Application, Implementation and Integration of Discrete-Event Systems Control Theory

by

Michael Matthias Wood

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Master of Science (Engineering)

Queen's University
Kingston, Ontario, Canada
August 2005

Abstract

Discrete-Event Systems control theory provides automated control solutions for systems that are characterized by asynchronous and instantaneous changes of state; for example, a cluster of machines that each may be *"idle"*, *"busy"*, or *"broken"*. Goals expressed in the language of this theory permit the automatic generation of control solutions which guarantee that illegal behaviour will never occur (within the assumptions of the framework). Despite intensive research on and expansion of the theoretical aspects of this field, a limited amount of research has been reported on its implementation and integration into existing systems.

This thesis focuses on identifying classes of systems upon which the theory may be applied and isolates and examines the implementation issues that arise. Methodology for the implementation of the theory is provided and supported by concrete examples. A software suite with an emphasis on human-computer interaction was developed to facilitate the application of the theory. The work described herein constitutes some of the first steps in making the use of Discrete-Event Systems control theory accessible outside of the academic realm.

Acknowledgments

I would like to thank my supervisor, Professor Karen Rudie, for her insight and guidance. Without her, I would never have ventured down this path. To Lenko Grigorov I would like to extend my thanks for his advice and contribution throughout my research and especially in the development of the IDES software. For Miss Amanda Eddington, I am infinitely grateful; without her, I would be lost in the void. I would also like to respectfully acknowledge the financial support I received from NSERC, the Department of Electrical and Computer Engineering at Queen's University, and Professor Karen Rudie. Finally I would like to thank the members of my defense committee for their essential contribution: A. Afsahi, A. Bakhshai, H. Meijer, K.S. Novakowski and K. Rudie.

Contents

List of Tables

List of Figures

Glossary

Alphabet — A set of symbols that act as labels for **events** within a **plant**. *Page 16.*

Centralized control — Classic closed-loop control with a single supervisor and a single plant. *Page 19.*

Closed-loop — The standard connection between supervisor and plant. The plant generates events from Σ which are accepted by the supervisor which, in turn, generates disablement commands which are accepted by the plant, thus forming a closed loop. *Page 14.*

Controllable — 1. When in reference to a **language** with respect to a **plant**, it implies that there exists a **supervisor** that can achieve the **language** when acting on the **plant**. 2. When in reference to an **event** or an element of an **alphabet**, it implies that some **supervisor** can prevent the **event** from occurring. *Page 20.*

Correct — The correctness of a structure depends only on the data upon which it is founded. An automaton's correctness is dependent on the event set Σ. A plant is correct if all strings $s \in \Sigma^*$ representing "possible" behaviour are also in $L(\mathscr{G})$. A legal specification is correct if no strings that both

can occur ($s \in L(\mathscr{G})$) and represent "undesirable" behaviour are in $L(\mathscr{L})$. *Page 18.*

Decentralized control — Closed-loop control of a single plant by multiple supervisors. This often employs partial observation where not all supervisors are notified of all events. The standard theory employs disablement by disjunction (disabled if any supervisor disables), although disablement by conjunction (disabled if all supervisors disable) has also been explored. *Page 19.*

DES control theory — A generalized and unifying theoretical framework for the control of discrete-event systems developed by P.J. Ramadge and W.M. Wonham and later extended by many other researchers. *Page 7.*

Event — A spontaneous, asynchronous and instantaneous occurrence within a **plant**. *Page 6.*

Full observation — Classic closed-loop control where the supervisor is notified of all events (from Σ) occurring within the plant. *Page 19.*

Generated — The language generated by an automaton is the set of all strings that can be created starting at the automaton's initial state and obeying its transition function. *Page 16.*

Language — A collection of words or strings. *Page 16.*

Legal — The subset of a system's possible behaviour that is desirable or permitted. *Page 6.*

Marked — The language marked by an automation is the subset of the generated language where each string ends at a marked state. This is commonly used to indicate completed tasks, but could be used to arbitrarily partition the generated language. *Page 16.*

Maximally permissive supervisor — A Supervisor that when acting on a plant achieves the largest possible subset of the legal specification. *Page 21.*

Obedient component — A component that can be trusted to behave according to a predefined protocol. *Page 84.*

Partial observation — This variant considers the situation where a supervisor is not informed of all events (from Σ) generated by the plant. *Page 19.*

Plant — A model representing all behaviour of which a system is capable. *Page 6.*

Shuffle — The concurrent behaviour of two automata with disjoint alphabets *Page 18.*

State-based communication — Communication of a binary message where the state of the medium of communication is parallel to the state of the message, for example: a light that is lit for the duration of dinner time (unlit = it is not dinner time). *Page 84.*

String — A sequence of elements from an alphabet Σ. This term is interchangeable with **word**. *Page 16.*

Supervised plant — The resulting behaviour of a plant restricted by a supervisor indicated by $\mathscr{S}/\mathscr{G}$. For an implicit supervisor $\mathscr{S}$ with $L(\mathscr{S}) \subseteq L(\mathscr{G})$,

the supervised plant is equivalent to $\mathscr{S}$, otherwise it is the synchronous product $\mathscr{S}||\mathscr{G}$ *Page 14.*

Supervisor — An entity whose purpose is to restrict the behaviour of a plant such that no behaviour outside of the legal specification occurs. *Page 6.*

Supremal controllable sublanguage — The largest legal specification that both generates a sublanguage of the language generated by the desired legal specification and is found to be controllable with respect to the plant. *Page 21.*

Synchronous product — This operation is denoted by || and expresses the result of two automata functioning jointly, where events from the shared alphabet are only allowed to occur when both automata execute them simultaneously, and events not contained in both alphabets are always allowed to occur whenever they are defined in one of the automata. *Page 18.*

Transient communication — Communication of a binary message where the state of the medium of communication is not parallel to the state of the message, for example: a bell that is rung as the beginning of dinner time (not ringing $\neq$ it is not dinner time). *Page 84.*

Word — A sequence of elements from an alphabet Σ. This term is interchangeable with **string**. *Page 16.*

Chapter 1

Introduction

1.1 Opening Remarks

This dissertation is concerned with the application, implementation and integration of discrete-event systems control theory. It is necessary then to have a clear understanding of what is meant by a discrete-event system (DES). Let us first note that functional reality is limited to that which is observed; therefore, since we are able to sample any system either continuously or discretely, every system can be considered either continuous or discrete. For the purposes of this work, it is most reasonable to define discrete-event systems as those systems which are best or at least beneficially modeled as discrete-event. This, of course, begs the question of how to determine how closely a given system meets this criteria, and it is one of the issues investigated in this dissertation.

In the past century, the complexity of human-made systems has greatly increased. In recent decades the advent and progress of the digital computer and ever-increasing

communication networks have been tightly integrated with this growth. The mainstream transition from analog to digital technologies provides strong motivation for the discretization of systems. In the past, mathematics and modeling techniques have been used to interface with processes governed by the laws of nature in systems and control engineering. For many problems in the new digital environment, these techniques are less adequate [12].

There exist many approaches to the modeling of discrete-event systems; these include Boolean models [3], Petri nets [36] and real-time temporal logic [31]. In the 1980s, Ramadge and Wonham published a generalized and unifying theory for the control of discrete-event systems [37, 47, 38]. This abstract framework is based upon automata [24] and has gained considerable popularity in the academic environment. In it, a system is modeled (by a human) as an automaton and called the **plant**. The plant expresses all the behaviour that is possible. Next a set of goals is generated (by a human) as an automaton and called the **legal** specification. The legal specification expresses the subset of possible behaviour that should be allowed to occur. Then, a third automaton called the **supervisor** can be automatically generated (using software) from the plant and legal specification. The plant is sometimes termed a generator because (at runtime) it will spontaneously generate **events** (which are the output symbols of the automaton). The supervisor automaton observes the events generated by the plant, and these serve as its input. Hence a supervisor is sometimes termed an acceptor. The supervisor automaton does not generate events. At any given state, the supervisor implicitly contains disablement information (in the form of events from the automaton's alphabet that are not defined in that state). If the plant observes and obeys this disablement information, its behaviour will be restricted

such that it will never generate sequences of events that are not contained within the legal specification. The core theory therefore allows the automatic generation of a supervisor that may achieve arbitrary goals in an arbitrary system.

The abstract framework proposed by Ramadge and Wonham has been expanded to included concepts such as observability [32, 16], modular construction [48], decentralized control [33, 16, 43, 42], communication between controllers [40], dynamic models [15, 22, 23], and models that include the concept of time [7, 41]. Since there has been considerable interest and advancement of this framework it provides a reasonable alternative for any individual who wishes to implement a real system modeled as a DES. Henceforth I will use the term **DES control theory** to refer to the framework developed by Ramadge and Wonham and its various extensions. For those unfamiliar with the framework, complete definitions for its various components are given in Chapter 2.

Unfortunately, there exists a real gap between the academic and the industrial application of DES control theory. The researchers of [28] state that "despite intensive research on the theoretical aspects ... a limited amount of research has been reported on the implementation of DES-based supervisory controllers". That said, several unconnected investigations into implementation have occurred. The works of [29, 4, 13, 34, 1] are all reviewed in Chapter 3, and other relevant investigations into application of DES control theory include [6, 8, 9, 41].

The investigation into application is further motivated by the fact that DES control theory has limitations and is best suited only for a subset of systems that one might conveniently model as discrete-event. It is therefore necessary to have a

methodology first to determine if DES control theory is a good model for a given problem, and second to map the real world application into the model, and the model's solution back to the real world. Finally, in order to actually use the theory with real systems, a software suite is required to assist in all phases of design, automatic generation, testing and implementation. These concerns are the primary focus of this thesis.

This work will confirm that DES control theory is most amenable to pre-existing systems that function in a non-optimal way and admit control. It will also be confirmed that the best performance can be achieved when high-level control is required and the specification does not imply the solution. Despite the disadvantages associated with systems that do not meet these criteria, it will be shown that the theory can still be advantageously applied. Specifically, several methodologies will be provided for systems in which a portion of the plant and the entire supervisory entity exist within an implementation device such as a microcontroller. This work demonstrates how DES control theory may be applied to a wide variety of systems and how these systems should be approached.

In analysis of the currently available software tools, it will be demonstrated that a real need exists for usable and intuitive software. As a result, the Integrated Discrete-Event Systems (IDES) software was developed, which functions as an effective interface for specifying DES components in a manner analogous to pen and paper drawing, and demonstrates the integrated use of DES control theory with custom hardware components for research and pedagogical purposes. To date, other academic tools (such as CTCT [49], UMDES [27] and GIDDES [39]) have been unconcerned with integrating hardware and software for control and testing purposes. Hence, the IDES

software is unique in its attempt to provide an interface for all of modeling, testing and control. The primary goal in the development of the IDES software was to achieve an interface analogous to pen and paper drawing and to export the defined components to various graphical formats. The software solidly achieves these requirements; however, it could be made vastly more powerful. A standardized and usable conglomerate design tool is necessary for the application of DES control theory to be feasible, and IDES could serve as a starting point for such a tool.

1.2 Contributions

The contributions of this work are as follows:

- We provide a means of classifying systems and determining their suitability as candidates for DES control theory and appropriate methodologies for its application with each class of systems. Clear application methodologies are necessary for low-level, programmable systems because there is no straightforward means to apply DES control theory to such systems.
- Concrete examples with a PIC16F84 microcontroller demonstrate beginning–to–end application of the theory including automatic machine code generation. In the majority of the related work, in the literature, insufficient information is provided for the reproduction of the modeling exercise.
- We introduce (in Chapter 7) the Initiated-Event Methodology, which is a new means of approaching the application of DES control theory to low-level, programmable systems. In it, three different solutions are provided: monolithic, modular, and integrated. Suggestions for automatic code generation for all three are also provided. The integrated variant is a very different approach to the implementation of the theory and can realize data and runtime complexity advantages over the other methods.
- The IDES Software was developed. It is a robust and usable software modeling and pedagogical tool that is able to output graphs for use in research documents and interface with real systems for demonstration of DES control theory

principles. It could serve as a starting point for a standardized and usable conglomerate design tool for many facets of DES control theory.

1.3 Organization of this Document

This thesis is organized as follows. In Chapter 2, the background theory is laid out in detail, and obvious implementations of the theory are discussed. In Chapter 3, a variety of related work is examined and the relevant details are extracted. Chapter 4 introduces the problem of implementation. Following this, in Chapters 5, 6 and 7, three disparate systems are closely examined and methodologies are proposed. The analysis demonstrates that the theory is most amenable to existing systems with existing high level control and non-optimal behaviour. Complications arise when modeling systems that are themselves in the design stage. It is found that the nature of the supervisor impacts the nature of the plant and vice versa. This is found to be most striking in the definition of events. Integration of the plant and supervisor is proposed as a means of improving the efficiency of some systems.

In consideration of the exposed issues, system properties, means of classification and solution methodologies are summarized in Chapter 8. Having this established, the focus moves to the actual act of modeling and the tools available. Chapter 9 details the development and use of Integrated Discrete-Event Systems software both as a modeling and as a pedagogical tool. The tool is evaluated and a case study of its use is provided. Finally, Chapter 10 summarizes and concludes the work.

1.4 Conventions in this Document

Several layout conventions have been used in this document. Finite-state machines are represented as directed graphs consisting of circles and arrows. In these diagrams, a dashed line is used to represent an uncontrollable transition, while a solid line is used to represent a controllable transition. In the case where both a controllable event and an uncontrollable event are listed on the same arc, it is drawn as a dashed line. Also in these diagrams, the initial state is denoted by a single, short, incoming arrow, and marked states are denoted by a double line along the circumference of the circle.

When discussing events they are displayed in *"quoted italics"*. Also, in this text several terms are displayed in **bold** font, which implies that these terms are included in the glossary at the end of this document.

Chapter 2

Background Theory

2.1 An Informal Overview

In the original paper on DES control theory [37], Ramadge and Wonham state

> In this paper we study the control of a class of systems broadly known as discrete-event processes. The principal features of such processes are that they are discrete, asynchronous and (possibly) nondeterministic. Typical instances include computer networks, flexible manufacturing systems, and the start-up and shut-down procedures of industrial plants.

The framework they developed employs notation and standard ideas from automata and language theory [24]. First, they define all the possible behaviour of the system as the **plant** and model this as an automaton, which can equivalently be expressed as a directed graph. It is assumed that some part of all possible behaviour is undesirable. It is the responsibility of the human modeler to accurately define the plant and the

subset of its behaviour that is **legal**. The legal specification is also represented as an automaton.

Given a plant and a legal specification, DES control theory can automatically synthesize a **supervisor** which provides information on how to prevent illegal things from occurring. This is a core benefit of DES control theory. The entire purpose of the theory is to automatically produce a correct control solution (the supervisor) when given a correct description of the problem (the plant and legal specification). The alternative is to have a human generate a control solution for a given problem in an ad hoc manner which (considering the current and increasing complexity of real discrete-event systems) is difficult and error prone.

Once synthesized, the supervisor may also be expressed as an automaton and, in order to impose control, must have been synthesized before runtime. (Note, several branches of DES control theory have investigated dynamically generating or replacing supervisors, but that work is beyond the scope of this investigation). At runtime, both the plant and the supervisor start at their initial states. The online relationship between plant and supervisor is demonstrated in Figure 2.1.

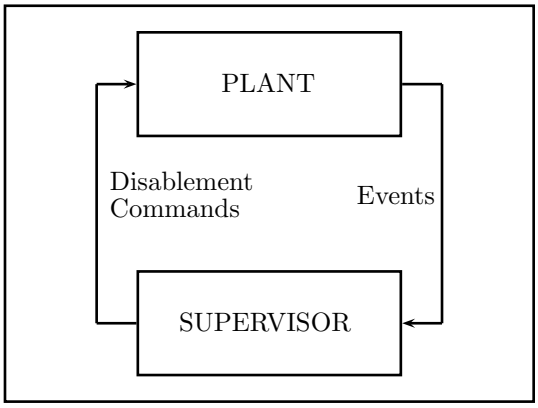

Figure 2.1: The relationship between plant and supervisor denoted by $\mathscr{S}/\mathscr{G}$.

As events occur within the plant the supervisor is notified (in an unspecified manner) and synchronously changes state. For this reason, the plant and supervisor are sometimes respectively termed generator and acceptor. At each state the supervisor communicates to the plant (in an

unspecified manner) a list of events that are disabled (i.e., should not be generated) thereby restricting the behaviour of the plant. The behaviour resulting from the imposition of this control can be expressed by another automaton known as the **supervised plant** or the **closed-loop** behaviour.

It is useful to present a simple example to see how this framework can map to the real world and how control might be implemented. The following is a summary of the example used in [30].

Consider the owner of a factory. In this example the owner is the **supervisor** and the factory, its workers, its interaction with the marketplace and with product supply and transport, etcetera are encompassed by the **plant**. One may view the plant as a discrete-event system. Example events include the purchase of new equipment, the hiring of new employees, the initiation of a new product line, the signing of a business agreement, the unexpected failure of equipment, a union strike, etcetera. Some of these things can be controlled by the supervisor and some cannot. The owner also knows of certain scenarios that are undesirable and are understood as illegal sequences of events within the plant. The problem of deciding which events to disallow at any given state in order to prevent illegal sequences of events from occurring is solved by DES control theory.

It is important to realize that DES control theory implicitly assumes that the plant exists independent of any set of goals or even the existence of a supervisor; furthermore, in this framework the supervisor has no means of initiating events. All events are generated independently by the plant. In this example, the supervisor cannot spontaneously decide that a new machine should be purchased. Instead, an employee must bring the supervisor a proposal for the purchase of new equipment,

and the supervisor must then allow or disallow the proposal.

Since the supervisor may only act by disablement, it cannot guarantee liveness. It can prevent deadlock, but since the framework does not incorporate the concept of time, and since the supervisor cannot generate events, it is quite possible for the plant to remain in the same state indefinitely.

A naive implementation of DES control theory for the factory example is obvious. The owner could hire a DES control theory professional who would model the system as a plant $\mathscr{G}$ and based on the verbal goals of the owner determine a maximally permissive implicit supervisor $\mathscr{S}$ which generates the supremal controllable sublanguage. This supervisor would be provided to the owner as a directed graph drawn on paper with an associated list describing what each transition meant. The owner would put a token on the start node of the graph, and whenever proposals were delivered, if the relevant outgoing transition was defined, the owner would grant approval and reposition the token. This implementation is unwieldy, but feasible.

Solutions produced by DES control theory are "correct by design" but depend on the initial models generated by humans. How a plant actually corresponds to the real system, how events map to actual occurrences in the plant, how controllable events and their disablement actually translate to the real system are all determined by a human and are error prone. Furthermore, the language of DES control theory does not express when and why events occur, only the order in which they occur. This can impede the realization of certain objectives.

2.2 Formal Definitions

The material in this section is adapted from [12].

2.2.1 An Automaton

- An automaton $\mathscr{A}$ is defined as a five tuple $(Q, \Sigma, \delta, q_0, Q_m)$.
- Q is the set of states in $\mathscr{A}$.
- Σ is the **alphabet** or set of output symbols of $\mathscr{A}$ and is always assumed to be finite. In the context of DES control theory, these elements correspond to logical events in some system.
- δ is the transition function of $\mathscr{A}$ and is defined from $\Sigma \times Q$ to Q. In general, δ is only a partial function, meaning that $\delta(\cdot, q)$ is only defined for some subset of Σ for any fixed $q \in Q$.
- $q_0 \in Q$ is the initial state of $\mathscr{A}$.
- $Q_m \subseteq Q$ is the set of marked states which serves only to denote some states as special in some way.
- $\mathscr{A}$ is equivalent to a directed graph with Q as the set of nodes and with edges defined by triplets in the form (*start node*, *edge*, *end node*) according to the rule $(q,\ \sigma,\ \delta(\sigma, q))$ providing $\delta(\sigma, q)$ is defined.
- Σ^* is the set of all finite **strings** s of elements of Σ. Note that s may represent a zero length string denoted by ε. Note also that strings are often called **words** and groups of strings are called **languages**.
- The transition function δ can easily be extended to $\Sigma^* \times Q \to Q$ by observing that for $s \in \Sigma^*$, $\delta(s\sigma, q)$ can be computed via $\delta(\sigma, \delta(s, q))$ down to the base case which is just the transition function from the initial state.
- The language **generated** by $\mathscr{A}$ is all possible sequences of events that can occur and is denoted by $L(\mathscr{A}) = \{w : w \in \Sigma^* \; and \; \delta(w, q_0) \; is \; defined\}$.
- The language **marked** by $\mathscr{A}$ is all possible sequences of events that end at a marked state and is denoted by $L_m(\mathscr{A}) = \{w : w \in L(\mathscr{G}) \; and \; \delta(w, q_0) \in Q_m\}$.

2.2.2 Operations

Notation

The prefix closure of a language K is denoted by an over-bar as in $\overline{K}$.

Equivalence

Two automata $\mathscr{A}_1$ and $\mathscr{A}_2$ are equivalent if they generate and mark the same languages. That is, $L(\mathscr{A}_1) = L(\mathscr{A}_2)$ and $L_m(\mathscr{A}_1) = L_m(\mathscr{A}_2)$.

Accessibility

The definition of an automaton does not prevent the existence of states and transitions that cannot be reached from the start state. The accessible component of $\mathscr{A}$, denoted by $Ac(\mathscr{A})$ is defined to be

$$Ac(\mathscr{A}) = (Q_{ac}, \Sigma, \delta_{ac}, q_0, Q_{ac,m})$$

where

$$\begin{aligned} Q_{ac} &= \{q : \exists s \in \Sigma^*, \delta(s, q_0) = q\}, \\ Q_{ac,m} &= Q_{ac} \cap Q_m, \\ \delta_{ac} &= \delta|(\Sigma \times Q_{ac}). \end{aligned}$$

Blocking

An automaton $\mathscr{A}$ is blocking if it can reach a non-marked state from which no marked state is reachable, i.e., $\mathscr{A}$ is blocking if $L(\mathscr{A}) \neq \overline{L_m(\mathscr{A})}$. Conversely, $\mathscr{A}$ is nonblocking if $L(\mathscr{A}) = \overline{L_m(\mathscr{A})}$.

Synchronous Product

This synchronous product operation is denoted by $||$ and expresses the result of the automata functioning jointly, where events from the shared alphabet are only allowed to occur when both automata execute them simultaneously, and events not contained in both alphabets are always allowed to occur whenever they are defined in one of the automata. Formally,

$$\text{if } \mathscr{A}_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, Q_{m_1}) \text{ and } \mathscr{A}_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, Q_{m_2}), \text{ then}$$
$$\mathscr{A}_1||\mathscr{A}_2 = Ac((Q_1 \times Q_2,\ \Sigma_1 \cup \Sigma_2,\ \delta,\ (q_{0_1}, q_{0_2}),\ Q_{m_1} \times Q_{m_2}))$$

where

$$\delta(\sigma, (q_1, q_2)) = \begin{cases} (\delta_1(\sigma, q_1), q_2) & \text{if } \sigma \in \Sigma_1 \backslash \Sigma_2 \text{ and } \delta_1(\sigma, q_1) \text{ is defined} \\ (q_1, \delta_2(\sigma, q_2)) & \text{if } \sigma \in \Sigma_2 \backslash \Sigma_1 \text{ and } \delta_2(\sigma, q_2) \text{ is defined} \\ (\delta_1(\sigma, q_1), \delta_2(\sigma, q_2)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2 \text{ and } \delta_1(\sigma, q_1) \text{ is defined and } \delta_2(\sigma, q_2) \text{ is defined} \\ undefined & \text{otherwise} \end{cases}$$

Note that if $\Sigma_1 = \Sigma_2$ then $\mathscr{A}_1||\mathscr{A}_2$ yields an automaton that recognizes the intersection of the languages recognized by $\mathscr{A}_1$ and $\mathscr{A}_2$. If $\Sigma_1 \cap \Sigma_2 = \emptyset$, we call the synchronous product the **shuffle**.

2.2.3 A Plant

- A plant $\mathscr{G}$ is represented by an automaton $(Q, \Sigma, \delta, q_0, Q_m)$ and indicates all the behaviour of which a system is capable.
- A plant is **correct** if the real system it represents never exhibits behaviour that

could be described as a string $s \notin L(\mathscr{G})$ were $s \in \Sigma^*$.

