown t_l^*. Note that the discontinuity function g^* does not involve the time derivative of the algebraic variables $\dot{\mathbf{y}}$ in this formulation. A Newton iteration requires only from zero to two iterations to solve this system of equations because extremely good initial guesses for the unknowns are available from the detection phase. An alternative formulation of the consistent event location problem is:

$$\mathbf{f}(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\mathbf{u},t_l^*)=0 \tag{6.25a}$$

$$\mathbf{u}=\mathbf{u}(t_l^*) \tag{6.25b}$$

$$\dot{\mathbf{x}}=\alpha_0\mathbf{x}/(t_l^*-t_k)+\sum_{i=1}^{q}\alpha_i\mathbf{x}(t_{k+1-i})\Big/(t_l^*-t_k) \tag{6.25c}$$

$$\dot{\mathbf{y}}=\alpha_0\mathbf{y}/(t_l^*-t_k)+\sum_{i=1}^{q}\alpha_i\mathbf{y}(t_{k+1-i})\Big/(t_l^*-t_k) \tag{6.25d}$$

$$g^*(\mathbf{x},\dot{\mathbf{x}},\mathbf{y},\dot{\mathbf{y}},\mathbf{u},t_l^*)=\pm\varepsilon_g \tag{6.25e}$$

where α_i, $i=0..q$, are coefficients of the BDF method. Eqn (6.25) is effectively a corrector iteration of the BDF method with an unknown step size $t_l^*-t_k$. The step size or the consistent event time t_l^* is simultaneously determined with other unknowns. Eqns (6.25b), (6.25c), and (6.25d) can be used to eliminate $\mathbf{u}$, $\dot{\mathbf{x}}$, and $\dot{\mathbf{y}}$ from Eqns (6.25a) and (6.25e), and the augmented Jacobian matrix of the reduced system of equations is:

$$\mathbf{J}=\left[\begin{array}{cc:c} \dfrac{\partial\mathbf{f}}{\partial\mathbf{x}}+\dfrac{\partial\mathbf{f}}{\partial\dot{\mathbf{x}}}\dfrac{\partial\dot{\mathbf{x}}}{\partial\mathbf{x}} & \dfrac{\partial\mathbf{f}}{\partial\mathbf{y}} & \dfrac{\partial\mathbf{f}}{\partial t}+\dfrac{\partial\mathbf{f}}{\partial\dot{\mathbf{x}}}\dfrac{\partial\dot{\mathbf{x}}}{\partial t}+\dfrac{\partial\mathbf{f}}{\partial\mathbf{u}}\dfrac{\partial\mathbf{u}}{\partial t} \\ \hdashline \dfrac{\partial g^*}{\partial\mathbf{x}}+\dfrac{\partial g^*}{\partial\dot{\mathbf{x}}}\dfrac{\partial\dot{\mathbf{x}}}{\partial\mathbf{x}} & \dfrac{\partial g^*}{\partial\mathbf{y}}+\dfrac{\partial g^*}{\partial\dot{\mathbf{y}}}\dfrac{\partial\dot{\mathbf{y}}}{\partial\mathbf{y}} & \dfrac{\partial g^*}{\partial t}+\dfrac{\partial g^*}{\partial\dot{\mathbf{x}}}\dfrac{\partial\dot{\mathbf{x}}}{\partial t}+\dfrac{\partial g^*}{\partial\dot{\mathbf{y}}}\dfrac{\partial\dot{\mathbf{y}}}{\partial t}+\dfrac{\partial g^*}{\partial\mathbf{u}}\dfrac{\partial\mathbf{u}}{\partial t} \end{array}\right] \tag{6.26}$$

It is unnecessary to evaluate and factor the whole Jacobian matrix $\mathbf{J}$. The leading principal sub-matrix $\mathbf{J}_{n+m}$ of order $n+m$ of $\mathbf{J}$ is equivalent to the Jacobian matrix $\mathbf{J}_c$ of the corrector iteration of the BDF method, and $\mathbf{J}_{n+m}(t)$, $t\in[t_k,t_{k+1}]$, can be approximated effectively by $\mathbf{J}_c(t_{k+1})$, the Jacobian matrix used in the corrector iteration at t_{k+1}, which is already available in a factored form. We only need to evaluate the last column and row of $\mathbf{J}$ and to perform elimination operations on the last column and row of $\mathbf{J}$ in order to factor the Jacobian matrix $\mathbf{J}$. Therefore, solution of Eqn (6.25) requires less computational effort compared to Eqn (6.24).

This additional calculation should determine the consistent state event time t_l^* as close as possible to the true event time t_a^*, at which the zero crossing actually occurs, while eliminating discontinuity sticking problems. This objective can be accomplished by selecting the discontinuity tolerance ε_g correctly as shown in Figure 6.7. The discontinuity tolerance is determined from *state event tolerance* δ and the slope of the discontinuity function $\dot{g}^*$ as follows:

$$\begin{aligned} &\varepsilon_g := \tfrac{1}{2}\left|\dot{g}^*\right|\delta \\ &\varepsilon_g \in \left[\varepsilon_g^{\min}, \varepsilon_g^{\max}\right] \end{aligned} \tag{6.27}$$

The discontinuity function g^* is assumed to be linear in the small interval $\left[t_a^*, t_a^* + \delta\right]$. Note that the discontinuity tolerance is bounded by $\varepsilon_g^{\min}$ and $\varepsilon_g^{\max}$. While the upper bound $\varepsilon_g^{\max}$ is necessary to prevent the value of g^* at t_l^* from being too far from zero, the lower bound $\varepsilon_g^{\min}$ is necessary to prevent the discontinuity sticking problem. Even though $\varepsilon_g^{\max}$ can be assigned by the user, $\varepsilon_g^{\min}$ is determined as follows:

$$\varepsilon_g^{\min} := 2\varepsilon \tag{6.28}$$

where ε is the numerical tolerance of the equation solver employed in the consistent event location calculation and initialization calculation. This relationship is necessary because the consistent state event time t_l^* can be converged to any value in the interval shown in Figure 6.7 and the initialization calculation is performed at this converged event time. In summary, consistent event location determines the consistent state event time t_l^*, which is guaranteed to be within the bound specified in Eqn (6.29) while eliminating the discontinuity sticking problem completely with minimal computational cost.

$$\begin{aligned} &t_l^* \in [t_a^*, t_a^* + \delta] \text{ if } \tfrac{1}{2}|\dot{g}^*|\delta \geq \varepsilon_g^{\min} \\ &t_l^* \in [t_a^*, t_a^* + \Delta] \text{ if } \tfrac{1}{2}|\dot{g}^*|\delta \leq \varepsilon_g^{\min} \text{ where } \Delta \geq \delta \end{aligned} \qquad (6.29)$$

After the consistent event location phase, it is necessary to check whether there exists any other state condition L^{**} such that $\gamma(L^{**}, t_l^*) = 1$ where $L^{**} \neq L^*$. This can be done efficiently using the converged results from the consistent event location calculation (basically repetition of the event detection phase). If there exists such L^{**}, then the consistent event location problem (Eqn (6.25)) is solved again using a new limiting discontinuity function g^{**}, which is one of discontinuity functions in L^{**}. A finite number of state conditions ensures termination of this algorithm in finite time. However, this phenomenon is very rare as will be demonstrated later.

6.5 Performance of Algorithm

The proposed algorithm has been implemented in the combined discrete/continuous modeling environment ABACUSS[1] and is applied to a series of test problems from the literature (Birta et al., 1985; Carver, 1978) and some chemical engineering problems to demonstrate its correctness and efficiency. We have also implemented and tested the bisection algorithm (Pantelides, 1988). The latter algorithm is valuable by way of comparison because it is currently employed by widely used DAE-based simulators (e.g., gPROMS (Barton and Pantelides, 1993) and SpeedUp (AspenTech, 1993)), and, if events are located correctly, represents the minimal computational cost associated with state event handling.

Table 6.6 shows an overview of the problems tested with respect to problem size and computational measures. Regarding representation of state conditions, one literature problem (Example 3 of Carver (1978)) and all the chemical engineering problems require

[1] ABACUSS (Advanced Batch and Continuous Unsteady-State Simulator) process modeling software, a derivative work of gPROMS software, © 1992 by Imperial College of Science, Technology, and Medicine.

state conditions expressed in terms of logical propositions rather than simple relational expressions.

6.5.1 Event Detection

Some of the earlier algorithms frequently fail to detect and locate the state event when there are multiple state condition transitions in an integration step. There can be two situations in which the state condition experiences multiple transitions. The first case is when one of its constituent discontinuity functions has multiple zero crossings in the integration step. For example, the state condition in Eqn (6.4) will change its value two times if $H - H_{MIN} = 0$ has two roots and $\Delta P \geq 0$ in the integration interval. It is critical to find the first root of $H - H_{MIN} = 0$ in this case. Even though this potential problem has been recognized (Birta et al., 1985; Carver, 1978; Shampine et al., 1987), only the Sturm sequence algorithm of Shampine et al. (1987) and our algorithm *guarantee* the detection of the state event in such a case. However, although it is possible to create simple example problems that have multiple zero crossings in an integration step, in our experience this phenomenon is extremely rare in the large-scale problems typically encountered in engineering applications. This is because it is extremely unlikely for a linear multi-step integration method to generate an integration step with a stationary point in its interpolation polynomials.

On the other hand, a state condition expressed as a logical proposition containing a conjunction can frequently experience multiple transitions in an integration step even if its constituent discontinuity functions cross zero only once (Angulo and Torkzadeh., 1988). The bisection algorithm fails to detect and locate the earliest transition in this case. For example, consider the circuit example (Birta et al., 1985; Carver, 1978). Figure 6.8 and Figure 6.9 show solution profiles produced by the bisection algorithm and our algorithm respectively. There are 10 state events in this problem (Birta et al., 1985) as can be seen in Figure 6.9. However, the bisection algorithm detected only the first state event, which results in completely incorrect solution profiles as can be seen in Figure 6.8. The bisection algorithm failed to detect 9 state events because it detects a state event by checking the transition of a state condition only at mesh points. For example, consider the failure to detect the second state event. The state condition that triggers the second state event is:

$$\left((i_2 \geq 0) \vee (v_2 \geq v_3)\right) \wedge \left((i_1 = 0) \wedge (v_1 \leq v_3)\right) \tag{6.30}$$

and the discontinuity function that actually triggers the state event is $i_1 = 0$[2] (Birta et al., 1985). Table 6.1 shows integration steps taken by the bisection algorithm around the second state event. It is clear that the state condition of Eqn (6.30) is satisfied in the interval [0.0066444,0.0066974] even though it is not at the mesh points. In this case, the time period in which the state condition is satisfied is completely enclosed by the integration step, and the bisection algorithm failed to detect the event because the state condition does not change its value at the mesh points. The bisection algorithm failed to detect the remaining 8 state events for the same reason. The same error has also been observed in the Weir simulation problem.

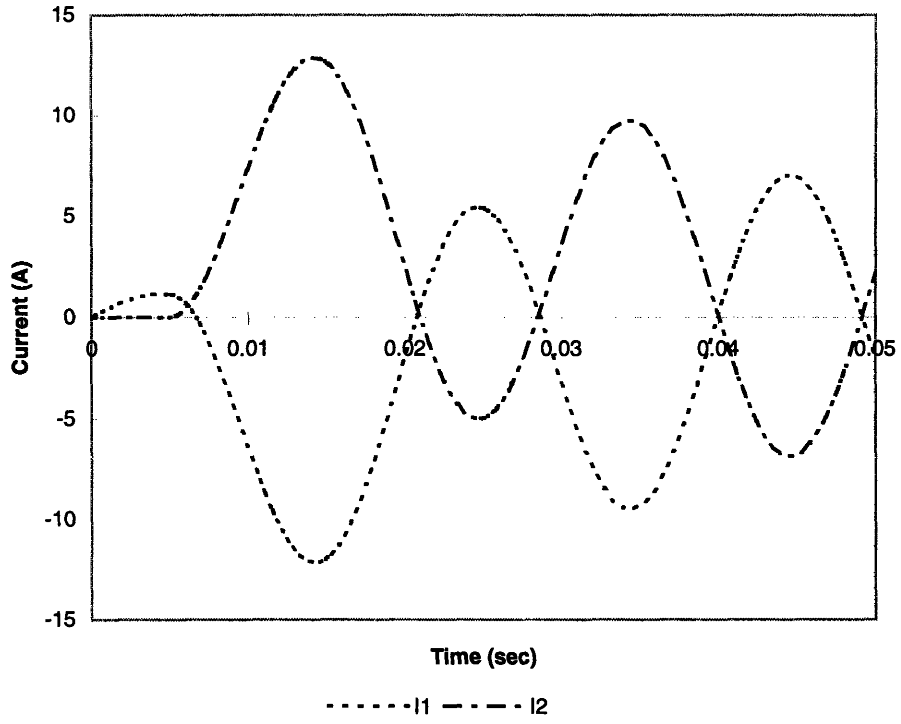
In general, any algorithm that detects a state event by monitoring the transition of the state condition only at mesh points fails to detect and locate the state event when the state condition experiences multiple transitions in the integration step. To be more specific, it fails to detect the state event when the state condition changes its value an even number of times, and it does not guarantee location of the earliest transition when the state condition changes its value an odd (greater than one) number of times.

Unlike the bisection algorithm, our detection algorithm detected and located all the state events correctly. In fact, our algorithm *guarantees* the detection and location of the state event even if the state condition experiences multiple transitions in the integration step because our algorithm rigorously searches for the state event in the whole integration step using interval arithmetic.

Table 6.1: Integration steps taken by the bisection algorithm for Example 3 (Carver, 1978)

Time	i_1	i_2	v_1	v_2	v_3
0.0063996	0.308957	0.747689	-42.5670	42.5670	0.0000
0.0065915	0.087537	0.959023	-47.9400	47.9400	0.0000
0.0066444	0.022226	1.021566	-49.3937	49.3937	0.0000
0.0066974	-0.044848	1.085879	-50.8336	50.8336	0.0000
0.0067504	-0.113660	1.151938	-52.2593	52.2593	0.0000

[2]Note that $i_1 = 0$ is formulated as $ABS(i_1) \leq Tol$ in ABACUSS where *Tol* is a small numerical tolerance.



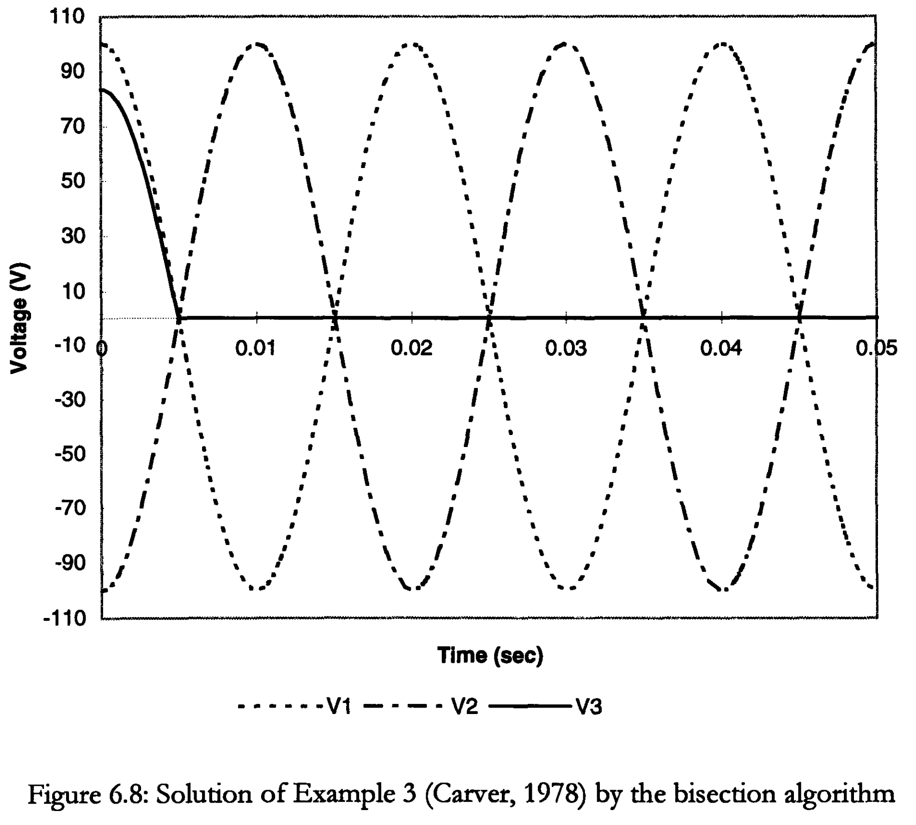
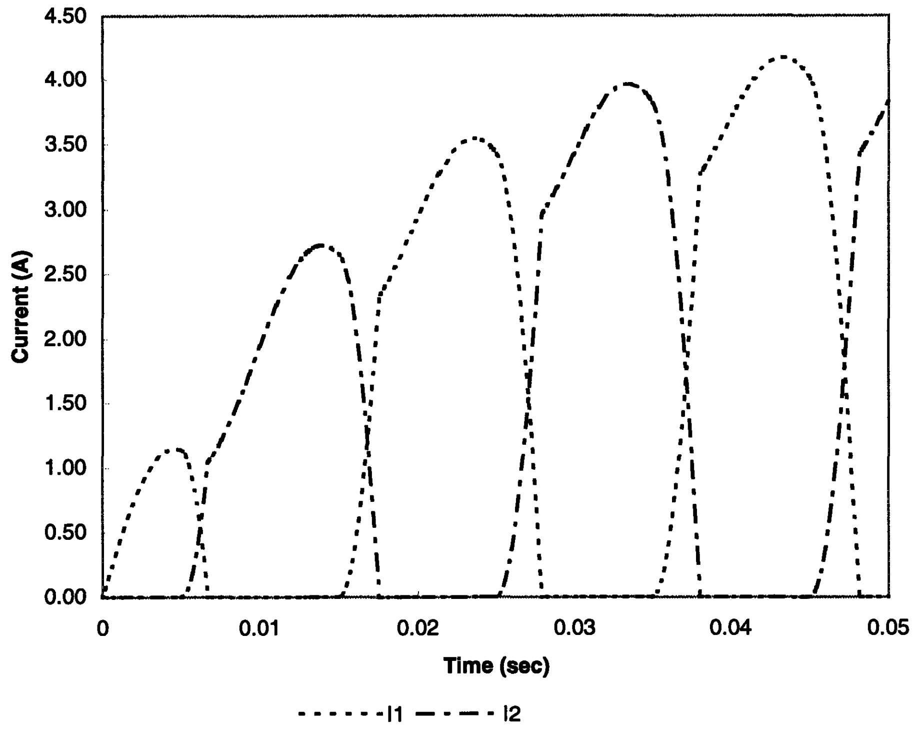


Figure 6.8: Solution of Example 3 (Carver, 1978) by the bisection algorithm



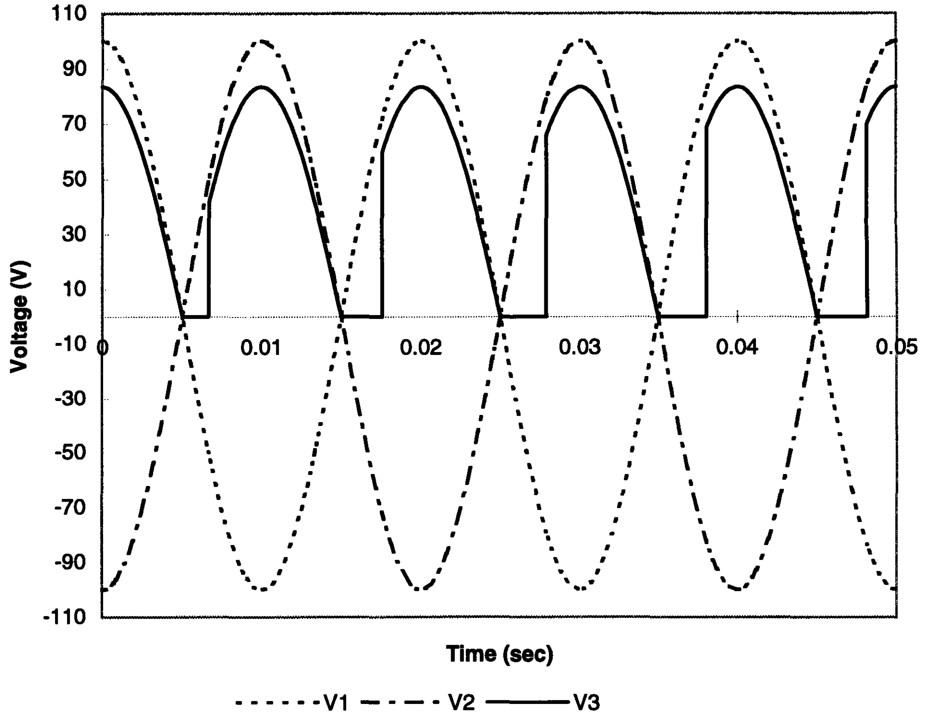


Figure 6.9: Solution of Example 3 (Carver, 1978) by the new algorithm

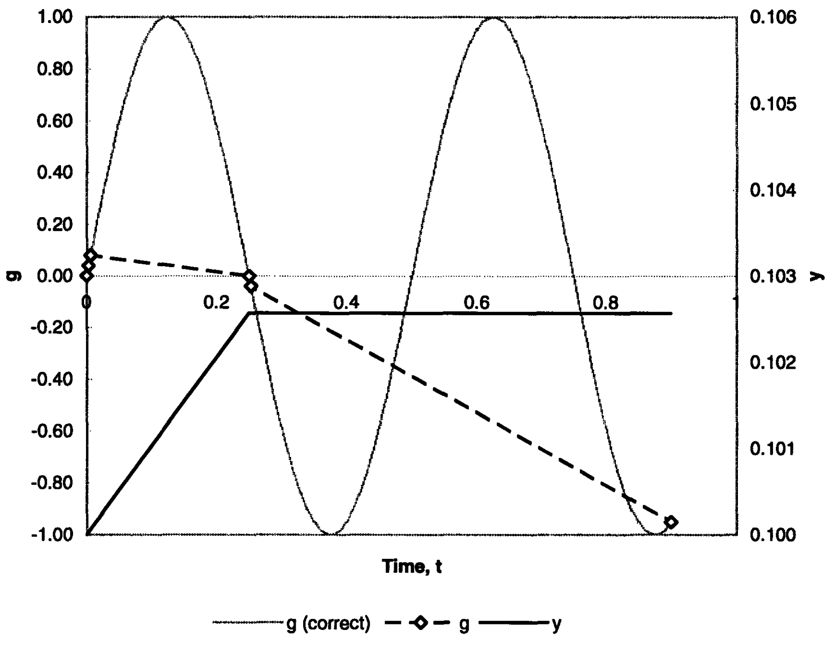


Figure 6.10: Solution of Example 1 (Carver, 1978) by the bisection algorithm

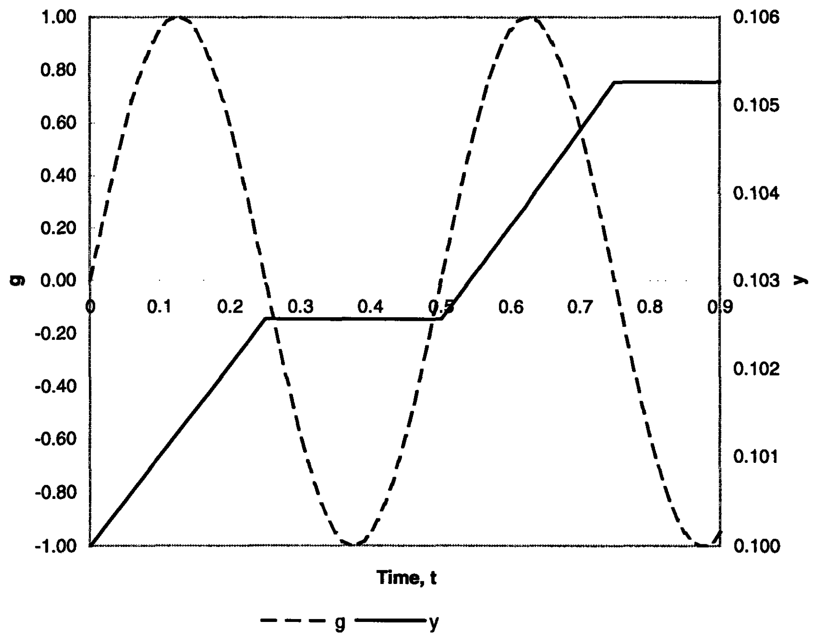


Figure 6.11: Solution of Example 1 (Carver, 1978) by the new algorithm

Table 6.2: Integration steps taken by the bisection algorithm for Example 1 (Carver, 1978)

Time, t	$\sin(4\pi t)$	y
0.0000000	0.000E+00	0.100000
0.0000062	7.791E-05	0.100000
0.0031680	3.980E-02	0.100032
0.0063310	7.947E-02	0.100063
0.2500020	-2.136E-05	0.102566
0.2531640	-3.975E-02	0.102566
0.9000000	-9.511E-01	0.102566

6.5.2 Interpolation Formulae for Discontinuity Functions

The detection and location phases of most algorithms rely on the interpolation formulae for the discontinuity functions $\mathbf{z}^p(t)$. However, the importance of the accuracy of $\mathbf{z}^p(t)$ has often been overlooked. In fact, interpolation formulae $\mathbf{z}^p(t)$ that are accurate over the entire integration step are essential to detect and locate the state event correctly (Shampine et al., 1987). For example, consider Example 1 (Carver, 1978) consisting of one differential equation:

$$\dot{x} = \begin{cases} x^2, & \sin(4\pi t) > 0 \\ 0, & \sin(4\pi t) \le 0 \end{cases} \quad x(0) = 0.1,\ t \in [0,0.9] \tag{6.31}$$

There are three state events, which occur at t =0.25, 0.50, 0.75 according to the analytical solution. Figure 6.10 and Figure 6.11 show the solutions obtained by the bisection algorithm and our algorithm respectively. Table 6.2 shows the integration steps taken by the bisection algorithm. Clearly, the bisection algorithm skips over the last two events, whose time of occurrences are completely enclosed in the last integration step. This happens because the integration steps taken by the bisection algorithm are far from being appropriate to provide an accurate approximation for the discontinuity function $\sin(4\pi t)$ by linear interpolation. In general, any algorithm will fail to detect the state events if it takes integration steps inappropriate for the discontinuity function $\sin(4\pi t)$ and constructs the interpolation formula using the data of $\sin(4\pi t)$ at mesh points.