2.2.4 Events

- Σ is a set of events within a system, such as: *"the mouse moves from room one to room two"*, *"the bank vault door opens"*, *"a motor begins spinning in the forward direction"*, etcetera.
- Events are considered to occur spontaneously, asynchronously and instantaneously; however, they are not required to be atomic and thereby provide further abstraction. For example, the opening of a bank vault door may in fact be a series of smaller events.
- Σ is partitioned into disjoint sets $\Sigma_c \subseteq \Sigma$ and $\Sigma_{uc} \subseteq \Sigma$ to distinguish between those events which are controllable (can be prevented from occurring by a particular supervisor) and those which are uncontrollable (cannot be prevented from occurring by a particular supervisor).
- Σ may also be partitioned into disjoint sets $\Sigma_o \subseteq \Sigma$ and $\Sigma_{uo} \subseteq \Sigma$ to distinguish between those events which are observable (are reported to a particular supervisor) and those which are unobservable (are not reported to a particular supervisor).
- In a system with **decentralized control**, each supervisor may have a different partitioning of Σ regarding both controllability and observability. The following material only covers the theory necessary for **centralized control** (synthesis of a single supervisor).
- In systems with **partial observation** ($\Sigma_o \neq \Sigma$) the requirements for synthesis of a supervisor are more complicated. The following material only covers the theory necessary for **full observation** ($\Sigma_o = \Sigma$).

2.2.5 A Legal Specification

- A legal specification $\mathscr{L}$ is represented by an automaton $(Y, \Sigma, \xi, y_0, Y_m)$ and indicates behaviour that is not forbidden.

- Note that the legal specification has only Σ in common with the plant. While it is possible for elements of Y to be logically in common with elements of Q this is not guaranteed.
- A legal specification is **correct** if it satisfies the following requirement: if $s \in L(\mathscr{G})$ and s describes behaviour that is forbidden then $s \notin L(\mathscr{L})$. This definition does not preclude a legal specification from being correct even if $L(\mathscr{L}) \not\subseteq L(\mathscr{G})$.
- Note that in general a set of arbitrary goals *cannot* be modeled simply by deleting states from $\mathscr{G}$.
- Not every $\mathscr{L}$ can be achieved via control. Specifically, if $\mathscr{L}$ can generate a string s, and $\mathscr{G}$ can generate a string $s\sigma$ where $\sigma \notin \Sigma_c$ and $\mathscr{L}$ cannot generate $s\sigma$, then $\mathscr{L}$ is clearly not achievable via control because $s\sigma$ cannot be prevented from occurring once s (a legal sequence) has occurred.
- $\mathscr{L}$ is said to be **controllable** with respect to $\mathscr{G}$ if $(\forall s \in L(\mathscr{L}))(\forall \sigma \in \Sigma_{uc}) s\sigma \in L(\mathscr{G}) \Rightarrow s\sigma \in L(\mathscr{L})$.

2.2.6 An Implicit Supervisor

- An implicit supervisor $\mathscr{S}$ is represented by an automaton $(X, \Sigma, \zeta, x_0, X_m)$ and when functioning in conjunction with the plant guarantees safe behaviour.
- Note that the supervisor has only Σ in common with the plant. While it is possible for elements of X to be logically in common with elements of Q this is not guaranteed.
- Every $x \in X$ is associated with an implicit control pattern that contains binary values for every $\sigma \in \Sigma$ according to the following rule. If $\zeta(\sigma, x)$ is defined then it is assigned a control value of 1 indicating that the event should be enabled (allowed to occur), otherwise it is assigned a control value of 0 indicating that the event should be disabled (not allowed to occur). Note that for all $\sigma \in \Sigma_{uc}$ the control pattern value must be 1 and hence $\zeta(\sigma, x)$ must be defined.
- $\mathscr{S}/\mathscr{G}$ denotes the supervised plant as described by the closed-loop behaviour of $\mathscr{S}$ and $\mathscr{G}$ with $\mathscr{S}$ accepting the event sequences of $\mathscr{G}$ and $\mathscr{G}$ restricted by the implicit control patterns of $\mathscr{S}$. This relationship is depicted in Figure 2.1. The structure $\mathscr{S}/\mathscr{G}$ itself can be described as a single automaton. By definition, $\mathscr{S}/\mathscr{G} = \mathscr{S}||\mathscr{G}$ when $\mathscr{S}$ and $\mathscr{G}$ share the same alphabet Σ.

- It is well known that if $\mathscr{L}$ is controllable with respect to $\mathscr{G}$ and $L(\mathscr{L}) \subseteq L(\mathscr{G})$ then $L(\mathscr{L}/\mathscr{G}) = L(\mathscr{L})$, which implies that $\mathscr{L}$ is an implicit supervisor which achieves itself.
- In the case where $\mathscr{L}$ is not controllable with respect to $\mathscr{G}$, it is desirable to determine a variant of it which generates the largest sublanguage of $L(\mathscr{L})$ that is controllable with respect to $\mathscr{G}$. This is termed the **supremal controllable sublanguage** (the largest legal specification that both generates a sublanguage of the language generated by the desired legal specification and is found to be controllable with respect to the plant) and is achieved by the **maximally permissive supervisor** (a supervisor that when acting on a plant achieves the largest possible subset of the legal specification).
- Algorithms and formulas for the computation of the maximally permissive supervisor are given in [47], [10] and [12].

2.2.7 An Explicit Supervisor

- An explicit supervisor $\mathscr{S}$ is a pair (S, Φ)
- S is represented by an automaton $(X, \Sigma, \zeta, x_0, X_m)$.
- Φ is a complete function that maps supervisor states to control patterns. For any $x \in X$, $\Phi(x)$ returns a control pattern that contains binary values for each $\sigma \in \Sigma$. A 1 indicates that the event should be enabled (allowed to occur), and a 0 indicates that the event should be disabled (not allowed to occur). Note that for all $\sigma \in \Sigma_{uc}$ the control pattern values must be 1.
- $\mathscr{S}/\mathscr{G}$ is the supervised plant as describe by the closed-loop behaviour of $\mathscr{S}$ and $\mathscr{G}$ with $\mathscr{S}$ accepting the event sequences of $\mathscr{G}$ and $\mathscr{G}$ restricted by Φ as shown in Figure 2.1.
- Explicit supervisors were used in the original theory of [37].

2.2.8 Reduced State Models

Once a supervisor has been obtained, it is reasonable to wish to reduce its complexity if at all possible. In general it is assumed that a supervisor with a smaller state

space that achieves the same control objective is more desirable. Since a supervisor acts in conjunction with events generated by the plant, much of the structure of the supervisor may be redundant. The state space can be reduced by allowing sequences that (although illegal) can never actually occur in $\mathscr{G}$. Reduction methods are detailed in [37] and [45].

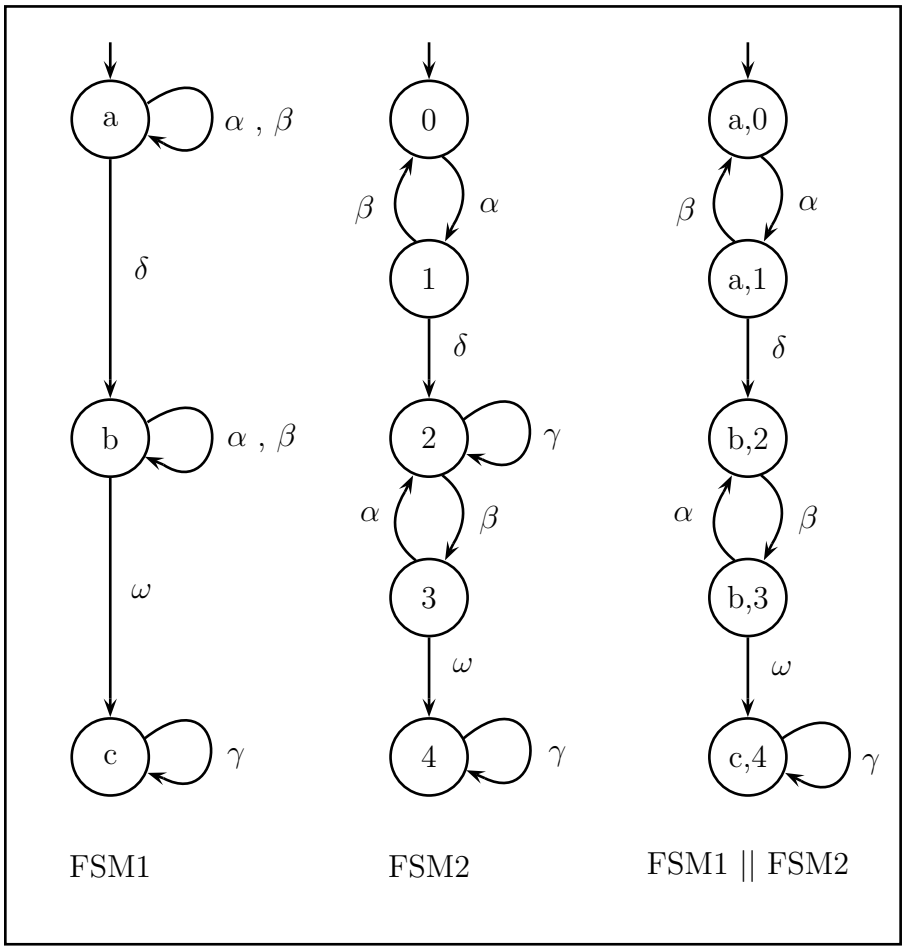

Figure 2.2: Before reduction: FSM1, FSM2, and FSM1 || FSM2

A reduction method can be used to decrease the state size of legal models. In a large system with several plant and legal models specified in a modular way, reduction

to minimal-state models can greatly impact the computation time for subsequent processes. A legal module need only communicate that certain sequences should never occur. Often this is achieved by eliminating transitions from a plant module. Such a legal module likely contains considerable redundant information about the plant structure.

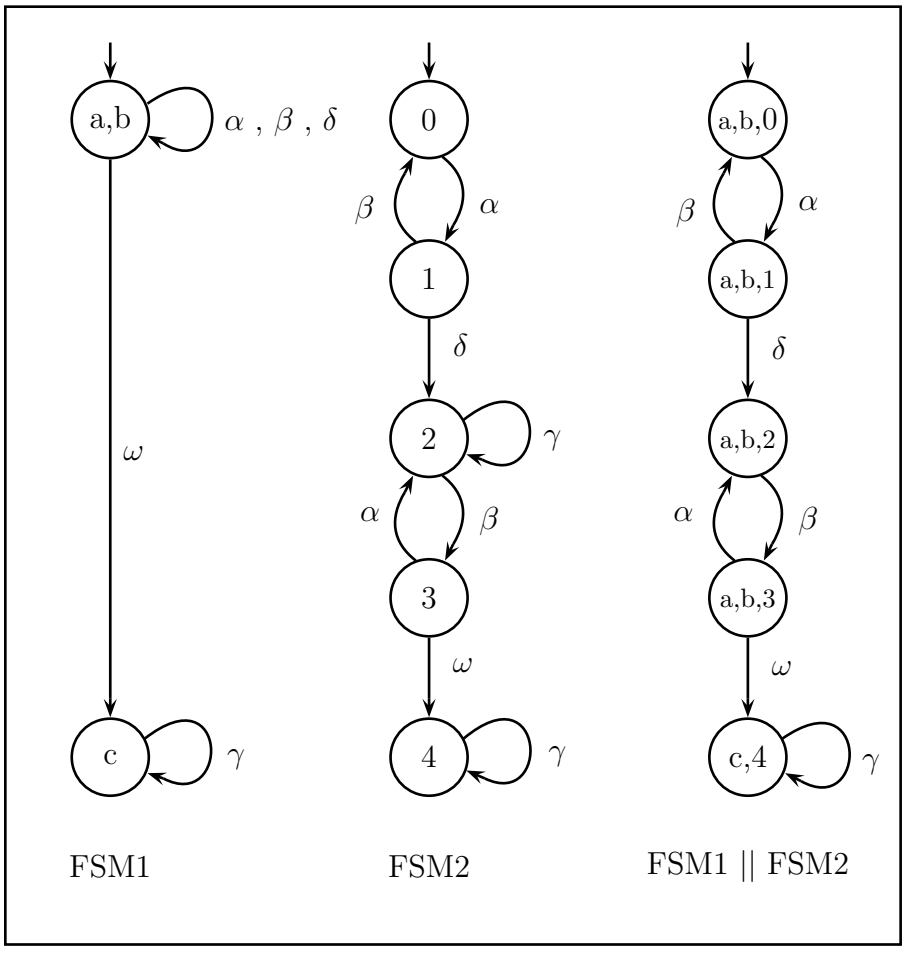

Figure 2.3: Valid Reduction: FSM1(reduced), FSM2, and FSM1(reduced) || FSM2

The synchronous product of a plant and supervisor describes the controlled plant. Similarly, the synchronous product of a plant and a legal module describes all behaviour deemed permissible by the legal module. For a given plant, many legal finite-state machines (FSMs) may achieve the same synchronous product result. Simple reduction can be achieved by merging all adjacent states in the legal FSM that can be merged without changing the synchronous product result. For small models, this can be achieved by inspection utilizing the following rules.

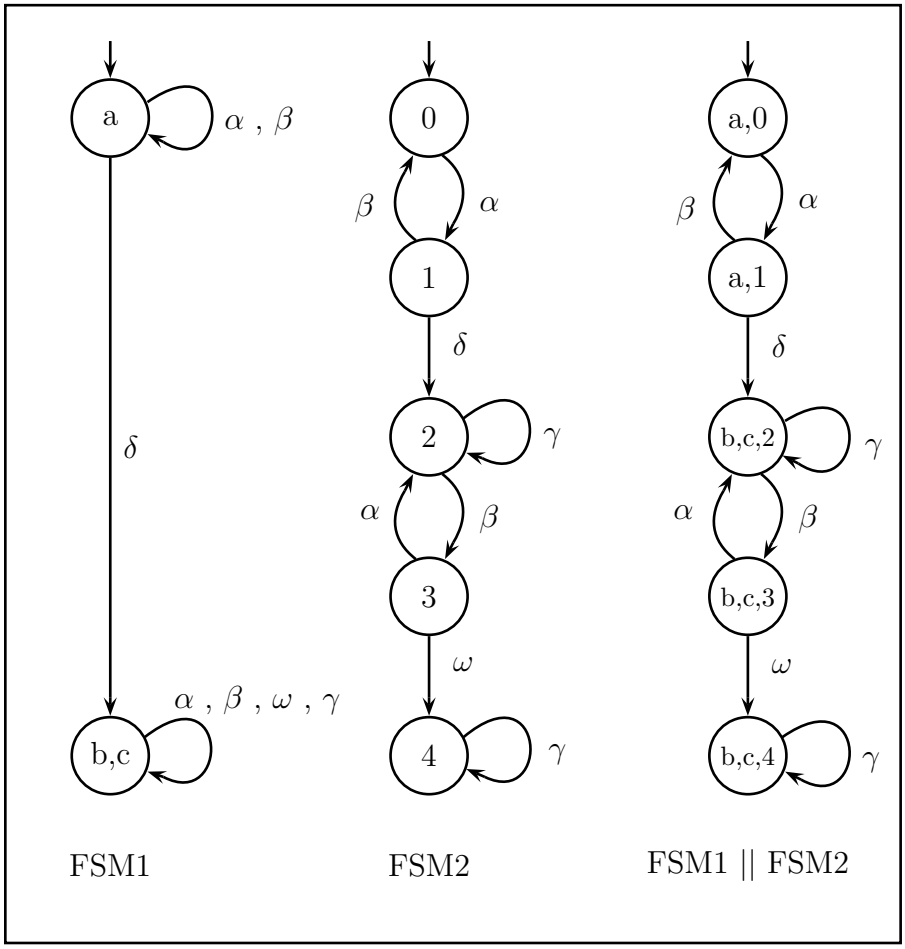

Figure 2.4: Invalid Reduction: FSM1(reduced), FSM2, and FSM1(reduced) || FSM2

1. Each state in the result of a synchronous product between $FSM1$ and $FSM2$ may be labeled (X, Y) where X was the current state of $FSM1$ and Y was the current state of $FSM2$; therefore, a given state X from $FSM1$ corresponds to a set of states X_s in $FSM2$. For example, state a in $FSM1$ of Figure 2.2 corresponds to the set of states $\{0, 1\}$ in $FSM2$ of Figure 2.2, as is visible by inspection of $FSM1||FSM2$, which is provided in Figure 2.2.

2. The only effect of a merger of two states X and Y in $FSM1$ on its synchronous product is that transitions effectively originate from states X or Y in the new (X, Y) from which they did not previously originate. Call the set of transitions for which X (respectively, Y) was not previously an originator X_t (respectively, Y_t). For example, in the merger of states a and b in $FSM1$ (shown in Figure 2.3), b effectively becomes an originator of δ whereas previous to the merger it was not; consequently, in this example $a_t = \{\omega\}$ and $b_t = \{\delta\}$. If we consider the merger of states b and c, b_t would contain only γ and c_t would equal $\{\alpha, \beta, \omega\}$.

3. If no events from X_t originate from any states in X_s in $FSM2$ and no events from Y_t originate from any states in Y_s in $FSM2$ then the merger of X and Y in $FSM1$ cannot affect the synchronous product of $FSM1$ and $FSM2$. For example, in the merger of states a and b in $FSM1$ (shown in Figure 2.3), $a_t = \{\omega\}$ and $b_t = \{\delta\}$. Checking against $b_s = \{2, 3\}$ in Figure 2.3, it is clear that neither states 2 nor 3 in $FSM2$ are originators of δ (and similarly, neither states 0 nor 1 are originators of ω) and therefore the merger is safe. In contrast, if we consider the merger of b and c, $b_t = \{\gamma\}$ and $c_t = \{\alpha, \beta, \omega\}$. Since $b_s = \{2, 3\}$ and 2 is an originator of γ the merger is not safe. Specifically, the merger of b and c in $FSM1$ (shown in Figure 2.4) would add γ in self-loop

at state $(b, 2)$, thereby altering the synchronous product.

4. In plain language, two states can be merged in one model, if the transitions added by the merger cannot occur in the corresponding states in the other model. Finally, note that a minor layer of complexity must be added to this algorithm when considering marked states.

2.3 Implications of Primary Assumptions

The primary assumptions discussed below are more suitable for certain types of problems. An overview of the assumptions is given here and system classification and solutions to the difficulties raised by the assumptions are given in later chapters. Suffice it to say that integration of the plant and supervisor in the implementation, and some modifications to the standard modeling methodology ease the difficulties exposed below.

The Plant Exists

It is a fundamental assumption of the framework that the plant exists independent of supervision and control objectives. This is appropriate for certain problems, such as the classical cat and mouse maze of [47] (overviewed in Section 2.4); however, it is less appropriate for other systems, such as a vending machine (which is probably the most classical example of a finite-state machine). Some researchers (such as the author of [29]) would describe these misfit systems as programmable plants, but the term is misleading. Anything that can be achieved by programming a digital computer can also be achieved by manufacturing a fixed electronic machine. The

important property is that some portion of expected behaviour does not yet exist.

Events are Generated by the Plant

In the framework, the plant serves as the sole generator of events. Again this choice is appropriate for systems such as the cat and mouse maze but is less amenable to systems such as a vending machine. Analysis of this assumption exposes the fact that certain systems simply don't exist without the influence of control. A vending machine—whether driven by a programmable microcontroller or simply realized as an asynchronous mechanical device—does not exist without control objectives. In the cat and mouse system, entities interact independent of control objectives. In a vending machine system, one of the necessary entities (the machine behaviour) is inexorably coupled to the control objectives. In problems such as these, it is sometimes difficult to avoid the situation where the specification is the solution.

Events Occur Spontaneously

Again, in a system such as the cat and mouse maze, the concept of spontaneous event generation is appropriate. Unfortunately, for a system such a vending machine this is problematic. Because some events are necessarily coupled to control objectives, it is unreasonable to ignore causality. Consider first designing a pop machine to randomly generate *"dispense pop"* events and second coupling it with a supervisor in closed loop. Not only is such a system needlessly complex, it makes certain goals (such as "dispense the appropriate brand of pop shortly after it is requested when payment has been received") inexpressible due to the random generation of events.

Events Occur Asynchronously

This assumption is not as problematic as the others but could raise timing issues. Many implementations of DES control theory utilize synchronous machines such as microcontrollers. It is quite possible for two events to occur between successive clock cycles. For practical purposes, the supervisor must interpret these as simultaneous. Furthermore, based on arbitrary implementation, the supervisor will consistently be notified of the occurrence of one event before the other. A rigorous programmer might implement a notification algorithm that randomizes the order of notification, but these concerns are probably unnecessary. As long as it is acceptable to interpret arbitrarily closely spaced consecutive events in an arbitrary order (as is often the case), then the assumption of asynchronism is acceptable.

Events Occur Instantaneously

In fact, events do not occur instantaneously. This can often be ignored by appropriately defining the events. Consider the event *"the bank vault door closes"*. In reality, this may entail a complex series of events over an arbitrary amount of time. It is acceptable to model this occurrence as a single instantaneous event provided its generation (and more importantly notification to the supervisory entity) occurs at a time when the real system correlates to the logical conclusions implied by it in the model. For example, if the purpose of the vault door is to prevent humans from passing through it, the event can safely be generated at a time when humans can no longer pass through it (which is the logical implication of the event in the plant model). This probably correlates to a point early in the locking process of the door. If, however, the purpose of the vault door is to contain an explosion of nerve gas, the

event must not be generated until the door is closed to a point of gas impermeability and sturdy resistance to explosive force.

Events Are Abstract

This assumption is closely related to the assumption of instantaneity. Events will often represent a sequence of smaller events or the occurrence of several conditions within the system. For example, it may be unnecessarily complex to model a timer within a system. Instead a single alarm event may be generated at some fixed interval after some condition has been met within the plant. This approach defers complexity from the DES portion of the solution to the nature of the plant itself. While this approach has many benefits, it can lead to complications as discussed in Chapter 5.

Control is Imposed by Disablement

This assumption ties in with plant existence and spontaneous event generation. As with several of the other assumptions, it is appropriate for problems such as the cat and mouse maze; however, when the system is tightly coupled to the control objectives it is more difficult to apply the framework. Since it is unreasonable to imagine a pop machine that randomly dispenses pop, it is clear that some entity must generate or force such an event based on some control logic.

2.4 Overview of Illustrative Systems

Cat And Mouse

This classical, theoretical system introduced in [47] is ideal for straightforward application and implementation of DES control theory. In this system, there exist a cat and a mouse in a maze. The doors connecting the various areas of the maze may be uni-directional, bi-directional, usable only by the cat or mouse, and controllable (meaning a supervisor could automatically force them shut). A representation of the classical maze is given in Figure 2.5. It is a screen-capture from software (by this author) for the specific purpose of using DES to automatically solve arbitrary maze problems. The unrestricted behaviour of the cat shuffled with the unrestricted behaviour of the mouse represents the plant. Control objectives may include mutual exclusion (so that the cat doesn't eat the mouse) and non-blocking (so that each animal can always return to its home room). To ask a human designer to achieve these goals in a maximally permissive manner (allowing the animals as much freedom as possible without violating the control objectives) may prove difficult, especially in an arbitrary maze of fifty rooms.

Discrete-event systems control theory can automatically solve this problem. The modeling of the plant and legal specification is very straightforward for this system. A software toolkit can automatically generate the maximally permissive supervisor. Aside from the problem of non-instantaneous events (for example if the mouse decides to sleep in a door way) the implementation of the solution is straightforward. One could simply monitor the supervisor and in each state, open all the doors that correspond to enabled transitions and close all the doors that correspond to disabled transitions.

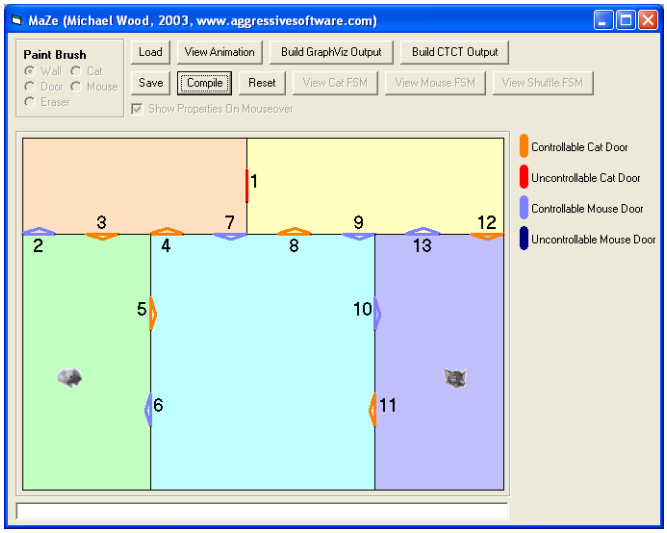

Figure 2.5: A representation of the classical cat and mouse maze.

The cat and mouse maze is a good problem because it requires high level control. It is characterized by an existing system containing independent entities that can function independent of each other's existence and independent of control objectives.

Similar systems include unordered processing systems such as lines at bank tellers. Each teller may have varying abilities to serve various customers' needs. By adding a higher level of control where client needs are observed and they are assigned to respective lines, efficiency can be increased. These systems represent uncoordinated, correct, non-optimal behaviour. In these systems some of the behaviour of the various entities can be restricted (controllable events) and some cannot (uncontrollable events).

Vending Machine

This classical example of a finite-state machine is less amenable to DES control theory and is analyzed in detail in Chapter 7. Consider a vending machine that accepts only

one type of token and dispenses only one type of pop when a request button has been pressed and the sufficient number of tokens have been inserted into the machine. The straightforward application of DES control theory would be to first build the vending machine modeled as the plant, and second attach it in closed loop to a supervisory entity. Because a vending machine is closely coupled to control objectives, this approach doesn't work. It would be bizarre to design a vending machine that generated output such as dispensed product and indicator light values at random. There is a clear need to integrate the plant and supervisor.

A vending machine system has three components, the machine itself, the humans that use the machine and the humans that administer, refill and repair the machine. From a control solution perspective the machine can be seen as a set of uncontrollable inputs (such as *"button pressed"* or *"token inserted"*) and controllable outputs (such as *"pop dispensed"* or *"insufficient funds message displayed"*). This separation is not found in good problems such as the cat and mouse maze.

The vending machine is a bad problem because it requires low-level control. It is characterized by entities (such as sensors and actuators) that have no purpose and do not function without control objectives. This coupling requires integration of plant and supervisor in order to produce a reasonable control solution.

Chapter 3

Related Work

3.1 Desco

The researchers of [21] described Desco (a Discrete-Event Systems Controller) as a software suite developed at the Control and Automation Laboratory at Chalmers University of Technology. It provides a mathematical manipulation engine and a graphical user interface. With this software, one is able to model various discrete-event systems as both state automata and Petri nets. The tool supports several functions in standard DES control theory, and it can be used to generate a maximally permissive supervisor for a given system. It is further able to execute the supervisor on a system that supports its methods of communication.

Methodology

Desco has been used to control a number of industrial systems as described in [1] and [34]. It communicates with a given plant through the Manufacturing Message Specification [26] which is an internationally standardized messaging system for exchanging

real-time data and supervisory control information between entities in a manner that is independent of the application function being performed and the developer of the device or application. In the case of [1], a commercial batch control system notified Desco of system events and periodically requested a list of available resources. This provides a perfect analogue to enabled events in standard DES control theory. In the case of [34], the programmable logic controller (PLC) of a group of spot-welding robots notified Desco of system events and periodically requested a single resource to which Desco replied with "yes" or "no". While this doesn't exactly parallel the structure of DES control theory, the translation is obvious.

Application Issues

The examples just described are "good problems". As in the cat and mouse maze, there exist discrete entities which permit control by disablement. Considering their existence without control, it is easy to see that the single-state supervisor that always returns all resources as available or always responds "yes" permits the systems to function independent of control objectives. While the resultant behaviour may be disastrous, it is a different situation than a pop machine that randomly dispenses pop. The difference is causality versus the lack of it. In [1] and [34] the plants are goal-based entities that request permission to take actions. In a vending machine, when a human requests a pop (by pushing a button) and the supervisor (perhaps realized in a microcontroller) enables the event, what entity ejects the pop from the machine? This entity must be defined (and will likely be realized in the same microcontroller). Desco requires that the plant exists and conforms to the standard notification/disablement protocol.

3.2 LEGO

The researchers of [13] expressed concern that "the design of logic controllers for event-driven systems continues to rely largely on intuitive methods rather than on formal techniques". Consequently, they built an educational test-bed that simulates an automated car assembly line using LEGO and Dacta products for the purpose of demonstrating the usefulness of DES control theory in manufacturing systems.

The physical system they created transports a roof and a chassis component to a press where they are combined to represent a completed automobile. The completed component is then transported to an exit location. The system utilizes eight motors and eight sensors and is controlled by LOGO code running on a personal computer to which the system is connected.

Methodology

In modeling the system the authors of [13] took a modular approach, partitioning the system into five plant modules, although they gave no justification or methodology for their particular choice of partitions. To create their legal specification they first designed sixteen safety specifications, some of which pertained to only one plant module, while others pertained to multiple modules. Finally they defined a progress specification which expressed a necessary ordering of events to produce an automobile. The safety and progress specifications combined represent the legal specification.

In order to avoid the computational problem of state-space explosion, they employed the modular approach of [48] and were able to synthesize a maximally permissive supervisor without computing the monolithic plant and legal specification. Having computed a supervisor, it remained to generate a control solution for their

real system (a LOGO program used to control the Dacta interface box). To achieve this, they examined their supervised plant and chose a single path from the start state to a marked state. This was accomplished by selecting at most one controllable event when possible while traversing the graph of the supervised plant. They then translated the reduced supervised plant by hand into LOGO code.

Application Issues

Note the necessity of the progress specification. Since all states in their plant modules were marked, with only the safety specifications, the supervised result would be largely chaotic, and it would be very difficult and error prone for a human to make the ad hoc conversion to a control solution.

This application of DES control theory illustrates the problem of a non-existent plant. In this case, the physical model and the DES model were necessarily created simultaneously. Even if one had first snapped together the LEGO blocks and hooked up the motors and sensors to the Dacta interface box, without a LOGO program, the plant would remain an empty set. The connected blocks, motors and sensors (without control) generate nothing.

It also illustrates the problem of event granularity. At the lowest level one might model every execution cycle of the LOGO program as an event. At the highest level, one might model the complete construction of an automobile as a single controllable event. The choice of event granularity is completely arbitrary; nevertheless, it has considerable impact on the effectiveness of the theory.

This application is also a good example of the imposition of asynchronous untimed DES control theory on an inherently synchronous and timed system. While the

manufacturing plant can be abstractly viewed as an asynchronous system, the final control solution takes the form of a LOGO program which is highly synchronous and timed.

A final notable aspect of their approach was the use of modular modeling techniques in the design of the plant and legal specification for the purpose of creating several modular supervisors, rather than a single monolithic supervisor. Because this system is a relatively simple example of an application of DES control theory to a low-level system, a variant of it is examined in detail in Chapter 5 for the purpose of highlighting the issues that beset low-level control.

3.3 Prometheus

Prometheus is a model train system developed in [29] by Ryan Leduc for the purpose of investigating the issues involved in modeling and designing supervisors for large, real systems. To facilitate this investigation the author created a PLC-based manufacturing plant with a monolithic DES plant model on the order of 10^{16} states. A significant contribution of his work was the definition of several model reduction theorems to aid in handling such a large plant; nevertheless, the primary focus of the work was the investigation of the issues involved in converting a supervisor into a physical implementation on PLCs.

The testbed was designed to simulate a manufacturing work-cell and focused on expressing the problems of routing and collision. Composed primarily of model railroad components, the trains function as automated vehicles that provide and remove material to and from three interacting manufacturing units represented by cranes.

No material is actually transported or processed, but the simulated loading and unloading processes do consume real time. To add further complexity, the tracks are interconnected, requiring switching and allowing the possibility for collision. The testbed is controlled by two MC68332 microcontrollers and an Allen-Bradley PLC. The microcontrollers control the trains and cranes directly and communicate with the PLC which is responsible for the control of the overall system.

Methodology

The work offers an overview of the implementation process as three main steps: first the synthesis of the supervisors, second their translation into clocked Moore synchronous state machines (CMSSMs) and finally their implementation on PLCs as relay ladder logic (RLL) programs. As justification for the CMSSM step Leduc argues that a CMSSM can easily be implemented on most digital logic devices, thereby guaranteeing usefulness in applications that do not employ PLCs.

The work further analyzes some issues involved in modeling a real plant. The author argues that it is important to model a system as several plant modules and to avoid unnecessarily complicating the model. As evidence he notes that a large system would be difficult, error prone and time-consuming for a human to model as a monolithic plant. He considers a 300 state system to be too large.

He suggests a methodology for the modular definition of the plant. First he suggests the definition of "fundamental" models (models of the basic components ignoring how they interact in the overall plant). Second he suggests the definition of "interaction" models (models describing how the fundamental models interact). Following this approach, some modular models may still be prohibitively large and

require further ad hoc partitioning. He notes that "not every detail of the plant's behaviour is required in every model, but every detail must be in at least one model".

He also comments on the granularity of events. He argues that by encapsulating several sub-events as a single event (whenever possible) one reduces the complexity and thereby improves the quality of the model. He does note that by reducing granularity, one limits the ability to satisfy future requirements.

The author argues that DES need not handle everything and that it may be beneficial to defer complexity to other components. If something is hard to model in the DES framework, one might consider encapsulating it as a stand-alone hardware component that interfaces with the rest of the plant in an event-driven way. For example, it would be extremely inefficient to model the filling of a tank as a DES using events such as *"one unit added"* and *"one unit removed"*. For the sake of massively simplifying the DES portion of the solution, one might install a subsystem that monitors the tank and generates a single event *"marker reached"*. This approach implicitly assumes the designer's ability to change the system.

He does attempt to classify problems, noting that for plants that are fixed in their operation, the plant model simply describes their possible behaviour. For example, a machine on an assembly line may *"start"*, *"finish"*, *"break"* and *"get repaired"*. Similarly in the cat and mouse maze, each animal may move between specific rooms via specific doors. Prometheus is described as a programmable plant. Leduc claims that such programmable plants must be defined by an interface specification. This means that the event space must map to a software interface; which is to say that the plant is every program that can be written using the interface specified by the event space. He notes that this abstraction layer allows the ability to plug in and out the

high and low level components at a later date.

Application Issues

Leduc makes a good argument that there is no guarantee that any plant model will ever be accurate, and further notes that there exists an infinite number of correct plant models for any given system. This follows from the ability to continually add information about the system, further describing subsystems, until presumably one must stop at the molecular level.

It is interesting to question why it is that a model can be correct even though it lacks information. The answer is that a plant's correctness is based only on the arbitrary definition of the events Σ. Providing the real system it represents never exhibits behaviour that could be described as as string $s \notin L(\mathscr{G})$ were $s \in \Sigma^*$, then the plant is correct. While highly granular and abstract event definitions ease the task of defining the plant, they reduce the set of possible control objectives that can be expressed and/or achieved.

Leduc notes that a synthesized supervisor enforces the specification given, not necessarily the specification intended. This is the designer's responsibility and is often very difficult; nevertheless, one presumes that for complex systems all design methods are nontrivial, so this is not a problem unique to DES control theory.

The problem of time arises in the Prometheus system. If the track connection at a track junction is switched while a train is over that portion of track, then the train will be derailed. If it is required to switch the track junction, it is not acceptable to simply list the events *"train at junction"* and *"junction switched"* in order due to the unknown time between execution of the events. Consequently two train location

events are required *"train enters danger zone"* and *"train exits danger zone"*.

Forced Events

In [4], a complex modification of standard DES control theory is proposed to aid in modeling systems that are tightly coupled to control objectives. It is suggested that the plant should generate outputs (uncontrollable events) termed responses and accept inputs (controllable events) termed commands. This framework allows controllable events to be generated (forced) by the supervisor. While this modification is theoretically much more amenable to systems like Prometheus, Leduc argues that the modification is unnecessarily complicated. He claims it is more difficult to specify a system in the modified framework and notes that many existing DES control theory results would have to be modified for its use. Leduc claims "this is re-inventing the wheel. There is no need to leave current [DES control theory]".

The approach used in the Prometheus project was to divide the controllable events into disjoint sets of those generated by the plant $\Sigma_{c,p}$ and those generated by the supervisor $\Sigma_{c,s}$. As far as application of the standard theory, one ignores this distinction by assuming all events are generated by the plant. It is only in the implementation of the supervisor that the partitioning is used.

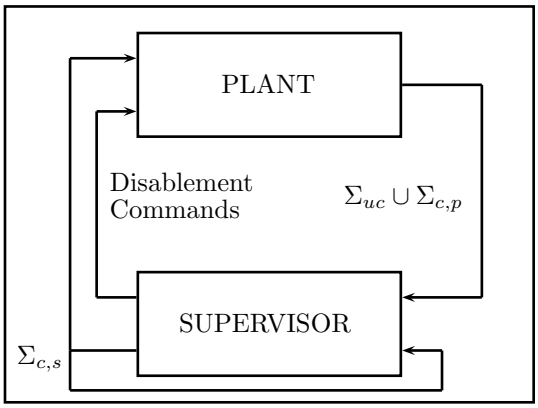

Figure 3.1: The relationship between plant and supervisor in the Prometheus project

When the supervisor reaches a state where there exists an outgoing event of the set that can be generated by the supervisor, it both instructs the plant that the event

should occur and feeds back notification of the event to itself as shown in Figure 3.1.

This approach is interesting, but like all approaches that allow both the plant and supervisor to generate events, it raises questions about ordering, priority and communication delays. Consider a system where in state one the supervisor wishes to generate an event but in state two wishes to prevent it. During the time that the supervisor communicates to the plant that the event must be forced, the plant may be communicating to the supervisor that an event that would have taken the supervisor to state two has occurred. Consequently the plant receives the forcing event in a undesirable sequence. Since this complication was not discussed in the work, one must assume that such occurrences were impossible either due to the implementation of the plant and supervisor or simply due to the nature of the system.

Chapter 4

The Problem of Implementation

When one has a system that is tightly coupled to control objectives, then the modeling of the plant, legal specification and supervisor is a more arbitrary process where each affects the other's design. First, the designer must decide upon an event space for the system. This phase involves a trade-off between simplification of the modeling process and reduced automation in the generation of a final complete control solution. This means that a complex subcomponent can be abstracted as a set of events, thereby simplifying the DES process, but leaving its implementation as ad hoc future work. Second, the designer must define a correct plant. In doing so it is usually advantageous to make the plant as inaccurate as possible while still accurate enough to express the control objectives. Correctness of a plant is based on the event space alone. As long as the real system never exhibits behaviour that could be interpreted as a string in Σ^* but not in $L(\mathscr{G})$ then the plant is correct. Abstraction in event definitions allows a less complex plant model, but this loss of accuracy can make certain control objectives inexpressible. In developing legal specifications, the models should be made as unrestrictive as possible to ensure maximum automation in supervisor synthesis.

When an acceptable supervisor is obtained it must in the end be translated into a complete control solution.

4.1 What is an Event?

If one views the system as an automaton, an event can be said to be the occurrence of $q' = \delta(\sigma, q)$. If one views the system as a formal language, an event can be said to be the generation of an element σ from the alphabet Σ. If one views the system as a directed graph, an event can be said to be a transition from a vertex to an adjacent vertex along an adjoining edge in an allowable direction. In [37] the authors state: "Events are considered to occur spontaneously (no auxiliary forcing mechanism is postulated), asynchronously (i.e., without reference to a clock) and instantaneously."

A common example of an event is *"a button is pressed"*. Consider a machine that has a button that can be depressed and released by a human. Suppose that the button is part of an electric circuit and suppose that it exhibits ideal behaviour (no debouncing issues). The button as described is an interface for a human to control a square wave within the circuit. How should this be modeled? What are the associated events? Are they uncontrollable?

At a fairly high level of detail, a "use" of the button might generate a short pulse corresponding to a press and a release: two events. The button might be logically tied to some behaviour within the machine, an action that the human requests by "using" the button. At a very abstract scale one might consider the press, release and all components of the action as a single event. Certainly the lumped event is not instantaneous, but perhaps it can be functionally considered so by assigning a particular instant of its existence as its instantaneous time of occurrence. From the

perspective of a human and a computer, a great many things may occur synchronously within the computer in a time frame that is effectively instantaneous to the human; furthermore, effective meaning of an event may allow an instantaneous marker to be applied. For example *"the user walks away"* could be keyed to the instantaneous time at which the user can be considered to be safely out of hearing range.

Events can easily be confused with system state. Consider a microcontroller with an input pin that reads zero. At some later time the input pin reads one. The rising edge may be considered the occurrence of the event, but the effective occurrence is the time when the microcontroller reads the value. This is necessarily sometime later but for certain systems may be assumed to be arbitrarily close. Alternatively the designer might intentionally decide to not sample the input value until some considerable time later (perhaps after some sequence of events). This later sampling blurs the line between event and state. This technique is useful when it is part of the design requirement to ignore an input for a period of time, this can be achieved at design time by creating a plant that simply does not sample the input during that period. Consider a keyboard, with 26 keys (one for each letter). By ignoring all other key events in between any keydown-keyup sequence of a particular key, a plant of 27 states (i.e., *"nothing active"*, *"a is active"*, *"b is active"*, ..., *"z is active"*) is achieved instead of the fully accurate plant of 2^{26}.

Another important consideration in event definition is plant existence. Many systems described in this work do not fully exist until the use of DES control theory is completed. Consequently the act of modeling the system participates in defining the system. Imagine a machine where a button (a square wave on an input pin) is related to exactly one action (some signals on output pins). The character of the

input wave is no doubt beyond the control of the machine, but it can optionally be ignored. Consider some lines of code that first test the input, then (perhaps in the case of a falling edge) generate some output. If these are to be modeled as separate observable events, lines of code must be inserted to communicate their occurrence to the supervisory entity. Alternatively they could be modeled as a single controllable event. When the sensor conditions are met, the supervisor can be asked whether or not to carry out the output actions. From a plant design point of view, it may be an equivalent amount of work to make an event observable or controllable.

What is Controllability?

The difference between a push button that delivers an uncontrollable signal to a microcontroller and, say, a push button that is mechanically connected to a container of highly concentrated acid and uncontrollably causes it to empty due to mechanical action is an important consideration. This is the difference between occurrences that can reasonably be ignored (input on a pin) and occurrences that cannot (a component fails). Uncontrollable events that may be ignored can be lumped into neighboring controllable events and seen as the initiating circumstances of the combined event.

Consider the cat and mouse system. The assumption of instantaneity is difficult to meet. One would have to go to great lengths to provide transportation capsules between the rooms (similar to air locks) for the purpose of avoiding the "lie in the doorway" problem. Surely the heart of this example relates to resource management between obedient entities. The cat is not forcefully prevented from entering a room, instead it asks for permission to enter by a shared protocol. This could be accomplished by making the supervisor's current feedback map values readable to the cat

and mouse, or by a query/response system such as that employed by DESCO [21]. The query/response strategy supports the idea that controllable events should be associated with uncontrollable initiation.

What is Observability?

Observability is a matter of taste. If a microcontroller is connected to a motor in such a way that it can cause it to begin spinning in the forward direction, is this controllable *"forward"* event also observable? Implicitly *"forward"* has occurred when the command is given, but if the motor is broken, the event is observed when it has not occurred. A sensor could be added to detect the start of forward motion, but if the sensor fails the event is not observed when it has occurred. The answers to questions on observability must fall to system assumptions, just as the promise of a correct solution rests on the assumption that the ad hoc initial models are correct.

4.2 What is a Control Solution?

In many cases the control solution is supervisor, legal specification, and part of the plant. In a vending machine, control objectives may affect the design of the plant. Consider a system that cannot prevent the insertion of tokens. In the DES control theory modeling process, the objective to not accept tokens when the machine is empty cannot be realized. The event must be made controllable, perhaps by having an uncontrollable initiating component (a sensor that observes a falling token) and a controllable output component (a path switch directing the token to the bank or exit location). The specification is part of the solution and affects the plant. The supervisor provides logic for order of behaviours, but some events may need to be generated

by the same entity that manages the supervisory logic. This is the case in many microcontroller implementations (where the plant is composed of humans interacting with components directed by a microcontroller). While the microcontroller's portion of the plant's behaviour could be programmed separately from the supervisor logic, integration of the two is a naturally desirable solution.

The problem of implementation is examined in three disparate systems in the following chapters. Each system is approached with a different methodology, and each exposes different facets of the implementation problem.

Chapter 5

Analysis of a LEGO Factory

5.1 Introduction

The researchers of [13] built a simplified assembly line using LEGO and Dacta products for the purpose of demonstrating the usefulness of DES control theory in manufacturing systems. Their implementation presented challenges in capturing (in the language of DES control theory) the physical system they had built, dealing with the problem of state-space explosion in their DES models, and translating the DES solution (a computed supervisor) into a control solution for their real system (a LOGO program used to control the Dacta interface box). The information in [13] describes some of the components used and suggests some methodologies.

The following discussion summarizes their work, discusses alternate approaches and highlights the difficulties that arise. The first section gives a straightforward review of what they actually accomplished. The second section focuses on a simplified version of a component in their system, and discusses how to implement the control

theory. While the information here parallels their result, a different naming strategy is used in order to simplify the discussion. Their methodology is for the most part not explicitly stated. Several methodologies for achieving their result are proposed and analyzed. It is demonstrated in Section 5.4.2 that without consideration of implementation, control objectives can become unachievable.

LEGO

In 1989, the educational products department of the LEGO group changed its name to Dacta and produced various electronic components that could easily be interfaced with their standard connectible block products. These components included touch sensors, light sensors, angle sensors and small motors. The components may be controlled with a special interface box which can be connected to a personal computer. Special software running on a personal computer can run programs written in a version of the LOGO programming language to control the interface box; consequently, LEGO can be used to produce custom programmable hardware. Today the Dacta product line has been replaced by the MindStorms product line which was developed in association with MIT. MindStorms provides many more physical and electronic building blocks, and replaces the interface box with a custom microcomputer. The microcomputer has fewer input/output connections than the Dacta interface box, but unlike the interface box, it functions stand-alone from the personal computer once it has been programmed.

5.2 Summary of the Assembly Line

The simplified assembly line constructed with LEGO products was intended to emulate an automobile manufacturing plant. The system was designed to input two different physical parts at two separate locations. It was then able to transport the two parts independently into a press, which was used to combine them into a single part. The completed part was then transported to a third separate exit location. Their complete model used all eight sensor inputs and all eight motor outputs of the Dacta interface box and was controlled by LOGO code running on a personal computer to which it was connected.

In translating their implementation into the language of DES control theory, they partitioned the entire system into five sections: a transporter module having twenty one states, a chassis module having thirty states, a roof module having fifteen states, a press module having twenty three states, and an unloader module having eight states. These modules imply a monolithic plant with 1,738,800 states.

Next they composed sixteen safety specifications and a single progress specification, the combination of which represented the complete legal specification. Of the safety specifications, some pertained only to a single plant module and were termed local, others pertained to multiple modules and were termed global. Since the prefixes of safe operations must themselves be safe, the safety specifications were required to be prefix closed, and consequently, all states in the safety specifications were marked. Since they chose to also mark all states in their plant modules, the progress specification was required to communicate the purpose of the plant. Combined, these modular specifications imply a monolithic legal specification with 11,206,656 states.

To combat the problem of state-space explosion, they employed the modular methods presented in [48]. Their implementation is a good example of the effective use of this theory, and it is the focus of their paper. Once they had computed a supervisor, it remained to translate it into a control solution for their real system (a LOGO program used to control the Dacta interface box). To achieve this, they examined their supervised plant (in which only the initial state was marked) and chose a single path from the initial state to the marked state. This was accomplished by selecting at most one controllable event when possible while traversing the graph of the supervised plant. They then translated the reduced supervised plant by hand into LOGO code.