This example illustrates that interpolation formulae $\mathbf{z}^p(t)$ that are accurate over the entire integration step are critical to detect and locate the state event correctly. Our algorithm guarantees the construction of $\mathbf{z}^p(t)$ as accurate as $\mathbf{x}^p(t)$ and $\mathbf{y}^p(t)$ by forcing the BDF method to select the integration step sizes appropriate to $\mathbf{z}(t)$, $\mathbf{x}(t)$, and $\mathbf{y}(t)$.

Table 6.3: Number of discontinuity stickings

Simulation problems	Bisection algorithm	New algorithm
FlashDrum	1	0
PilotPlant	3	0
BatchDistillation	8	0

Table 6.4: Numerical tolerances for consistent event location

Numerical tolerances	Values
Tolerance for an equation solver ε	1.0×10^{-6}
Lower bound on discontinuity tolerance $\varepsilon_g^{\min}$	2.0×10^{-6}
Upper bound on discontinuity tolerance $\varepsilon_g^{\max}$	1.0×10^{-4}
State event tolerance δ	1.0×10^{-4}

```
                .
                .
Integrating from 6237.8032240 to 6270.9173360
Integrating from 6270.9173360 to 6304.0314480
IF condition PLANT.CALANDRIA.DELTA_T > 40 no longer satisfied
Performing Reinitialisation calculation at time: 6304.0011311
IF condition PLANT.CALANDRIA.DELTA_T > 40 satisfied
Reinitialisation calculation completed.

Integrating from 6304.0011311 to 6304.0042934
IF condition PLANT.CALANDRIA.DELTA_T > 40 no longer satisfied
Performing Reinitialisation calculation at time: 6304.0011867
Reinitialisation calculation completed.

Integrating from 6304.0011867 to 6304.0043490
Integrating from 6304.0043490 to 6304.0075113
                .
                .
```

Figure 6.12: ABACUSS output of PilotPlant simulation problem solved by the bisection algorithm

6.5.3 Discontinuity Sticking Problem

Table 6.3 shows the number of discontinuity stickings for three of the test problems. Note that the bisection algorithm suffers from a number of discontinuity stickings. Figure 6.12 is a portion of the ABACUSS output script for the PilotPlant simulation problem, which shows one instance of discontinuity sticking. On the other hand, the consistent event location calculation completely eliminates discontinuity stickings. Table 6.4 lists satisfactory values of numerical tolerances used in the consistent event location calculation.

Regarding the activation of a different state condition after the consistent event location, it does not occur in any of the problems tested.

Table 6.5: Performance of exclusion test

Simulation problems	Solved	NoRoot	Excluded
Example1 (Carver, 1978)	255	248	248
Example2 (Carver, 1978)	217	210	209
Example3 (Carver, 1978)	1694	1664	1644
Example1 (Birta et al., 1985)	66	64	64
Example2 (Birta et al., 1985)	308	290	287
BouncingBall	46	39	39
OverflowTank	280	276	276
SafetyValve	88	83	83
Weir	370	369	367
FlashDrum	2449	2439	2439
PilotPlant	23965	23948	23946
BatchDistillation	49802	49788	49782

Solved : the number of polynomials solved excluding zero-order and linear polynomials
NoRoot : the number of polynomials that have no roots
Excluded : the number of polynomials excluded by the exclusion test as having no roots

6.5.4 Computational Efficiency

Computational efficiency of the algorithm can be evaluated by considering the computational costs associated with three calculations: root-finding, consistent event location, and integration of the augmented system of DAEs. Each subject will be discussed.

As shown in Table 6.5, it is necessary to solve a large number of polynomials during the simulation experiment. Note that the polynomials have no roots at all in most cases (99.85%). Therefore, it is important to exclude those polynomials as efficiently as possible. Table 6.5 shows performance of the exclusion test we employed, and it excluded 99.96% of polynomials that have no roots.

Recursive interval bisection with existence and nonexistence tests requires a series of evaluations of the Krawcyk operator. Even though interval arithmetic generally requires more computational effort than real arithmetic, evaluation of the Krawcyk operator requires only one interval extension of the Jacobian of a polynomial and one polynomial evaluation, which are efficiently computed by the Horner scheme. Regardless of the computational cost related with evaluating the Krawcyk operator, the overall efficiency of the root-finding procedure is dominated by the efficiency of the exclusion test because most polynomials have no roots at all.

Comparing our root-finding algorithm with the Sturm sequence algorithm of (Shampine et al., 1987) that can be used as an *exact* exclusion test, both algorithms guarantee that all roots in an integration interval will be found. However, the latter is computationally expensive. In order to perform an exclusion test by Sturm sequence for a q-th order polynomial, it is necessary to construct a Sturm sequence, which requires $(q+1)(q-2)/2$ multiplications, and to evaluate $2q$ polynomials, while our exclusion test requires only q additions.

Additional computational effort required for the consistent event location calculation is insignificant because it converges within zero to two iterations due to extremely good initial guesses for the unknowns, and it is unnecessary to evaluate and factor the whole Jacobian matrix for this calculation. Furthermore, consistent event location completely eliminates discontinuity stickings that cause computational inefficiency primarily due to additional Jacobian factorizations in the unnecessary consistent initialization calculations and unnecessary restarts of the numerical integration.

Table 6.6: Comparison of computational cost associated with integration

	Problem size		Bisection algorithm			New algorithm (**g** attached)			New algorithm (**g** not attached)		
Simulation problems	NEqn	NDis	NStep	NRes	NJac	NStep	NRes	NJac	NStep	NRes	NJac
Example1 (Carver, 1978)	2	1	Fail	Fail	Fail	312	1250	167	Fail	Fail	Fail
Example2 (Carver, 1978)	2	2	154	281	34	161	321	37	267	267	31
Example3 (Carver, 1978)	6	10	Fail	Fail	Fail	275	990	104	245	802	83
Example1 (Birta et al., 1985)	3	1	101	167	14	115	194	17	111	181	17
Example2 (Birta et al., 1985)	2	2	345	514	52	370	702	68	337	499	49
BouncingBall	4	2	35	39	20	74	95	34	35	38	20
OverflowTank	4	4	113	192	24	110	194	24	108	185	23
SafetyValve	3	2	122	218	32	133	257	38	121	204	32
Weir	5	3	Fail	Fail	Fail	165	488	48	Fail	Fail	Fail
FlashDrum	54	15	414	1058	137	451	1159	154	418	1044	143
PilotPlant	113	35	1558	2966	432	1642	3222	422	1413	2697	389
BatchDistillation	462	124	438	995	124	415	931	111	416	916	110

NEqn : the number of equations
NDis : the number of discontinuity functions
NStep : the number of integration steps taken
NRes : the number of residual evaluations
NJac : the number of jacobian evaluations

Table 6.6 shows the effect on computational cost of appending the discontinuity functions to the system of DAEs and integrating the augmented system. Compared with the bisection algorithm, our algorithm required more integration steps for the problems tested except the BatchDistillation problem. Note that our algorithm required less integration steps for the BatchDistillation problem, which has numerous discontinuity stickings, because it eliminated the unnecessary integration restarts completely. However, the issue of computational efficiency is irrelevant because the bisection algorithm fails as demonstrated earlier whereas our algorithm detected and located all the state events correctly. For the purposes of experimentation, we tested our algorithm without inserting the discontinuity functions in the error control mechanism. Therefore, the accuracy of the interpolation polynomials for the discontinuity functions is not guaranteed. Table 6.6 shows the result. In terms of computational cost, this approach compares very favorably with the bisection algorithm. However, it also fails for the same reason on two problems. Hence, it is not recommended.

6.6 Conclusions

Mathematical models of physical systems frequently contain discontinuities (Grossman et al., 1993) and accurate simulation of these models requires exact location of these discontinuities or state events. An integrated state event location algorithm for initial value problems in differential-algebraic equations has been presented. The algorithm is designed to be used in modern general-purpose simulation tools, it supports flexible representation of state conditions using propositional logic, and guarantees the efficient location of all state events in strict time order.

The state event location algorithm consists of two phases: 1) event detection, and 2) consistent event location. The event detection identifies the state condition and its constituent discontinuity function that triggers the state event. Interpolation polynomials for the discontinuity functions are solved directly to find the earliest transition of the state condition in the integration interval. The polynomials are solved by a hierarchical root-finding procedure based on interval arithmetic, which consists of an exclusion test and Newton's method with recursive interval bisection. Interpolation polynomials as accurate as those of model variables are generated automatically by the BDF method. A consistent event location calculation is developed to accurately locate the time occurrence of the state event detected earlier while completely eliminating discontinuity sticking, which is a term coined to describe the numerical phenomenon of incorrect reactivation of the same state event immediately after the initialization calculation that will follow.

Results from various test problems have been presented to demonstrate the guaranteed performance of the algorithm. The event detection phase detected all state events correctly even if the state condition experiences multiple transitions in an integration step. The overall efficiency of the event detection phase is enhanced by the extremely efficient exclusion test employed to fully exploit the fact that there are no roots at all in most integration steps. Consistent event location determines the state event time accurately and eliminates all discontinuity sticking with minimal computational cost.

Chapter 7

Preliminary Study of Combinational Logic System Verification

7.1 Introduction

A VLSI (Very Large Scale Integrated) circuit is an integral part of any modern electronic system. The design of such circuits is a complicated and time consuming process, and thus there are many sources of error that can produce an incorrectly functioning circuit. A human designer or an automatic design tool makes an error, and consequently produces an incorrectly functioning circuit. Design errors might also be introduced due to the misuse of an automatic design tool. Design verification is the process of determining whether the designed circuit is the same as what was specified. In particular, logic verification is the process of verifying the equivalence of two circuits at the logic level, usually the optimized and the unoptimized circuits. This design verification step is essential because it is extremely expensive and time consuming to fix errors found in the fabrication process. With the advances in integrated circuit technology, the number of devices that can be put on a chip has increased rapidly. This has greatly increased the complexity of the circuit design and verification process. Design verification has relied mostly on simulation, though formal methods are emerging. However, validation via simulation is still used in practice for circuits that cannot be verified by formal methods, for example, arithmetic circuits (e.g., multipliers) and large-scale sequential circuits. Therefore, it is necessary to find new techniques to verify large and complex circuits.

The implicit model checking technology developed in chapter 3 is applied to formal verification of combinational circuits at the logic level. The problem is formulated as a Boolean equivalence checking problem and is then solved as an integer programming feasibility problem. A general priority setting strategy for the IP feasibility problem is developed. The proposed methodology is demonstrated by verifying a 2-bit multipler. Then, the methodology is applied to verify a 16-bit multipler.

7.2 Problem Formulation as Boolean Equivalence Checking Problem

Unlike sequential logic systems, the outputs of combinational logic systems only depend upon the current inputs as shown in Figure 7.1. Combinational logic systems are widely used to model digital circuits at the logic level, especially arithmetic logic functions. Formal verification of combinational logic systems is rather different from that of sequential logic systems where the description or model of the system is checked against a set of specifications. In combinational logic system verification, the model of the system is checked against a reference model or another representation of the same system. Therefore, formal verification of combinational logic systems is usually formulated as a decision problem in which, given two different descriptions of a combinational logic system, the question is whether the two descriptions have the same functionality.

Consider a combinational logic system in Eqn (7.1) computing a vector of l Boolean outputs $\mathbf{Y} \in \{\mathrm{F},\mathrm{T}\}^l$ given a vector of m Boolean inputs $\mathbf{U} \in \{\mathrm{F},\mathrm{T}\}^m$:

$$\mathbf{Y} \leftrightarrow \mathbf{f}(\mathbf{U}) \tag{7.1}$$

where $\mathbf{f}:\{\mathrm{F},\mathrm{T}\}^m \mapsto \{\mathrm{F},\mathrm{T}\}^l$ is a vector of logical propositions. The reference model can be represented similarly by a set of Boolean functions in Eqn (7.2):

$$\mathbf{Y}_r \leftrightarrow \mathbf{f}_r(\mathbf{U}) \tag{7.2}$$

where $\mathbf{f}_r : \{\mathrm{F},\mathrm{T}\}^m \mapsto \{\mathrm{F},\mathrm{T}\}^l$ is a vector of logical propositions, and $\mathbf{Y}_r \in \{\mathrm{F},\mathrm{T}\}^l$ is a vector of outputs calculated by the reference model given the same inputs $\mathbf{U}$. The formal verification problem is then formulated as a Boolean equivalence checking problem that determines whether the two combinational logic systems implementing the equivalent functionality produce equivalent outputs for all possible valuations of inputs, as illustrated in Figure 7.2. In order to show that two implementations are not equivalent, it is sufficient to find an input combination that asserts different outputs when applied to the two systems. If no such input combination exists, the systems are formally proved to be equivalent.

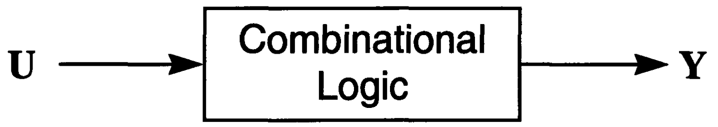


Figure 7.1: Combinational logic system

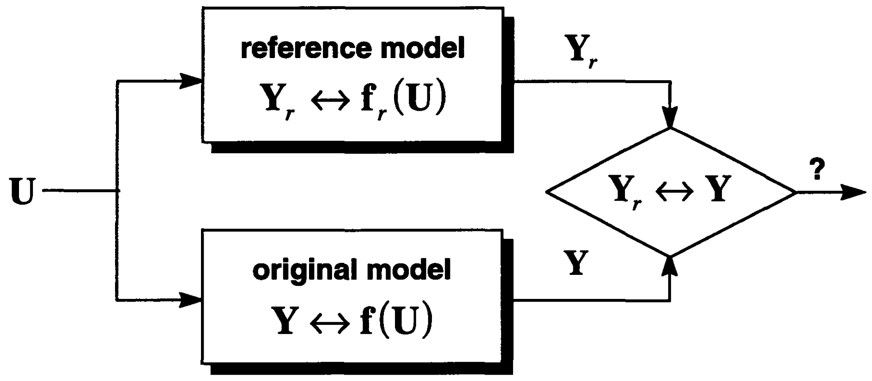

reference model
$\mathbf{Y}_r \leftrightarrow \mathbf{f}_r(\mathbf{U})$
$\mathbf{Y}_r$
U
$\mathbf{Y}_r \leftrightarrow \mathbf{Y}$
?
original model
$\mathbf{Y} \leftrightarrow \mathbf{f}(\mathbf{U})$
Y

Figure 7.2: Formal verification of combinational logic systems

The formal verification problem for combinational logic systems can thus be formulated mathematically as a Boolean satisfiability problem:

$$\begin{gathered} \exists[\mathbf{U}, \mathbf{Y}_r, \mathbf{Y}] \\ \mathbf{Y}_r \leftrightarrow \mathbf{f}_r(\mathbf{U}) \\ \mathbf{Y} \leftrightarrow \mathbf{f}(\mathbf{U}) \\ \bigvee_{i=1}^{l} \left(Y_{r,i} \otimes Y_i\right) \end{gathered} \tag{7.3}$$

where the symbol $\otimes$ represents an exclusive OR function. The last constraint in Eqn (7.3) can be replaced by $\bigvee_{i=1}^{l}\left(\neg\left(Y_{r,i} \leftrightarrow Y_i\right)\right)$, which becomes TRUE if and only if $Y_{r,i}$ and Y_i possess different valuations for the same inputs. If the problem in Eqn (7.3) is unsatisfiable, then the original model is formally verified against the reference model. On the other hand, the existence of any satisfiable solutions means inconsistency between the two models, and feasible valuations correspond to errors in the original model against the reference model.

In general, the Boolean satisfiability problem in Eqn (7.3) is very difficult to solve due to the combinatorial number of inputs to consider. Instead of solving the problem in Eqn (7.3), the Boolean satisfiability problem is transformed into its equivalent integer programming feasibility problem, and it is solved in the domain of binary variables.

7.3 Branch and Bound Priority Setting for Verification of Combinational Logic Systems

The IP feasibility problem is solved by a standard branch and bound search. It is well known that proper branch and bound priority setting for binary variables can significantly reduce the computational efforts required to solve IP problems because it will dramatically reduce the number of nodes explored during branch and bound search required to find a solution. In minimization problem, branch and bound search uses lower bounds on the objective function to avoid exploring certain parts of the set of feasible integer solutions. The original problem is partitioned into a tree of finite sub-problems (the branching part of the algorithm), and each sub-problem is solved separately for its lower bound. In the course of the algorithm, an upper bound on the objective function is maintained. If the lower bound corresponding to a particular sub-problem is greater than the current upper bound, then this sub-problem and all the sub-problems originated from this sub-problem need not to considered further (the bounding part of the algorithm). Different priority setting for binary variables leads to different tree of sub-problems, and consequently shows different performance in terms of the number of sub-problems (or nodes) required to solve in order to find an optimum solution.

The formal verification of combinational logic systems or Boolean equivalence checking problem requires complete enumeration of the input space. Therefore, setting priorities correctly is essential to solve the problem. Due to the topological characteristics of a combinational logic system, we can determine priorities for all variables systematically by *levelizing* the system. The combinational logic systems are directed acyclic graphs, where nodes and edges correspond to logic elements and logic signals or variables respectively. The level, which is an integer number, is assigned to each logic signal or edge according to the depth of the edge from the input nodes. By definition, the level of each input node is set to zero, and the outputs will have the highest value of level. The entire system can be systematically levelized by applying the formula in Eqn (7.4) recursively to each logic element or node:

$$\text{level of outgoing edge} := \max(\text{level of incoming edges}) + 1 \tag{7.4}$$

For example, consider a simple combinational logic system levelized according to this formula in Figure 7.3, where the level of the output signal Y_1 is 3. An efficient algorithm is listed as a `Modula-2` program in Appendix D. The algorithm requires only l depth first searches where l is the number of outputs in the combinational logic system.

Once the value of level for each variable is determined, it is straightforward to determine priorities for variables. Variables of low level have higher priorities than variables of high level. In combinational logic systems, all the other variables including outputs are uniquely determined once the inputs are determined. Therefore, the inputs have the highest priority and should be explored first in branch and bound search. This strategy is based on the natural computational sequence for combinational logic systems.