According to [13], the steps necessary to develop a control solution using DES control theory are as follows.

1. Construction of FSM models of the system to be controlled.
2. Construction of FSM models of the safety and progress control specifications of the system.
3. Use of supervisory control theory to obtain the maximally permissive supervisor for the system.
4. Extraction of a controller from the supervisor, which permits at most one controllable event to be enabled at each state.
5. Translation of the controller into control code or PLC.

The following discussion focuses on methodologies for actually generating a plant and legal specification. It examines the idea of a progress specification, and considers the implications of translating the supervisor into a control solution. While this discussion parallels their result, it considers only a single simplified version of

their transporter module. This avoids the difficulties of state-space explosion and is sufficient to expose the relevant design issues.

5.3 A Physical Model

An abstracted model of a simplified transporter component is given in Figure 5.1. Its purpose is to transport a payload through a linear distance. The model contains two devices: a motor which generates output and a sensor which generates input. The motor employs a rack and pinion to transport a payload backwards or forwards in a single linear direction. The sensor is used to determine when the payload passes certain points in certain directions. In reality this is accomplished by monitoring the rotation of the motor shaft. For the sake of discussion, physical limitations and failures, such as the length of the rack, possible obstructions of the payload, slipping of the gear and miscounting of motor shaft rotations shall be temporarily ignored. With these assumptions, the payload can be transported infinitely along the x-axis, and the sensor always accurately notes its passing of the marked points. The payload can be modeled as a single point because any modification of its size can be accommodated by adjusting the locations of the sensor's firing points.

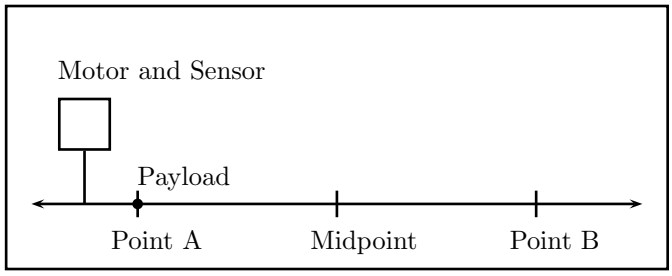

Figure 5.1: An abstract view of a transporter with a payload at point A.

In the following discussion, the task required of the transporter is to move the payload from point A to point B and then return to its initial position (which is the same a moving an empty payload from point B to point A). The midpoint is

remarked for the purpose of providing reusability. Although it is irrelevant to this discussion, some future user may have a different goal, such as moving from A to B to the midpoint, back to B and finally back to A. Because it so often occurs in reality, it is a useful constraint that the plant should be capable of behaviour other than that which is required to realize the current task.

5.4 Attempt and Complication

5.4.1 The Plant And Its Events

Since the motor is an output device, its associated events (*"forward"*, *"stop"*, *"reverse"*) are controllable. These events are instantaneous (as all events must be) and signify a change in the motor's desired behaviour. For example *"forward"* corresponds to the interface box instructing the motor to start spinning in the forward direction, and implies that it will continue to spin in the forward direction until instructed otherwise. Uncontrollable events such as *"the motor burns out"* are not modeled.

Similarly, since the sensor is an input device, its associated events (*"reachedA"*, *"reachedB"*, *"reachedMid"*) are uncontrollable. The definition of these events implies constraints on the implementation of this DES model. Specifically, this implies that some entity must exist which periodically monitors the sensor (which simply counts rotations) and notifies the central controlling entity when any of the logical events (*"reachedA"*, *"reachedB"*, *"reachedMid"*) occur. In this case the implied entity will likely take the form of a subroutine that is called every iteration of the main loop in the LOGO code running on the personal computer connected to the LEGO model.

A DES model of the simplified transporter is given in Figure 5.2. This model is

a relabeled version of a model discussed in [13]. When it is presented in [13] it is not immediately clear that an event such as *"reachedA"* may have any more specific meaning than can be implied from its name.

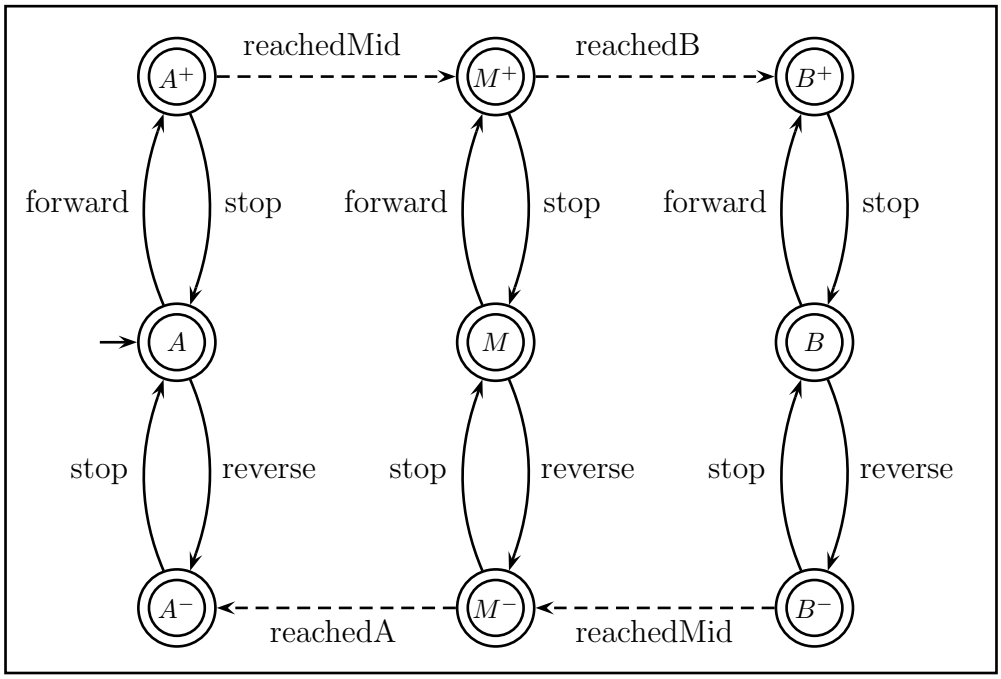

Figure 5.2: The plant model used to represent the transporter. In this figure, a dashed line is used to represent an uncontrollable transition, while a solid line is used to represent a controllable transition. The initial state is denoted by a single, short, incoming arrow, and marked states are denoted by a double circle.

It is assumed that initially the payload is at point A and the motor is stopped. In light of the event definitions, meaning can be derived from the current state of the plant. For example, state A^- indicates that the motor is running in the reverse direction and the payload is at or left of point A, while state A can only guarantee that the motor is stopped and the payload is somewhere left of the midpoint.

The purpose of the transporter is to move a payload from point A to point B and

then back to point A. It is important to note that this movement has physical implications; specifically, the LEGO model could be damaged due to maximally extending the rack or colliding with other components if the motor is run in the forward direction past point B or run in the reverse direction past point A. These possibilities must be described in a safety specification. A safety specification in conjunction with a progress specification can form a legal specification for the system. The specifications given later in this document are adaptations of those discussed in [13].

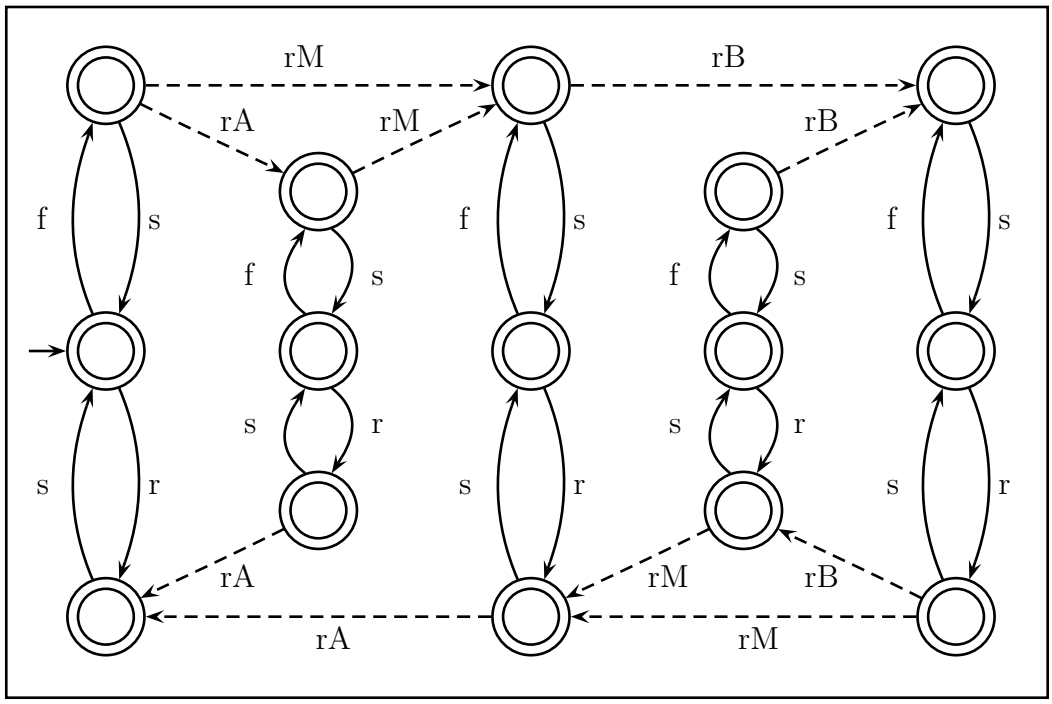

Figure 5.3: The plant model for the transporter that expresses passing beyond points A and B. Event labels have been shortened for readability with f, s, r corresponding to the motor events and rA, rM, rB corresponding to the sensor events.

The plant model of Figure 5.2 should have been constructed by considering all possible sequences of events. Consider the uncontrolled behaviour of the plant. When not constrained by a safety specification it is quite possible for the payload to extend

past point B, regardless of any catastrophic effect that may occur. Assume that the payload does extend past point B some distance, and then is reversed back to point A without suffering any harm. A possible string achieving this scenario is *"forward"*, *"reachedMid"*, *"reachedB"*, *"stop"*, *"reverse"*, *"reachedB"*, *"reachedMid"*, *"reachedA"*. This sequence cannot be generated by the plant model of Figure 5.2, but is represented in an alternate plant model given in Figure 5.3.

Symbol	Attribute	Type	Description
"forward"	controllable	output	The motor (initially stopped) is caused to begin (and continue) spinning in the forward direction.
"stop"	controllable	output	The motor (initially spinning) is caused to stop.
"reverse"	controllable	output	The motor (initially stopped) is caused to begin (and continue) spinning in the reverse direction.
"reachedA"	uncontrollable	input	The payload is sensed to have reached Point A while the motor is spinning in the reverse direction.
"reachedMid"	uncontrollable	input	The payload is sensed to have reached the midpoint.
"reachedB"	uncontrollable	input	The payload is sensed to have reached Point B while the motor is spinning in the forward direction.

Table 5.1: Complete definition of events in the transporter model.

By our current understanding of the events, the plant of Figure 5.2 is incorrect. Careful reading of [13] reveals that the simpler plant model of Figure 5.2 is justified by increasing the complexity of the definition of the sensor events. Specifically, the condition is added that *"reachedA"* and *"reachedB"* should only occur (be reported by the sensor subroutine) when traveling in the outgoing directions. This follows from the fact that the safety specification (given later) is only concerned with preventing movement in the outgoing directions. With *"reachedB"* only reported in the forward direction, the earlier example string collapses to a string that can be generated by the simple plant of Figure 5.2. A complete description of all events for this plant model

is given in Table 5.1.

Upon further inspection, it can be demonstrated that even in light of the more restrictive event definitions, the plant of Figure 5.2 (adapted from [13]) is (strictly speaking) incorrect. Consider driving the payload past point B, then reversing it left of point B but not past the midpoint, then driving it forward past point B again. The following events would be observed: *"forward"*, *"reachedMid"*, *"reachedB"*, *"stop"*, *"reverse"*, *"stop"*, *"forward"*, *"reachedB"*. This string cannot be generated by the simple plant model of Figure 5.2. The error is due to the fine line between all behaviour that could occur in this programmable system if one considers arbitrary programs (restricted by the definition of the event space), and the behaviour that could occur in this programmable system if one considers programs aligned with the designer's goals. In essence Figure 5.2 is a step in approaching the final controlled plant from the original true plant of Figure 5.3. In this circumstance, the incorrect model still leads to a correct solution but that can only be verified manually, after the fact, and is based on the final (ad hoc) implementation. The entire purpose of employing DES control theory is to automatically generate solutions that are guaranteed to be correct by construction. Since the simple plant model of Figure 5.2 does not, in fact, represent all possible sequences of events, this guarantee is lost, and the benefit of DES control theory is greatly diminished.

As a further detractor, this approach defers complexity from the DES control theory portion of the solution to the ad hoc human-made portion of the solution. Recall that the rotational sensor is a simple input device that merely monitors the rotation of the motor shaft. From this information, a human programmer could write ad hoc LOGO code to determine when each of the sensor events has occurred. Specifically

one could count rotations and generate the *"reachedB"* event (for example) only when the correct number of rotations occurs and the motor is known to be spinning in the forward direction. In contrast, the automatically generated DES supervisor will itself eventually be translated into LOGO code. Consequently, the final solution (a LOGO program) contains both the DES supervisor and the ad hoc events. By choosing the simpler plant model of Figure 5.2, the ad hoc portion of the solution (realization of abstract events) is made more complex.

5.4.2 The Legal Specification

Parallel to the example in [13], the plant of Figure 5.2 has all its states marked. This means that the plant model itself contains no information on what strings represent completed units of behaviour. The purpose of the safety specification is to remove behaviour that is damaging to the system; whereas, the purpose of the progress specification is to indicate desirable orderings of events and to indicate which strings represent completed tasks. These combined describe the subset of the plant's behaviour that is desirable.

Safety

For the transporter, damaging behaviour is limited to the possible extension of the payload some distance beyond point A or point B. Note that due to the simplified definition of events *"reachedA"* and *"reachedB"* it is not possible for the system to determine how far past these points the payload has or will travel.

The safety specification is generated by first removing portions of the plant model as shown in Figure 5.4, and second removing redundant information in a manner

parallel to the computation of a reduced-state supervisor. The goal in generating the safety specification is to express everything of which the plant is capable, minus the specific strings that are damaging.

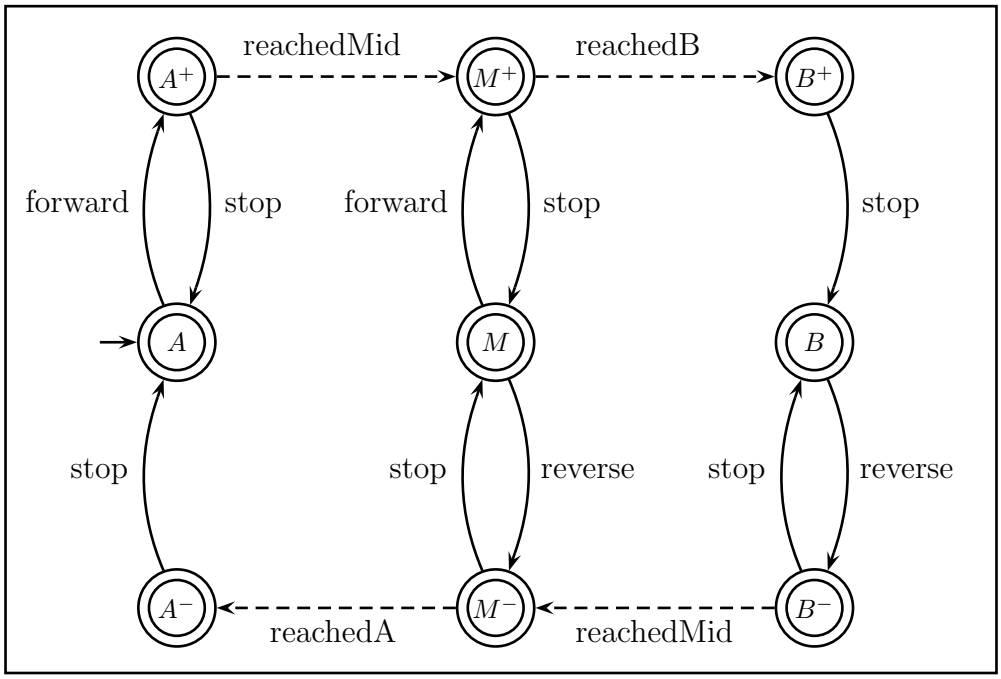

Figure 5.4: The construction of the safety specification.

Consider Figure 5.4. The only information guaranteed by state A is that the payload is left of the midpoint. In order to prevent the payload from possibly reversing past point A, the *"reverse"* event is deleted. Similarly, *"forward"* is deleted from state B. At this point the model communicates "don't move the payload past points A or B". By observation, this specification is controllable with respect to the plant and therefore serves as an implicit supervisor.

The safety specification may be reduced as described in Section 2.2.8, yielding the final safety specification of Figure 5.5. The purpose for this reduction is to simplify subsequent computations on the model. The reduction is valid because the safety specification need only communicate what is disallowed. While the reduced model of Figure 5.5 generates strings not generated by the plant, this is irrelevant because it is only used to eliminate strings from the plant, and is never used to generate them on its own.

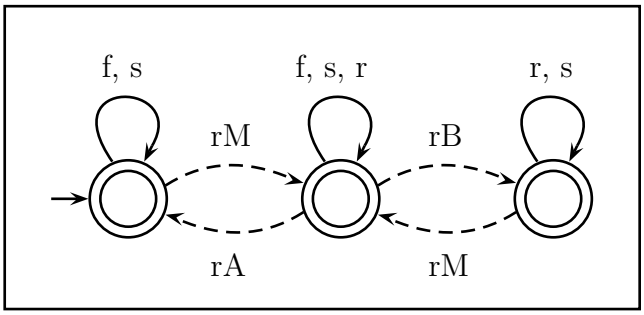

Figure 5.5: The safety specification for the transporter.

The Problem of Time

In the end, when a LOGO program is generated, and the assembly line is set in motion, the generated solution and hence the safety specification do, in fact, succeed. This success however is based on an implementation assumption that is not contained within or considered by the employed DES control theory. Consider state B^+ of Figure 5.4 or the corresponding state in the plant model. In the final solution, in some iteration of the main loop, the subroutine that monitors the sensor may generate the *"reachedB"* event causing the system to logically change state to B^+. In some later iteration of the main loop, by the methodology of [13] the program will generate *"stop"* and the system will safely change state to B. It so happens that the LOGO program runs iterations of its main loop at a rate much faster than the transporter motor spins; consequently, the *"stop"* event is generated shortly after the *"reachedB"* event

occurs. This is not in any way guaranteed to be the case by the DES control theory used to generate the solution. In a purely theoretical view, the system is allowed to happily remain in state B^+ (with the motor running in the forward direction) for all eternity.

Remember that this plant model is only one small module of a potentially massive system. In [13] they require five plant modules for the simple task of connecting two lego blocks. In a more complicated system it is reasonable to expect long strings of events from other modules while a given module remains in a single state. No part of the DES control theory employed in this example guarantees any degree of liveness. With this consideration, it is impossible to achieve the human requirement "don't move the payload past points A or B" in the language of the DES models employed. It is only in consideration of implementation (or by the use of considerably more complex timed DES theory) that it is possible to sweep problems such as these under the rug. That is to say that the fact that this requirement is not actually guaranteed can be ignored by making and verifying the assumption that the system will never linger in states B^+ or A^- longer than a known and sufficiently short amount of time. This can only be verified once the implementation is complete, and if this verification is not provable, the "correct by construction" assumption must be lost.

Progress

The researchers of [13] suggest the definition of a progress specification which has the goal of expressing how the plant should behave. This necessarily precludes the plant from behaving in any other way. Since it would defeat the automation benefits of DES control theory to completely express exactly what the plant should do, some degree

of freedom is necessary. In an input/output system such as the transporter, it is convenient to consider only the events generated by the unprogrammed plant. In the case of the transporter, these are the uncontrollable sensor events and are necessarily caused by the motor events. By rigidly specifying all sensor events and arbitrarily allowing all motor events, the specification can communicate "use the motor however you want in order to make this occur".

For the transporter, the required task is to move the payload from point A to point B and then back to point A. This task is described by the sensor sequence *"reachedMid"*, *"reachedB"*, *"reachedMid"*, *"reachedA"*, and by allowing all motor events. Progress is communicated by the specification shown in Figure 5.6. Note that since this progress specification separates the initial and final states, it communicates that the process should only be carried out once. This is simply an arbitrary design decision.

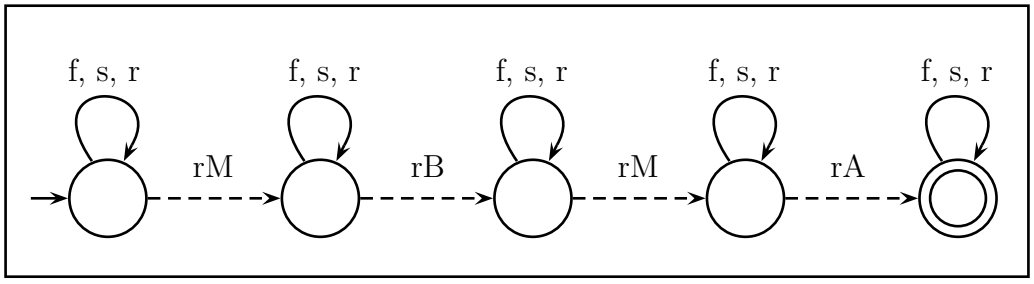

Figure 5.6: The progress specification for the transporter.

This progress specification has several physical implications. First it happily allows the supervisor to stop and start the motor arbitrarily throughout the progress of the task. One could view this as flexibility in the specification—perhaps another component later added to the system will need to stop the transporter and do some

work before allowing the transporter to complete its task. Second it indirectly determines (due to uncontrollability) that it is not acceptable for the transporter to backtrack in the completion of its task. This may be seen as reduced flexibility in the specification—perhaps another component needs to use the space at point B before the payload is unloaded but the transporter has already moved the payload to point B. This particular specification allows some flexibility while sufficiently limiting allowable behaviour such that the ad hoc process of converting the final solution to LOGO code is simplified.

5.4.3 A Control Solution

As stated previously the legal specification is formed from the synchronous product of the safety specification and the progress specification. It is shown in Figure 5.7. At this point the designer has produced correct DES models for both the plant and the legal specification. This is the standard input for DES control theory and the automatic generation of a solution can commence.

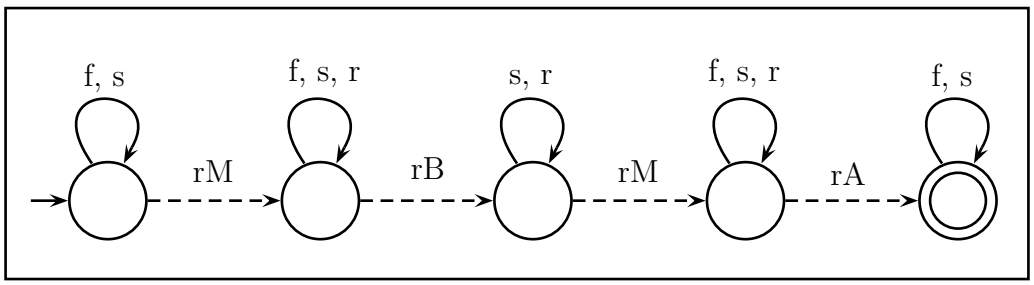

Figure 5.7: The legal specification for the transporter.

The final legal specification of Figure 5.7 is found to be *not* controllable with respect to the plant. From it, a maximally permissive supervisor (that allows the

largest possible sublanguage of the legal specification) is automatically generated as shown in Figure 5.8. According to conventional wisdom, one would next compute a reduced supervisor as shown in Figure 5.9. This is the standard output of DES control theory. For these models the various operations were performed by hand, but for larger models, one would use software such as CTCT as described in Chapter 9.

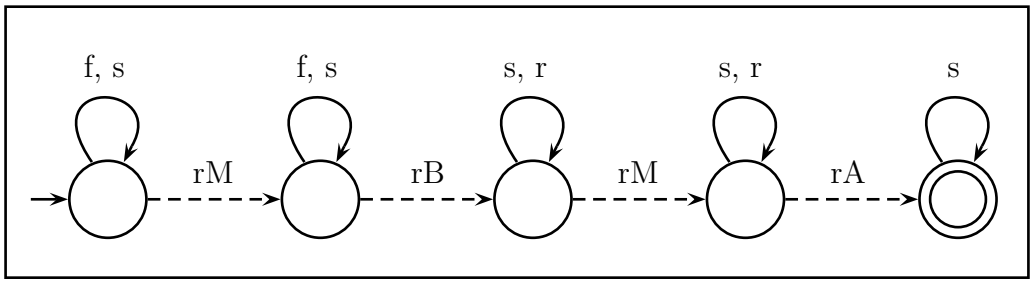

Figure 5.8: Maximally permissive supervisor.

In this implementation, the desired final solution is a LOGO program that will correctly control the hardware model. The reduced-state supervisor is used to compute the supervised plant as shown in Figure 5.10. From this, a reduced supervised plant (shown in Figure 5.11) is generated in an ad hoc manner from which LOGO code can be derived. In order to achieve this, it was suggested in [13] to remove controllable events from the supervised plant until each state has at most one outgoing controllable event while preserving a path from the initial state to a marked state that signifies completion of the required task. The choice of which controllable events to remove is ad hoc. In general, this task could be arbitrarily

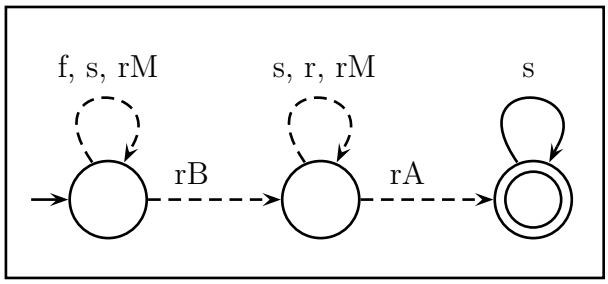

Figure 5.9: Reduced-state supervisor.

complex. In this situation, it is trivial due to the small size of the model and especially due to the form of the progress specification. By incorporating into the legal specification a sequence of uncontrollable events (inputs) separated by states with all controllable events (outputs) in self-loop, a single path is guaranteed to emerge in the controlled plant. While this strategy was effective for this system, other systems with varying behaviour may not permit such a sequential specification. In these cases, the reduction would be more difficult.

Given existing subroutines to monitor the sensors, pseudocode for a complete control solution is provided in Listing 5.1. This was extracted from the reduced supervised plant by hand in an ad hoc manner. The approach for beginning-to-end implementation of DES control theory just described parallels the work in [13] in several ways. A physical system was proposed and modeled as a DES plant while deferring some complexity to the event definitions. All states in the plant model were marked, and the legal specification was formed from a safety and a progress specification. The supervised plant was examined and reduced (in an ad hoc manner) in order to isolate a desirable control sequence, which was translated into a final control solution (LOGO code) by hand (in an ad hoc manner).

In reverse analysis of this methodology, it is clear that the generation of the LOGO code depends only on the information contained within the supervised plant of Figure 5.11. Recall that when $L(\mathscr{X}) \subseteq L(\mathscr{G})$ and $\mathscr{X}$ is controllable with respect to $\mathscr{G}$ then $\mathscr{X}$ is equivalent to the supervised plant $\mathscr{X}/\mathscr{G}$. Consequently, considerable computation could be saved by intentionally generating $\mathscr{L}$ such that $L(\mathscr{L}) \subseteq L(\mathscr{G})$.

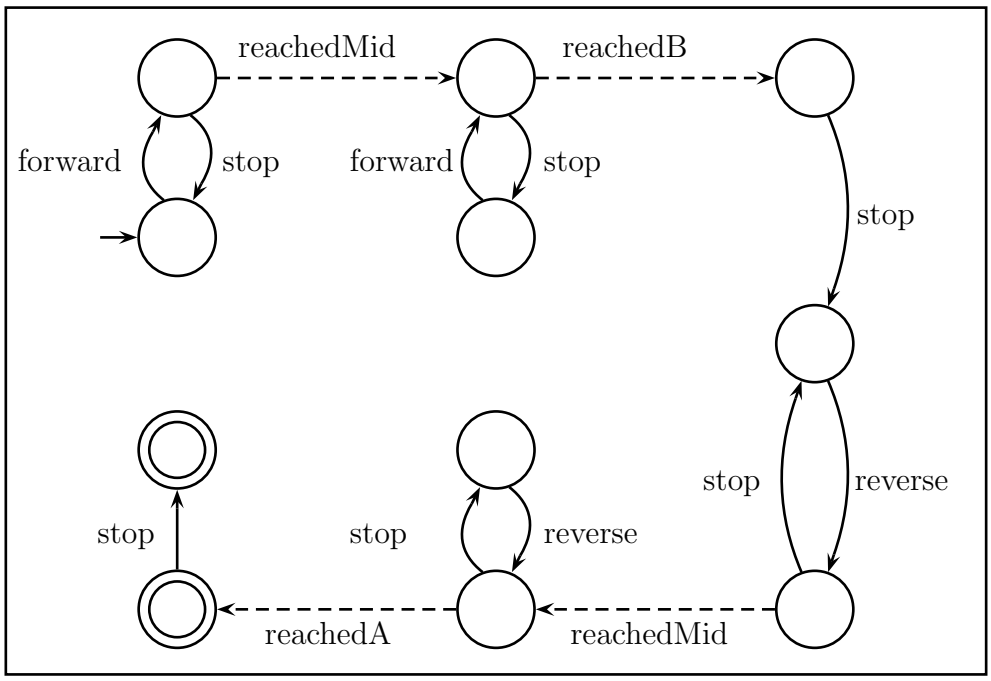

Figure 5.10: Supervised plant.

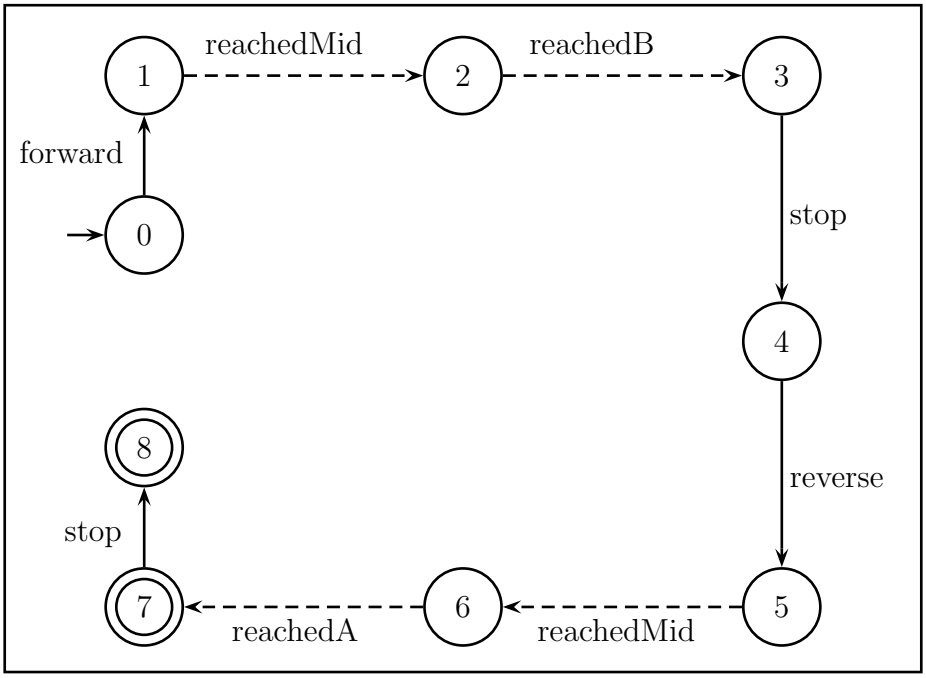

Figure 5.11: Reduced supervised plant.

Listing 5.1 A final control solution in pseudocode. This code assumes access to the motor and sensor in an object-oriented manner, and further assumes that code has been written to test for the logical sensor events unique to this system.

```
int state = 0;

void init()
{
  state = 0;
  motor.stop();
  sensor.reset();
}

boolean run()
{
  if (state == 0)
  {
    motor.goForward();
    state = 1;
  }
  else if (state == 1) { if (sensor.reachedMid()) { state = 2; } }
  else if (state == 2) { if (sensor.reachedB())   { state = 3; } }
  else if (state == 3)
  {
    motor.stop();
    state = 4;
  }
  else if (state == 4)
  {
    motor.goReverse();
    state = 5;
  }
  else if (state == 5) { if (sensor.reachedMid()) { state = 6; } }
  else if (state == 6) { if (sensor.reachedA())   { state = 7; } }
  else if (state == 7)
  {
    motor.stop();
    state = 8;
  }

  sensor.update();
  if (state == 8) { return false; }
  else { return true; }
}

void main()
{
  init();
  while(run()) {}
}
```

Finally, it is natural to ask the question, "When the language of the final control solution is known, can the solution be automatically generated from a reduced supervised plant model (such as Figure 5.11), thereby avoiding human error?" Certainly, in the context of this simple example the answer is 'yes', as demonstrated by the simplicity and standardized nature of the pseudocode in Listing 5.1. The same cannot be said for the ad hoc process of reducing the supervised plant. This is achieved by arbitrarily removing outgoing controllable events based on intuitive desires of the human designer.

5.5 Alternate Approaches

The Reality of Design

There is a wide variety of systems that can be conveniently modeled as discrete-event, and of these, some classes are very different than others with respect to implementation of DES control theory. Consider some of the properties of the transporter problem. It is very important to realize that the physical model and the DES model are created simultaneously. Even if a previous employee had first snapped together the LEGO blocks and hooked up the motor and sensor to the Dacta interface box, the plant would still be the empty set. The connected blocks, motor and sensor (without control) generate nothing. There is no plant until the DES control theory is implemented. This is very different from a class of problems where there is an existing system functioning under existing and imperfect control (such as is the case with DESCO discussed in Chapter 3). One might assume that a third employee might have written an ad hoc and faulty LOGO program to control the transporter but this

is not the same as an existing system. Either the faulty program must be deleted (which reverts to the original case) or the goal changes to modeling and controlling the LOGO code itself. This is quite different than modeling the actual LEGO system and, in cases such as this, would be strikingly inefficient.

Consideration of the Goal

In the case of the LEGO transporter, the ultimate goal is to connect together some physical components and write a LOGO program that correctly instructs the physical components to transport the payload from point A to point B and back to point A. Once the physical setup has been defined and has been translated into a DES model such as the plant shown in Figure 5.2, the remaining task is to generate the LOGO code. In order to generate the LOGO code, it is only necessary to obtain a model of an appropriately supervised plant. In the previous discussion this was accomplished by constructing and reducing a safety specification, constructing a progress specification, obtaining a legal specification as the synchronous product of the safety and progress specifications, synthesizing a maximally permissive supervisor from the legal specification and plant, reducing the supervisor, and finally computing the supervised plant as the synchronous product of the reduced-state supervisor and the plant.

Consider an alternate methodology for achieving the same result. First recall that an appropriately supervised plant should not drive the transporter outside of points A or B (for safety reasons) and further should generate no other string of sensor events than *"reachedMid"*, *"reachedB"*, *"reachedMid"*, *"reachedA"* (for correct progress). In this, as in the original discussion, it shall be assumed that the process should only be carried out once. With the control objectives in mind, consider the construction of a

legal specification by first imposing safety and second incorporating progress.

Safety can be imposed as shown in Figure 5.12. This model is the same as the original construction of the safety specification from Figure 5.4. In this stage the plant is examined and two events are removed in order to guarantee safety. This ad hoc process is necessarily the job of a human designer. Previously once safety was expressed redundant information was removed converting it into a reduced-state model. In this new methodology, the constraint is added that the ad hoc expression of safety must generate a language that is a subset of or equal to $L(\mathscr{G})$. Figure 5.12 is guaranteed to comply with this constraint because it was constructed by only removing states and/or transitions from the plant. Assuming the plant is a minimal-state representation of itself, any state-space reduction of Figure 5.12 is guaranteed to violate the new sublanguage constraint.

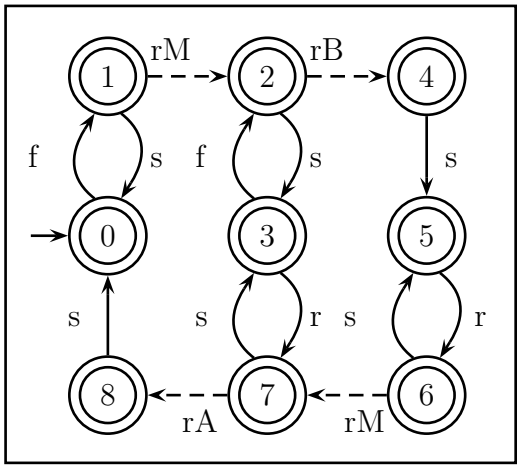

Figure 5.12: Safety imposed by removing events from the plant.

Next consider modifying the model to communicate progress. This can again be achieved in an ad hoc manner by a human designer. One must simply split all states that allow paths that generate unwanted sequences of sensor events—namely states zero and three. It is, however, perhaps too much to ask of the designer to manually incorporate the requirements for progress into the legal specification. As in the previous example it may be easier to express it as a progress specification and automatically compute the final legal specification. The progress specification

of Figure 5.6 is reproduced here as Figure 5.13. The final legal specification is the synchronous product of safety and progress and is shown in Figure 5.14.

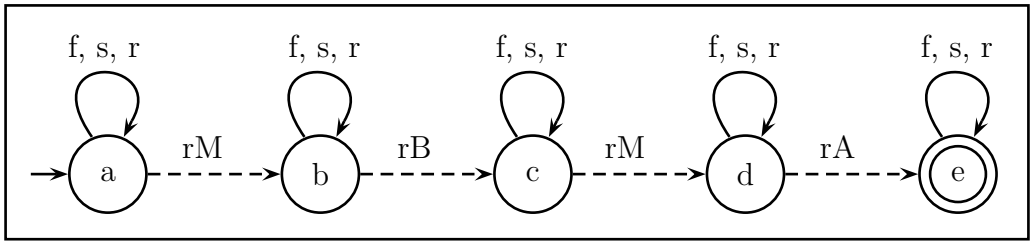

Figure 5.13: Adding progress to the legal specification.

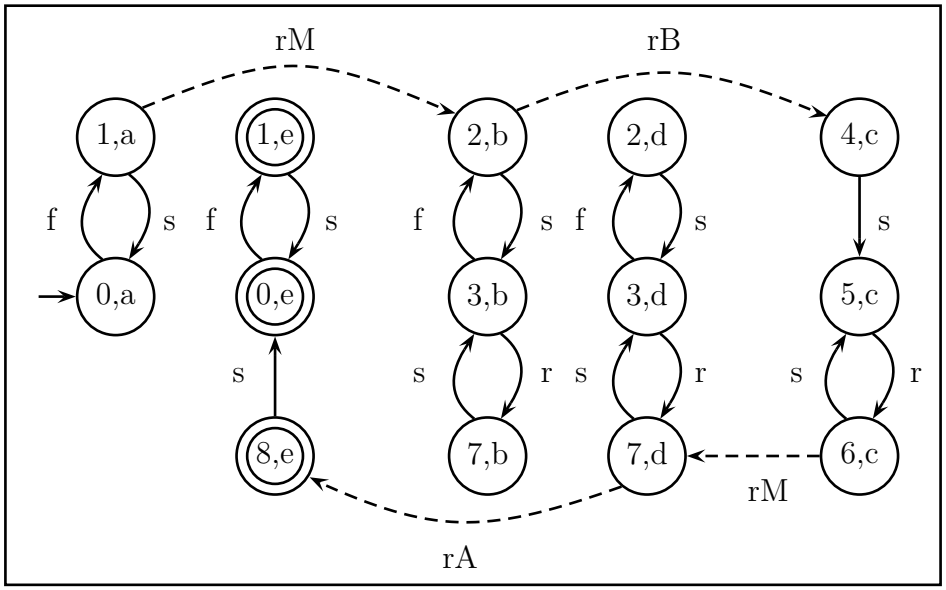

Figure 5.14: The synchronous product of safety and progress.

This legal specification is different than the previous specification of Figure 5.7 because the safety specification was not reduced. This legal specification is not controllable with respect to the plant (nor was the previous legal specification). It is, however, highly desirable to achieve a controllable result. The synchronous product

of the safety and progress specifications is guaranteed to generate a sublanguage of $L(\mathscr{G})$ because the safety specification itself generates a sublanguage of $L(\mathscr{G})$. If it were also controllable, it would not only serve as an implicit supervisor, but also would represent the supervised plant, which is the goal of this exercise.

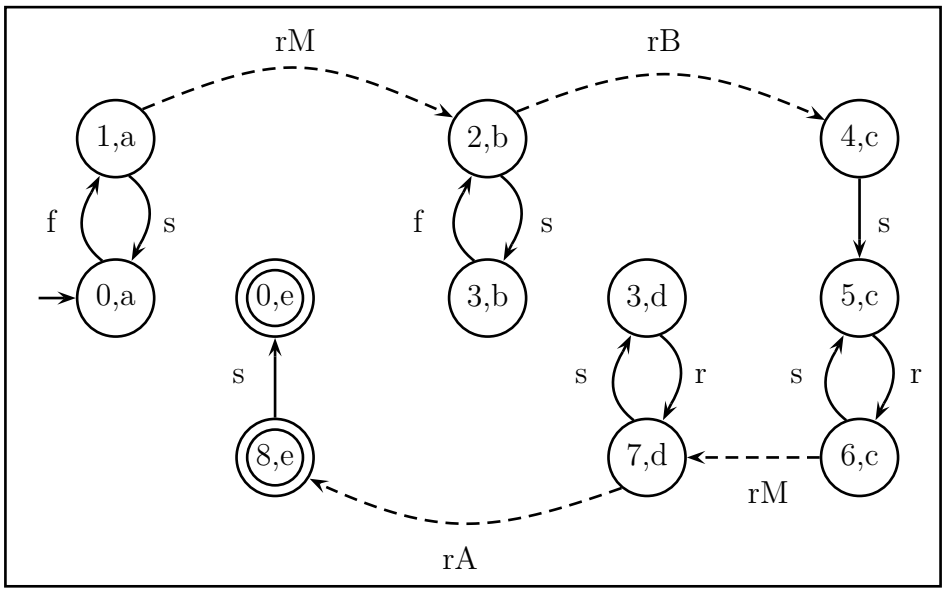

Figure 5.15: The legal specification, supervisor, and supervised plant.

Consider modifying the synchronous product algorithm to delete from the result any states that fail to include any outgoing uncontrollable events that exist in the corresponding states of the input automata and further deleting all states that uncontrollably lead to these deleted states. This is effectively the same as computing the synchronous product and the maximally permissive supervisor simultaneously. It has the benefit of reduced computational complexity and guarantees that the language produced by the supervisor is a sublanguage of the language produced by the plant. This achieves the goal of automatically synthesizing a model of an appropriately supervised plant, and the result is shown in Figure 5.15.

Summary

All the work in this chapter has hinged on the assumption (given in [13]) that in the end a human designer would choose a single path from the controlled plant and generate a LOGO program from it. An important property of this system is that no plant actually exists. There will never exist a plant and supervisor in the standard closed-loop form. The solution for this system is a LOGO program that is unrelated to the idea of interacting finite-state machines. In light of this fact, the human designer needs only to acquire a model of the controlled plant from which to extract a LOGO program by an ad hoc method. The standard application of theory that one might assume to employ is inefficient for a system such as this. An alternate methodology is summarized below. While it only decreases the complexity of the task by some constant factor, one could find the decreased workload appreciable in practice. The reality is that a human must give a description of the plant, a human must give a description of what is legal, and a human must extract a single path from the controlled plant. The automation therefore lies in the generation of the controlled plant model from the human's description of what is possible and what is legal. The key difference here is that this automation is carried out in a single step. The user need only input the safety and progress models as described. Tedious tasks such as the separate computation of the final legal specification, the supervisor and the reduced supervisor could be avoided.

1. Define the event space.

2. Define the plant FSM $\mathscr{G}$.

3. From a copy of the plant define a safety FSM that generates a language contained in or equal to $L(\mathscr{G})$ (which can be achieved by paring down the transition structure of the plant).

4. Define a progress FSM that is a series of uncontrollable events with all controllable events in self-loop.

5. Run an algorithm that simultaneously computes the synchronous product of safety and progress and removes all uncontrollability with respect to $\mathscr{G}$. That is to say that, while computing the synchronous product, if a state from the safety specification is not included in the result and there exists an uncontrollable transition to that state (in the original safety specification), then the state that originates the uncontrollable transition must also be removed from the result (and so on recursively).

6. The result is an appropriately supervised plant from which the designer may choose a path and generate a LOGO program.

The methodologies examined and proposed in this Chapter provide means of applying DES control theory to a class of systems. These procedures require considerable ad hoc components and, applied to problems of this scale, are arguably more complicated and error prone than a straightforward ad hoc solution. Nevertheless, they do promise to scale to larger systems and would presumably realize an advantage in such cases. These strategies along with the others reviewed in previous chapters are listed and categorized in Chapter 8.

Chapter 6

Resource Management

6.1 System Description

In [37] Ramadge and Wonham illustrate the principles of DES control theory by considering "two users of a single resource". They model the system, define a legal specification, compute a supervisor and further compute a reduced-state supervisor. This is the standard procedure by which DES control theory would theoretically be used to solve problems, and a suitable supervisor is considered the final solution. It can be demonstrated that there is a gap between this solution and its implementation.

First consider the behaviour of a single user as shown in Figure 6.1. Note that it is assumed that granting use of the resource is controllable and that once the user desires the resource, the user does not cease to desire the resource before acquiring the resource. With these two assumptions in place it is clear that the abstract model in Figure 6.1 is correct.

The **shuffle** of two such models (shown in Figure 6.2) represents the concurrent behaviour of the two users under the assumption that actions are asynchronous and independent. This is valid if any implementation interprets two simultaneous events as arbitrarily one before the other. Note that this model contains all possible behaviour including potentially undesirable behaviour.

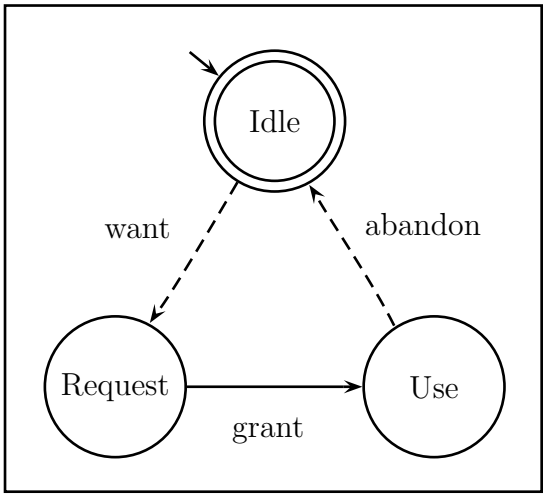

Figure 6.1: Behaviour of a user and a resource.

In [37] the control objectives for this system are mutual exclusion (only one user should have access to the resource at any given time) and fair usage (the user that first requests the resource should first receive the resource, as in a queue). These objectives are formalized in Figure 6.3 which generates a sublanguage of the language generated by Figure 6.2. Hence Figure 6.2 represents the plant and Figure 6.3 represents the legal specification.

In the legal specification, mutual exclusion was achieved by the deletion of the state *"UU"* and fair usage was achieved by splitting the state *"RR"*, thereby eliminating any ambiguity as to which user first requested the resource. Recall that if the legal specification is found to be controllable with respect to the plant, then it can serve as an implicit supervisor. In this case it is indeed found to be controllable.