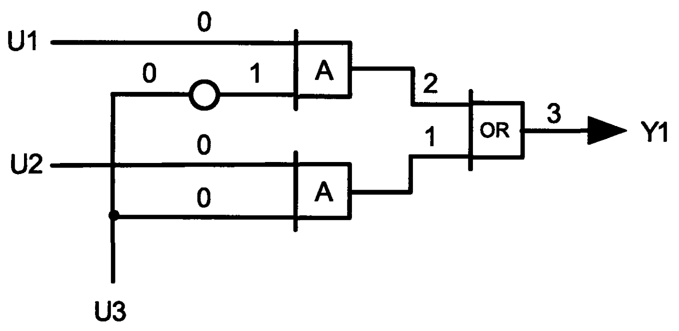


Figure 7.3: Levelized combinational logic system

7.4 Verification of a 2-bit multiplier

Consider the verification of a 2-bit multipler as an example. Figure 7.4 shows a 2-bit multiplier implemented by adder modules while Figure 7.5 shows a reference model for the 2-bit multipler. The 2-bit multiplier is designed to compute the output according to the formula:

$$[y_4 y_3 y_2 y_1]_2 = [u_4 u_3]_2 \times [u_2 u_1]_2 \tag{7.5}$$

where u_i and y_i are binary variables corresponding to Boolean variables U_i and Y_i respectively, and the subscript 2 denotes the binary number system. The set of logical propositions can be derived as following:

$$\begin{aligned} &Y_{r,1} \leftrightarrow U_1 \wedge U_3 \\ &Y_{r,2} \leftrightarrow (U_2 \wedge U_3) \otimes (U_1 \wedge U_4) \\ &Y_{r,3} \leftrightarrow \neg Y_{r,1} \wedge U_2 \wedge U_4 \\ &Y_{r,4} \leftrightarrow Y_{r,1} \wedge U_2 \wedge U_4 \end{aligned} \tag{7.6}$$

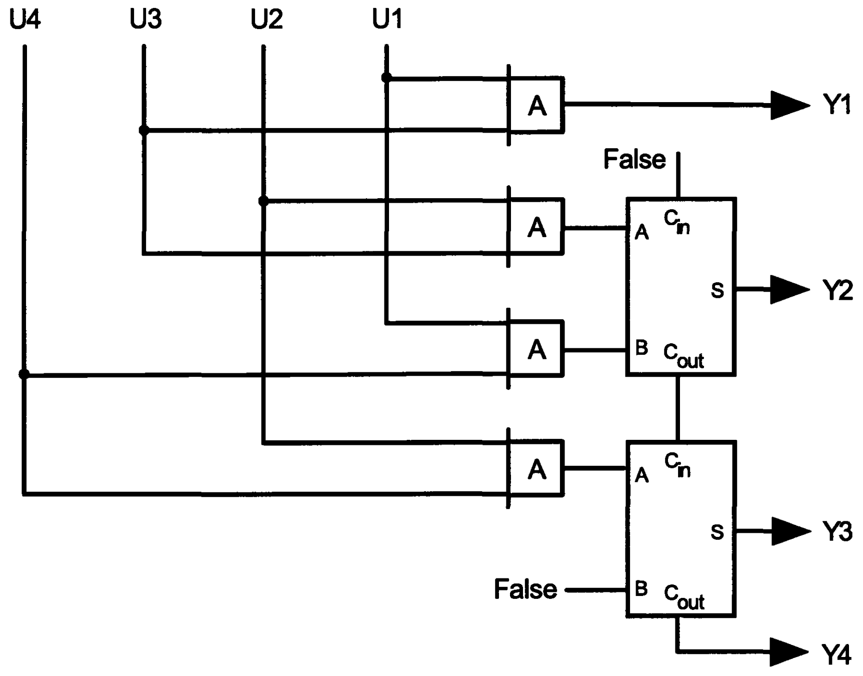


Figure 7.4: Modular implementation of 2-bit multiplier

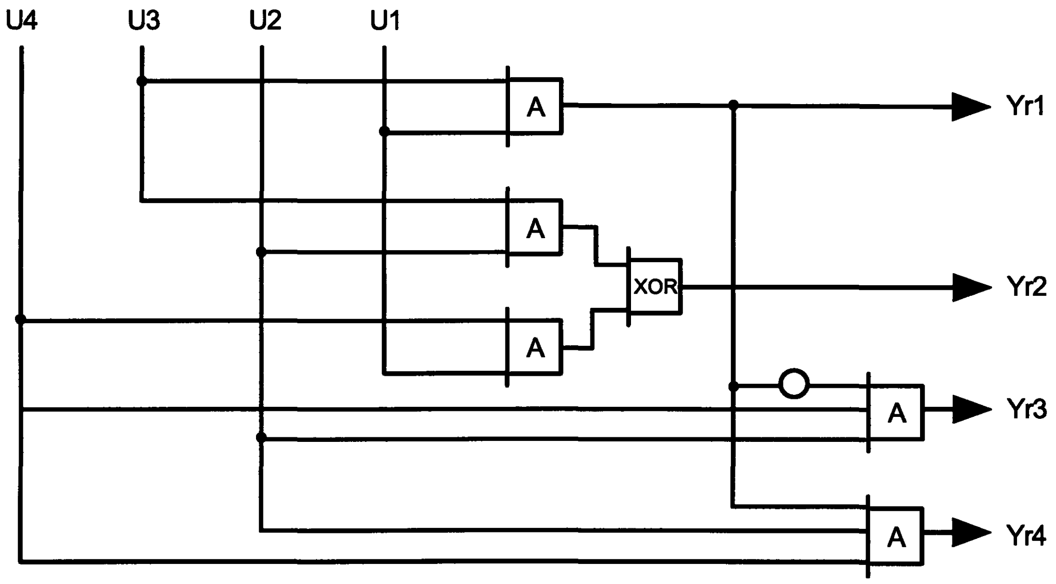


Figure 7.5: Gate-level implementation of 2-bit multiplier

$$\begin{array}{ll}
Y_1 \leftrightarrow U_1 \wedge U_3 & Z_6 \leftrightarrow U_2 \wedge U_4 \\
Z_1 \leftrightarrow U_2 \wedge U_3 & \neg Z_7 \\
Z_2 \leftrightarrow U_1 \wedge U_4 & Z_8 \leftrightarrow Z_4 \\
\neg Z_3 & Z_{10} \leftrightarrow Z_8 \otimes (Z_6 \otimes Z_7) \\
Z_5 \leftrightarrow Z_3 \otimes (Z_1 \otimes Z_2) & Z_9 \leftrightarrow (Z_6 \wedge Z_7) \vee ((Z_6 \otimes Z_7) \wedge Z_8) \\
Z_4 \leftrightarrow (Z_1 \wedge Z_2) \vee ((Z_1 \otimes Z_2) \wedge Z_3) & Y_3 \leftrightarrow Z_{10} \\
Y_2 \leftrightarrow Z_5 & Y_4 \leftrightarrow Z_9
\end{array} \tag{7.7}$$

where Eqn (7.6) and Eqn (7.7) are derived from Figure 7.5 and Figure 7.4 respectively. Note that the adder module asserts the following logical propositions:

$$\begin{array}{c}
S \leftrightarrow C_{in} \otimes (A \otimes B) \\
C_{out} \leftrightarrow (A \wedge B) \vee ((A \otimes B) \wedge C_{in})
\end{array} \tag{7.8}$$

The verification problem is formulated as a Boolean satisfiability problem by combining the set of logical propositions from each model (Eqn (7.6) and Eqn (7.7)) with the constraint:

$$(Y_{r,1} \otimes Y_1) \vee (Y_{r,2} \otimes Y_2) \vee (Y_{r,3} \otimes Y_3) \vee (Y_{r,4} \otimes Y_4) \tag{7.9}$$

The Boolean satisfiability problem is transformed into its equivalent integer programming feasibility problem by employing the procedure in section 3.2. The IP feasibility problem is solved by the branch and bound code in GAMS/CPLEX (Brooke et al., 1992). The LP sub-problems during branch and bound search are solved by the dual simplex method. It takes 0.08 seconds to solve this IP feasibility problem involving 22 binary variables and 88 inequalities on HP9000/J200. A total of 28 nodes are explored during the branch and bound search. The IP feasibility problem is infeasible, which proves formally that the original model (Eqn (7.7)) is logically equivalent to the reference model (Eqn (7.6)).

Due to the small size of the problem in terms of the number of variables, the reduction in computational efforts by setting priorities to variables according their values of level is not noticeable.

Table 7.1: Problem size of C6288 and C6288NR

Category	C6288	C6288NR
No. of Propositions	2416	2399
No. of Inputs	32	32
No. of Outputs	32	32
No. of Intermediates	2384	2367
No. of Inequalities	7216	7147

7.5 Verification of a 16-bit multiplier

Symbolic model checking technology based on OBDDs (Bryant, 1986) has been successfully used for verifying combinational logic systems (Fujita et al., 1988; Malik et al., 1988). However, there are classes of combinational logic systems that cannot be verified by the symbolic model checking technique because the size of OBDDs representing the system is too large to be practical even though the system can be represented by Boolean expressions of reasonable size. For example, the combinational logic systems describing integer multiplication have OBDDs that grow exponentially in the number of inputs (Bryant, 1986).

In this case study, the implicit model checking methodology is applied to verify the 16-bit multiplier, C6288 circuit from the ISCAS85 benchmarks (Brglez and Fujiwara, 1985). The multiplier is an importance class of circuits that cannot be verified efficiently with the standard OBDD-based verification method because multiplication cannot be represented compactly with OBDDs. Note that this is the only ISCAS85 benchmark circuit that could not be verified by the symbolic model checking based on OBDD representation (Fujita et al., 1988; Malik et al., 1988).

The ISCAS85 benchmark has two 16-bit multipliers, the original circuit C6288 and the optimized circuit C6288NR. The two combinational logic systems are checked for Boolean equivalence. Table 7.1 shows the problem size for C6288 and C6288NR circuits. The Boolean satisfiability problem can be formulated by combining logical propositions from each model with the constraint:

$$\bigvee_{i=1}^{32}\left(Y_{r,i} \otimes Y_i\right) \tag{7.10}$$

The proposition in Eqn (7.10) will result in 2^{32} disjunctions when it is converted into the conjunctive normal form (see section 3.2.2). In order to avoid this situation, the constraint in Eqn (7.10) is changed into the following form by introducing additional 32 Boolean variables, P_i for $i = 1..32$:

$$P_i \leftrightarrow \left(Y_{r,i} \otimes Y_i\right),\ i = 1..32 \tag{7.11}$$

Then, the Boolean satisfiability problem is formulated as an optimization problem:

$$\begin{gathered} \max \sum_{i=1}^{32} p_i \\ s.t. \\ \mathbf{q}(\mathbf{u},\mathbf{y},\mathbf{z},\mathbf{p}) \end{gathered} \tag{7.12}$$

where $\mathbf{q}(\mathbf{u},\mathbf{y},\mathbf{z},\mathbf{p})$ is a set of inequalities derived from each logical proposition in the Boolean satisfiability problem including the propositions in Eqn (7.11), and $\mathbf{u}$, $\mathbf{y}$, $\mathbf{z}$ (intermediate variables), and $\mathbf{p}$ are vectors of binary variables corresponding to their Boolean variables $\mathbf{U}$, $\mathbf{Y}$, $\mathbf{Z}$, and $\mathbf{P}$. If the optimum solution found is zero, then the original Boolean satisfiability problem is infeasible. Otherwise, the satisfiability problem is feasible.

The priorities for all the variables are assigned automatically by applying the program in Appendix D. Note that the IP feasibility problem is very large with respect to the number of variables and constraints; it contains 4,847 binary variables and more than 14,363 inequalities. We applied the branch and bound code in GAMS/CPLEX (Brooke et al., 1992) to the IP feasibility problem with LP sub-problems solved by the dual simplex method. This approach failed to find a solution because the branch and bound search was not able to explore sufficient number of nodes to find a solution. Due to the large problem size and the strong coupling of variables in the problem, it is expected to find a solution only after exploring a large number of nodes. However, the branch and bound code was only able to explore approximately 2,000 nodes per day, which means that it is extremely expensive to explore each node or solve each LP sub-problem. By analyzing the LP sub-problems, we identified that they are massively degenerate. In the simplex method, the basic solution of a LP problem is said to be degenerate if the number of active constraints are greater than the number of variables. Note that the performance of the simplex method is very sensitive to the presence of degeneracy because the number of simplex iterations (or the number of extreme points on the boundary of the feasible set) required to find an optimum solution increases substantially with the presence of degeneracy (Bertsimas and Tsitsiklis, 1997). This massive degeneracy causes the poor performance of the simplex method and then the branch and bound search. We tried variants of simplex method with perturbation, but failed to improve the performance of the simplex method for this problem.

It is well known that interior point algorithms are much less sensitive to the presence of degeneracy in LP problems because interior point algorithms find a solution in the interior of the feasible set, not on the boundary of the feasible set (Bertsimas and Tsitsiklis, 1997). Furthermore, certain large and sparse problems are solved faster using interior point algorithms (Bertsimas and Tsitsiklis, 1997). Interior point algorithms will be well suited to solve our LP sub-problems because they are large, sparse, and highly degenerate. The branch and bound code is again applied to the IP feasibility problem with LP sub-problems solved by interior point algorithms. The cost of exploring each node or solving each LP sub-problem is significantly reduced. The branch and bound code explored approximately more than 20,000 nodes per day, which is an order of magnitude performance increase. However, it still was unable to find a solution after exploring more than 500,000 nodes. This is not unexpected because this instance of this IP feasibility problem is difficult to solve due to strong coupling of variables and large problem size. In fact, IP feasibility problems are known to be NP-complete in worst cases. It has been observed that most of the LP sub-problems have zero or very close to zero as their optimal objective functions. This means that the implicit enumeration performed by the branch and bound search is not effective compared to explicit enumeration.

In order to confirm that the verification problem of C6288 circuit has an exponential computational complexity, the original problem is decomposed into 32 sub-problems by exploiting the structure of the constraint in Eqn (7.10), and each sub-problem is solved in series. Note that the constraint in Eqn (7.10) is a disjunction of 32 propositions. Therefore, 32 propositions (or 32 output bits) can be checked separately for their equivalence. Therefore, each sub-problem is formulated by combining the set of logical propositions from each circuit with the constraint:

$$\left(Y_{r,i} \otimes Y_i\right) \tag{7.13}$$

for $i = 1..32$. If all 32 sub-problems are infeasible, then the original problem is infeasible, which verifies the logical equivalence of two circuits. If there exist any feasible sub-problems, then the original problem is feasible, which provides a counter-example to logical equivalence of two circuits. The branch and bound code in GAMS/CPLEX (Brooke et al., 1992) on HP9000/J200 is applied to the first five sub-problems. Table 7.2 shows the verification results. It has been observed that setting priorities is essential to solve the problem because it failed to find a solution even for the first sub-problem without setting priorities. All sub-problems solved are infeasible. However, there are significant increases in the computational costs as the output-bit is increased. Figure 7.6 plots the number of nodes explored (in logarithmic scale) versus the output-bit of the multiplier verified. The number of nodes explored are increasing exponentially in the number of the output-bit. Figure 7.7 plots the CPU time (in logarithmic scale) versus the output-bit. The CPU time increase is also exponential. From the plot in Figure 7.6, we can extrapolate the number of nodes required to solve the whole problem, which is about 5.9×10^{20}. Note that the number of exhaustive simulations required to solve the problem is about 8.6×10^{9}. Even though the number of nodes cannot be compared directly with the number of simulations, this comparison shows the inferior performance of the implicit enumeration performed by the branch and bound search.

Table 7.2: Verification results and computational statistics for each sub-problem

Sub-problem No.	Feasibility	Nodes explored	CPU time (m)
1	infeasible	4	1.46
2	infeasible	22	8.49
3	infeasible	80	30.69
4	infeasible	400	158.76
5	infeasible	1578	528.72

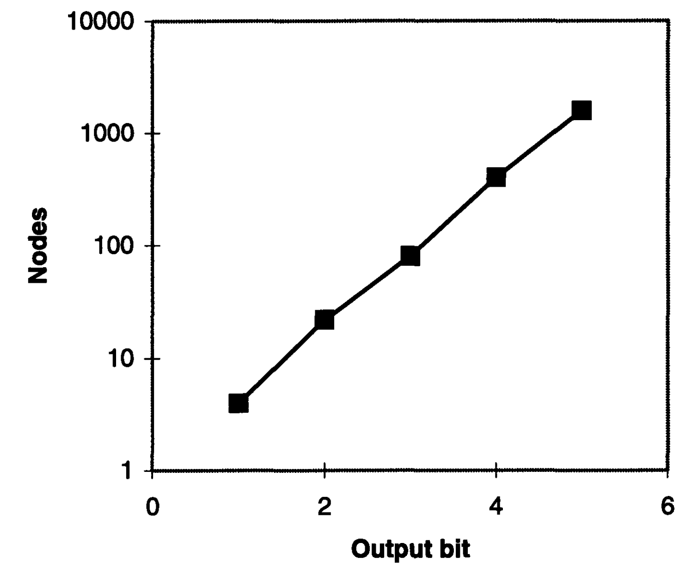


Figure 7.6: The number of nodes explored with respect to output-bit

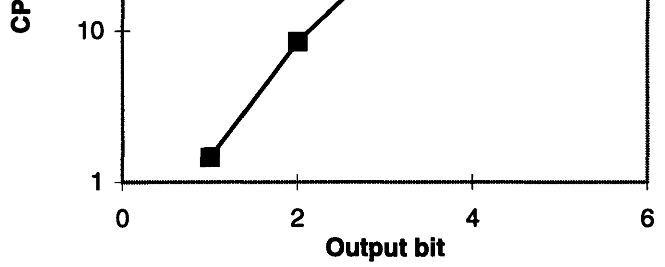


Figure 7.7: CPU time with respect to output-bit

7.6 Conclusions

The implicit model checking technology is applied to formal verification of combinational logic systems. The Boolean satisfiability problem checks the Boolean equivalence between the system in question and the reference system by comparing the outputs for all possible input combinations. The Boolean satisfiability problem is then transformed into its equivalent integer programming feasibility problem to solve it in the domain of binary variables. The 2-bit multiplier is used to illustrate this approach, which has been successfully verified against its reference model.

In order to reduce the computational efforts required to solve the IP feasibility problem, priorities for variables should be set properly. A general priority setting strategy for combinational logic systems is developed, which is based on the value of level of each signal. The 16-bit multiplier case study demonstrated that this priority setting strategy is very effective.

The proposed methodology for combinational logic systems is applied to the C6288 16-bit multiplier from the ISCAS85 benchmarks. Unlike extremely huge OBDDs used in the symbolic model checking, the implicit model checking provides a reasonably sized model for the C6288 circuit. Due to the massive degeneracy of the LP sub-problems, the branch and bound search with simplex method for LP sub-problems failed to solve the problem. In order to circumvent this difficulty, an interior point algorithm is used to solve each LP sub-problem during branch and bound search. Even though this reduces the cost of solving each LP sub-problem significantly, it still failed to find a solution. This is primarily due to the inferior performance of implicit enumeration by the branch and bound search. In order to identify the inherent complexity of this problem, the original problem is decomposed into 32 sub-problems. The solution of the first five sub-problems shows that the formal verification of C6288 circuit has exponential complexity.

In summary, the proposed methodology for verification of combinational logic systems failed to verify the 16-bit multiplier like OBDD based-verification method. Other solution algorithms such as cutting plane algorithms could be applied to solve the problem. However, any integer programming algorithms will not be effective unless the nature of multiplication is embedded in the problem formulation, which is impossible in the bit-level formulation (Bertsimas, 1997). Alternatively, semi-definite programming approaches (Adams and Sherali, 1986) can be used to tighten the gap in the LP relaxations. However, they will not be practical on large-scale problems (Bertsimas, 1997).

Chapter 8

Conclusions and Recommendations

8.1 Conclusions

This thesis has argued that logic-based control systems play an essential role in safety-related and sequence applications, and that the demand for systematic formal approaches to correct design of the LCS is ever increasing due to the growing complexity of LCS applications. However, little attention has been paid to the rigorous design and verification of LCSs with respect to functionality despite the fact that there is an urgent demand for the development of such tools. In order to meet this demand, this thesis has prototyped a unified framework for rigorous verification and validation of LCSs. In particular, this thesis has delivered two major technical contributions towards realization of this goal. First, a formal methodology has been developed that can verify the functionality of a complex large-scale LCS with respect to its formal specifications. Second, a rigorous simulation technology has been developed that can validate the overall performance of the LCS considering its dynamic interaction with its underlying physico-chemical process.

As a first step toward these objectives, the binary logic diagram is selected to represent the implementation-independent functionality of the LCS. The binary logic diagram supports declarativity, implicitness, and concurrency, which are all essential to represent the functionality of a complex large-scale LCS without suffering from the state explosion problem and while maintaining the reactive feature of the LCS. However, it has been recognized that a certain class of BLDs is not deterministic, which is extremely undesirable considering its area of safety-related applications. Analyzing BLDs as sequential logic systems, it has been identified that the BLDs exhibit non-determinisitic behavior when there exist steady state hazards, which are caused by non-unique feedback cut sets in the BLDs. In order to eliminate this ambiguity, an explicit declaration of the location of feedback cut sets is added to the standard BLDs, which makes the revised BLD completely deterministic.

8.1.1 Formal Verification of Logic-based Control Systems

The implicit model checking technique has been developed that can formally verify large-scale LCSs with respect to their specifications. The functionality of LCSs is abstracted as a sequential logic system, which is in turn represented using an implicit Boolean state space model that can be automatically derived from the revised BLD (or, in fact, any standard representation such as ladder logic, programming languages, etc.). The implicit Boolean state space model encapsulates only the relevant set of time invariant relationships between variables rather than a partial or full enumeration of the state space that is liable to increasing number of system states. Therefore, our model embeds all possible states and transitions in a compact closed form. This notion of implicitness in the model formulation is the key property that distinguishes our methodology from previous efforts. The language of temporal logic is adopted to specify formally correctness properties for the LCS, including shutdown logic, permissive logic, and sequences. In order to verify these specifications against the model, the Boolean satisfiability problem is formulated by combining the model with specifications. Instead of solving the Boolean satisfiability problem in the domain of Boolean variables, the problem is transformed into its equivalent integer programming feasibility problem, which is then solved by standard branch and bound techniques. The original specification can be proved or disproved depending upon the feasibility of the problem, yielding counter-examples or witnesses as necessary. If the specification is violated against the model, then the model or the design must be modified so that the specification is satisfied. Even though the implicit model checking technique can identify systematically any inconsistency between the model and specifications, users are currently responsible for modifying the design.

In principle, formal verification of the LCS is a combinatorial problem, and any formal verification method will suffer from this combinatorial nature of the problem. The major contribution of our approach compared to other approaches is to confine the combinatorial nature of the problem to the solution of the IP feasibility problem, whereas in other approaches model formulation is also combinatorial. Further, all the intermediate steps, including model formulation, in our approach can be performed efficiently in polynomial time. This implies that the entire verification problem can be solved efficiently if we can solve the IP feasibility problem efficiently.

The proposed implicit model checking methodology has been successfully applied to two industrial-scale burner management systems, MOD5™ and Throx sequence. In order to test the empirical complexity of our model checking technique, a series of burner management systems of increasing size that combine a number of a single burner management system have been solved. The problem size in terms of the number of inequalities in the IP feasibility problem increases linearly with respect to the number of state variables. It has been shown empirically that the solution time required for the branch and bound search increases quadratically with respect to the number of state variables. Even though the IP feasibility problem is combinatorial in the worst case, our empirical studies indicate that the IP feasibility problem arising from typical industrial applications can be solved very efficiently in polynomial time due to extremely efficient implicit enumeration conducted by branch and bound search.

8.1.2 Dynamic Validation of Logic-based Control Systems

Even if the functionality of the LCS is fully consistent with its complete set of specifications, the LCS may fail to provide a corrective action when necessary and/or may not respond quickly enough, because the LCS does not exist in isolation, instead being coupled dynamically with its underlying physico-chemical process. In order to resolve this potential problem, a dynamic validation technique has been developed based on hybrid discrete/continuous simulation. In particular, the validation of the LCS is formulated as the hybrid discrete/continuous dynamic simulation problem. The hybrid model is constructed by combining the model of the LCS and the model of the physico-chemical process. The LCS is modeled using the revised binary logic diagrams, and the physico-chemical process is modeled as a system of differential-algebraic equations. Rigorous input/output interfaces have been developed between the two different models. An efficient algorithm has been developed to solve the resulting hybrid model. Due to the passive nature of the LCS and its relatively fast response time, initialization and reinitialization problems are further complicated compared to those for hybrid dynamic simulation problems without the LCS. Solution algorithms for both sub-problems are addressed.