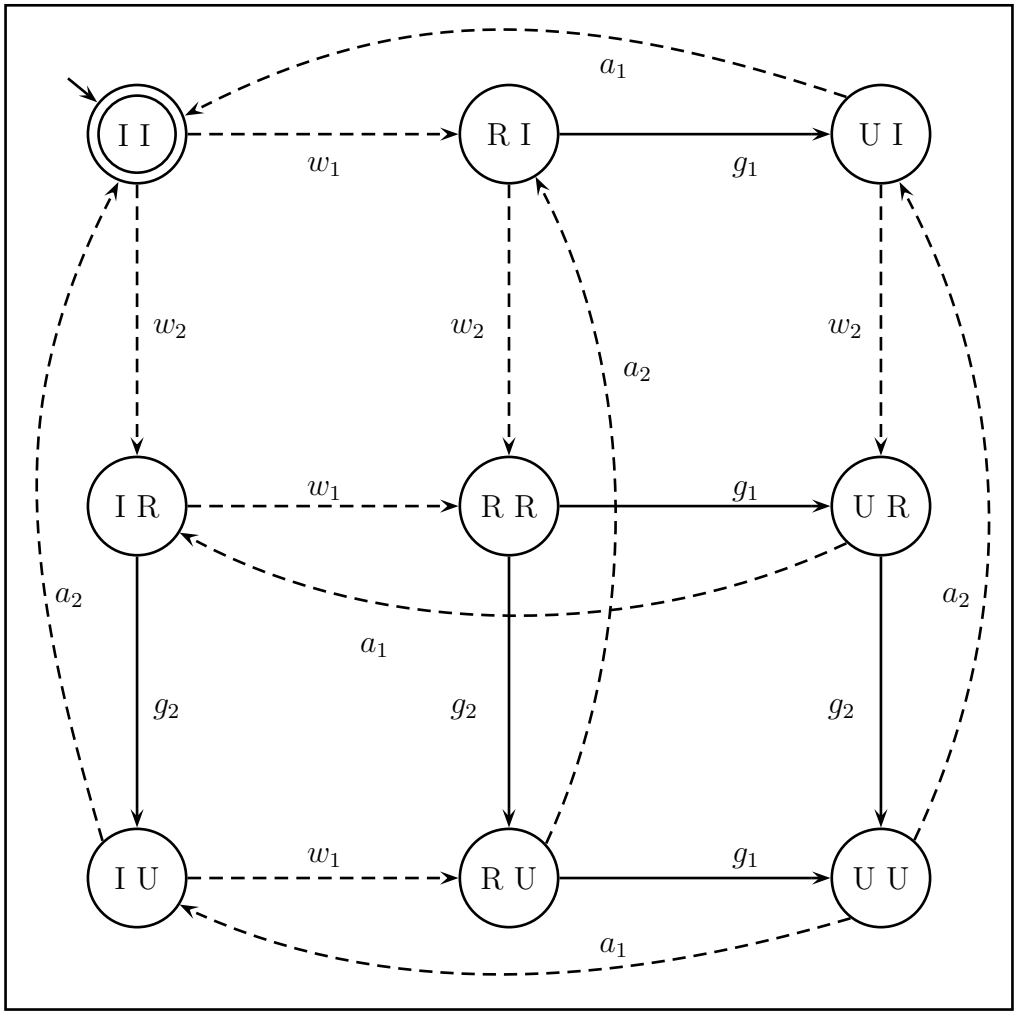

Figure 6.2: Behaviour of two users and a single resource. The symbol w_i indicates user i suddenly wanting the resource. The symbol g_i indicates user i suddenly being granted the resource. The symbol a_i indicates user i suddenly abandoning the resource. States are labeled USER1 USER2 with I indicating the idle state, R indicating the request state and U indicating the use state.

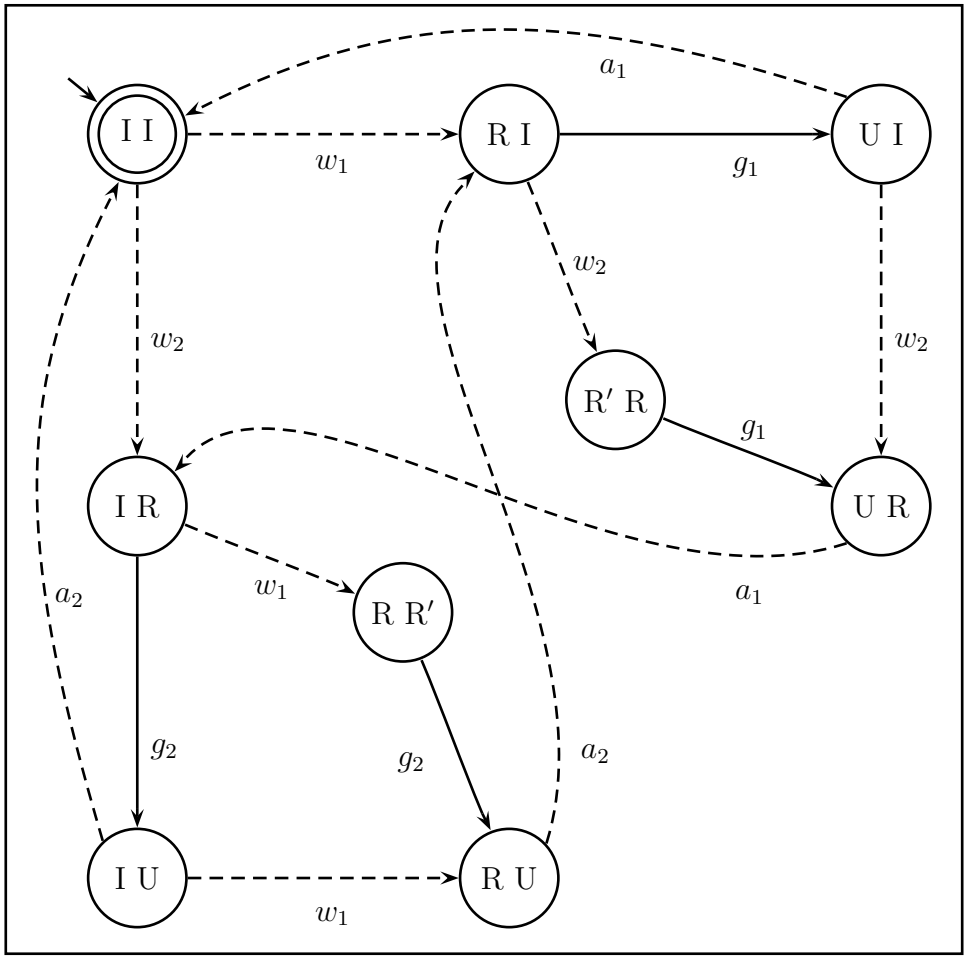

Figure 6.3: Legal behaviour of two users with a single resource. R′ is used to indicate that the respective user was the first to enter the request state. The remaining symbols are as defined in previous figures. This figure models the control objectives of mutual exclusion and fair usage. Since this specification is found to be controllable with respect to the plant it also serves as an implicit supervisor.

In general it is assumed that a supervisor with a smaller state space is more desirable, and since a supervisor acts in conjunction with events generated by the plant, much of the structure in Figure 6.3 is redundant. For this reason, the reduced-state supervisor shown in Figure 6.4 is computed as the final solution. Note that while the automaton in Figure 6.4 could generate illegal strings, this never occurs in closed-loop, because all events are generated by the plant.

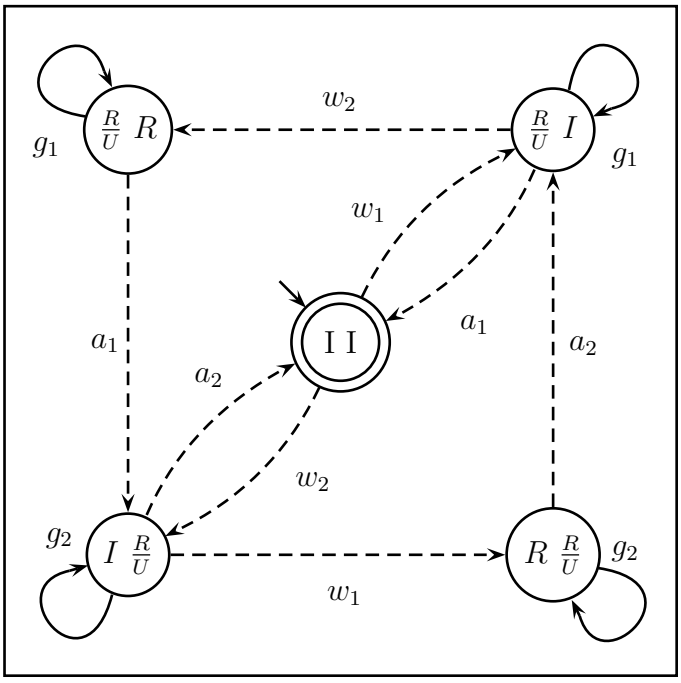

Figure 6.4: Reduced-state supervisor of two users with a single resource. Here $\frac{R}{U}$ simply indicates that the respective user may be in either the request state or the use state. The remaining symbols are as defined in previous figures.

6.2 Plausible Implementations

Consider a concrete example of this abstract model, namely two humans in a clothing store with only one change room as shown in Figure 6.5. In this case the supervisor might be implemented by the store clerk. Consider now what exactly is the plant? Certainly the individual humans may generate the *"want"* and *"abandon"* events, but what entity generates the *"grant"* events? If the store clerk is the supervisor then strictly speaking, it cannot generate events. Following the framework, there must exist a fourth entity within the plant that generates *"grant"* events. In the worst

case, these might be generated randomly. Alternatively they might be triggered by any *"want"* event. Clearly, in an implementation it is desirable that there exist some notion of causality, but if the *"grant"* generating component is more intelligent and only generates grant events according to the legal specification, then the plant is self-controlled and has no need for DES control theory. This concern exposes the possibility of integrating DES supervision into the plant as the control solution.

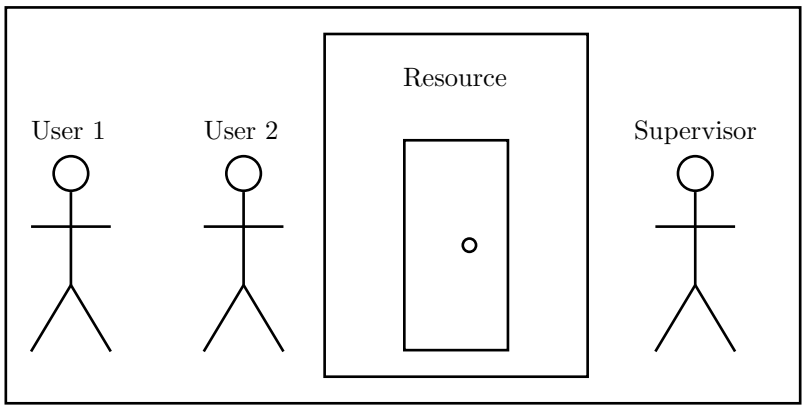

Figure 6.5: Two human users and a resource with a human supervisor.

In the scenario of Figure 6.5 a single store clerk could serve as both a supervisor and a component of the plant. If the store clerk were given a copy of the legal specification in Figure 6.3 (which is an implicit supervisor) and was instructed to generate *"grant"* events as soon as possible whenever allowable, a correct implementation would be complete. This is not the case for the reduced-state supervisor in Figure 6.4 because it would continually generate *"grant"* events in all states other than the initial state. The store clerk is both the supervisor and part of the plant, so it must have knowledge of not only what should occur but also what can occur. The latter is exactly the redundant information that was removed in order to create

the reduced-state supervisor. This demonstrates that if the plant is not fully defined independent of the control objectives, then DES control theory cannot be applied in the standard manner. This issue is revisited at the end of this chapter.

6.2.1 Automatic Solutions

Up to this point the discussion has focused on high-level implementation utilizing existing complicated objects such as humans. While existence of a solution has been shown, the idea of handing a human a paper copy of Figure 6.3 is somewhat clumsy and doesn't scale to automated systems. This motivates investigation into low-level and automatic implementations such as the scenario depicted in Figure 6.6. Here control is imposed by a machine, but in order to achieve an implementation such as this, the system must be more strictly defined. In the human scenario it is not necessary to specify exactly how *"want"*, *"grant"* and *"abandon"* are realized because the human components are versatile and complex. In a civilized situation, "$w_1, w_2, g_1, a_1, g_2, a_2$" might map to the following interchange: "May I (user1) use the room", "May I (user2) use the room", "Go ahead (user1)", "I'm (user1) done", "Go ahead (user2)", "I'm (user2) done". In this exchange addressing is determined by eye contact, and protection of the resource is achieved by a shared protocol where the users don't use the resource without permission. In a less civilized example, the clerk might protect the resource by main

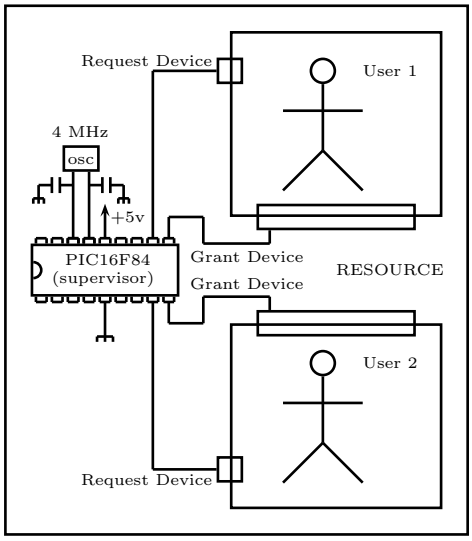

Figure 6.6: Two human users and a resource with a machine supervisor.

force, but the dynamics of the situation are the same.

In the machine example of Figure 6.6 the users are contained in separate rooms, each with a request device and a grant device. Consider the situation where the request device is a binary switch connected to an input pin and the grant device is a lockable door connected to an output pin. A human user indicates *"want"* by flipping the switch (input reads one), and indicates *"abandon"* by returning the switch to the initial position (input reads zero). Similarly *"grant"* corresponds to the unlocking of the door (output zero) and a necessary component of *"abandon"* must be a re-locking of the door (output one). Note that the state of the lock, not the position of the human, must indicate possession of the resource. Once the door is unlocked, the human has uncontrollable access to the resource and therefore even if the human never exits the room it must be considered to be in possession of the resource. For rigor, assume that the human cannot revert the switch and rush through the door before it is locked, and assume that the door is the only access to the resource and cannot be compromised.

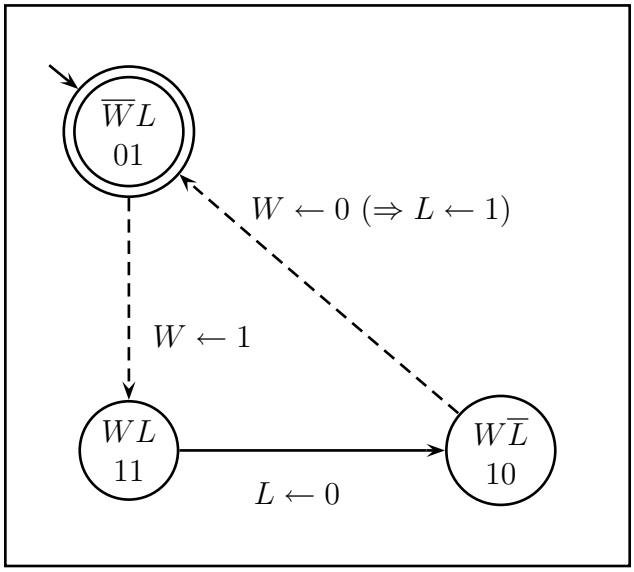

Figure 6.7: Complete behaviour of a user and a resource. State labels show the values of the system variables want and locked. In the initial state $\overline{W}L$ or 01 indicates that the resource is un-wanted and locked. The transition "$W \leftarrow 0$ ($\Rightarrow L \leftarrow 1$)" represents two changes of state (namely $10 \rightarrow 00 \rightarrow 01$) and reads: "The user ceases to want the resource which implicitly denies access to the resource."

This description implies that a single user is associated with two binary variables describing four possible states. This conflicts with the

three state model of Figure 6.1 until one realizes that *"abandon"* actually represents two changes of state. First the user indicates *"abandon"* and second *"abandon"* is enforced as shown in Figure 6.7. This difference is, in fact, an artifact of the implementation. Consider an alternate situation where the grant device is a door without a lock but having a buzzer that can be made to ring for three seconds by a very short pulse on the output pin. In this situation, *"grant"* corresponds to such a pulse and there is no machine restriction on the *"abandon"* event. This is accomplished by deferring complexity to a different component of the system, namely the human.

6.2.2 Implementation Options

These nuances of implementation re-occur in many other systems and can be summarized by two concepts: obedience and communication. Components of a system may or may not be obedient. An **obedient component** may be trusted to behave according to predefined protocols and therefore control can be achieved via communication. For example, a human may be trusted to not pass through a door unless a light is green, or a software process may be trusted to not access an area of memory unless a predefined boolean value it can read is set to true. Similarly some humans cannot be trusted to not use doors (hence the invention of locks), and some software processes (viruses) cannot be trusted to not modify data (hence the physical read-only switch on 3.5-inch floppy disks).

The concept of control by communication itself has two components: **state-based communication** and **transient communication**. A light that is lit when the user is not allowed to access the resource and unlit otherwise is an example of state-based communication; whereas a light that briefly flashes or a bell that briefly rings to

communicate that the user is now allowed to access the resource is an example of transient communication. Note that transient communication requires an additional assumption that the relevant component never misses the communication.

In transient communication the lack of a flashing light, or the lack of a ringing bell does not imply that the user does not have access to the resource, and hence the state of the communicator (the means of communication) is not consistent with the state of the communication (the message received). In all transient communication, the message received (in this case the granting of access to a resource) must implicitly expire at some later predefined state. In the case of the two users and a resource system, each entity may know that once the resource is abandoned, further access is implicitly denied.

Finally, consider a working solution realized by machine code in the microcontroller for the switches and locks system previously defined, and note that replacing the locks with lights that are lit when the locks would be unlocked and unlit otherwise, and replacing the humans with obedient humans who never enter the room unless their respective lights are lit demonstrates that the control solution for a non-obedient system is identical to the control solution for an obedient system with state-based communication. Further note that this is not the case for obedient systems with transient communication, which defer complexity to sub-components in order to simplify the control solution.

The re-labeled specification of Figure 6.8 contains enough information for any firmware programmer to realize an implementation of the switches and locks scenario of Figure 6.6 without any knowledge of the DES control theory with which it was generated and verified. While this may not provide an efficient solution or even a

scalable solution, it demonstrates a bridge from theory to application.

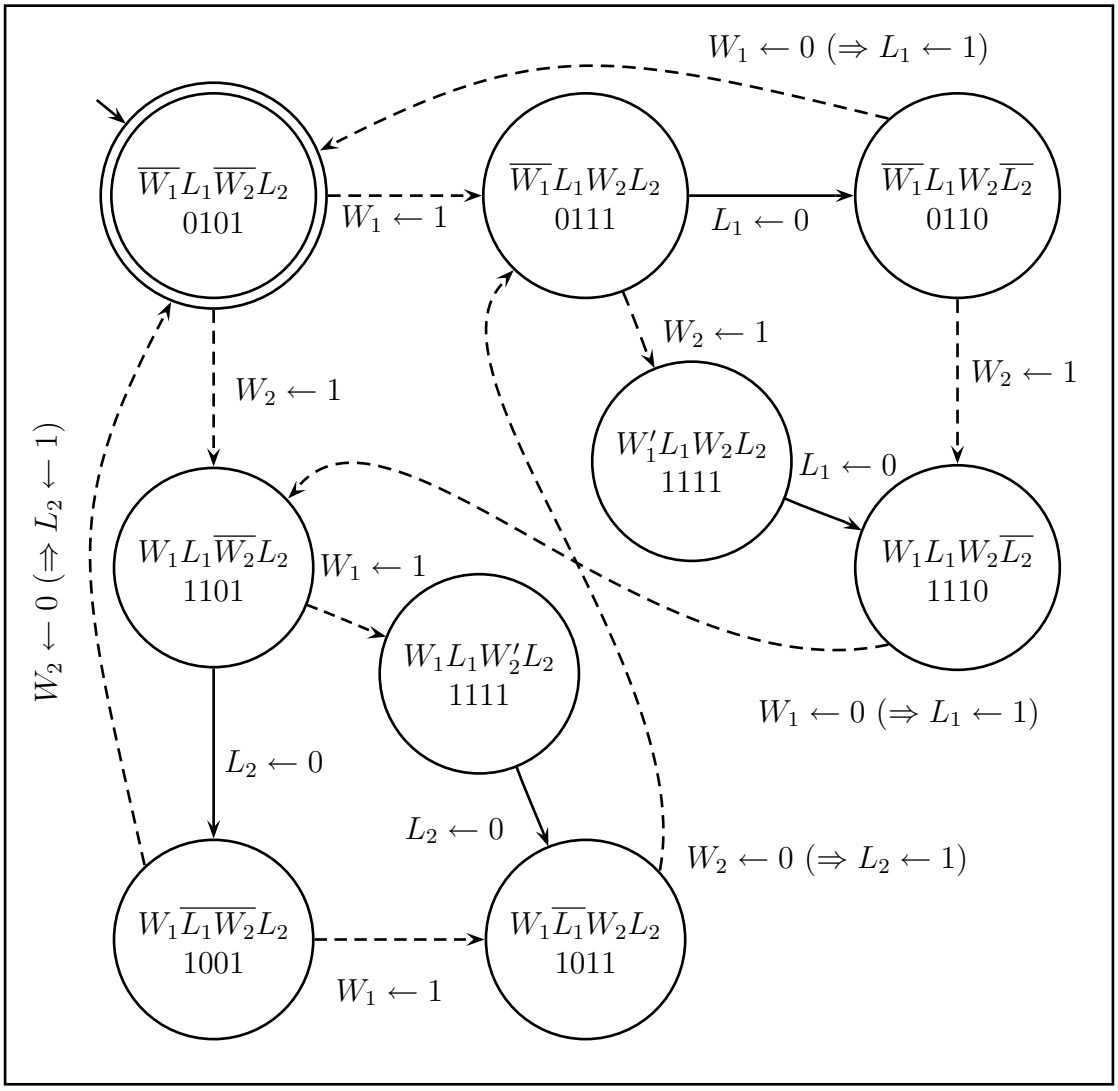

Figure 6.8: Complete legal behaviour of two users with a single resource. Each state records the current value of all system variables, and each transition documents the nature of their changes. This figure models the control objectives of mutual exclusion and fair usage. It serves as an integrated plant and supervisor; namely specifying what outputs to generate and which inputs to ignore.

6.3 The Plant Myth

At this point, the plant, legal specification and supervisor seem well understood and various implementations seem straightforward. Reconsider the plant of Figure 6.7 in the context of the switches and locks implementation of Figure 6.6. Does the plant model define all possible behaviour? Before the microcontroller is programmed, no behaviour exists. After the microcontroller is correctly programmed only legal behaviour exists and DES control theory is unnecessary.

Consider a single user plant model expanded to show all system states and extended to a complete graph as shown in Figure 6.9. The transitions labeled X represent two simultaneous changes of state. For simplicity, assume the microcontroller always functions quickly enough to distinguish arbitrarily close events as one before the other. This eliminates the possibility of the X transitions. Furthermore, the human behaviour is assumed to be restricted by the rule that a user does not cease to desire the resource before acquiring the resource. This means the user will never revert the switch before the resource is granted and guarantees that $11 \xrightarrow{\overline{W}} 01$ will never occur. Finally, based on the high-speed microcontroller assumption, the door will always be locked very soon after the switch is moved to the abandonment position. That is to say that $00 \xrightarrow{L} 01$ will always quickly

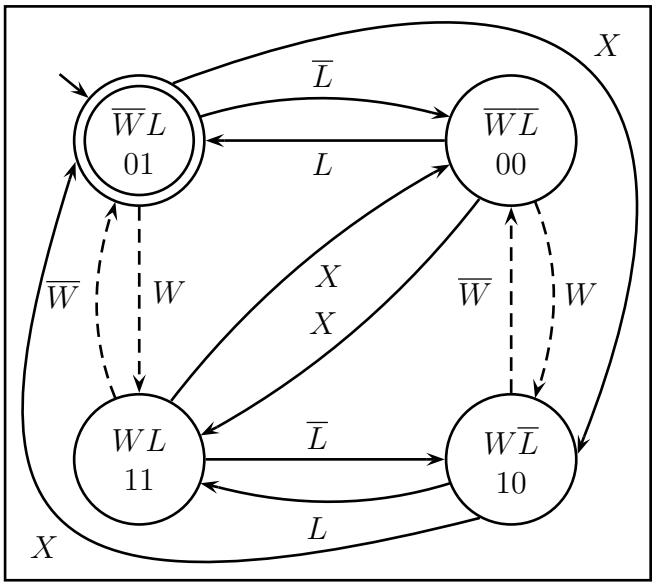

Figure 6.9: Unrestricted plant.

follow $10 \xrightarrow{\overline{W}} 00$, thereby guaranteeing that $00 \xrightarrow{W} 10$ will never occur.

At this point the plant is defined by Figure 6.10, but it includes more transitions than were expected by the analysis at the beginning of this chapter. What is it about the system that could preclude the possibility of $10 \xrightarrow{L} 11$ and $01 \xrightarrow{\overline{L}} 00$ other than the fact that the behaviour is intuitively not useful? In fact, there is nothing to preclude these events. They are quite possible based on the software interface defined by the event space. The definition of the event space is of great importance, and the following may only hold for systems (such as this) where controllable events map to output pins on a programmable device and uncontrollable events map to input pins on a programmable device.

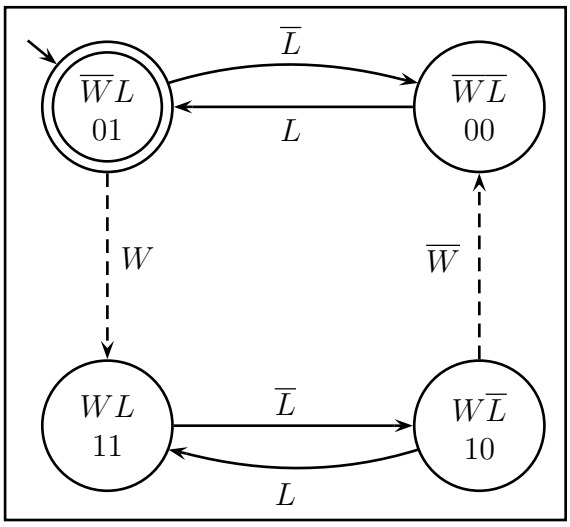

Figure 6.10: Restricted plant.

In this case, the boundary between plant and legal specification has become blurred because the microcontroller component (still unprogrammed) could be capable of taking actions not only in response to the actions of other entities (such as the humans) but also of its own initiative. Since it is desirable to automatically generate an appropriate program for the microcontroller, strictly speaking, one should not disregard possible events simply because they are not useful.

On the other hand, one could simply view this as iterative construction of a controlled plant. Consider a plant $\mathscr{G}$. Now suppose that $\mathscr{L}$ is constructed by copying $\mathscr{G}$

and then only removing some controllable transitions. Such an $\mathscr{L}$ is not only controllable with respect to $\mathscr{G}$ but is in fact equal to the controlled plant $\mathscr{L}/\mathscr{G}$. Furthermore, one might choose to call this new model the plant, because, once programmed, the microcontroller will not in fact be capable of generating the removed events.

In light of this, it is safe to iteratively construct a specification (that, in essence, constitutes the plant, the legal requirement, the supervisor, *and* the controlled plant) in the following manner. Once the event space has been defined, start by defining $\mathscr{X}$ as an automaton that expresses the widest possible range of behaviour. Next construct an automaton $\mathscr{L}_1$ that removes some behaviour (either unreasonable or undesirable) and confirm $\mathscr{L}_1$ controllable with respect to $\mathscr{X}$. Finally, define $\mathscr{X} = \mathscr{L}_1/\mathscr{X}$. Repeat for more requirements $\mathscr{L}_i$ until all requirements have been included. The automaton $\mathscr{X}$ contains all required implementation information for plant components that do not yet exist (code on the microcontroller) and further contains all necessary supervisor information implicit in those plant behaviours that could have existed but did not survive the generation process into the final system specification. Note that in this scenario, no attempt is made to generate a reduced-state supervisor, because it relies on the assumption of classic closed-loop control; which is to say, it requires that the plant exists independent of the control objectives.

Summary

This material provides specific means of applying DES control theory to specific systems. As in any problem, the choice of how to view the system and the tools to employ involves trade-offs between various goals. These strategies along with the others reviewed in previous chapters are listed and categorized in Chapter 8.

Chapter 7

Initiated-Event Methodology

In this chapter a new methodology is presented for application of DES control theory to low-level systems with programmable components. The methodology is developed in detail for one system, and then its application to two other systems is described in less detail, for the purpose of demonstrating both its flexibility and its weaknesses. This new approach, called *the initiated-event methodology*, is presented using a vending machine as a running example. Its fundamental precepts include a particular style of event space definition (namely, abstracting uncontrollable inputs and controllable outputs into a single controllable event that is uncontrollably initiated) and the goal of automatic code generation (for the programmable component of the plant). As it is described (with the vending machine example), justification for the approach is provided. Furthermore, this new methodology is summarized (along with the other approaches examined in this work) in Chapter 8.

The core human component of the approach is in the event space definition, and three separate styles of automatic result are proposed: monolithic, distributed and integrated. The monolithic and distributed results follow from classical DES control

theory. The integrated result applies to a smaller set of systems and requires more input from the human designer, but it provides savings in both time and data complexity. All three results are accompanied by pseudocode that indicates how such results could be obtained in a general way. Furthermore, Appendix C contains concrete representations of each result. The combination of these should be sufficient for an individual to implement automated code generation from the initiated-event methodology for an arbitrary platform.

Finally, one should note that the initiated-event methodology does not provide a completely automated solution. For each event definition it is required that the conditions for initiation and the work necessary for fulfillment both be provided (in code) by the human designer. Such code would presumably be inserted into the automatically generated framework as the final phase of the DES solution.

7.1 Vending Machine

Discrete-Event Systems control theory is founded upon FSM models, and the vending machine is a favorite real-world example of textbooks introducing finite-state machines. Furthermore, the vending machine is a low-level, non-intelligent device, and as such is illustrative of many of the complications that can arise during the application and implementation of DES control theory. In this section the initiated-event methodology is introduced implicitly via a vending machine example.

7.1.1 System Description

Consider a fairly high-level view of a very simple vending machine, depicted in Figure 7.1. The vending machine can accept one type of token and dispense one type of pop. The pop costs two tokens. The human users may insert tokens (which may be rejected). The users may also request pop (using a button that may be ignored). The machine contains a maximum of three pop and may be refilled by a human technician.

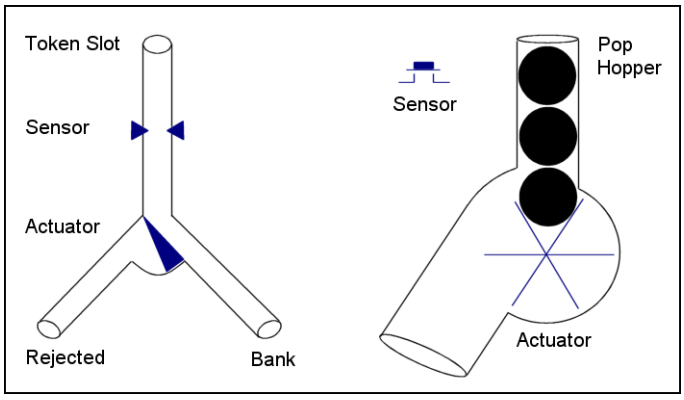

Figure 7.1: A representation of a simple vending machine.

From the point of view of a control solution (such as a microcontroller), the inputs caused by the human users are uncontrollable (values read from input pins) and the outputs generated by the machine are controllable (values written to output pins). This point of view, however, results in a rather unwieldy application of DES control theory. From a more abstract point of view, difficulties such as controllable event generation and separation of the plant from the control objectives may be resolved. The event space is the aspect of the modeling language that is determined by the chosen point of view. The event space defined in Table 7.1 is representative of the more abstract (and beneficial) point of view.

Symbol	Description
"token"	**controllable.** A token is received into the machine's bank. Assume that a human may asynchronously and uncontrollably input tokens into the machine, but that a subsystem always senses this and always has the option of accepting or rejecting the token. Further assume that this decision is always carried out fully before the human is able to take any other action. Further assume that no user ever inputs any matter other than a valid token. Finally, the token bank is assumed to be of infinite size. The instantaneous time of this event is taken to be the time after which the machine is no longer able to control the position of an inserted token. Note that if the token is rejected, then this event has not occurred.
"pop"	**controllable.** A pop is delivered from the machine. This physically corresponds to a mechanical action in the ejection device which can occur whether or not any pop exists in the machine. This event is considered to have occurred regardless of whether or not a pop was actually dispensed from the machine. All possibility of jamming or mechanical failure is ignored. Assume that a human user may asynchronously and uncontrollably request the vending of a pop by pressing and releasing a button, but that a subsystem always senses this and always has the option of ignoring the input or dispensing a pop. Further assume that this decision is always carried out fully before the human is able to take any other action. The instantaneous time of this event is taken to be the time after which the ejection device stops moving. Note that if the ejection device does not attempt to dispense a pop, then this event has not occurred.
"refill"	**uncontrollable.** The pop hopper is manually refilled such that the machine contains three pop. It is assumed that during the act of refilling, all other input is prevented from occurring. The instantaneous time of this event is taken to be the time after which the refiller stops preventing other users from interacting with the machine.

Table 7.1: Events in the vending machine.

7.1.2 Rules

The description of the vending machine system can be summarized as a list of rules that communicate the machine's goals and physical limitations. Figure 7.2 demonstrates the rule that pop costs two tokens. It contains two pieces of information that can be read as **Rule 1.1** *"pop events must be separated by at least two token events"* and **Rule 1.2** *"pop events must be separated by at most two token events"*.

The former is indicated by the lack of *"pop"* transitions from states zero and one. The latter is indicated by the lack of a *"token"* transition from state two. The structure of the automaton facilitates the counting of un-spent tokens that have been received into the machine's bank and implicitly represents the numerical cost of pop. The counting depends only on the *"token"* and *"pop"* events. The *"refill"* event has no impact on this rule and is included in self-loop only to be unrestrictive. Note that this rule is actually the synchronous product of the two pieces of information it represents. Rule 1.1 is described by Figure 7.2 with the addition of the *"token"* event in self-loop at state two. Rule 1.2 is described by Figure 7.2 with the addition of the *"pop"* event in self-loop at state zero and transitioning from state one to zero.

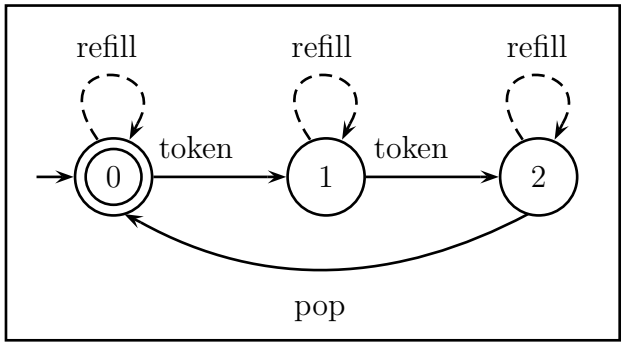

Figure 7.2: Rule: pop costs two tokens.

Figure 7.3 demonstrates the rule that the machine should not steal money. It contains two pieces of information that can be read as **Rule 2.1** *"don't receive tokens into the machine's bank when there is no pop in the machine"* and **Rule 2.2** *"don't attempt to deliver pop (i.e., drive the ejection device) when there is no pop in the*

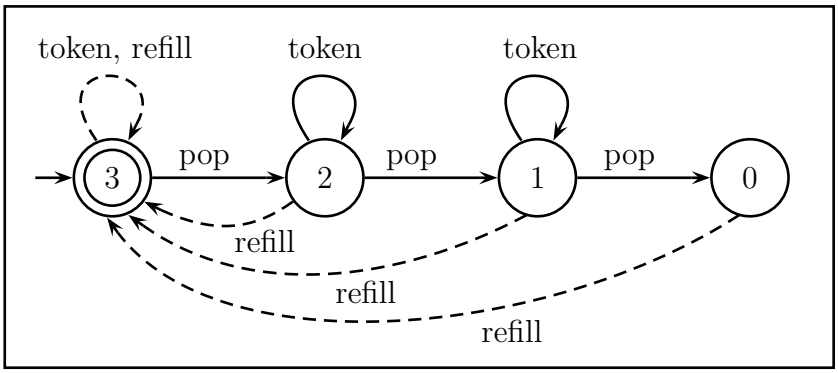

Figure 7.3: Rule: the machine should not steal money.

machine". The former is indicated by the lack of a *"token"* transition from state zero. The latter is indicated by the lack of a *"pop"* transition from state zero. The structure of the automaton facilitates the counting of remaining pop in the machine and implicitly represents the physical limitation that the machine can contain a maximum of three pop. The counting depends only on the *"pop"* and *"refill"* events. Note that this rule is actually the synchronous product of the two pieces of information it represents. Rule 2.1 is described by Figure 7.3 with the addition of the *"pop"* event in self-loop at state zero. Rule 2.2 is described by Figure 7.3 with the addition of the *"token"* event in self-loop at state zero.

The rules represented in Figures 7.2 and 7.3 communicate the desired behaviour of the machine (the legal specification). What then is the possible behaviour (the plant)? Considering the event space, the plant must be defined by Σ^* (a single state automaton with all events in self-loop). This definition seems unfortunately broad but follows from the fact that each event may by asynchronously, independently and uncontrollably initiated.

7.1.3 Monolithic Supervision

Assume that the control solution is to be achieved with a microcontroller. The uncontrollable refilling of the machine by a technician might cause a pulse on an input pin after its completion. In the modeling language, this single pulse maps to the generation of the *"refill"* event. The controllable events *"token"* and *"pop"* correspond to complicated series of actions on both input and output pins whenever the event is initiated. It very well may be that the microcontroller is required to do work even

when the event is not allowed to occur. In both cases, the series starts with uncontrollable signals on input pins and branch to carry out one of two possible series of actions on output pins. This means that each event corresponds to a single decision, namely whether to follow the *"yes"* branch or the *"no"* branch. It is this property that allows them to map to controllable events in the modeling language.

Considering the subsystems in this manner and with a perspective at the level of machine code on the microcontroller, one can imagine the system assembled, programmed and in place. Because events are assumed to be atomic, the main loop could simply poll a *"refill indicator"* sensor, a *"token inserted"* sensor, and a *"pop request button"* sensor, executing the appropriate subroutine whenever each occurred. Pseudocode for a general plant of this form is given in Listings 7.1 and 7.2. All the solution would lack is the answer to the *"yes/no"* questions when choosing a branch in the controllable event routines; which is to say, all it would lack is a supervisor.

Listing 7.1 Pseudocode snippet for a general plant

```
...
void main()
{
  doEvent0();
  doEvent1();
  ...
  doEventN();
}
```

Pseudocode for a general supervisor of this form is given in Listing 7.3. All events in the event space are initiated by external systems (human users and a human technician causing signals to appear on the input pins) so there is no question as to how the microcontroller (viewed as the plant) should generate events. It is the manner in which the event space is defined that allows the DES control theory to be applied in a straightforward way.

Listing 7.2 Pseudocode snippet for a general event

```
// # is an integer on [0..N]
void doEvent#()
{
  boolean initiated = testSensor#();  // test if this event is being initiated
  if (initiated)
  {
    boolean enabled = notifySupervisor(#);
    if (enabled)
    {
      // processing for "yes" branch
      // although uncontrollable events are always enabled, the supervisor still needs to be notified
    }
    else
    {
      // processing for "no" branch
    }
  }
}
```

Listing 7.3 Pseudocode snippet for a general supervisor

```
const NUM_EVENTS = #; // the number of events in the event space
int state = 0;        // global integer initialized to zero

// data in the form [state0],[state1],...,[stateM]
// where [stateI] = [event0_destination,event1_destination,...,eventN_destination]
// and where -1 implies disablement (i.e. no destination)
int[] data = {1,-1,0,2,-1,1,-1,3,2,4,-1,0,5,-1,1,-1,6,2,7,-1,0,8,-1,1,-1,9,2,-1,-1,0};

// event is the ID of the event that is notifying the supervisor (i.e. 0..N)
boolean notifySupervisor(int event)
{
  int next_state = data[state*NUM_EVENTS + event];  // get the requested data
  if (next_state == -1)
  {
    // disabled
    return false;
  }
  else
  {
    // enabled
    state = next_state;
    return true;
  }
}
```

The complete legal specification for the vending machine can be computed as the synchronous product of Figures 7.2 and 7.3 as shown in Figure 7.4, and since it is found

to be controllable with respect to the plant, it also serves as an implicit supervisor. A partial implementation of the plant and supervisor in assembly language for a PIC16F84 microcontroller is given in Appendix C as Listing C.1.

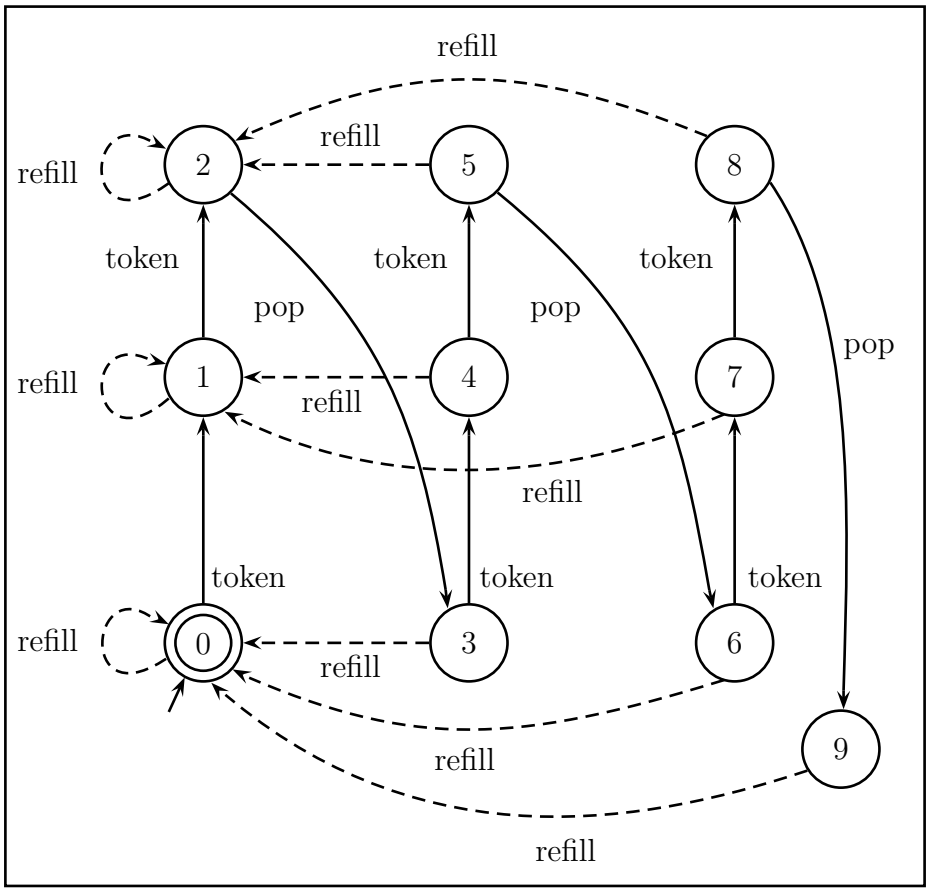

Figure 7.4: The complete legal specification (via synchronous product) which also serves as an implicit supervisor.

Concrete code solutions for the vending machine problem (such as those listed in Appendix C) can, to a considerable degree, be generated automatically given the event space (Table 7.1), plant (Σ^*), and legal specification (Figures 7.2 and 7.3). The only ad hoc components of the solution are the testing of the event initiation sensors and the content of the *"yes/no"* branches of the event subroutines (which are simply

the specific realization of the abstract event definitions).

7.1.4 Distributed Supervision

The current solution, however, can be improved upon by employing principles from distributed control theory to decrease the amount of data required to achieve supervision. The monolithic supervisor of Figure 7.4 was generated as the synchronous product of the set of rules that define the system. In distributed control, an event is disabled if any of a set of supervisors disables the event. Under full observation (as is the case) this policy is equivalent to the synchronous product operation. Therefore instead of precomputing a monolithic supervisor from the set of rules, each rule can be treated as a distributed supervisor.

Listing 7.4 Pseudocode snippet for a general event with distributed supervisors.

```
// # is an integer on [0..N]
void doEvent#()
{
  boolean initiated = testSensor#();  // test if this event is being initiated
  if (initiated)
  {
    boolean enabled = true;  // disable if any supervisor disables
    enabled = enabled && notifySupervisor1(#);
    enabled = enabled && notifySupervisor2(#);
    ...
    enabled = enabled && notifySupervisorX(#);
    if (enabled)
    {
      confirm();  // notify all supervisors that the event was not disabled by anyone
      // processing for "yes" branch
      // although uncontrollable events are always enabled, the supervisors still need to be notified
    }
    else
    {
      // processing for "no" branch
    }
  }
}
```

Pseudocode for a general event (which is part of the plant) that supports distributed supervision is given in Listing 7.4, and pseudocode for a group of distributed supervisors is given in Listing 7.5.

Listing 7.5 Pseudocode snippet for distributed supervisors

```
const NUM_EVENTS = #; // the number of events in the event space
int s1state = 0, s2state = 0, ..., sXstate = 0;
int next_s1state = 0, next_s2state = 0, ..., next_sXstate = 0;

// data in the form [state0],[state1],...,[stateM]
// where [stateI] = [event0_destination,event1_destination,...,eventN_destination]
// and where -1 implies disablement (i.e., no destination)

int[] s1data = {1,-1,0,2,-1,1,-1,0,2};
int[] s2data = {0,1,0,1,2,0,2,3,0,-1,-1,0};
...
int[] sXdata = {0,0,0};

boolean notifySupervisor1(int event)
{
  int next = s1data[s1state*NUM_EVENTS + event];  // get the requested data
  if (next == -1) { return false; }
  else { next_s1state = next; return true; }
}
boolean notifySupervisor2(int event)
{
  int next = s2data[s2state*NUM_EVENTS + event];  // get the requested data
  if (next == -1) { return false; }
  else { next_s2state = next; return true; }
}
...
boolean notifySupervisorX(int event)
{
  int next = sXdata[sXstate*NUM_EVENTS + event];  // get the requested data
  if (next == -1) { return false; }
  else { next_sXstate = next; return true; }
}

void confirm()
{
  s1state = next_s1state;
  s2state = next_s2state;
  ...
  sXstate = next_sXstate;
}
```

A partial implementation of the plant under distributed supervision in assembly language for a PIC16F84 microcontroller is given in Appendix C as Listing C.2. These listings demonstrate a savings in data complexity at a cost of increased source code

and runtime complexity. Specifically, let E be the size of the event space, let R be the number of rules, let S_i be the number of states in rule i, and let K_n be constants. Then monolithic supervision has a data complexity of $O(S_1 \times S_2 \times ... \times S_R)$ while distributed supervision has a data complexity of $O(S_1 + S_2 + ... + S_R)$. For example, in a five rule system, where each rule has five states, distributed supervision requires 25 units of data whereas monolithic supervision could require 3125 units of data. This gain comes at a cost of $E \times (R \times K_1 + K_2)$ instructions which both consumes program memory and increases runtime. Specifically, under monolithic supervision an event decision is $O(1)$, versus $O(R)$ under distributed control. While these are both constant time, the difference is real and can have an important impact on systems where runtime is vastly more important than storage space.

In the simple vending machine application as detailed in the assembly listings of Appendix C, the overhead of distributed supervision (in increased bytes of program memory) is not worth the gain; however in a system with more rules the opposite could easily be the case. Furthermore, the relative overhead could be reduced by optimizing the implementation of the separate supervisory subroutines as a single indexed routine.