The location of state events or implicit discontinuities is an important sub-problem in the solution of hybrid discrete/continuous dynamic simulation problems. All the state events must be located efficiently in strict time order because numerous state events occur in the course of hybrid simulation, and missing events can radically change the future evolution of the overall system behavior. An efficient state event location algorithm has been developed for initial value problems in differential-algebraic equations. The algorithm supports flexible representation of state conditions in propositional logic, and guarantees the location of all state events in strict time order. The algorithm consists of two phases: 1) event detection and 2) consistent event location. In the event detection phase, the entire integration step is searched for the state event by solving the interpolation polynomials for the discontinuity functions generated by the BDF method. An efficient hierarchical polynomial root-finding procedure based upon interval arithmetic guarantees detection of the state event even if multiple state condition transitions exist in an integration step, in which case many existing algorithms may fail. In the second phase of the algorithm, a consistent event location calculation is developed that accurately locates the state event detected earlier while completely eliminating discontinuity sticking or incorrect reactivation of the same state event immediately after the consistent initialization calculation that may follow. The guaranteed performance of the algorithm has been illustrated by solving a set of test problems. All the state events are correctly detected even if the state condition experiences multiple transitions in an integration step. The event detection phase is efficient due to an extremely efficient exclusion test employed in the hierarchical polynomial root-finding procedure. Consistent event location determines the state event time accurately and eliminates all discontinuity sticking with minimal computational effort.

8.2 Recommendations for Future Research

8.2.1 Systematic Design Modification

Implicit model checking can find inconsistency between the specification and the model systematically. The ultimate goal of formal verification, including implicit model checking, is to identify and fix any bugs in the design. When formal verification finds no inconsistency between the specification and the model, there is no further work required. However, if it detects violation of the specification in the model, then the model or the design must be modified to resolve this inconsistency[1]. Even though the verification algorithm provides counter-examples to assist in this design modification process, there is currently no systematic way to incorporate the design change so that the same specification need not to be verified again. Instead, users need to modify the design based on the verification results, and to reverify the modified design against the same specification. Therefore, it is necessary to address the problem of systematic design modification based on defects found in the formal verification step.

The first step toward this goal will be to identify a subsystem that causes this inconsistency in order to reduce the search space. This idea is practicable because it is very rare that a single specification relates to all the properties of the system. Instead, the specification usually checks a subset of the properties of the system. It is likely that there exist non-unique subsystems that cause the inconsistency. In this case, the smallest system in terms of the number of logical propositions or inequality constraints can be found by formulating the problem as an optimization problem.

The existence of defects in the design can be found in some cases without verifying the model against the specifications. The model can be inconsistent by itself. Inconsistency in the model can be found by solving the problem:

[1] Here the specification is assumed to be correct.

$$\begin{gathered}\exists\left[\mathbf{U}_{k-1},\mathbf{U}_k,\tilde{\mathbf{X}}_{k-1},\tilde{\mathbf{X}}_k,\mathbf{Y}_k\right]\\ s.t.\\ \mathbf{v}\left(\mathbf{U}_{k-1},\mathbf{U}_k,\tilde{\mathbf{X}}_{k-1},\tilde{\mathbf{X}}_k,\mathbf{Y}_k\right)\end{gathered}\tag{8.1}$$

which is a Boolean satisfiability problem including only the model without any specification. For any model to function properly, the model should include a set of stable states and associated transitions. The unsatisfiability of the problem in Eqn (8.1) indicates that the model does not embed any stable states, and that the model is self-inconsistent. There can be several causes for this inconsistency. It can come from poor designs or from simple errors introduced while formulating problem. As an example, consider the model for the tank interlock of Figure 3.1 represented as a set of inequalities in Eqn (8.2):

$$\begin{gathered}1-u_{1,k}+1-\tilde{x}_{1,k}\geq 1\\ 1-u_{2,k}+1-\tilde{x}_{1,k}\geq 10\\ u_{1,k}+u_{2,k}+1-u_{3,k}+\tilde{x}_{1,k}\geq 1\\ 1-u_{1,k-1}+1-\tilde{x}_{1,k-1}\geq 1\\ 1-u_{2,k-1}+1-\tilde{x}_{1,k-1}\geq 1\\ u_{1,k-1}+u_{2,k-1}+1-u_{3,k-1}+\tilde{x}_{1,k-1}\geq 1\\ u_{1,k}+u_{2,k}+u_{3,k}+1-\tilde{x}_{1,k}+\tilde{x}_{1,k-1}\geq 1\\ u_{1,k}+u_{2,k}+u_{3,k}+\tilde{x}_{1,k}+1-\tilde{x}_{1,k-1}\geq 1\end{gathered}\tag{8.2}$$

where there is an error in the 2nd inequality. The right hand side of the 2nd equality should be 1 instead of 10. This kind of error can be introduced if the problem is formulated manually or if an automatic translation tool has bugs. The IP feasibility problem of Eqn (8.2) is infeasible before adding any constraint from a specification. Even though it is easy to find the source of inconsistency in this small example, it is not straightforward to find it for large problems.

After discovering the inconsistency of a model, the next task is to find the causes of inconsistency or to isolate the subsystem (a set of logical propositions) that may cause the problem. As usual, the Boolean satisfiability problem in Eqn (8.1) will be solved as an IP feasibility problem. Therefore, it is necessary to find the set of inequalities that are responsible for infeasibility of the IP problem. The optimization problem in Eqn (8.3) can be used to identify these inequalities:

$$\begin{gathered}\min \sum_{j=1}^{N_C} s_j \\ s.t. \\ \mathbf{A} \cdot \left[\mathbf{u}_{k-1}, \mathbf{u}_k, \tilde{\mathbf{x}}_{k-1}, \tilde{\mathbf{x}}_k, \mathbf{y}_k\right]^T + \mathbf{s} \geq \mathbf{a}\end{gathered} \tag{8.3}$$

where $\mathbf{s} \in \mathbf{N}^{N_c}$ is a vector of integer variables, $\mathbf{A}$, $\mathbf{a}$ are respectively an integer matrix and an integer column vector of coefficients, and $\mathbf{u}_{k-1}, \mathbf{u}_k \in \{0,1\}^m$, $\tilde{\mathbf{x}}_{k-1}, \tilde{\mathbf{x}}_k \in \{0,1\}^n$, $\mathbf{y}_k \in \{0,1\}^l$ are vectors of binary variables. A zero optimal objective function indicates that the problem is feasible. If the optimal objective function is greater than zero, then the problem is infeasible and the sub-system causing inconsistency in the model is a set of constraints with non-zero artificial variables $\mathbf{s}$. Even though the set of inequality constraints causing inconsistency in the model can be identified systematically, this formulation is not practical for large problems because such problems involve a large number of inequality constraints and the artificial variable should be introduced for each inequality constraint. It is necessary to develop another formulation that can be applied to large-scale problems. This pre-processing step checking inconsistency of the model should precede verifying specifications against the model.

8.2.2 Systematic Formulation of Correct Specifications

Any formal verification algorithm verifies the model of the LCS against the specifications. Therefore, the guarantee of the verification results is with respect to the specifications. Furthermore, the set of specifications must be *complete* in order to verify the entire functionality of the model. It has been reported (Burch et al., 1994), and also observed in the work leading to this thesis, that the formulation of specifications takes most of the human effort in solving the verification problem. Currently, it is entirely the responsibility of users to formulate *correct* specifications. This is partially due to the current design practice, where the LCS design comes from previously employed designs or is based on informal information collected from several sources rather than coming from a set of formal specifications. Even though the importance of correct specification formulations has been recognized (Clarke et al., 1993), this problem has not been considered seriously so far. For the formal verification technique to be more useful, the development of computer-aided tools is necessary that can help users to formulate the specifications correctly. For example, the systematic derivation of global specification inferred from a set of local specifications is necessary.

8.2.3 Automatic Design of the LCS

The ultimate goal in the LCS design process is to generate LCS logic automatically and systematically from the set of specifications, thus eliminating the need for the formal verification step. As a dual form of the formal verification problem, automatic design of the LCS from its specifications is a generic transformation problem from one type of representation into another type of representation. This transformation is extremely difficult, and the nature of this transformation problem can be quite different depending upon the types of representations chosen for specification and design. For LCSs, the common choice for specifications will be a set of rules and the desirable choice for design will be any configurable representation such as binary logic diagrams. In general, the specification includes less variables than the design because a set of auxiliary variables is introduced during the design process. Therefore, automatic transformation should provide a systematic way to add additional variables as necessary, the number of which should be minimized. Furthermore, the automatic transformation through intermediate canonical representations will be helpful to decompose the original problem.

8.2.4 Development of Software Tool

For implicit model checking to be used in many applications, it is necessary to develop an integrated software tool implementing implicit model checking because it requires a lot of human efforts to verify even small size applications manually even though some of intermediate steps (e.g., conjunctive normal form translation) are automated.

Within the integrated software tool, implicit model checking should take as inputs any standard representation of a LCS design and a specification, and prove or disprove the specification automatically. The software tool should automatically derive the implicit Boolean state space model from any standard representation after checking that the representation is deterministic. The Boolean satisfiability problem should be formulated by combining the model with a specification, and then should be translated automatically into the IP feasibility problem. After solving the IP feasibility problem, the software tool should provide an effective user interface for the analysis of the verification results.

8.2.5 Digital Circuit Verification

LCSs and digital circuits at the logic level can be described by the common framework of Boolean equations. Therefore, implicit model checking can be applied to digital circuit verification. In chapter 7, a preliminary study has been performed to find out the potential applications of implicit model checking in digital circuit verification. However, it was unable to verify a 16-bit multiplier circuit. There are many digital circuits including multipliers that cannot be verified by existing techniques.

In order to verify all these difficult digital circuits, it is essential to analyze the characteristic of the Boolean space embedded in the system of interest. For example, symbolic model checking is able to verify certain classes of digital circuits because it exploits the regularity of the state space characterized by this class of circuits.

The formal verification problem is extremely difficult, and it is unlikely that there exists a universal algorithm that can solve all classes of problems. Therefore, it is necessary to develop tailored algorithms for each class of problem rather than applying one algorithm to all classes of problems.

Appendix A

Transformation of Logical Propositions into Conjunctive Normal Form

A.1 Program Usage

The code listed in section A.2 transforms any logical propositions into their conjunctive normal forms. Note that the logical proposition in conjunctive normal form is a conjunction of clauses where each clause is a disjunction of atomic propositions. A conjunction is a set of logical propositions connected by the AND operator while a disjunction is a set of logical propositions connected by the OR operator. An atomic proposition is a logical proposition that does not contain any Boolean connectives.

The set of valid Boolean connectives are {~ (NOT), & (AND), # (OR), $ (XOR), -> (IMPLICATION), <-> (EQUIVALENCE)}. The program can be run in an interactive mode or in a batch mode. In the interactive mode, the program can transform a single logical proposition into its conjunctive normal form. For example, the proposition, `y<->u1#u2`, can be converted into its conjunctive normal form as:

```
| ?- [cnf].
Consulting. . . .cnf.pl

yes
```

```
| ?- translate (y1 <-> u1 # u2).
~y1#u1#u2
~u1#y1
~u2#y1

yes
| ?-
```

In the batch mode, the program can transform a set of logical propositions into their conjunctive normal forms. The following are sample input and output files with program script:

input file "logic.in":

```
(y1 <-> u1 # u2).
(y2 <-> u1 & u3).
```

program script:

```
| ?- [cnf].
Consulting. . . .cnf.pl

yes
| ?- run.

yes
| ?-
```

output file "logic.out":

```
/* Original Proposition : y1<->u1#u2 */

~y1#u1#u2
~u1#y1
~u2#y1

/* Original Proposition : y2<->u1&u3 */

~y2#u1
~y2#u3
(~u1# ~u3)#y2
```

A.2 Program Listing

```
/*==============================================================
     Purpose  : Transform any logical propositions into their CNFs
     File     : cnf.pl
     Language : PROLOG
     Author   : Taeshin Park
     Date     : May, 1996
==============================================================*/
```

```
% NOT
:- op(100,fy,~).
% AND
:- op(110,xfy,&).
% OR
:- op(120,xfy,#).
% XOR
:- op(125,xfy,$).
% IMPLICATION
:- op(130,xfy,->).
% EQUIVALENCE
:- op(140,xfy,<->).

run :-
   see('logic.in'),
   tell('logic.out'),
   iterate.

iterate :-
   read(Clause),
   process(Clause).

process(Clause) :-
   Clause=end_of_file;
   write('/* Original Proposition : '),
   write(Clause), write(' */'), nl, nl,
   translate(Clause),
   nl,
   iterate.

translate(Clause1 & Clause2) :- !,
   translate(Clause1),
   translate(Clause2).

translate(Clause) :-
   transform(Clause,NewClause), !,
   translate(NewClause).

translate(end_of_file) :- !.

translate(Clause) :-
   check_tautology(Clause);
   simplify_clause(Clause).

check_tautology(Clause) :-
   assert(Clause),
   literal_exist(X,Clause),
   literal_exist(~X,Clause),
   write('$Tautology : '),
   write(Clause),
   nl.

simplify_clause(Clause) :-
   assert(Clause),
   reduce_clause(X,Clause,ReducedClause),
   literal_exist(X,ReducedClause),
   simplify_clause(ReducedClause);
   write_clause(Clause), nl.
%   write_inequality(Clause).
```

```
write_inequality(Clause) :-
   write('EQ!..   '),
   write(Clause),
   write(' =G= 1;'),
   nl.

write_clause(Clause) :-
   write(Clause),
   nl.

literal_exist(X,X).
literal_exist(X,Y) :-
   reduce_clause(X,Y,_).

reduce_clause(X,X#Y,Y).
reduce_clause(X,Y#X,Y).
reduce_clause(X,Y#Z,Y#Z1) :-
   reduce_clause(X,Z,Z1).
reduce_clause(X,Y#Z,Y1#Z) :-
   reduce_clause(X,Y,Y1).

% Transformation rules for propositional formulas

transform(~(~X),X) :- !.
transform(X <-> Y,(X->Y)&(Y->X)) :- !.
transform(X -> Y,~X # Y) :- !.
transform(X $ Y, (X & ~Y)#(~X & Y)) :- !.
transform(~(X & Y), ~X # ~Y) :- !.
transform(~(X # Y), ~X & ~Y) :- !.
transform(X&Y#Z, (X#Z)&(Y#Z)) :- !.
transform(X#Y&Z, (X#Y)&(X#Z)) :- !.
transform(X#Y,X1#Y) :-
   transform(X,X1), !.
transform(X#Y,X#Y1) :-
   transform(Y,Y1), !.
transform(~X,~X1) :-
   transform(X,X1).
```

Appendix B

Mod5 Burner Management System

The implicit Boolean state space model derived from the burner management system is listed here with the list of variables. Note that equations for $\tilde{\mathbf{X}}_{k-1} \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}}_{k-1}, \mathbf{U}_{k-1})$ is not included here because they are same as those for $\tilde{\mathbf{X}}_k \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}}_k, \mathbf{U}_k)$. Note that the logical connectives {NOT, AND, OR} are denoted by {~, &, #} respectively.

Equations for $\tilde{\mathbf{X}}_k \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}}_k, \mathbf{U}_k)$ and $\mathbf{Y}_k \leftrightarrow \mathbf{g}(\tilde{\mathbf{X}}_k, \mathbf{U}_k)$:

```
(x12 <-> (xs6 & (x13 # x12))).
(z23 <-> ((u4 # u60) & (u6 # u61) & (x12 # z32))).
(z32 <-> (xs7 # xs8 # xs9)).
(x1 <-> ((z23 & ~x1) # (~u47 & x1))).
(ti2 <-> (xs5 & (x14 # ti2))).
((~ti2 & ~to2)#(~ti2_p & ti2 & ~to2)#(ti2_p & ti2 & to2)).
(ti3 <-> (xs6 & (x13 # ti3))).
((~ti3 & ~to3)#(~ti3_p & ti3 & ~to3)#(ti3_p & ti3 & to3)).
(z24 <->((to2&(~u17#u36))#(to3&(~u17#~u20#u36#u39))
      #(((~u17#u36)&(~u20#u39))&z32))).
(x2 <-> ((z24 & ~x2) # (~u47 & x2))).
(z25 <-> ((u1#u53)&(u5#u55)& ~xs1 & ~xs2)).
(x3 <-> ((z25 & ~x3) # (~u47 & x3))).
(z26 <-> (((~u18 & ~u37)#(~u21&~u40)#((u18#u21)&u54)#(u37&u40)
      #((u37#u40)&u2))& ~xs1 & ~xs2)).
(x4 <-> ((z26 & ~x4) # (~u47 & x4))).
(z27 <-> (z33&z34)).
(z33 <-> (((~u19#u38)&(~u28#u43))#((~u24#u41)&(~u29#u44)))).
(z34 <-> (xs5#xs6#xs7#xs8#xs9#xs10)).
(x5 <-> ((z27 & ~x5) # (~u47 & x5))).
(z28 <-> (z35&(u4#u62)&(u6#u63))).
(z35 <-> (xs6#xs7#xs8#xs9)).
(x6 <-> ((z28 & ~x6) # (~u47 & x6))).
(z29 <-> (((u62&u61)#(u60&u63))& ~u4 & ~u6 & ~xs1)).
(x7 <-> ((z29 & ~x7) # (~u47 & x7))).
(z30 <-> ((u7#u64)&(u9#u65))).
```

```
(x8 <-> ((z30 & ~x8) # (~u47 & x8))).
(z31 <-> ((u3#u56)&(u8#u50))).
(x9 <-> ((z31 & ~x9) # (~u47 & x9))).
(y1 <-> (((u19 & ~u28)#(~u19 & u28))#((u24 & ~u29)#(~u24 & u29)))).
(y7 <-> (xs1_p & u45 & u46)).
(y7 -> xs2).
((xs1_p & ~y7) -> xs1).
(y16 <-> (xs2_p & ~u45 & u46)).
(y16 -> xs1).
((xs2_p & ~y16) -> xs2).
(z20 <-> (u28 & u19 & u24 & u29)).
(z22 <-> (u10&u11&u12&u14&u15&u16& ~u30 & ~u31 & ~u32 & ~u34 & ~u35)).
(y8 <-> (z22 & z20 & ~u17 & ~u20 & u45 & u46 & u47 & xs2_p)).
(y8 -> xs3).
(xs2_p & ~y8 -> xs2).
(y17 <-> (to4 & ~u45 & u46)).
(ti4 <-> (xs4_p & u47)).
((~ti4 & ~to4)#(~ti4_p & ti4 & ~to4)#(ti4_p & ti4 & to4)).
(y17 -> xs2).
(xs4_p & ~y17 -> xs4).
((~ti1 & ~to1)#(~ti1_p & ti1 & ~to1)#(ti1_p & ti1 & to1)).
(ti1 <-> ((xs3_p#xs10_p)&u13&u57&u26&u49&z22
      & ~u17& ~u20& ~u33& ~u1& ~u42& ~u36& ~u39)).
(y9 <-> (to1 & xs3_p & u47 & z20)).
(y9 -> xs4).
(xs3_p & ~y9 -> xs3).
(y18 <-> (((u17#u20# ~z22# ~u47)&xs4_p)#to5#to9)).
(ti5 <-> (xs10_p & u47 & z20)).
(ti9 <-> xs4_p).
((~ti5 & ~to5)#(~ti5_p & ti5 & ~to5)#(ti5_p & ti5 & to5)).
((~ti9 & ~to9)#(~ti9_p & ti9 & ~to9)#(ti9_p & ti9 & to9)).
(y18 -> xs3).
(xs4_p & ~y18 -> xs4).
(xs10_p & ~y18 -> xs10).
(y10 <-> (to6 & u45 & u46 & u47 & z20)).
(ti6 <-> (xs4_p&u23)).
((~ti6 & ~to6)#(~ti6_p & ti6 & ~to6)#(ti6_p & ti6 & to6)).
(y10 -> xs5).
(xs4_p & ~y10 -> xs4).
(y11 <-> (to10 & u22 & u23 & u45 & u46)).
(ti10 <-> (xs5_p & u17)).
((~ti10 & ~to10)#(~ti10_p & ti10 & ~to10)#(ti10_p & ti10 & to10)).
(y11 -> xs6).
(xs5_p & ~y11 -> xs5).
(y12 <-> to8).
(ti8 <-> (u17 & u20 & xs6_p)).
((~ti8 & ~to8)#(~ti8_p & ti8 & ~to8)#(ti8_p & ti8 & to8)).
(y12 -> xs7).
(xs6_p & ~y12 -> xs6).
(y13 <-> (to7 & xs7_p)).
(ti7 <-> (u58 # u59)).
((~ti7 & ~to7)#(~ti7_p & ti7 & ~to7)#(ti7_p & ti7 & to7)).
(y13 -> xs8).
(xs7_p & ~y13 -> xs7).
(y19 <-> (xs9_p & u45 & u46)).
(y19 -> xs7).
(xs9_p & ~y19 -> xs9).
(y14 <-> (~u45 & u46 & xs8_p)).
(y14 -> xs9).
(xs8_p & ~y14 -> xs8).
(y15 <-> (xs9_p & u48 & ~u45 & u46)).
```

```
(y15 -> xs10).
(xs9_p & ~y15 -> xs9).
(z21 <-> (x1 # x2 # x3 # x4 # x5 # x6 # x7 # x8 # x9)).
(y20 <-> ((xs5_p # xs6_p # xs7_p # xs8_p # xs9_p) & z21)).
(y20 -> xs10).
(xs5_p & ~y20 -> xs5).
(xs6_p & ~y20 -> xs6).
(xs7_p & ~y20 -> xs7).
(xs8_p & ~y20 -> xs8).
(xs9_p & ~y20 -> xs9).
(z15 <-> (((xs6&u22&u23)#xs7#xs8#xs9) & ~z21)).
(z1 <-> (~z15 # z21 # z36)).
(z36 <-> xs1#xs2#xs3#xs4#xs5#xs10).
(z2 <-> z1).
(y2 <-> (z15 & ~z1)).
(x10 <-> ((x10 # u25) & z15 & ~z21)).
(z5 <-> (~x10 # z21 # z36)).
(z6 <-> z5).
(y3 <-> (x10 & ~z5)).
(z16 <-> (x10 & ~z21)).
(z3 <-> (~z16 # z21 # z36)).
(z4 <-> z3).
(x13 <-> (z16 & ~z3)).
(z17 <-> ((xs5#(xs6&u52)) & ~z21)).
(z7 <-> ~z17 # z21 # z37)).
(z37 <-> (xs1#xs2#xs3#xs4#xs7#xs8#xs9#xs10)).
(z8 <-> z7).
(y4 <-> (z17 & ~z7)).
(x11 <-> ((x11 # u27) & z17 & ~z21)).
(z9 <-> (~x11 # z21 # z37)).
(z10 <-> z9).
(y5 <-> (x11 & ~z9)).
(z18 <-> (x11 & ~z21)).
(z11 <-> (~z18 # z21 # z37)).
(z12 <-> z11).
(x14 <-> (z18 & ~z11)).
(z19 <-> (xs5 & z18 & u51 & ~z21)).
(z13 <-> (~z19 # z21 # z37 # xs6)).
(z14 <-> z13).
(y6 <-> (z19 & ~z13)).
```

Equations for $\mathbf{h}(\tilde{\mathbf{X}}_k, \mathbf{U}_k) \rightarrow (\tilde{\mathbf{X}}_k \leftrightarrow \tilde{\mathbf{X}}_{k-1})$:

```
((xs6 & ~x13) -> (x12 <-> x12_p)).
((z23 & u47) -> (x1 <-> ~x1_p)).
((~z23 & ~u47) -> (x1 <-> x1_p)).
((xs5 & ~x14) -> (ti2 <-> ti2_p)).
((xs6 & ~x13) -> (ti3 <-> ti3_p)).
((z24 & u47) -> (x2 <-> ~x2_p)).
((~z24 & ~u47) -> (x2 <-> x2_p)).
((z25 & u47) -> (x3 <-> ~x3_p)).
((~z25 & ~u47) -> (x3 <-> x3_p)).
((z26 & u47) -> (x4 <-> ~x4_p)).
((~z26 & ~u47) -> (x4 <-> x4_p)).
((z27 & u47) -> (x5 <-> ~x5_p)).
((~z27 & ~u47) -> (x5 <-> x5_p)).
((z28 & u47) -> (x6 <-> ~x6_p)).
((~z28 & ~u47) -> (x6 <-> x6_p)).
((z29 & u47) -> (x7 <-> ~x7_p)).
((~z29 & ~u47) -> (x7 <-> x7_p)).
```

```
((z30 & u47) -> (x8 <-> ~x8_p)).
((~z30 & ~u47) -> (x8 <-> x8_p)).
((z31 & u47) -> (x9 <-> ~x9_p)).
((~z31 & ~u47) -> (x9 <-> x9_p)).
((~u25 & z15 & ~z21) -> (x10 <-> x10_p)).
((~u27 & z17 & ~z21) -> (x11 <-> x11_p)).
```

Variable List:

Variables used in the Boolean state space model are listed in Tables B.1-B.4.

Table B.1: Inputs for MOD5™

Variables	Meaning
u1	combustion air flow in manual
u2	boiler level in manual
u3	steam pressure
u4	main fuel gas pressure to burner in manual
u5	combustion air blower motor amps in manual
u6	main fuel gas pressure to burner in manual
u7	stack temperature in manual
u8	steam temperature in manual
u9	stack temperature in manual
u10	main fuel gas vent emv TRUE if open
u11	upstream main fuel gas ebv TRUE if closed
u12	downstream main fuel gas ebv TRUE if closed
u13	TRUE if blower motor is running is a fail-on motor
u14	pilot gas vent ebv TRUE if open
u15	upstream pilot gas ebv TRUE if closed
u16	downstream pilot gas ebv TRUE if closed
u17	TRUE if pilot or main showing flame
u18	low low boiler level TRUE if not low low
u19	Mod5 emergency stop switch TRUE
u20	TRUE if main showing flame
u21	low low boiler level TRUE if not low low
u22	main fuel control valve in light off position
u23	air dampers in light off position
u24	field emergency stop switch TRUE
u25	main vent ebv showing closed
u26	air damper in purge position
u27	pilot vent ebv showing closed
u28	Mod5 emergency stop switch TRUE
u29	field emergency stop switch TRUE
u30	DIM(101)
u31	DIM(102)
u32	DIM(103)
u33	DIM(106)
u34	DIM(111)
u35	DIM(112)

u36	DIM(141)
u37	DIM(142)
u38	DIM(143)
u39	DIM(151)
u40	DIM(152)
u41	DIM(155)
u42	DIM(162)
u43	DIM(243)
u44	DIM(255)
u45	DK(11), permissive switch for step terminations
u46	DM(11), permissive switch for step terminations
u47	nsdn
u48	plant specific logic for cooldown step
u49	AC(2) LE AP(1998,-0.1,4)
u50	AC(1920) GT AP(1920)
u51	AC(1907) LT AP(1927,10,32767)
u52	AC(1908) LT AP(1928,10,32767)
u53	AI(101,300) LT AP(1911,50,300)
u54	AI(102,100) LT AP(1902,2,100)
u55	AI(111,75) LT AP(1910,7.5,75)
u56	AI(102,600) GT AP(1920,500,600)
u57	AI(101) GE AP(2000,75,100)
u58	AI(262) GE AP(1996)
u59	AI(272) GE AP(1996)
u60	AI(109,50) LT AP(1909,3,50)
u61	AI(119) LT AP(1909)
u62	AI(109) GT AP(1917,30,50)
u63	AI(119) GT AP(1917)
u64	AI(121,1000) GT AP(1919,700,1000)
u65	AI(131) GT AP(1919)

Table B.2: Outputs for MOD5™

Variables	Meaning
y1	emergency stop switch disagreement
y2	main vent
y3	main upstream ebv
y4	pilot vent
y5	pilot upstream ebv
y6	ignition electrode
y7	leave maintenance wait
y8	leave process wait
y9	leave purge
y10	leave purge complete
y11	leave light pilot
y12	leave main lightoff
y13	leave warmup
y14	leave run

y15	leave cooldown
y16	return to STEP(103)
y17	return to STEP(104)
y18	return to STEP(105)
y19	return to STEP(109)
y20	jump to STEP(112)

Table B.3: State variables for MOD5™

Variables	Meaning
x1	ALM(109), low low fuel gas pressure
x2	ALM(110), loss of flame
x3	ALM(111), low low combustion air
x4	ALM(112), low low boiler level
x5	ALM(113), emergency stop
x6	ALM(117), high high fuel gas pressure
x7	ALM(118), high/low fuel gas pressure
x8	ALM(119), high high stack temperature
x9	ALM(120), high high steam pressure
x10	main upstream ebv open
x11	pilot upstream ebv open
x12	TRUE when main fg downstream block first opens
x13	main fuel gas downstream ebv
x14	pilot downstream ebv
xs1	STEP(103), maintenance wait
xs2	STEP(104), process wait
xs3	STEP(105), purge
xs4	STEP(106), purge complete
xs5	STEP(107), light pilot
xs6	STEP(108), main lightoff
xs7	STEP(109), warmup
xs8	STEP(110), run
xs9	STEP(111), cooldown
xs10	STEP(112), shutdown
ti1	input to purge timer
to1	output from purge timer
ti2	input to timer for seeing pilot flame in STEP(107)
to2	output from timer for seeing pilot flame in STEP(107)
ti3	input to timer for seeing main flame in STEP(108)
to3	output from timer for seeing main flame in STEP(108)
ti4	input to timer in STEP(104)
to4	output from timer in STEP(104)
ti5	input to timer in STEP(105)
to5	output from timer in STEP(105)
ti6	input to timer in STEP(106)
to6	output from timer in STEP(106)
ti7	input to timer for terminating STEP(109)
to7	output from timer for terminating STEP(109)

ti8	input to timer for terminating STEP(108)
to8	output from timer for terminating STEP(108)
ti9	input to timer for purge complete step
to9	output from timer for purge complete step
ti10	input to timer in STEP(107)
to10	output from timer in STEP(107)

Table B.4: Intermediate variables for MOD5™

Variables	Meaning
z1	abort for main fuel vent
z2	parallel abort for main fuel vent
z3	abort for main fuel downstream block
z4	parallel abort for main fuel downstream block
z5	abort for main fuel upstream block
z6	parallel abort for main fuel upstream block
z7	abort for pilot vent
z8	parallel abort for pilot vent
z9	abort for pilot upstream block
z10	parallel abort for pilot upstream block
z11	abort for pilot downstream block
z12	parallel abort for pilot downstream block
z13	abort for ignition electrode
z14	parallel abort for ignition electrode
z15	main vent ebv close
z16	main downstream ebv open
z17	pilot vent ebv close
z18	pilot downstream ebv open
z19	pilot ignition electrode
z20	TRUE if emergency stop switches in run
z21	TRUE whenever a shutdown alarm is TRUE
z22	TRUE if all fuel valves are in their failsafe state
z23	SET(109), low low fuel gas pressure
z24	SET(110), loss of flame
z25	SET(111), low low combustion air
z26	SET(112), low low boiler level
z27	SET(113), emergency stop
z28	SET(117), high high fuel gas pressure
z29	SET(118), high/low fuel gas pressure
z30	SET(119), high high stack temperature
z31	SET(120), high high steam pressure
z32	intermediate variables introduced
z33	intermediate variables introduced
z34	intermediate variables introduced
z35	intermediate variables introduced
z36	intermediate variables introduced
z37	intermediate variables introduced

Appendix C

Throx Sequence in MDI Plant

The implicit Boolean state space model derived from the SEQ(36) in the Throx sequence is listed here with the list of variables. Note that equations for $\tilde{\mathbf{X}}_{k-1} \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}}_{k-1}, \mathbf{U}_{k-1})$ is not included here because they are same as those for $\tilde{\mathbf{X}}_k \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}}_k, \mathbf{U}_k)$. Note that the logical connectives {NOT, AND, OR} are denoted by {~, &, #} respectively.

Equations for $\tilde{\mathbf{X}}_k \leftrightarrow \mathbf{f}(\tilde{\mathbf{X}}_k, \mathbf{U}_k)$ and $\mathbf{Y}_k \leftrightarrow \mathbf{g}(\tilde{\mathbf{X}}_k, \mathbf{U}_k)$:

```
(z75 <-> u46 # u47).
(z76 <-> u48 # u49).
(z77 <-> u50 # u51).
(z48 <->  u23 # u24).
(z49 <->  u25 # u26).
(z50 <->  u27 # u28).
(z51 <->  u29 # u30).
(z52 <->  u31 # u32).
(z53 <->  u33 # u34).
(z54 <->  u35 # u36).
(z55 <->  u37 # u38).
(z69 <->  u40 # u41).
(z70 <->  u42 # u43).
(z71 <->  u44 # u45).
(z78 <->  u52 # u53).
(z112 <->  u57 # u58).
(z113 <->  u59 # u60).
(z126 <-> xs29#xs30#xs31#xs32#xs33#xs34#xs35).
(z23 <->  (~x29 & ~x7 & z67) & z126).
(z10 <->  z23 & u328).
(z24 <-> u206 & ~z25).
(z25 <-> u206).
(z7 <-> xs29 & ~z8).
(z8 <-> xs29).
```

```
(z26 <-> z10 & u95 & u94).
(y60 <->  ~y64 # x2#x4#x30#x5#x17).
(z109 <->  ((u344 & ~u116) # ~z68) & xs31).
(z111 <-> x7 & xs29 & ~z110).
(z110 <-> x7 & xs29).
(z144 <-> (u346 & xs29 & ~u99) # ( ~z67 & (xs30#xs29))).
(x1 <->  z144 # z109 # (x1 & ~u147)).
(z127 <-> xs31#xs32#xs33#xs34#xs35).
(z128 <-> xs32#xs33#xs34#xs35).
(z129 <-> xs29#xs30#xs31).
(z142 <-> (x17 & ~xs30) # (x17 & x29 & z127)).
(z143 <-> (x29 & z129) #  (~z67 & ~z68 & z128)).
(x2 <->  z142 # z143 #  (x2 & ~u210 & ~u148)).
(y1 <->  ~u113 ).
(z162 <-> x16#x18#x20#x21#x22#x23#x24#x27#x31#z58#u4#u412# ~z61).
(x3 <-> (z162 & z127) # (x3 & ~u149)).
(z145 <-> (x3 & ~xs30) # (x3 & xs31) # (x3 & x29 & z126)).
(x4 <->  z145 # (x4 & ~u210 & ~u150)).
(z163 <-> y20#x25#x26#x28# ~z74).
(x5 <-> (z163 & y64) # (x5 & ~u151)).
(x52 <->  (u238 #  (x52 & u237)) & z128).
(y2 <-> (u99 & ~z126) # (u116 & ~z127) # x52).
(x53 <-> (u337 # (x53 & u336)) & ~(xs24#xs25#xs26#xs36)).
(y3 <-> x53).
(x33 <-> (u219 # (x33 & u218)) ).
(y4 <-> x33 & xs28 & ~u55 ).
(x79 <-> (u244 # (x79 & u243)) ).
(x56 <-> (u268 # (x56 & u267)) ).
(x54 <-> (u242 # (x54 & u240))).
(x55 <-> (u265 # (x55 & u263)) ).
(y5 <-> x54 # x55 # x56 # x79 ).
(y6 <->   (~u80 & ~xs24) ).
(x60 <->  (u227 # (x60 & u226))).
(x61 <->  ((u230 # (x61 & u228)) & y67)).
(y7 <->  ((x60 # x61) & y67) ).
(x6 <->  (xs28 & ~(x84 # u55) & u411) #  (x6 & ~u152)).
(x7 <->  (u343 & (xs29# xs32# xs33# xs34 # xs35)) #  (x7 & ~u112)).
(z130 <-> xs1#xs2#xs3).
(z131 <-> xs5#xs6#xs7).
(z132 <-> xs8#xs9#xs11).
(z133 <-> xs12#xs13#xs15).
(z134 <-> xs20#xs21#xs23).
(z135 <-> xs16#xs17#xs19).
(z30 <->  xs37 & z130 & z131 & z132 & z133 & z134 & z135).
(z136 <-> xs25#xs26#xs27#xs28).
(z137 <-> z136#xs29#xs30#xs31#xs32).
(z17 <-> ((u94&u95&u96&u97&z136)#(u96&u97&(xs29#xs30)))&
      u105&u104&u103&u102&u101&u100&u109&u107&u79&u89&u90&u91&u92&u67
      &u68&u65&u66&u63&u64&u69&u70&u71&u72&u73&u74&u75&u76 & z137).
(z123 <->  ~(y7#y9#x8#y10#x11# y12#x12#x13#x14#y21#y22#x16#u3#u4)).
(z138 <-> xs26#xs27#xs28#xs29#xs30#xs31).
(y8 <->  ((~z30 # ~z123 # u93) & z138) # ( ~z17&(xs26#xs27#xs28#xs30))).
(x34 <->  (u221 #  (x34 & u220))).
(y9 <->  (x34 & u201) #   (~u108 & u201)).
(z139 <-> xs24#xs25#xs26).
(z146 <-> (~u114 & ~z139) # (~u115 & ~z139)).
(x8 <->  ((u224 #  (x8 & u223)) & u201) # z146).
(x9 <->  ((u235 # (x9 & u234)) & z127) ).
(x10 <->  (u232 # (x10 & u231)) ).
(x42 <->  (u317 # (x42 & u316)) ).
(x43 <->  (u319 # (x43 & u318) & ~u207) ).
```

```
(z31 <->  ~u126 ).
(z164 <-> x42#x43#u15#u16#u17#z31#u413).
(y10 <->   z164 & ~xs24 & ~u207).
(x48 <->  ((u332) # (x48 & (u331)))).
(x49 <->  ((u330) # (x49 & (u329))) ).
(y11 <->  (x48 # x49) & ~(xs24#xs25#xs26#xs27#xs28#xs36)).
(x11 <->  ((u326 # (x11 & u324)) & u122 & ~u125)).
(z28 <->  (u347 & u200) ).
(x50 <->  ((u212 # (x50 & u211)) & u200) ).
(x51 <->  ((u214 # (x51 & u213)) & u200) ).
(z29 <->  (u200 $ ~u81)).
(y12 <->  (z28#x50#x51#x29) & ~xs24).
(x57 <->  (u216 #  (x57 & u215)) & ~xs24).
(x58 <->  (u260 # (x58 & u259)) & (u408)).
(y13 <->  x57 # x58 # ~u61).
(x12 <->  (u248 # (x12 & u246) # ~u117) ).
(x13 <->  (u251 # (x13 & u250) # ~u118 # ~u120) & ~z139).
(x14 <->  (u262 # (x14 & u261)) ).
(x64 <->  u254 # (x64 & (xs26#xs27#xs28#xs29 #xs30#xs31#xs32))
      # xs33 #xs34 # xs35).
(x62 <->  (u254 # (x62 & u253) # ~u119) ).
(x63 <->  (u258 # (x63 & u256)) & x64).
(y14 <->  x62 # x63).
(x65 <->  (u241 # (x65 & u239)) ).
(x66 <->  (u266 # (x66 & u264)) ).
(z147 <-> (x65 & x66) # ((u11 # u7) & x65) # ((u10 # u6) & x66)).
(x15 <->  z147 #  (x15 & ~u153)).
(x35 <->  (u249 # (x35 & u247)) ).
(y15 <->  ((~u117 & x35 & ~u137 & ~z52)#(~u117 & z52)#(x35 & u137))).
(x67 <->  (u287 #  (x67 & u286))).
(x69 <->  (u288 #  (x69 & xs32))).
(x68 <->  ((u291 #  (x68 & u289)) & (x69 # xs33 # xs34 # xs35)) ).
(y16 <->  (x67 # x68) & ~xs24).
(x70 <->  (u273 & u296 & u280 & u305) # (x70 & xs32)).
(x36 <->  ((u275 # (x36 & u274)) & (x70#y64))).
(x71 <->  ((u298 # (x71 & u297)) & (x70#y64))).
(x37 <->  ((u283 # (x37 & u281)) & (x70#y64))).
(x72 <->  ((u307 # (x72 & u306)) & (x70#y64))).
(y17 <->  x36 # x71 # x37 # x72).
(x38 <->  (u270 # u293 # (x38 & (u269 # u292)))).
(x39 <->  (u284 # u302 # (x39 & (u282 # u301)))).
(y18 <->  (x38 # x39) & ~(xs24 # xs25)).
(x74 <->  (u311 & u370 & u396) # (x74 & xs32)).
(x73 <->  ((u315 # (x73 & u314)) & (x74#y64)) ).
(x75 <->  ((u394 # (x75 & u393)) & (x74#y64))).
(x78 <->  ((u402 # (x78 & u401)) & (x74# y64))).
(x44 <->  (u312 # (x44 & u310)) ).
(x45 <->  (u390 # (x45 & u380)) ).
(x46 <->  (u398 # (x46 & u397)) ).
(y19 <->  (x44 # x45 # x46 # x73 # x75 # x78) & ~xs24).
(x47 <->  (u325 # (x47 & u323)) ).
(y20 <->  x47 & u122 & ~u125 & ~u21).
(x40 <->  (u342 #  (x40 & u341)) ).
(x41 <->  (u322 # (x41 & u321))).
(y21 <->  (x40 # x41) & ~(xs24 # xs25)).
(y22 <->  ((~u110 # ~u84) & ~(xs24 # xs25))).
(x16 <->  (z46 & ~z139) # (x16 & ~u154)).
(z148 <-> ((~u116 # ~z68) & z128) # (~z68 & xs31) ).
(x17 <->  z148 # (x17 & ~u155)).
(z149 <-> (z60 & y65 & ~(xs24 # xs25)) ).
(x18 <->   z149 # (x18 & ~u156)).
```

```
(z150 <-> (z124 & ~y83 & ~(xs24#xs25)) ).
(x19 <->  z150 # (x19 & ~u157)).
(z9 <->  u354 & ~z75).
(z32 <->  ~u108 # u133).
(z27 <->  u222 # z76).
(z33 <->  u233 # z77).
(z6 <->  xs31 # xs32 # xs33 # xs34 # xs35).
(z1 <->  z32 & z27 & z6).
(z2 <->  z27 & z33 & z6).
(z3 <->  z32 & z33 & z6).
(z4 <->  u98 & u205 & z6).
(z5 <->  z9 & y67).
(x20 <->  (z1#z2#z3#z4#z5) # (x20 & ~u158)).
(z34 <->  ((~u114 & u201) # u136) ).
(z35 <->  ((u225 # z76) & u201) ).
(z36 <->  ((u236 # z77) & u204 & u205 & u409)).
(z151 <-> (((z34 & z35) # (z35 & z36) # (z34 & z36)) & z127) ).
(x21 <->  z151 # (x21 & ~u159)).
(z152 <-> (z57 & ~(xs24 # xs25 # xs26)) ).
(x22 <->  z152 # (x22 & ~u160)).
(x23 <->  z56 # (x23 & ~u161)).
(x24 <->  z59 # (x24 & ~u162)).
(x82 <->  (u359 # (x82 & u358))).
(x81 <->  (u364 # (x81 & u363))).
(x80 <->  (u362 # (x80 & u361))).
(z43 <->  u365 ).
(z41 <->  u368 ).
(z42 <->  u369 ).
(y23 <->  ((x82 # x80 # x81 # z43 # z41 # z42) & y64) ).
(z38 <->  ((u271 # u294) & (u278 # u303) & xs32) # xs33 # xs34 # xs35).
(z11 <->  (u276 & z38)).
(z12 <->  (u299 & z38)).
(z13 <->  (u285 & z38)).
(z37 <->  (u308 & z38)).
(x25 <->  (z11#z12#z13#z37) # (x25 & ~u163)).
(z14 <->  u272).
(z15 <->  u295).
(z16 <->  u279).
(z39 <->  u304).
(x26 <->  (z14#z15#z16#z39) # (x26 & ~u164)).
(x27 <->  z120 # (x27 & ~u165)).
(z40 <->  ((u313 # u391 # u399) & xs32) # xs33 # xs34 # xs35).
(x28 <->  (z73 & z40) # (x28 & ~u166)).
(z140 <-> (~z67 & x59 & z129) # ((~z67 # (~u99 & ~x76)) & z128) ).
(z141 <-> (u202 & u203 & ~u115) # (u202 & u203 & ~u87) ).
(x59 <->  (u99 & (xs29#xs30)) # (x59 & (xs29#xs30))).
(x76 <->  (z128 & u112) #  (x76 & (z128 & ~u99))).
(x29 <->  z140 # z141 #  (x29 & ~u112 & z128) # (x29 & ~u167 & xs36)).
(z153 <-> u5 # x19 # (~u110 & ~z62) # ~xs37 # (z9 & ~(xs24#xs25)) #
(~u113 & (xs30#xs31#xs32#xs33#xs34#xs35)) ).
(x30 <-> z153 # (x30 & ~u210 & ~u168)).
(x31 <->  (z72 & ~(xs24#xs25)) # (x31 & ~u169)).
(x32 <->  z58 # (x32 & ~u170)).
(z122 <->  x1 # x2 # x4 # x30 # x29).
(z121 <->  x3 # x4 # x30 # x17).
(z44 <->  (~u110 & ~u84 & ~u134 & ~u130)#(~u110 & u130)#(~u84 &u134)).
(z45 <->  (~u111 & ~u85 & ~u135 & ~u131)#(~u111 & u131)#(~u85 & u135)).
(z79 <->  (u327 & ~y83) # z78).
(z80 <->  (u320 & ~y83) # z50).
(z81 <->  (~u121 & ~y83) # u142).
(z124 <->  (z79 & z80) # (z80 & z81) # (z79 & z81)).
```

```
(z47 <->  ~z44 & ~z45 & ~z124 & y77).
(z56 <->  u356 & ~z51).
(z85 <->  (~u118 # u138) # u82).
(z86 <->  (~u120 # u140) # u83).
(z87 <->  (u355 # z52) ).
(z57 <->  (z85 & z86) # (z85 & z87) # (z86 & z87)).
(z58 <->  u360 & ~z53).
(z59 <->  ((u255# ~u119)& ~u139& ~z55)#(u255&u139)
       #(~u119&z55)#(u139&z55) ).
(z88 <->  (u81 # u129) ).
(z89 <->  u351 # z54).
(z90 <->  u348 # u141).
(z60 <->  (z88 & z89) # (z88 & z90) # (z89 & z90)).
(z82 <->  u352 # z48).
(z83 <->  u353 # z49).
(z84 <->  (~u106 # u132) ).
(z46 <->  (z82 & z83) # (z82 & z84) # (z83 & z84)).
(z72 <->  (u357 & u349) # (u357 & z70) # (u349 & z69)).
(z117 <->  u366 # z71).
(z118 <->  u392 # z112 # u127 # ~u123).
(z119 <->  u400 # z113 # u128 # ~u124).
(z120 <->  ((z117 & z118) # (z118 & z119) # (z117 & z119))).
(z61 <->  ~z56 & ~z57 & ~z58 & ~z59 & ~z60 & ~z46 & ~z72 & ~z120).
(z62 <->  z47 & z61).
(z64 <->  u75 & u76 & u73 & u74 & u71 & u72 & u69 & u70 & u63 & u64
       & u65 & u66 & u67 & u68 & u89 & u90 & u91 & u92).
(z65 <->  u107 & u79 & u77 & u78).
(z63 <->  z64 & z65).
(z91 <->  z63 & z62).
(z66 <->  u350 & ~z54).
(z125 <->  z66 # u55).
(z93 <->  ~u99 & u95).
(z94 <->  u55 & u412 ).
(z92 <->  z93 # z94).
(y62 <->  x83 & ~z101).
(z101 <->  x83).
(z108 <->  u112 ).
(z153 <-> (z91 & u97 & ~u116 & z92 & z108) ).
(x83 <->  z153 #  (x83 & z91 & u97 & ~u116 & z92 & u345)).
(z96 <->  u345).
(z97 <->  z125 & z96).
(z98 <->  u99 # u116).
(z99 <->  z98 & u55).
(z100 <->  z99 # z97).
(z154 <-> (z47 & z100 & z108) ).
(x84 <->  z154 # (x84 & z100 & z47)).
(z106 <->  u412 # u99).
(z107 <->  ~u115 ).
(z67 <->  u55 & z106 & ~z107).
(z102 <->  u56 & u99 & z63).
(z103 <->  z102 # u116).
(z104 <->  ~u108 ).
(z105 <->  ~u114 ).
(z68 <->  z103 & ~z104 & ~z105 & u55 & z61).
(z114 <->  u367 # z71).
(z115 <->  u395 # z112).
(z116 <->  u403 # z113).
(z73 <->  ((z114 & z115) # (z114 & z116) # (z115 & z116)) ).
(z74 <->  ~z73 & u116 & u55 & z61).
(z155 <-> ~u8 # u9 # ~z74 # xs1 # xs2 # xs4).
(y27 <->  ~u171 # z155).
```

```
(y28 <->  ~u172 # z155).
(y26 <->  (~u173 # ~u64 # ~u63) & ~xs1 ).
(y24 <->  ~u174 # z155).
(y25 <->  ~u175 # z155).
(y29 <->  (~u176 # ~u65 # ~u66) & ~xs1 ).
(y30 <->  ~u177 # z155).
(y31 <->  ~u178 # z155).
(y32 <->  (~u179 # ~u67 # ~u68) & ~xs1 ).
(y33 <->  ~u180 # z155).
(y34 <->  ~u181 # z155).
(y35 <->  (~u182 # ~u69 # ~u70) & ~xs1 ).
(y36 <->  ~u183 # z155).
(y37 <->  ~u184 # z155).
(y38 <->  (~u185 # ~u71 # ~u72) & ~xs1 ).
(y39 <->  ~u186 # z155).
(y40 <->  ~u187 # z155).
(y41 <->  (~u188 # ~u73 # ~u74) & ~xs1 ).
(y42 <->  ~u189 # z155).
(y43 <->  ~u190 # z155).
(y44 <->  (~u191 # ~u75 # ~u76) & ~xs1).
(y58 <->  ~y75 # ~z26 # x29 # z122 # z121 # ~z67 #  xs25 # xs26
      # xs27 # xs28 # xs30 # xs36).
(y52 <->  (~y70 # z122 # ~z67 # xs25 # xs26 # xs27 # xs28 # xs36)
      & (u95 & u94)).
(y54 <->  ~y68 # z122 # ~z67 # xs25 # xs26 # xs27 # xs28 # xs36).
(y53 <->  ~y69 # z122 # ~z67 # xs25 # xs26 # xs27 # xs28 # xs36).
(y55 <->  (~y73 # z121 # ~z68 # xs25#xs26#xs27#xs28#xs29#xs30#xs36)
      & u97 & u96).
(y57 <->  ~y71 # z121 # ~z68 # xs25#xs26#xs27#xs28#xs29#xs30#xs36).
(y56 <->  ~y72 # z121 # ~z68 # xs25#xs26#xs27#xs28#xs29#xs30#xs36).
(y59 <->  ~u199 # u1 # ~z74 # xs9 # xs11).
(y51 <->  ~u192 # u1 # ~z74 # xs9 # xs11).
(y46 <->  ~u193 # u2 # ~z74 # xs13 # xs14 # xs15).
(y47 <->  ~u194 # u2 # ~z74 # xs13 # xs14 # xs15).
(y45 <->  (~u195 # ~u89 # ~u90) & ~xs12).
(y49 <->  ~u196 # u2 # ~z74 # xs13 # xs14 # xs15).
(y50 <->  ~u197 # u2 # ~z74 # xs13 # xs14 # xs15).
(y48 <->  (~u198 # ~u91 # ~u92) & ~xs12).
(xs24 <->  u146 & u404 & xs25 & u290).
(xs25 <->  (u146 & u405 & xs24)#(u146&u405&xs27&u334&u335&u257)).
(xs26 <->  u146 & u406 & xs25 & ~y13 & ~x12).
(z156 <-> ((y8 # x6 # (~x84 & ~u55)) & xs28) # (y8 & xs29)).
(xs27 <->  (u245 & ~x12 & xs26) # z156 # (u146 & u407 & u210 & xs36)).
(xs28 <->  ~y8 & x84 & xs27).
(z18 <->  u55 & xs28).
(z19 <->  u338 & ~y3).
(z20 <->  (x84 & u55) # z67).
(z21 <->  ~y8 & z18 & z19 & z20).
(y61 <->  z21 & ~z22).
(z22 <->  z21).
(xs29 <->  z21 & xs28).
(xs30 <->  (u99 & xs29) # ((x3 # x17) & z128)).
(x77 <->  (xs29 # x77) & ~xs31).
(z157 <->  (~x77 & ~(x3 # x17 # y8) & u410 & u112 & xs30)).
(xs31 <->  z157 # (u410 & ~y8 & x77 & xs30 & ~u145)).
(xs32 <->  u205 & xs31).
(z158 <-> ((((u13 # ~u20) & ~xs18) # (u19 & ~xs10) # x5 # (~u208 & ~xs18)
      # (~u209 & ~xs10)) & xs34)).
(xs33 <->  (u277 & u300 & u333 & u309 & u252 & xs32)
      # z158 # (u14 & xs35)).
(xs34 <->  (u12 # u18) & xs33).
```

```
(xs35 <->  u62 & xs33 & u143 & xs22).
(y85 <->  xs26 # xs27 # xs28 # xs29 # xs30 # xs31 # xs32 # xs33
      # xs34 # xs35).
(y85 -> xs36).
(y63 <->  u113 # (~u113 & xs36)).
(y75 <->  (xs29 # xs32 # xs33 # xs34 # xs35) & z26).
(y70 <->  (xs29 # xs30 # xs31 # xs32 # xs33 # xs34 # xs35) & z10).
(y68 <-> ((z126 & z10) # (~u94 # ~u95)) & u87).
(y69 <-> z126 & z10 & u87).
(y67 <->  ~xs24 & ~z9 & u210 & ~xs36 & u144 & (u229)).
(y84 <->  u144).
(y73 <->  (z127 & z68 & ~(x17 # x3)) # (~u96 # ~u97)).
(y71 <->  (z127 & z68 & ~(x17 # x3)) & u86).
(y72 <->  (z127 & z68 & ~(x17 # x3)) & u86).
(y74 <->  ~(xs24#xs25)).
(y65 <->  ~(xs24#xs25) & ~u5 & ~x18).
(y66 <->  y65 & (xs33#xs34#xs35#xs36) &
      (~u187 & u74 & ~u172 & u64 & ~u175 & u66)).
(y64 <->  xs33#xs34#xs35 ).
(y83 <->  ~(~(xs24 # xs25) & ~u5 & ~x19)).
(y76 <->  xs27 & u217 & u111 & z62 & z66).
(z161 <-> u111 & u85 & ~y8 & xs27).
(z159 <->  z161 # (z91 & ~y8 & xs27) # ((x84 # u55) & ~y8 & xs28)).
(z160 <-> (z67 & ~y8 & ~x1 & (xs29 # xs30)) # ((z67 # z68) & z6)).
(y77 <->  (z159 # z160) & u210).
(y79 <->  xs28 & u217).
(y80 <->  u412 & z10).
(y81 <->  u99 & y69 & y68).
(y82 <->  u116 & y72 & y71).
(y78 <->  ~u110 & ~u84).
```

Equations for $\mathbf{h}\left(\tilde{\mathbf{X}}_k, \mathbf{U}_k\right) \rightarrow \left(\tilde{\mathbf{X}}_k \leftrightarrow \tilde{\mathbf{X}}_{k-1}\right)$:

```
((z144#z109# ~u147)& ~(z144#z109) -> (x1<->x1_p)).
((z142#z143# ~u210)&(z142#z143# ~u148)& ~(z142#z143) -> (x2<->x2_p)).
((z162# ~u149)&(z127# ~u149)& ~(z162&z127) -> (x3<->x3_p)).
((z145# ~u210)&(z145# ~u150)& ~z145 -> (x4<->x4_p)).
((z163# ~u151)&(y64# ~u151)& ~(z163&y64) -> (x5<->x5_p)).
((xs28# ~u152)&(~x84# ~u152)&(~u55# ~u152)&
      (u411# ~u152)& ~(xs28& ~x84& ~u55&u411) -> (x6<->x6_p)).
((u343# ~u112)&(xs29#xs32#xs33#xs34#xs35# ~u112)&
      ~(u343&(xs29#xs32#xs33#xs34#xs35)) -> (x7<->x7_p)).
((u221#u220)& ~u221 -> (x34<->x34_p)).
((u224#u223#z146)&(u201#z146)& ~(u224#z146) -> (x8<->x8_p)).
((u235#u234)&z127& ~u235 -> (x9<->x9_p)).
((u232#u231)& ~u232 -> (x10<->x10_p)).
((u326#u324)&u122& ~u125& ~u326 -> (x11<->x11_p)).
((u248#u246# ~u117)& ~(u248# ~u117) -> (x12<->x12_p)).
((u251#u250# ~u118# ~u120)& ~z139& ~(u251# ~u118# ~u120)
      -> (x13<->x13_p)).
((u262#u261)& ~u262 -> (x14<->x14_p)).
((z147# ~u153)& ~z147 -> (x15<->x15_p)).
((z46# ~u154)&(~z139# ~u154)& ~(~z139&z46) -> (x16<->x16_p)).
((z148# ~u155)& ~z148 -> (x17<->x17_p)).
((z149# ~u156)& ~z149 -> (x18<->x18_p)).
((z150# ~u157)& ~z150 -> (x19<->x19_p)).
((z1#z2#z3#z4#z5# ~u158)& ~(z1#z2#z3#z4#z5) -> (x20<->x20_p)).
((z151 # ~u159) & ~z151 -> (x21<->x21_p)).
((z152# ~u160)& ~z152 -> (x22<->x22_p)).
```

```
((z56# ~u161)& ~z56 -> (x23<->x23_p)).
((z59# ~u162)& ~z59 -> (x24<->x24_p)).
((z11#z12#z13#z37# ~u163)& ~(z11#z12#z13#z37) -> (x25<->x25_p)).
((z14#z15#z16#z39# ~u164)& ~(z14#z15#z16#z39) -> (x26<->x26_p)).
((z120# ~u165)& ~z120 -> (x27<->x27_p)).
((z73# ~u166)&(z40# ~u166)& ~(z40&z73) -> (x28<->x28_p)).
((z140#z141# ~u112# ~u167)&(z140#z141# ~u112#xs36)&
      (z140#z141#z128# ~u167)&(z140#z141#z128#xs36)&
       ~((z140#z141# ~u112)&(z140#z141#xs36)&(z140#z141# ~u167)
      &(z140#z141)&(z140#z141#z128)) -> (x29<->x29_p)).
((z153# ~u210)&(z153# ~u168)& ~z153 -> (x30<->x30_p)).
((z72# ~u169)&(~xs24# ~u169)&(~xs25# ~u169)& ~(~xs25& ~xs24&z72)
      -> (x31<->x31_p)).
((z58# ~u170)& ~z58 -> (x32<->x32_p)).
((u219#u218)& ~u219 -> (x33<->x33_p)).
((u221 # u220) & ~u221 -> (x34<->x34_p)).
((u249#u247)& ~u249 -> (x35<->x35_p)).
((u275#u274)&(x70#y64)& ~u275 -> (x36<->x36_p)).
((u283#u281)&(x70#y64)& ~u283 -> (x37<->x37_p)).
((u270#u293#u269#u292)& ~(u270#u293) -> (x38<->x38_p)).
((u284#u302#u282#u301)& ~(u284#u302) -> (x39<->x39_p)).
((u342#u341)& ~u342 -> (x40<->x40_p)).
((u322#u321)& ~u322 -> (x41<->x41_p)).
((u317#u316)& ~u317 -> (x42<->x42_p)).
((u319#u318)& ~u207& ~u319 -> (x43<->x43_p)).
((u312#u310)& ~u312 -> (x44<->x44_p)).
((u390#u380)& ~u390 -> (x45<->x45_p)).
((u398#u397)& ~u398 -> (x46<->x46_p)).
((u325#u323)& ~u325 -> (x47<->x47_p)).
((u332#u331)& ~u332 -> (x48<->x48_p)).
((u330#u329)& ~u330 -> (x49<->x49_p)).
((u212#u211)&u200& ~u212 -> (x50<->x50_p)).
((u214#u213)&u200& ~u214 -> (x51<->x51_p)).
((u238#u237)&z128& ~u238 -> (x52<->x52_p)).
((u337#u336)& ~xs24 & ~xs25 & ~xs26 & ~xs36 & ~u337 -> (x53<->x53_p)).
((u242#u240)& ~u242 -> (x54<->x54_p)).
((u265#u263)& ~u265 -> (x55<->x55_p)).
((u268#u267)& ~u268 -> (x56<->x56_p)).
((u216#u215)& ~xs24& ~u216 -> (x57<->x57_p)).
((u260#u259)&u408& ~u260 -> (x58<->x58_p)).
((u99#xs29#xs30)&(xs29#xs30)& ~((xs29#xs30)&u99) -> (x59<->x59_p)).
((u227#u226)& ~u227 -> (x60<->x60_p)).
((u230#u228)&y67& ~u230 -> (x61<->x61_p)).
((u254#u253# ~u119)& ~(u254# ~u119) -> (x62<->x62_p)).
((u258#u256)&x64& ~u258 -> (x63<->x63_p)).
((u254#xs26#xs27#xs28#xs29#xs30#xs31#xs32#xs33#xs34#xs35)&
      ~(u254#xs33#xs34#xs35) -> (x64<->x64_p)).
((u241#u239)& ~u241 -> (x65<->x65_p)).
((u266#u264)& ~u266 -> (x66<->x66_p)).
((u287#u286)& ~u287 -> (x67<->x67_p)).
((u291#u289)&(x69#xs33#xs34#xs35)& ~u291 -> (x68<->x68_p)).
((u288#xs32)& ~u288 -> (x69<->x69_p)).
((u273#xs32)&(u296#xs32)&(u280#xs32)&(u305#xs32)& ~(u305&u280&u296&u273)
      -> (x70<->x70_p)).
((u298#u297)&(x70#y64)& ~u298 -> (x71<->x71_p)).
((u307#u306)&(x70#y64)& ~u307 -> (x72<->x72_p)).
((u315#u314)&(x74#y64)& ~u315 -> (x73<->x73_p)).
((u311#xs32)&(u396#xs32)&(u370#xs32)& ~(u396&u370&u311)
      -> (x74<->x74_p)).
((u394#u393)&(x74#y64)& ~u394 -> (x75<->x75_p)).
```

```
(z128&(z128# ~u99)&(u112#z128)&(u112# ~u99)& ~(u112&z128)
      -> (x76<->x76_p)).
(~xs31 & ~xs29 -> (x77<->x77_p)).
((u402#u401)&(x74#y64)& ~u402 -> (x78<->x78_p)).
((u244#u243)& ~u244 -> (x79<->x79_p)).
((u362#u361)& ~u362 -> (x80<->x80_p)).
((u364#u363)& ~u364 -> (x81<->x81_p)).
((u359#u358)& ~u359 -> (x82<->x82_p)).
((z153#z91)&(z153#u97)&(z153# ~u116)&(z153#z92)&(z153#u345)& ~z153
      -> (x83<->x83_p)).
((z154#z100)&(z154#z47)& ~z154 -> (x84<->x84_p)).
```

Variable List:

Variables used in the Boolean state space model are listed in Tables C.1-C.3.

Table C.1: Inputs for Throx sequence

Variable	Meaning	Variable	Meaning
u1	ALM(303)	u2	ALM(324)
u3	ALM(425)	u4	ALM(432)
u5	ALM(445)	u6	AIM(324)
u7	AIM(329)	u8	DC(687)
u9	DC(688)	u10	DC(1042)
u11	DC(1060)	u12	DC(1113)
u13	DC(1185)	u14	DC(1223)
u15	DC(1252)	u16	DC(1253)
u17	DC(1254)	u18	DC(1308)
u19	DC(1309)	u20	DC(1330)
u21	DC(2092)	u22	DC(2402)
u23	DC(2406)	u24	DC(2408)
u25	DC(2409)	u26	DC(2411)
u27	DC(2412)	u28	DC(2414)
u29	DC(2415)	u30	DC(2417)
u31	DC(2418)	u32	DC(2420)
u33	DC(2421)	u34	DC(2423)
u35	DC(2425)	u36	DC(2427)
u37	DC(2428)	u38	DC(2430)
u39	DC(2444), rinsing is running	u40	DC(2455)
u41	DC(2457)	u42	DC(2458)
u43	DC(2460)	u44	DC(2461)
u45	DC(2463)	u46	DC(2467)
u47	DC(2469)	u48	DC(2470)
u49	DC(2472)	u50	DC(2473)
u51	DC(2475)	u52	DC(2477)
u53	DC(2479)	u54	DC(2493)
u55	DC(2494), rinse ended	u56	DC(2509)
u57	DC(2521)	u58	DC(2523)
u59	DC(2524)	u60	DC(2525)
u61	DC(2650)	u62	DC(2905)

u63	DI(201)	u64	DI(202)
u65	DI(223)	u66	DI(224)
u67	DI(226)	u68	DI(227)
u69	DI(229)	u70	DI(230)
u71	DI(242)	u72	DI(243)
u73	DI(245)	u74	DI(246)
u75	DI(248)	u76	DI(249)
u77	DI(302)	u78	DI(303)
u79	DI(304)	u80	DI(307)
u81	DI(308)	u82	DI(315)
u83	DI(316)	u84	DI(318)
u85	DI(319)	u86	DI(321)
u87	DI(322)	u88	
u89	DI(324)	u90	DI(325)
u91	DI(327)	u92	DI(328)
u93	DI(331)	u94	DI(332)
u95	DI(333)	u96	DI(335)
u97	DI(336)	u98	DI(337)
u99	DI(340)	u100	DI(342)
u101	DI(343)	u102	DI(344)
u103	DI(345)	u104	DI(346)
u105	DI(347)	u106	DI(348)
u107	DI(349)	u108	DI(361)
u109	DI(364)	u110	DI(374), emergency shutdown switch
u111	DI(376), start button pressed	u112	DI(377), start button pressed
u113	DI(380)	u114	DI(381), low natural gas pressure
u115	DI(382)	u116	DI(384), main flame on
u117	DI(386)	u118	DI(387)
u119	DI(388)	u120	DI(391)
u121	DI(401), blower break down	u122	DI(423)
u123	DI(426)	u124	DI(430)
u125	DI(433)	u126	DI(435)
u127	DI(464), blower break down	u128	DI(465), speed low
u129	DIM(308)	u130	DIM(318)
u131	DIM(319)	u132	DIM(348)
u133	DIM(361)	u134	DIM(374)
u135	DIM(376)	u136	DIM(381)
u137	DIM(386)	u138	DIM(387)
u139	DIM(388)	u140	DIM(391)
u141	DIM(396)	u142	DIM(401)
u143	DK(34)	u144	DK(329)
u145	DK(360)	u146	DM(36)
u147	DM(333)	u148	DM(334), reset SHDN(334)
u149	DM(336)	u150	DM(337)
u151	DM(338)	u152	DM(346), reset ALM(346)
u153	DM(364)	u154	DM(377)
u155	DM(378), reset WARN(378)	u156	DM(379), reset WARN(379)
u157	DM(380)	u158	DM(381)
u159	DM(382)	u160	DM(383)

u161	DM(384)	u162	DM(385)
u163	DM(387)	u164	DM(388)
u165	DM(389)	u166	DM(390)
u167	DM(391)	u168	DM(392)
u169	DM(393)	u170	DM(394)
u171	DO(201)	u172	DO(202)
u173	DO(222)	u174	DO(223)
u175	DO(224)	u176	DO(225)
u177	DO(226)	u178	DO(227)
u179	DO(228)	u180	DO(229)
u181	DO(230)	u182	DO(241)
u183	DO(242)	u184	DO(243)
u185	DO(244)	u186	DO(245)
u187	DO(246)	u188	DO(247)
u189	DO(248)	u190	DO(249)
u191	DO(250)	u192	DO(304)
u193	DO(324)	u194	DO(325)
u195	DO(326)	u196	DO(327)
u197	DO(328)	u198	DO(329)
u199	DO(349)	u200	DOT(308)
u201	DOT(331)	u202	DOT(332)
u203	DOT(333)	u204	DOT(335)
u205	DOT(336)	u206	DOT(339)
u207	DOT(401)	u208	NSDN(30)
u209	NSDN(32)	u210	NSDN(36)
u211	AC(301) GT AC(1301)	u212	AC(301) GT AP(1335)
u213	AC(301) LT AC(1302)	u214	AC(301) LT AP(1336)
u215	AC(302) LT AC(1007)	u216	AC(302) LT AP(1008)
u217	AC(311) GT AP(1028,10000,15000)	u218	AC(311) LT AC(1001)
u219	AC(311) LT AP(1877)	u220	AC(314) GT AC(1004)
u221	AC(314) GT AP(1005)	u222	AC(314) GT AP(1853,3,5)
u223	AC(314) LT AC(1005)	u224	AC(314) LT AP(1006)
u225	AC(314) LT AP(1808,1.09,5)	u226	AC(319) GT AC(1003)
u227	AC(319) GT AP(1004)	u228	AC(319) LT AC(1319)
u229	AC(319) LT AP(1327,16,21)	u230	AC(319) LT AP(1356)
u231	AC(320) GT AC(1294)	u232	AC(320) GT AP(1329)
u233	AC(320) GT AP(1854,750,1000)	u234	AC(320) LT AC(1006)
u235	AC(320) LT AP(1007)	u236	AC(320) LT AP(1809,40,1000)
u237	AC(322) LT AC(1307)	u238	AC(322) LT AP(1347)
u239	AC(324) GT AC(1012)	u240	AC(324) GT AC(1313)
u241	AC(324) GT AP(1013)	u242	AC(324) GT AP(1349)
u243	AC(324) LT AC(1315)	u244	AC(324) LT AP(1351)
u245	AC(325) GE AC(999)	u246	AC(325) GT AC(1008)
u247	AC(325) GT AC(1013)	u248	AC(325) GT AP(1009)
u249	AC(325) GT AP(1026)	u250	AC(325) LT AC(1009)
u251	AC(325) LT AP(1010)	u252	AC(326) GE AP(1032,12,21)
u253	AC(326) GT AC(1011)	u254	AC(326) GT AP(1012)
u255	AC(326) GT 19,21	u256	AC(326) LT AC(1321)
u257	AC(326) LT AP(636,1.1,21)	u258	AC(326) LT AP(1357)

u259	AC(327) LT AC(1317)	u260	AC(327) LT AP(1353)
u261	AC(328) GT AC(1010)	u262	AC(328) GT AP(1011)
u263	AC(329) GT AC(1314)	u264	AC(329) GT AC(1322)
u265	AC(329) GT AP(1350)	u266	AC(329) GT AP(1358)
u267	AC(329) LT AC(1316)	u268	AC(329) LT AP(1352)
u269	AC(333) GT AC(1016)	u270	AC(333) GT AP(1015)
u271	AC(333) GT AP(1366,860,2000)	u272	AC(333) GT AP(1818,1600,2000)
u273	AC(333) GT '1200,2000'	u274	AC(333) LT AC(1015)
u275	AC(333) LT AP(1014)	u276	AC(333) LT AP(1816,870,2000)
u277	AC(334) GE AP(1030,1000,2000)	u278	AC(334) GT AP(1367,860,2000)
u279	AC(334) GT AP(1819,1350,2000)	u280	AC(334) GT '950,2000'
u281	AC(334) LT AC(1196)	u282	AC(334) GT AC(1325)
u283	AC(334) LT AP(1177)	u284	AC(334) GT AP(1361)
u285	AC(334) LT AP(1817,870,2000)	u286	AC(335) GT AC(1028)
u287	AC(335) GT AP(1025)	u288	AC(335) GT AP(1370,760,2000)
u289	AC(335) LT AC(1323)	u290	AC(335) LT AP(1346,200,2000)
u291	AC(335) LT AP(1359)	u292	AC(343) GT AC(1016)
u293	AC(343) GT AP(1015)	u294	AC(343) GT AP(1366)
u295	AC(343) GT AP(1818)	u296	AC(343) GT '1200,2000'
u297	AC(343) LT AC(1015)	u298	AC(343) LT AP(1014)
u299	AC(343) LT AP(1816)	u300	AC(344) GE AP(1030)
u301	AC(344) GT AC(1325)	u302	AC(344) GT AP(1361)
u303	AC(344) GT AP(1367)	u304	AC(344) GT AP(1819)
u305	AC(344) GT '950,2000'	u306	AC(344) LT AC(1196)
u307	AC(344) LT AP(1177)	u308	AC(344) LT AP(1817)
u309	AC(360) GE AP(1031,200,400)	u310	AC(360) GT AC(1017)
u311	AC(360) GT AC(1327)	u312	AC(360) GT AP(1016)
u313	AC(360) GT AP(1372,195,400)	u314	AC(360) LT AC(1327)
u315	AC(360) LT AP(979)	u316	AC(418) GT AC(1085)
u317	AC(418) GT AP(1076)	u318	AC(418) LT AC(1086)
u319	AC(418) LT AP(1077)	u320	AC(418) LT 10,300
u321	AC(422) LT AC(1328)	u322	AC(422) LT AP(1364)
u323	AC(423) GT AC(1018)	u324	AC(423) GT AC(1296)
u325	AC(423) GT AP(1017)	u326	AC(423) GT AP(1332)
u327	AC(436) LT 100,2000	u328	AC(971) GE AP(956,15,32767)
u329	AC(973) GT AC(1295)	u330	AC(973) GT AP(1331)
u331	AC(974) GT AC(1295)	u332	AC(974) GT AP(1331)
u333	AC(1273) GE AP(1399,1250,2000)	u334	AC(1273) LT AP(634,700,2000)
u335	AC(1274) LT AP(634)	u336	AC(1278) LT AC(1000)
u337	AC(1278) LT AP(1000)	u338	AC(1278) LE AC(998)
u341	AC(1817) LT AC(1019)	u342	AC(1817) LT AP(1018)
u343	AC(1832) GE AP(1964,4,32767)	u344	AC(1922) EQ ZERO
u345	AC(1955) GT ZERO	u346	AC(1957) GE AP(1875,2,32767)
u347	AC(1986) LT AP(1334,500,3000)	u348	AC(1986) LT 300,3000
u349	AI(306) LT 2,10	u350	AI(311) GE 10000,15000
u351	AI(311) LT 1500,15000	u352	AI(315) LT 30,712
u353	AI(316) LT 1.07,2	u354	AI(319) GT AP(1943,19,21)
u355	AI(325) LT AP(1985,55,100)	u356	AI(328) GT 13.5,14.2
u357	AI(331) LT 2,10	u358	AI(355) GT AC(980)

u359	AI(355) GT AP(964)	u360	AI(335) GT 1100,1200
u361	AI(355) LT AC(979)	u362	AI(355) LT AP(963)
u363	AI(356) GT AC(980)	u364	AI(356) GT AP(964)
u365	AI(356) LT AP(966,100,400)	u366	AI(360) GT 300,400
u367	AI(360) LT 180,400	u368	AI(371) LT AP(967,100,400)
u369	AI(372) LT AP(968,100,400)	u370	AI(437) GT AC(746)
u380	AI(437) GT AC(747)	u390	AI(437) GT AP(707)
u391	AI(437) GT AP(708,195,1050)	u392	AI(437) GT 300,1050
u393	AI(437) LT AC(746)	u394	AI(437) LT AP(709)
u395	AI(437) LT 180,1050	u396	AI(447) GT AC(746)
u397	AI(447) GT AC(747)	u398	AI(447) GT AP(707)
u399	AI(447) GT AP(708)	u400	AI(447) GT 300,1050
u401	AI(447) LT AC(746)	u402	AI(447) LT AP(709)
u403	AI(447) LT 180,1050	u404	AK(36,0,32767) EQ 351,32767
u405	AK(36) EQ 352,32767	u406	AK(36) EQ 353,32767
u407	AK(36) EQ 354,32767	u408	AOT(354) LT AP(1354,50,110)
u409	AOT(359) GT '27,110'	u410	AOT(359) GE AP(802)
u411	STIME(36) GT AP(837,500,32767)	u412	DC(2499)
u413	DC(1073)		

Table C.2: Outputs for Throx sequence

Variables	Meaning
y1	ALM(335), critical instrument not ok
y2	ALM(339), insufficient flame or low intensity
y3	ALM(340), low air flow
y4	ALM(341)
y5	ALM(342), burner chamber pressure high/low
y6	ALM(343), major disturbance
y7	ALM(344)
y8	ALM(348)
y9	ALM(349)
y10	ALM(353), PB-521 speed, vibration, temperature problems
y11	ALM(355), energy input high, Max power 8800 KW
y12	ALM(358), PB-520 Supercharger problem
y13	ALM(359), condensation supply problem or failure
y14	ALM(363), steam flow high/low
y15	ALM(365), condensate level in boiler very high
y16	ALM(366), Turning chamber temperature high/low
y17	ALM(367), burner temperature low
y18	ALM(368), burner temperature high
y19	ALM(369), smoke gas temperature high/low
y20	ALM(370), CO concentration in smoke gas high
y21	ALM(371), low O_2 concentration in smoky gas, page 48
y22	ALM(375), emergency shutdown prealarm
y23	ALM(386), PFTB-521 outer wall temperature high high or low
y24	DA(201), PD-514 vent 1 block valve

y25	DA(202), PD-514 vent 2 block valve
y26	DA(203), PD-512 nitrogen valve
y27	DA(204), PD-512 vent 1 block valve
y28	DA(205), PD-512 vent 2 block valve
y29	DA(206), PD-514 vent nitrogen bleed valve
y30	DA(207), PDY-358 vent 1 block valve
y31	DA(208), PDY-358 vent 2 block valve
y32	DA(209), PDY-358 vent nitrogen valve
y33	DA(210), Anilin vent 1 block valve
y34	DA(211), Anilin vent 2 block valve
y35	DA(212), Anilin vent nitrogen valve
y36	DA(213), Acid amin vent 1 block valve
y37	DA(214), Acid amin vent 2 block valve
y38	DA(215), Acid amin vent nitrogen valve
y39	DA(216), MCB vent 1 block valve
y40	DA(217), MCB vent 2 block valve
y41	DA(218), MCB vent nitrogen valve
y42	DA(219), Formalin vent 1 block valve
y43	DA(220), Formalin vent 2 block valve
y44	DA(240), Formalin vent nitrogen valve
y45	DA(321), PT-541 vent nitrogen bleed valve
y46	DA(322), PT-541 vent 1 block valve
y47	DA(323), PT-541 vent 2 block valve
y48	DA(324), PE-323 vent nitrogen bleed valve
y49	DA(325), PE-323 vent 1 block valve
y50	DA(326), PE-323 vent 2 block valve
y51	DA(331), Liquid residue burning 2 valve
y52	DA(332), Pilot gas ventilation
y53	DA(333), Pilot gas 2 valve
y54	DA(334), Pilot gas 1 valve
y55	DA(335), Main gas ventilation
y56	DA(336), Main gas 2 valve
y57	DA(337), Main gas 1 valve
y58	DA(338), Abort ignition
y59	DA(339), Liquid residue burning 1 valve
y60	DC(1116)
y61	DC(1121)
y62	DC(2507)
y63	DO(301), critical instrument reset
y64	DO(306), computer activation
y65	DO(308), air blower PB-520
y66	DO(310)
y67	DO(331), natural gas block valve in the limit
y68	DO(332), pilot gas blockvalve 1
y69	DO(333), pilot gas blockvalve 2
y70	DO(334), pilot gas bleed
y71	DO(335), main-gas block valve 1
y72	DO(336), main-gas block valve 2
y73	DO(337), main gas bleed valve

y74	DO(338), combustion air block valve
y75	DO(339)
y76	DO(371), rinse ready
y77	DO(372), Mov-V hardware ok
y78	DO(373), emergency shutdown
y79	DO(374), Rinse running
y80	DO(375), Ignition running
y81	DO(376), Burner ignition running
y82	DO(377), Main burner running
y83	DO(401)
y84	LAO(379)
y85	JSTEP(363), PFTB-521 Shutdown

Table C.3: State variables for Throx sequence

Variables	Meaning
x1	ALM(333), ignition stopped
x2	ALM(334), mainflame and pilotflame failed
x3	ALM(336), shutdown except pilot-burner
x4	ALM(337), illegal process conditions
x5	ALM(338), residue shutdown
x6	ALM(346), Rinse time problem
x7	ALM(347), Pilot flame ignition problem
x8	ALM(350), natural gas pressure low
x9	ALM(351), natural gas flow low
x10	ALM(352), natural gas flow high
x11	ALM(356), high CO concentration in the outlet
x12	ALM(360), condensate level in boiler high
x13	ALM(361), condensate level in boiler low
x14	ALM(362), steam flow high
x15	ALM(364), burner chamber pressure very high
x16	ALM(377), instrument air pressure very low
x17	ALM(378), main flame off
x18	ALM(379), PB-520 trip
x19	ALM(380), PB-521 smoke gas blower trip
x20	ALM(381), natural gas pressure/flow very high
x21	ALM(382), natural gas pressure/flow very low
x22	ALM(383), condensate level in boiler very low
x23	ALM(384), steam flow very high
x24	ALM(385), steam pressure very high
x25	ALM(387), burning temperature very low
x26	ALM(388), burning temperature very high
x27	ALM(389), smoke gas temperature very high
x28	ALM(390), smoke gas temperature very low
x29	ALM(391), Pilot-burner off
x30	ALM(392), process shutdown
x31	ALM(393), O2 concentration in smoke gas very low

x32	ALM(394), turn chamber temperature too high
x33	DC(1000)
x34	DC(1001)
x35	DC(1002)
x36	DC(1003)
x37	DC(1004)
x38	DC(1005)
x39	DC(1006)
x40	DC(1007)
x41	DC(1008)
x42	DC(1149)
x43	DC(1150)
x44	DC(1210)
x45	DC(1211)
x46	DC(1212)
x47	DC(1213)
x48	DC(1240)
x49	DC(1241)
x50	DC(1246)
x51	DC(1247)
x52	DC(1259)
x53	DC(1261)
x54	DC(1265)
x55	DC(1266)
x56	DC(1268)
x57	DC(1269)
x58	DC(1270)
x59	DC(1271)
x60	DC(1273)
x61	DC(1274)
x62	DC(1277)
x63	DC(1278)
x64	DC(1279)
x65	DC(1280)
x66	DC(1281)
x67	DC(1282)
x68	DC(1283)
x69	DC(1284)
x70	DC(1285)
x71	DC(1286)
x72	DC(1287)
x73	DC(1288)
x74	DC(1289)
x75	DC(1290)
x76	DC(1307)
x77	DC(1313)
x78	DC(1365)
x79	DC(1367)
x80	DC(1380)

x81	DC(1381)
x82	DC(1390)
x83	DC(2495)
x84	DC(2506), hardware release
xs24	STEP(351), Maintenance wait step PFTB-521
xs25	STEP(352), Process wait step PFTB-521
xs26	STEP(353), Boiler fill-up
xs27	STEP(354), operation condition check
xs28	STEP(355), Air rinse
xs29	STEP(356), Pilot-flame ignition
xs30	STEP(357), Only pilot-burner in operation
xs31	STEP(358), Main-flame ignition
xs32	STEP(359), Heat-up
xs33	STEP(360), Natural gas and vent gas burning
xs34	STEP(361), Liquid burning
xs35	STEP(362), Containment air burning
xs36	STEP(363), PFTB-521 Shutdown

Appendix D

Computation of Level of Variables in Combinational Logic Systems

The following program written in `Modula-2` compute the level of signals or variables in combinational logic systems by depth first search.

```
MODULE Levelize ;
(*=================================================================*)
(*    Purpose  : Compute level of signals in combinational circuit  *)
(*    File     : Levelize.mod                                       *)
(*    Language : Modula 2                                           *)
(*    Author   : Taeshin Park                                       *)
(*    Date     : March 20, 1997                                     *)
(*=================================================================*)
(*    Input File Format for circuit with N output signals:
      N
      OutputNodeLabel_1
      ...
      OutputNodeLabel_N
      NoChild ParentNodeLabel LabelofChildNode1 LabelofChildNode2 ...
      ...
      0
      Example :
      2
      12
      13
      2 7 2 3
      2 8 4 5
      1 9  6
      2 10  7 8
      2 11 8  9
      2 12 1  10
      2  13  9  11
      0
      Note : There should be no blank spaces at the end of each line.
  =================================================================*)
```

```
FROM Storage        IMPORT ALLOCATE,
                           DEALLOCATE ;

FROM InOut          IMPORT WriteCard,
                           WriteString,
                           WriteLn,
                           ReadCard,
                           OpenInput,
                           CloseInput,
                           OpenOutput,
                           CloseOutput;

CONST
  MaxOutputs  = 50 ;
  MaxEqns     = 5000 ;
  MaxChildren = 50 ;

TYPE
  PointerToNodeList = POINTER TO NodeList ;
  PointerToNode = POINTER TO Node ;
  NodeList = RECORD
               Node : PointerToNode ;
               Next : PointerToNodeList ;
             END ;
  Node = RECORD
           Left       : PointerToNode ;
           Right      : PointerToNode ;
           Balance    : INTEGER[-1..1] ;
           Label      : CARDINAL ;
           Level      : CARDINAL ;
           Leveled    : BOOLEAN ; (* Level is calculated *)
           Visited    : BOOLEAN ; (* Node is visited *)
           ChildList  : PointerToNodeList ;
         END ;
  CardArray = ARRAY[1..MaxChildren] OF CARDINAL ;
  Input = RECORD
            NodeLabel : CARDINAL ; (* output variable *)
            Incident  : CardArray; (* a list of inputs *)
          END ;

PROCEDURE InitialiseNodeTree(VAR Tree : PointerToNode) ;
(* Initialises a new tree of Nodes. *)
  BEGIN
    Tree := NIL ;
  END InitialiseNodeTree ;

PROCEDURE LocateNodeTreeEntry(VAR Tree  : PointerToNode ;
                              NewLabel  : CARDINAL ;
                              Create    : BOOLEAN     ;
                              VAR Found : BOOLEAN     ) : PointerToNode ;
(* LocateNodeTreeEntry searches the binary tree Tree for an entry
   of NewLabel. Found will return TRUE is this search is successful.
   If the search is not succuessful and Create is TRUE, a new entry
   will be made. *)
  VAR
    HeightIncrease : BOOLEAN    ;
    P1, P2, Node   : PointerToNode ;

  PROCEDURE LocateEntry(VAR Tree : PointerToNode) : PointerToNode ;
    BEGIN
      IF Tree = NIL THEN
    IF Create THEN
      (* Add new entry to the tree. *)
      NEW(Tree) ;
      HeightIncrease := TRUE  ;
      WITH Tree^ DO
        Left := NIL ;
        Right := NIL ;
        Label := NewLabel ;
        Level := 0 ;
        Leveled := FALSE ;
        Visited := FALSE ;
        ChildList := NIL ;
        Balance := 0 ;
      END (*with*) ;
    END (*if*) ;
    Found := FALSE ;
    RETURN Tree    ;
  ELSE
  (* Continue searching for the correct position of the new node. *)
    IF NewLabel < Tree^.Label THEN
      (* The Label of the new node is less than the present node,
         search the left branch of the tree. *)
      Node := LocateEntry(Tree^.Left) ;
      IF HeightIncrease THEN
        (* left branch has grown. *)
        CASE Tree^.Balance OF
          1 : Tree^.Balance  := 0       ;
              HeightIncrease := FALSE   |
          0 : Tree^.Balance  := -1      |
         -1 : (* Rebalance tree. *)
              P1 := Tree^.Left          ;
              IF P1^.Balance = -1 THEN
                (* single LL rotation. *)
                Tree^.Left    := P1^.Right ;
                P1^.Right     := Tree      ;
                Tree^.Balance := 0         ;
                Tree          := P1        ;
              ELSE
                (* double LR rotation. *)
                P2         := P1^.Right ;
                P1^.Right  := P2^.Left  ;
                P2^.Left   := P1        ;
                Tree^.Left := P2^.Right ;
                P2^.Right  := Tree      ;
                IF P2^.Balance = -1 THEN
                  Tree^.Balance := 1    ;
                ELSE
                  Tree^.Balance := 0    ;
                END (*if*)              ;
                IF P2^.Balance = 1 THEN
                  P1^.Balance := -1     ;
                ELSE
                  P1^.Balance := 0      ;
                END (*if*)              ;
                Tree := P2              ;
              END (*if*)                ;
              Tree^.Balance  := 0       ;
              HeightIncrease := FALSE   ;
        END (*case*) ;
      END (*if*) ;
      RETURN Node ;
```

```
        ELSIF NewLabel > Tree^.Label THEN
          (* The Label of the new node is greater than the present node,
             search the right branch of the tree. *)
          Node := LocateEntry(Tree^.Right) ;
          IF HeightIncrease THEN
            (* right branch has grown *)
            CASE Tree^.Balance OF
              -1 : Tree^.Balance  := 0        ;
                   HeightIncrease := FALSE    |
               0 : Tree^.Balance  := 1        |
               1 : (* Rebalance tree. *)
                   P1 := Tree^.Right          ;
                   IF P1^.Balance = 1 THEN
                     (* single RR rotation. *)
                     Tree^.Right   := P1^.Left ;
                     P1^.Left      := Tree     ;
                     Tree^.Balance := 0        ;
                     Tree          := P1       ;
                   ELSE
                     (* double RL rotation. *)
                     P2          := P1^.Left  ;
                     P1^.Left    := P2^.Right ;
                     P2^.Right   := P1        ;
                     Tree^.Right := P2^.Left  ;
                     P2^.Left    := Tree ;
                     IF P2^.Balance = 1 THEN
                       Tree^.Balance := -1    ;
                     ELSE
                       Tree^.Balance := 0     ;
                     END (*if*)               ;
                     IF P2^.Balance = -1 THEN
                       P1^.Balance := 1       ;
                     ELSE
                       P1^.Balance := 0       ;
                     END (*if*)               ;
                     Tree := P2               ;
                   END (*if*)                 ;
                   Tree^.Balance  := 0        ;
                   HeightIncrease := FALSE    ;
            END (*case*) ;
          END (*if*) ;
          RETURN Node ;
        ELSE
          (* Congratulations, you have found the node. *)
          Found := TRUE ;
          RETURN Tree   ;
        END (*if*) ;
      END (*if*)   ;
    END LocateEntry ;

  BEGIN
    (* LocateVertexTreeEntry *)
    HeightIncrease := FALSE  ;
    RETURN LocateEntry(Tree) ;
  END LocateNodeTreeEntry ;

PROCEDURE PushChild(VAR CurrentNode : PointerToNode ;
                        Child       : PointerToNode ) ;
(* pushes Child onto the adjacency list of CurrentNode. FIFO list. *)
  VAR
    Dummy : PointerToNodeList ;
  BEGIN
    NEW(Dummy) ;
    WITH Dummy^ DO
      Node := Child ;
      Next := CurrentNode^.ChildList ;
    END (*with*) ;
    CurrentNode^.ChildList := Dummy ;
  END PushChild ;

PROCEDURE PrintChild(Node : PointerToNode) ;
(* recursively print child list information. *)
  VAR
    Dummy : PointerToNodeList ;
  BEGIN
    WITH Node^ DO
    IF NOT Visited THEN
      Visited := TRUE;
      WriteString("Node: ") ;
      IF Label > 20000 THEN (* input *)
         WriteString("u");
         WriteCard(Label-20000,0);
      ELSIF Label > 10000 THEN (* output *)
         WriteString("y");
         WriteCard(Label-10000,0);
      ELSE (* intermediate *)
         WriteString("z");
         WriteCard(Label,0);
      END;
      WriteString("  Level: ") ;
      WriteCard(Level,0) ;
      WriteString("  Child List:") ;
      Dummy := ChildList ;
      WHILE Dummy <> NIL DO
      WriteString(" ") ;
      WITH Dummy^ DO
        IF Node^.Label > 20000 THEN (* input *)
           WriteString("u");
           WriteCard(Node^.Label-20000,0);
        ELSIF Node^.Label > 10000 THEN (* output *)
           WriteString("y");
           WriteCard(Node^.Label-10000,0);
        ELSE (* intermediate *)
           WriteString("z");
           WriteCard(Node^.Label,0);
        END;
        END (*with*) ;
        Dummy := Dummy^.Next ;
      END (*while*) ;
      WriteLn ;
      Dummy := ChildList ;
      WHILE Dummy <> NIL DO
        PrintChild(Dummy^.Node) ;
        Dummy := Dummy^.Next ;
      END (*while*) ;
    END; (*if *)
    END (*with*) ;
  END PrintChild ;

PROCEDURE LevelIs(Node : PointerToNode) : CARDINAL;
  VAR
    Dummy : PointerToNodeList;
    Value : CARDINAL;
  BEGIN
    WITH Node^ DO
    IF Leveled THEN
      RETURN Level;
    ELSE
      Leveled := TRUE;
      IF ChildList=NIL THEN
        Level := 0;
        RETURN Level;
      ELSE
        Dummy := ChildList ;
        Value := LevelIs(Dummy^.Node);
        WHILE Dummy <> NIL DO
          IF Value<LevelIs(Dummy^.Node) THEN
             Value := LevelIs(Dummy^.Node);
          END;
        Dummy := Dummy^.Next;
        END (*while*);
        Level := Value+1;
        RETURN Level;
      END; (*if*)
    END; (*if*)
    END ; (*with*)
  END LevelIs;

PROCEDURE PrintDependency(Node : PointerToNode) ;
  VAR
    Dummy : PointerToNodeList ;
  BEGIN
    WITH Node^ DO
    IF NOT Visited THEN
      Visited := TRUE;
      IF Label > 20000 THEN (* input *)
         WriteString("u.prior(IU)$(ord(IU) EQ ");
         WriteCard(Label-20000,0);
      ELSIF Label > 10000 THEN (* output *)
         WriteString("ynr.prior(IY)$(ord(IY) EQ ");
         WriteCard(Label-10000,0);
      ELSE (* intermediate *)
         WriteString("znr.prior(IZNR)$(ord(IZNR) EQ ");
         WriteCard(Label,0);
      END;
      WriteString(") = ");
      IF Label>20000 THEN
         WriteString('0.0001');
      ELSIF Level>=100 THEN
         WriteString('0.');
         WriteCard(Level,0);
      ELSIF Level>=10 THEN
         WriteString('0.0');
         WriteCard(Level,0);
      ELSE
         WriteString('0.00');
         WriteCard(Level,0);
      END;
      WriteString(" ; "); WriteLn;
      Dummy := ChildList ;
      WHILE (Dummy <> NIL) DO
        PrintDependency(Dummy^.Node) ;
        Dummy := Dummy^.Next ;
      END (*while*) ;
    END (*if*) ;
    END (*with*) ;
  END PrintDependency ;

PROCEDURE ReadAdjacency(VAR NoEqns : CARDINAL;
                        VAR NoOutputs : CARDINAL);
  VAR
    I, NoChildren, Number : CARDINAL;
  BEGIN
    ReadCard(NoOutputs);
    FOR I := 1 TO NoOutputs DO
      ReadCard(Number);
      OutputNode[I] := Number ;
    END;
    NoEqns := 0;
    ReadCard(NoChildren);
    WHILE NoChildren <> 0 DO
      ReadCard(Number);
      INC(NoEqns);
      InputData[NoEqns].NodeLabel := Number;
      FOR I := 1 TO NoChildren DO
        ReadCard(Number);
        InputData[NoEqns].Incident[I] := Number;
      END; (* for *)
      ReadCard(NoChildren);
    END; (* while *)
  END ReadAdjacency;

VAR
  Found                            : BOOLEAN ;
  InputData                        : ARRAY[1..MaxEqns] OF Input ;
  OutputNode                       : ARRAY[1..MaxOutputs] OF CARDINAL;
  I,J,NoInputs,NoEqns,NoOutputs    : CARDINAL ;
  Nodes,CurrentNode,IncidentNode   : PointerToNode ;

BEGIN
  InitialiseNodeTree(Nodes) ;
  FOR I := 1 TO MaxEqns DO
    InputData[I].NodeLabel := 0;
    FOR J := 1 TO MaxChildren DO
      InputData[I].Incident[J] := 0;
    END;
  END;

  OpenInput("");
  OpenOutput("");

  WriteLn; WriteString("Reading inputs ... ");
  ReadAdjacency(NoEqns,NoOutputs);
  WriteString("DONE"); WriteLn;

(* read the input data, constructing the graph as we go. *)
  WriteString('Constructing graph ... ');
  NoInputs := 0 ;
  FOR I := 1 TO NoEqns DO
    (* create the entry for this node if it doesn't already exist. *)
    CurrentNode :=
      LocateNodeTreeEntry(Nodes,InputData[I].NodeLabel,TRUE,Found) ;
    (* update the adjacency list of all the incident nodes *)
    J := 1 ;
    WHILE InputData[I].Incident[J] > 0 DO
      IncidentNode :=
        LocateNodeTreeEntry(Nodes,InputData[I].Incident[J],TRUE,Found) ;
      PushChild(CurrentNode,IncidentNode);
      INC(J) ;
    END (*while*) ;
  END (*for*) ;
  WriteString('DONE'); WriteLn;

  FOR I := 1 TO NoOutputs DO
    WriteLn; WriteString('Level of Output Node : ');
    WriteCard(OutputNode[I],0);
    WriteString(' = ');
    WriteCard(LevelIs(LocateNodeTreeEntry(Nodes,OutputNode[I],
      TRUE,Found)),0);
    WriteLn;
  END;

  FOR I := 1 TO NoOutputs DO
    WriteLn; WriteString('Dependency List of Output Node : ');
    WriteCard(OutputNode[I],0); WriteLn; WriteLn;
    PrintDependency(LocateNodeTreeEntry(Nodes,OutputNode[I],TRUE,Found));
    WriteLn;
  END;
END Levelize.
```

Bibliography

Abramovici, M., Breuer, M. A., and Friedman, A. D., *Digital Systems Testing and Testable Design*, New York: Computer Science Press, 1990.

Adams, W. P., and Sherali, H. D., "A tight linearization and an algorithm for zero-one quadratic programming problems," *Management Science*, vol. 32, no. 10, pp. 1274-1290, 1986.

AIChE/CCPS, *Guidelines for Chemical Process Quantitative Risk Analysis*, New York: AIChE, 1989.

AIChE/CCPS, *Guidelines for Hazard Evaluation Procedures*, 2nd ed, New York: AIChE, 1992.

AIChE/CCPS, *Guidelines for Safe Automation of Chemical Processes*, New York: AIChE, 1993.

Allgor, R. J., Barrera, M. D., Barton, P. I., and Evans, L. B., "Optimal batch process development," *Computers chem. Engng.*, vol. 20, no. 6/7, pp. 885-896, 1994.

Also, N., Camillocci, L. S., Sanchez, A., and Macchietto, S., "Synthesis of procedural controllers - application to a batch pilot plant," *Computers chem. Engng.*, vol. 20, no. Suppl., pp. S1481-S1486, 1996.

Alur, R., Courcoubetis, C., Henzinger, T. A., and Ho, P., "Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Systems, Lecture Notes in Computer Science, vol. 736*. New York: Springer-Verlag, 1993.

André, C., and Péraldi, M. A., "GRAFCET and synchronous languages," *Automatique Productique Informatique Industrielle*, vol. 27, no. 1, pp. 95-105, 1993.

Angulo, I., and Torkzadeh., G., "State event in combined simulation," in *Proceedings of the 1988 Winter Simulation Conference*, San Diego, CA, 1988.

ANSI/ISA, *Binary Logic Diagrams for Process Operations, ANSI/ISA-S5.2-1976(R1981)*: ISA, 1981.

AspenTech, *SpeedUp User Manual Issue 5.4*: Aspen Technology, 1993.

Barton, P. I., *The Modeling and Simulation of Combined Discrete/Continuous Processes*, Ph.D. Thesis, University of London, 1992.

Barton, P. I., and Pantelides, C. C., "gPROMS - a combined discrete/continuous modeling environment for chemical processing systems," *Simulation Series*, vol. 25, no. 3, pp. 25-34, 1993.

Barton, P. I., and Pantelides, C. C., "Modeling of combined discrete/continuous processes," *AIChE Journal*, vol. 40, no. 6, pp. 966-979, 1994.

Barton, P. I., and Park, T., "Analysis and control of combined discrete/continuous systems: progress and challenges in the chemical processing industries," *AIChE Symposium Series*, 1997.

Bell, R., "Safety in chemical process automation: HSE approach," in *International Symposium and Workshop on Safe Chemical Process Automation*, AIChE/CCPS, 1994.

Bertsimas, D., *Personal Communication*, 1997.

Bertsimas, D., and Tsitsiklis, J. N., *Introduction to Linear Optimization*, Belmont: Athena Scientific, 1997.

Birta, L. G., Ören, T. I., and Kettenis, D. L., "A robust procedure for discontinuity handling in continuous system simulation," *Transactions of the Society for Computer Simulation*, vol. 2, no. 3, pp. 189-205, 1985.

Bose, S., and Fisher, A., "Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic," in *Proceedings of IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, Leuven, 1989.

Boy de la Tour, T., "Minimizing the number of clauses by renaming," in *Proceedings of 10th International Conference on Automated Deduction*, Kaiserslautern, France, 1990.

Bradbury, R. K., Rominger, M. C., and Verhulst, J. D., "Promoting modularity in batch control software," *ISA Transactions*, vol. 28, no. 3, 1989.

Brenan, K. E., Campbell, S. L., and Petzold, L. R., *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*: North-Holland, 1989.

Brglez, F., and Fujiwara, H., "A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN," in *International Symposium on Circuits and Systems*, 1985.

Brooke, A., Kendrick, D., and Meeraus, A., *GAMS Release 2.25 A User's Guide*, San Francisco: The Scientific Press, 1992.

Brüll, L. and Pallaske, U., "On differential algebraic equations with discontinuities," *Z angew Math Phys*, vol. 43, pp.319-327, 1992.

Bryant, R. E., "Graph-based algorithms for Boolean function manipulation," *IEEE Trans on Comp.*, vol. C-35, no. 8, pp. 677-691, 1986.

Bullard, L. G., and Biegler, L. T., "Iterated linear programming approaches for non-smooth simulation," *Computers chem. Engng.*, vol. 17, 1993.

Burch, J. R., Clarke, E. M., and Long, D. E., "Representing circuits more efficiently in symbolic model checking," in *Proceedings of 28th ACM/IEEE Design Automation Conference*, 1991.

Burch, J. R., Clarke, E. M., Long, D. E., McMilan, K. L., and Dill, D. L., "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401-424, 1994.

Burch, J. R., Clarke, E. M., McMillan, K. L., and Dill, D. L., "Sequential circuit verification using symbolic model checking," in *Proceedings of 27th ACM/IEEE Design Automation Conference*, 1990.

Carver, M. B., "Efficient integration over discontinuities in ordinary differential equation simulations," *Mathematics and Computers in Simulation*, vol. XX, pp. 190-196, 1978.

Cavalier, T. M., Pardalos, P. M., and Soyster, A. L., "Modeling and integer programming techniques applied to propositional calculus," *Computers Opns. Res.*, vol. 17, no. 6, pp. 561-570, 1990.

Cellier, F. E., *Combined Continuous/Discrete System Simulation by Use of Digital Computers*, Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, 1979.

Clarke, E., and Emerson, E. A., "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proceedings of Workshop on Logics of Programs, Lecture Notes in Computer Science, 131*, Springer-Verlag, 1981.

Clarke, E., Emerson, E. A., and Sistla, A. P., "Automatic verification of finite state concurrent systems using temporal logic specifications," *ACM Trans. on Programming Languages and Systems*, vol. 8, no. 2, pp. 244-263, 1986.

Clarke, E., Grumberg, O., and Long, D., "Verification tools for finite-state concurrent systems," in *A Decede of Concurrency : Reflections and Perspectives, Proceedings of REX School/ Symposium*, Noordwijkerhout, The Netherlands, 1993.

Coudert, O., Madre, J. C., and Berthet, C., "Verifying temporal properties of sequential machines without building their state diagrams," in *Proceedings of the 1990 Workshop on Computer-Aided Verification*, 1990.

David, R., "Petri Nets and GRAFCET for specification of logic controllers," in *Proceedings of IFAC 12th Triennial World Congress*, Sydney, Australia, 1993.

Devadas, S., Ghosh, A., and Keutzer, K., *Logic Synthesis*, New York: McGraw-Hill, 1994.

Dow Chemical Company, *North American Guidelines for Application of MOD5 Burner Management*, 1994.

Drake, E. M., and Thurston, C. W., "A safety evaluation framework for process hazards management in chemical facilities with PES-based controls," *Process Safety Progress*, vol. 12, no. 2, 1993.

Eichelberger, E. B., "Hazard detection in combinational and sequential switching circuits," *IBM J. Res. Dev.*, vol. 9, no. 2, 1965.

Eikaas, T. I., "CADAS - A system for computer aided design, analysis and synthesis of industrial processes," in *Proceedings of the Nordic CACE Symposium*, Lyngby, Denmark, 1990.

Elmquist, H., *A Structured Model Language for Large Continuous Systems*, Ph.D. Thesis, Lund Institute of Technology, 1978.

Emerson, E. A., "Temporal and modal logic," in *Handbook of Theoretical Computer Science.* Amsterdam: Elsevier Science Publishers, 1990.

Evans, S. F., and Wylie, P., "A plant simulator for the THORP nuclear fuel reprocessing plant at Sellafield," in *Dynamic Simulation in the Process Industries*, UMIST Manchester, IChemE, 1990.

Felder, R. M., "Simulation - a tool for optimizing batch-process production," *Chemical Engineering*, pp. 79-84, 1983.

Felder, R. M., McLeod, G. B., and Moldin, R. F., "Simulation for the capacity planning of specialty chemicals production," *Chemical Engineering Progress*, pp. 41-46, 1985.

Fisher, T. G., *Batch Control Systems*, ISA, 1989.

Frederickson, A. A., "Fault tolerant programmable controllers for safety systems," *ISA Transactions*, vol. 29, no. 2, 1990.

Frederickson, T., and Beckman, L. V., "Comparison of fault tolerant controllers used in safety applications," *ISA Transactions*, vol. 30, no. 4, 1991.

Friedman, A. D., *Fundamentals of Logic Design and Switching Theory*, Rockville: Computer Science Press, 1986.

Friedrich, M., and Perne, R., "Design and control of batch reactors - an industrial viewpoint," *Computers chem. Engng.*, vol. 19, no. Suppl., pp. S357-S368, 1995.

Fujita, M., Fujisawa, H., and Kawato, N., "Evaluation and improvements of Boolean comparison method based on binary decision diagrams," in *International Conference on Computer-Aided Design*, 1988.

Gani, R., Sørensen, E. L., and Perregaard, J., "Design and analysis of chemical processes through DYNSIM," *Ind. Eng. Chem. Res.*, vol. 31, pp. 244-254, 1992.

Gear, C. W., "Simultaneous numerical solution of differential-algebraic equations," *IEEE Transactions on Circuit Theory*, vol. CT18, pp. 89-95, 1971.

Goldfarb, S., Stephens, G., and Turner, S., "A new dynamic simulation environment for design and operational analysis," in *AIChE Annual Meeting*, St. Louis, Missouri, 1993.

Gritsis, D., Pantelides, C. C., and Sargent., R. W. H., "The dynamic simulation of transient systems described by index two differential-algebraic equations," in *Proceedings of 3rd International Symposium on Process Systems Engineering*, Sydney, Australia, 1988.

Grossman, R. L., Nerode, A., Ravn, A. P., and Rischel, H., *Hybrid Systems, Lecture Notes in Computer Science, vol.736*, New York: Springer-Verlag, 1993.

Hähnle, R., "Short CNF in finitely-valued logics," in *Proceedings of 7th International Symposium on Methodologies for Intelligent Systems*, Trondheim, Norway, 1993.

Halbwachs, N., *Synchronous Programming of Reactive Systems*, Dordrecht: Kluwer Academic Publishers, 1993.

Halbwachs, N., Pilaud, D., and Ouabdesselam, F., "Specifying, programming and verifying real-time systems using a synchronous declarative language," in *Proceedings of*

International Workshop on Automatic verification methods for finite state systems, Grenoble, France, 1989.

Hansen, E., and Sengupta., S., "Bounding solutions of systems of equations using interval analysis," *BIT*, vol. 21, pp. 203-211, 1981.

Hay, J. L., and Griffin, A. W. J., "Simulation of discontinuous dynamical systems," in *Proceedings of 9th IMACS Conference on Simulation of Systems*, Sorrento, Italy, 1979.

Holl, P., Marquardt, W., and Gilles, E. D., "DIVA - a powerful tool for dynamic process simulation," *Computers chem. Engng.*, vol. 12, 1988.

Hooker, J. N., "Generalized resolution and cutting planes," *Ann. Opns. Res.*, vol. 12, pp. 217-239, 1988.

HSE, *Programmable Electronic Systems in Safety Related Applications*, London: Health and Safety Executive, 1987.

IEC, *Programmable Controllers, IEC 1131, Part 3: Programming Language*, 1993.

IEC, *Functional safety - safety-related systems, IEC 1508 Draft Standard 65A*, 1995.

IEE, *SEMSPLC Guidelines: Safety-related Applications Software for Programmable Logic Controllers*, 1996.

IEEE, *IEEE Standard VHDL Langauge Reference Manual*, New York: The IEEE, 1987.

Jagadeesan, L. J., Puchol, C., and Olnhausen, J. E. V., "Safety property verification of ESTEREL programs and applications to telecommunications software," in *Proceedings of 7th International Conference on Computer Aided Verification*, Liège, Belgium, 1995.

Jeong, S. H., Lee, K. W., and Moon, I., "Symbolic safety analysis of automated fuel cell processes," in *1995 AIChE Annual Meeting*, Miami, FL, 1995.

Joglekar, G. S., and Reklaitis., G. V., "A simulator for batch and semi-continuous processes," *Computers chem. Engng.*, vol. 8, no. 6, pp. 315-327, 1984.

Kern, A. G., "Batch automation in a PLC: Software design is the key," *ISA Transactions*, vol. 29, no. 2, 1990.

Kestne, Y., Pnueli, A., Sifakis, J., and Yovine, S., "Integration graphs: A class of decidable hybrid systems," in *Hybrid Systems, Lecture Notes in Computer Science, vol. 736.* New York: Springer-Verlag, 1993.

Kletz, T., *Plant Design for Safety*, New York: Hemisphere Publishing Corp., 1991.

Kletz, T., *What Went Wrong: Case Histories of Process Plant Disasters*, 3rd ed, Houston: Gulf Publishing Company, 1994.

Lawley, H. G., and Kletz, T. A., "High-pressure trip systems for vessel protection," *Chem. Eng.*, 1975.

Lees, F. P., *Loss Prevention in the Process Industries*, London: Butterworth, 1980.

Lesage, J. J., and Roussel, J. M., "Hierarchical approach to GRAFCET using forcing order," *Automatique Productique Informatique Industrielle*, vol. 27, no. 1, pp. 25-38, 1993.

Lygeros, J., Godbole, D., and Sastry, S., "Simulation as a tool for hybrid system design," in *Proceedings of IEEE Conference on AI, Simulation and Planning in High-Autonomy Systems*, Gainsville, FL, 1994.

Malik, S., Wang, A. R., Brayton, R., and Sangiovanni-Vincentelli, A., "Logic verification using binary decision diagrams in a logic synthesis environment," in *International Conference on Computer-Aided Design*, 1988.

Miczo, A., *Digital Logic Testing and Simulation*, New York: Harper & Row Publishers, 1986.

Moon, I., "Modeling programmable logic controllers for logic verification," *IEEE Control Systems Magazine*, vol. 14, no. 2, pp. 53-59, 1994.

Moon, I., Powers, G. J., Burch, J. R., and Clarke, E. M., "Automatic verification of sequential control systems using temporal logic," *AIChE Journal*, vol. 38, no. 1, pp. 67-75, 1992.

Moore, R. E., "A test for existence of solution to nonlinear systems," *SIAM J. Numerical Analysis*, vol. 14, pp. 611-615, 1977.

Moore, R. E., *Methods and Applications of Interval Analysis*: SIAM, 1979.

Morton, W., *Personal Communication*, 1992.

Nerode, A., and Kohn, W., "Hybrid systems as a substrate for DSSA," in *Proceedings of IEEE/IFAC Joint Symposium on Computer-Aided Control System Design*, Tucson, AZ, 1994.

Neumaier, A., *Interval Methods for Systems of Equations*, Cambridge: Cambridge University Press, 1990.

Nemhauser, G. L., and Wolsey, L. A., *Integer and Combinatorial Optimization*, New York: John Wiley & Sons Inc., 1988.

NFPA, *Standard for Prevention of Furnace Explosions in Fuel Oil- and Natural Gas-Fired Single Burner Boiler Furnaces, NFPA 85A-1982*: NFPA, 1982.

Otter, J. D., *Programmable Logic Controllers: Operation, Interfacing and Programming*, Englewood Cliffs: Prentice-Hall, 1988.

Pantelides, C. C., "SpeedUp - Recent advances in process simulation," *Computers chem. Engng.*, vol. 12, no. 7, pp. 745-755, 1988.

Pantelides, C. C., and Barton, P. I., "Equation-oriented dynamic simulation current status and future perspectives," *Computers chem. Engng.*, vol. 17, no. Suppl., 1993.

Plaisted, D. A., and Greenbaum, S., "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, pp. 293-304, 1986.

Preston, A. J., and Berzins, M., "Algorithms for the location of discontinuities in dynamic simulation problems," *Computers chem. Engng.*, vol. 15, no. 10, pp. 701-713, 1991.

Probst, S. T., and Powers, G. J., "Automatic verification of chemical processes in the presence of failure modes," in *AIChE Annual Meeting*, San Francisco, CA, 1994.

Probst, S. T., Powers, G. J., Long, D. E., and Moon, I., "Verification of a logically controlled solids transport system using symbolic model checking," *Computers chem. Engng.*, vol. 21, no. 4, 1997.

Probst, S. T., Turk, A. L., and Powers, G. J., "Formal verification of a furnace system standard," in *AIChE Annual Meeting*, Miami, FL, 1995.

Ray, W. H., "Dynamic modeling of polymerization process flowsheets using POLYRED," in *AIChE Annual Meeting*, St. Louis, Missouri, 1993.

Rosenof, H. P., and Ghosh, A., *Batch Process Automation*, New York: Van Nostrand Reinhold Company Inc., 1987.

Rovaglio, M., T.Faravelli, Gaffuri, P., Palo, C. D., and Dorigo, A., "Controllability and operability of azeotropic heterogeneous distillation systems," *Computers chem. Engng.*, vol. 19, no. Suppl., pp. S525-S530, 1995.

Saleh, R. A., Jou, S., and Newton, A. R., *Mixed-mode Simulation and Analog Multilevel Simulation*: Kluwer Academic Publishers, 1994.

Shampine, L. F., Gladwell, I., and Brankin, R. W., "Reliable solution of special event location problems for ODEs," Numerical Analysis Report 138, Department of Mathematics, University of Manchester, 1987.

Smith, S. E., "Triple redundant fault tolerance: A hardware implemented approach," *ISA Transactions*, vol. 30, no. 4, 1991.

Tarjan, R., "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, 1972.

Thomas, D. E., and Moorby, P., *The Verilog Hardware Description Language*, Norwell, MA: Kluwer Academic Publishers, 1991.

Troyan, J. E., and Vine, R. Y. L., *Loss Prevention*, vol. 2, 1968.

Unger, S. H., "Hazard detection in combinational and sequential switching circuits," *IRE Transactions on Circuit Theory*, vol. CT6, 1959.

Victor, J., "Designing safe interlock systems," *Chemical Engineering*, 1979.

Watkins, D., *Personal Communication*, 1993.

Yamalidou, E. C., Patsidou, E. P., and Kantor, J. C., "Modeling discrete-event dynamical systems for chemical process control - a survey of several new techniques," *Computers chem. Engng.*, vol. 14, no. 3, pp. 281-299, 1990.

Zaher, J. J., and Westerberg, A. W., "Conditional modeling in an equation-based modeling environment," in *AIChE Annual Meeting*, Los Angeles, CA, 1991.