7.1.5 Integrated Supervision

In an effort to further reduce the complexity of the final solution, consider a general supervisor. It has two main components: its structure which keeps track of important state information about the plant and its transitions which determine the actions allowable in various system states. In the case of the vending machine system, that information could alternatively be contained in two system variables (un-spent tokens

and available pop) and the four sub-rules previously defined. This is obvious in the ad hoc solution in Listing C.3 of Appendix C, which is notably less complex than the automatically generated solutions. In this solution, each event routine (doPop, doToken, doRefill) is responsible for accurately updating the system variables (tokens, pops), and the controllable routines (doPop, doToken) are further responsible to test all applicable rules (from 1.1, 1.2, 2.1, 2.2) in order to decide whether to process the *"yes"* or *"no"* branch. From this perspective, the ad hoc solution seems to be an integrated version of the plant/supervisor solution. In fact, by adding information to the "rules" or "supervisors" and limiting the scope of the modeling language, integrated solutions such as the ad hoc solution in Listing C.3 of Appendix C can also be automatically generated.

Consider the following general restriction: all integer system variables affected by events from Σ must be defined along with their initial values, as must the effect each element of Σ has on each system variable. Compliance to this restriction for the vending machine system is given in Tables 7.2 and 7.3.

System Variable	Initial Value	Description
"tokens"	0	The number of tokens received into the machine's bank since the last pop was dispensed.
"pops"	3	The number of pops currently in the vending machine.

Table 7.2: System variables of the vending machine.

Symbol	Impact on tokens	Impact on pops
"token"	tokens = tokens + 1	no change
"pop"	tokens = tokens - 2	pops = pops -1
"refill"	no change	pops = 3

Table 7.3: Events impact on system variables.

Listing 7.6 Pseudocode algorithm for generation of an integrated solution.

```
        // initialization code ////////////////////////////////////////////////
        for each system_variable
output:   system_variable = initial_value  // from Table 7.2
        next system_variable

        // main code //////////////////////////////////////////////////////////
output: loop
output: {
          for each event
output:     doEvent
          next event
output: }

        // event code /////////////////////////////////////////////////////////
        // need to compute state values
        for each rule
          initial_state.value = rule.affected_variable.initial_value
          for each remaining_state
            remaining_state.value = path_from_initial_state.value
            // any path will do  since we assume the fsm is designed to represent single values in each state
            // when calculating the value you consider only modifications to the affected_variable
            // by each transition on the path
          next remaining_state
        next rule

        for each event
output:   doEvent
output:   {
output:     if (isInitiated())  // this is not automated in any of the solutions
output:     {
              condition_list = nothing
              for each rule
                for each state
                  if (state.disables(event))
                  {
                    condition_list.add(rule.affected_variable != state.value)
                  }
                next state
              next rule

output:       if (condition_list)
output:       {
                for each system_variable
                  if event.hasModificationFor(system_variable)  // from Table 7.3
                  {
output:             system_variable = modification
                  }
                next system_variable

output:         // yes branch, do custom work.
output:       }
output:       else
output:       {
output:         // no branch, do custom work.
output:       }
output:     }
output:   }
        next event
```

Does recording this information actually constitute more work for the designer? Is it even possible to define the rules of Figures 7.2 and 7.3 without being conscious of this information. I believe it is not. The only remaining information necessary to automatically generate an integrated solution is to specify the relation of the structure of the rules to the system variables. Consider the following general restriction: the state structure of each rule must correspond to exactly one system variable and that variable must be noted. The monolithic legal specification does not comply with this restriction but its components (Figures 7.2 and 7.3) do, and are associated with system variables **tokens** and **pops**, respectively.

With these restrictions in place and the additional information collected, an integrated solution can be automatically generated by the algorithm in Listing 7.6. Integrated supervision has only been demonstrated to apply to systems under full observation whose legal specifications are controllable and can be represented by modular automata that each correspond to exactly one integer system variable. While this is only a subset of the systems to which DES control theory may be applied, many real systems do fall into this set. Furthermore, it may be possible to expand the use of integrated supervision to other classes of systems.

In the concrete code examples of Appendix C, the system under monolithic supervision required 78 words (for program and data) while distributed supervision required 108 words and integrated supervision required only 33 words. When considering scaling, one must consider an increasing event space, an increasing rule set, and (for monolithic and distributed supervision) an increasing data requirement to represent the supervision logic.

The runtime of distributed and integrated supervision increases linearly with the

number of rules but is constant under monolithic supervision. Conversely, the data requirement of integrated supervision increases linearly with the number of rules while the data requirement of distributed supervision increases with the sum of the sizes of the rules, and monolithic supervision increases with the product of the sizes of the rules. This implies that for complex systems, only distributed or integrated supervision provide a viable solution, and in any context where it is possible, integrated supervision provides the superior solution.

Finite-State Machines with Parameters

The benefit of the "integrated" methodology is derived from the replacement of state structure with integer values. A more general approach to this same optimization is presented in [14]. The "integrated" approach recognizes that for many low-level systems, the set of legal specifications (and hence modular supervisors) often corresponds to the recording of single integer system variables. The methodology exploits this observation by eliminating the state structure in exchange for appropriately manipulated parameters.

In [14], they note that "modeling in finite-state machines has long suffered the potential problem of state explosion that renders it unsuitable for many practical applications." They propose modeling finite-state machines with parameters. Specifically, they append finite sets of parameters to FSMs and show with examples how systems can be modeled efficiently with their approach while mitigating the problem of state-space explosion.

They demonstrate only the modeling of plants, and show how to perform basic operations (such as synchronous product) with their new structures. Their approach

allows integer counting in FSMs to be much more efficiently modeled, thereby eliminating exactly the property that is exploited by the "integrated" approach. The work in [14] suggests that all DES control theory be adapted to support this new core structure of FSMs with parameters. Certainly, such advances would improve DES control theory.

In the meantime, the action of the "integrated" approach is actually quite different than the ideas proposed in [14]. Specifically, instead of increasing the overall efficiency of modeling and computation within DES control theory, the "integrated" approach exploits a property of a class of solutions such that the specific, concrete realization of the solution, is implemented in a more efficient manner.

7.2 Application to Other Systems

The methodology here described works well for the vending machine system, and it has been shown that even at this scale, DES control theory can compete with ad hoc methodologies as an effective solution. One wonders how well these strategies can be applied to other problems. These methodologies and their implications are listed and categorized in Chapter 8, but concrete examples for various systems are briefly provided here to help convince the reader that the necessary properties are not unique to the vending machine system.

7.2.1 Resource Management

Reconsider the switches and locks scenario of Chapter 6 reproduced here as Figure 7.5. By the new methodology, the controllable outputs should be tied to uncontrollable

inputs. An appropriate event space is given in Table 7.4. Note that in the event definitions, initiating signals are clearly defined as are all assumptions restricting the behaviour of the other entities in the system (namely the users). A further requirement for the automatic generation of the integrated solution is the specification of system variables and their relationship to the event space. These are provided in Tables 7.5 and 7.6.

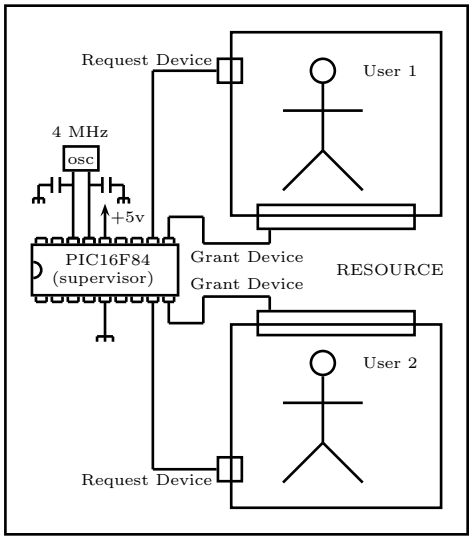

Figure 7.5: Two human users and a resource with a machine supervisor.

Symbol	Description
"g_1"	**controllable.** User #1 has moved the switch to the request position (switch1 = 1), and the grant device has been adjusted to allow access to the resource (lock1 = 0). Until lock1 is set to zero, this event has not occurred. While switch1 = 1, this event is continuously being initiated. Recall that once a user requests the resource (switch = 1) it is assumed they do not cease to request the resource (switch = 0) until they have been granted the resource (lock = 0)
"g_2"	**controllable.** User #2 has moved the switch to the request position (switch2 = 1), and the grant device has been adjusted to allow access to the resource (lock2 = 0). Until lock2 is set to zero, this event has not occurred. While switch2 = 1, this event is continuously being initiated.
"a_1"	**uncontrollable.** User #1 has moved the switch to the abandon position (switch1 = 0), and the grant device has been adjusted to prohibit access to the resource (lock1 = 1). This event has been arbitrarily labeled uncontrollable to indicate that its initiation should never be ignored. Assume that the microcontroller is always able to adjust the lock before any human is able to modify the system in any way.
"a_2"	**uncontrollable.** User #2 has moved the switch to the abandon position (switch2 = 0), and the grant device has been adjusted to prohibit access to the resource (lock2 = 1).

Table 7.4: Events in the switches and locks resource management scenario.

System Variable	Initial Value	Description
"owner"	0	The current owner of the resource. Zero implies no owner.
"lock1"	1	The output controlling the grant device for user #1. Zero implies unlocked.
"lock2"	1	The output controlling the grant device for user #2. Zero implies unlocked.

Table 7.5: System variables of the resource management system.

Symbol	Impact on owner	Impact on lock1	Impact on lock2
"g_1"	owner = 1	lock1 = 0	no change
"g_2"	owner = 2	no change	lock2 = 0
"a_1"	owner = 0	lock1 = 1	no change
"a_2"	owner = 0	no change	lock2 = 1

Table 7.6: Events impact on system variables of the resource management system.

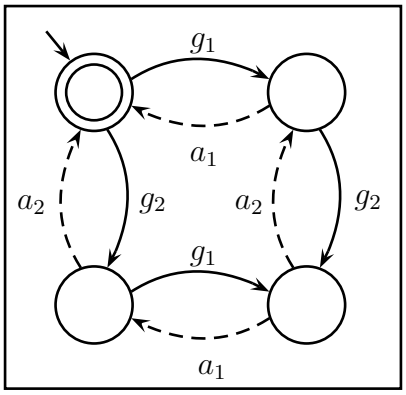

Figure 7.6: Resource management plant model.

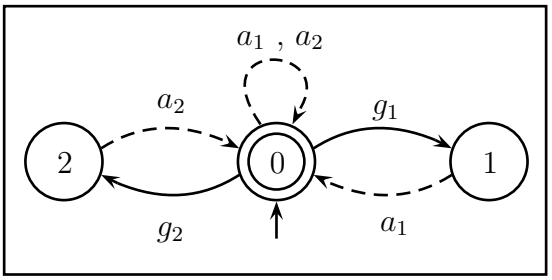

Figure 7.7: Rule: mutual exclusion.

With the event space fully defined, it remains to model the plant and the control objectives. An appropriate plant is given in Figure 7.6. This plant is more restrictive than Σ^* because of the restricting assumptions placed on the human users. The control objectives for this system were mutual exclusion and fair usage. The latter is

guaranteed simply by the definition of the event space, and the former is expressed as a rule in Figure 7.7.

Listing 7.7 Pseudocode for the automatically generated solution.

```
owner = 0; lock1 = 1; lock2 = 1;

loop { doGrant1(); doGrant2(); doAbandon1(); doAbandon2(); }

doGrant1()
{
  if (isInitiated()) // switch1 = 1
  {
    if (owner != 1 && owner != 2)
    { owner = 1; lock1 = 0; // yes branch, do custom work. none in this case. }
    else { // no branch, do custom work. none in this case. }
  }
}

doGrant2()
{
  if (isInitiated()) // switch2 = 1
  {
    if (owner != 1 && owner != 2)
    { owner = 2; lock2 = 0; // yes branch, do custom work. none in this case. }
    else { // no branch, do custom work. none in this case. }
  }
}

doAbandon1()
{
  if (isInitiated()) // switch1 = 0
  {
    if (true) { owner = 0; lock1 = 1; // yes branch, do custom work. none in this case. }
    else { // no branch, do custom work. none in this case. }
  }
}

doAbandon2()
{
  if (isInitiated()) // switch2 = 0
  {
    if (true) { owner = 0; lock2 = 1; // yes branch, do custom work. none in this case. }
    else { // no branch, do custom work. none in this case. }
  }
}
```

Recall that for automatic generation of the integrated solution each rule must correspond to exactly one system variable. This rule corresponds to the **owner** variable and its values are shown in each state. Since this rule is found to be controllable

with respect to the plant, it (along with the plant and the fully defined event space) can serve as valid input to the integrated solution generation algorithm. Pseudocode output according to the algorithm is given in Listing 7.7. This demonstrates that the necessary restrictions in this methodology for automatic integrated solutions admit systems quite different from the vending machine system, thereby suggesting that the restrictions are not severe.

7.2.2 LEGO Transporter

Recall the abstract Transporter of Chapter 5 reproduced here as Figure 7.8. Under the new methodology, the controllable outputs should be associated with initiating inputs. Recall that the goal of this system is to transport the payload from point A to point B and back to point A. While the sensor inputs were considered to be uncontrollable events, they are—from another point of view—merely the predictable result of the controllable motor output.

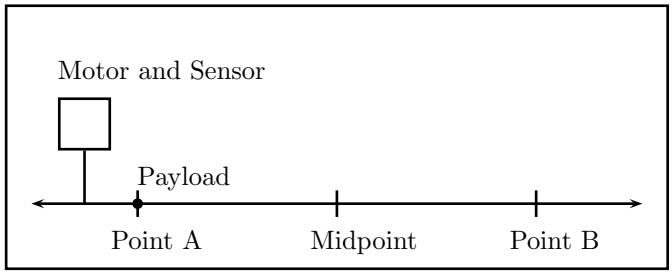

Figure 7.8: An abstract view of a transporter with a payload at point A.

From the point of view of Chapter 5, the new methodology cannot be applied to this system. Consider, however, what was actually achieved in the LEGO model. The designer expresses a desire for a transporter module that can move a payload from A to B and then return to point A. The system was modeled to include the motor commands *"forward"*, *"reverse"*, *"stop"*, and to report sensor conditions that could be interpreted as *"rotations = A"*, *"rotations = M"*, *"rotations = B"*. After employing DES control theory, the designer in an *ad hoc* manner determines from the

result that *"forward"*, *"rotations = M"*, *"rotations = B"*, *"stop"*, *"reverse"*, *"rotations = M"*, *"rotations = A"*, *"stop"* will achieve the goal. One might argue that this is obvious from the requirement. Note that this ad hoc process also removes extraneous but non-damaging behaviour that could have provided flexibility for differing future goals.

Symbol	Description
"goA"	**uncontrollable.** Initiated when an external component generates a pulse on the goA input pin, this event has occurred when the motor has begun to drive the payload to the A position.
"goM"	**uncontrollable.** Initiated when an external component generates a pulse on the goM input pin, this event has occurred when the motor has begun to drive the payload to the M position.
"goB"	**uncontrollable.** Initiated when an external component generates a pulse on the goB input pin, this event has occurred when the motor has begun to drive the payload to the B position.
	Assumptions and Notes
	Only one external component interfaces with this device, so the signals on goA, goM, goB and busy cannot suffer confusion of origin.
	The external component will never initiate goA, goM, goB while busy = 1.
	Pulses on goA, goM, goB input pins are never missed by the transporter.
	The external component viewed under DES control theory interprets a zero on the busy pin as an uncontrollable event indicating that the previous event (goA, goM, goB) has achieved the desired result.
	There is a sufficient time delay between the external component initiating one of goA, goM, goB and its testing of the busy pin such that the transporter can always set busy=1 before it is tested.

Table 7.7: Events for a reusable transporter.

From another point of view, a more advantageous model can be generated. Instead of designing for a specific task, consider designing the transporter as a general purpose tool. Tasks for the tool might include transportation to various positions. Assume that the transporter is implemented with a microcontroller, with an output pin attached to the motor, an input pin attached to the sensor, and for simplicity,

three input pins (goA, goM, goB) and one output pin (busy) attached to other modules in the system. An appropriate event space and extended event information for such a system is given in Tables 7.7, 7.8 and 7.9.

System Variable	Initial Value	Description
"position"	0	The current position of the platform. A=0, M=1, B=2.
"busy"	0	The current value of the busy pin.

Table 7.8: System Variables of the transporter component.

Symbol	Impact on position	Impact on busy
"goA"	position = 0	busy = 1
"goM"	position = 1	busy = 1
"goB"	position = 2	busy = 1

Table 7.9: Events impact on system variables of the transporter component.

By the event definitions, the occurrence of an event correlates to the beginning of the driving of the motor. Consequently, these events cannot occur when the payload is already at the destination. This physical limitation is captured in the plant model of Figure 7.9. This limitation raises the concern that the external component may not be aware of the limitation and may initiate the events at illegal states. This is solved by interpreting Figure 7.9 as both the plant and the legal specification, and noting that its structure corresponds to the position variable. Since the plant cannot generate the illegal events, the specification is technically controllable with respect to the plant and automatic solution generation can commence. The result produced by following the

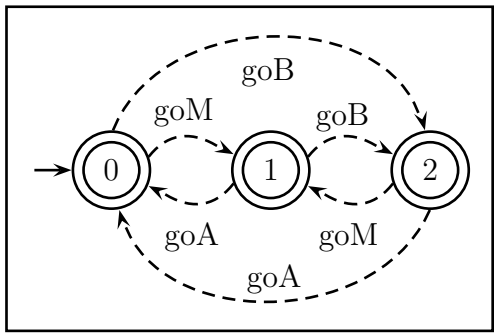

Figure 7.9: Plant: the payload can't move to the position it is already at.

integrated controller algorithm is shown in Listing 7.8. Note that while the result effectively ignores initiated *"goA"*, *"goM"* and *"goB"* when already at the specified destination, this is transparent to the external component because it simply requests an action, then monitors the busy pin until it reads zero. Since the busy pin was at zero before the request, this exchange seems valid to the external component.

To increase efficiency, the transporter and external component could alternatively be realized on the same chip by replacing the pins with memory locations and toggling between their main loops. Further note that in an all-encompassing point of view, one might wish to perceive events *"goA"*, *"goM"*, *"goB"* as composite transitions, initiated by the external device, remaining in a temporary busy state, and terminating when the rotational sensor indicates the destination is reached and the busy pin is returned to zero. While such a model is correct, the current model is also correct and provides a more efficient solution for code generation for the transporter component.

Finally, note that Listing 7.8 contains some custom code as required by the assumptions defined in the event space. While this must be added in an ad hoc manner, it parallels the ad hoc control of the token acceptance device in the vending machine system. Furthermore, it is left to the designer to provide ad hoc code to conditionally drive the motor forward or reverse in the *"yes"* branch of *"goM"*. While these constraints add a burden to the human designer, it is arguably no more a burden than that imposed by arbitrarily extracting a control pattern from a supervised plant as was suggested in Chapter 5.

While this example proves to be the least elegant of the three, it demonstrates that the necessary restrictions in this methodology for automatic integrated solutions admit a variety of systems, and by the existence of these demonstrates the usefulness

of the approach.

Listing 7.8 Pseudocode for the automatically generated solution.

```
position = 0; busy = 0;

loop { doGoA(); doGoM(); doGoB(); }

doGoA()
{
  if (isInitiated()) // pulse on goA pin
  {
    if (position != 0)
    {
      busy = 1; position = 0; // some ad hoc code would go before the modification to position.
      // yes branch, do custom work. (go reverse and count sensor rotations until A is reached).
      // important to set busy = 0 when finished as specified in the event space assumptions.
      }
    else { // no branch, do custom work. none in this case. }
  }
}

doGoM()
{
  if (isInitiated()) // pulse on goM pin
  {
    if (position != 1)
    {
      busy = 1; position = 1; // some ad hoc code would go before the modification to position.
      // yes branch, do custom work. (go forward or reverse based on previous state and monitor the sensor).
      // important to set busy = 0 when finished as specified in the event space assumptions.
      }
    else { // no branch, do custom work. none in this case. }
  }
}

doGoB()
{
  if (isInitiated()) // pulse on goB pin
  {
    if (position != 2)
    {
      busy = 1; position = 2; // some ad hoc code would go before the modification to position.
      // yes branch, do custom work. (go forward and count sensor rotations until B is reached).
      // important to set busy = 0 when finished as specified in the event space assumptions.
      }
    else { // no branch, do custom work. none in this case. }
  }
}
```

Chapter 8

Classifications

In the preceding analysis, properties of various systems have been noted and exploited in the methodologies employed. Here these properties and methodologies are summarized. Each methodology has advantages and disadvantages which are noted, as are the preferential system properties for each.

8.1 System Properties

Immutable Plant

An immutable plant is assumed to exist independent of the designer. It is fully defined and cannot be augmented in any way. In this circumstance the plant must admit observation and control in the classical closed-loop form, otherwise DES control theory cannot be applied. An immutable plant is a general assumption of classical DES control theory, and may in the future be the dominant form in high-level systems. As DES control theory comes into popular use, it is reasonable to expect that systems may be constructed for the purpose of functioning under DES control, and therefore

can be expected to conform to the classical interface. When employing DES control theory on an immutable plant, the definition of the event-space is fixed and therefore obvious.

Augmentable Plant

A lesser restriction than the immutable plant, it may be the case that the plant exists, is fully defined but can be augmented in various ways. Optionally, such a plant may be considered immutable, but the ability to augment the plant provides an avenue for optimization. Specifically, if components of the legal requirement are found to be uncontrollable, ad hoc or formal analysis may be employed to determine what changes to the plant are necessary in order to achieve the requirement, and what cost this entails. There has been little investigation into this topic, likely because of the usability and complexity issues still unsolved in the core theory. When employing DES control theory on an augmentable plant, the event-space is not fixed. This admits the possibility of optimization but is an added burden on the human designer.

Decoupled Plant

A decoupled plant can be described independent of supervision and control objectives. Whether it exists or is yet to be manufactured, this implies that such a system can reasonably function without the presence of supervisory control. These systems can be said to be decoupled from the control objectives. This corresponds with the primary assumption that events are generated by the plant. In highly coupled plants, it is unreasonable to ignore the causality between events, which conflicts with the primary assumption that events occur spontaneously. Some systems (such as the cat

and mouse maze) are obviously decoupled, while others (generally incomplete plants) intuitively appear highly coupled to control objectives. By redefining the event-space according to the initiated-event methodology such systems can be made to appear decoupled. For this reason, decoupled plants are defined to be those systems that are obviously and intuitively decoupled. In general, high-level decision-making systems are obviously decoupled, while low-level control implementations tend to be highly coupled.

Incomplete Plant

Generally, the problem of a coupled plant occurs because the plant is intuitively incomplete. Certain systems simply don't exist without the influence of control. In low-level control, such as with systems directed by a microcontroller, various intuitive events are associated with the supervisor rather than with the plant. This conflicts with the primary assumption that events are generated by the plant. Because of the coupling of these events to the control objectives, it is unreasonable to associate them with a system functioning without control. Similarly since the microcontroller can't be programmed until the supervisor is computed, these events can't even occur. Consequently, the plant is said to be incomplete.

Implied Solution

Some specifications imply a solution. This is usually the case in coupled or incomplete plants. For the LEGO and vending machine examples, this was the case. For example "pop costs two tokens" implies that a machine should allow two tokens, allow a pop, repeat. Merely stating the problem implies a solution. This is not the case with other

systems such as the cat and mouse maze. The specification "mutual exclusion" against an arbitrary maze with hundreds of rooms is nontrivial for a human to solve in an ad hoc manner, yet is trivially solved by DES control theory. In systems with an implied solution, many methodologies can lead to a DES solution that is more complicated, more work and more error-prone than a solution achieved by other methods.

8.2 Design Ideologies

Physical Disablement

The means of supervisory disablement is not addressed by standard DES control theory. One might assume that the supervisory entity is capable of physically altering the plant such that various events cannot occur. This approach is troublesome. The event set can be arbitrarily large and the disablement feedback map can arbitrarily change as the supervisor changes state. Considering this, and the fact that physical changes consume real-time, one must accept that the application of the disablement feedback map is not instantaneous and, in fact, may consume an arbitrarily large amount of time. The idea of physical disablement by the supervisor is infeasible for many systems. As stated previously, DES control theory is best suited for high-level decision making, and of all the systems researched for this dissertation, none employ this ideology.

Blanket Feedback

In the middle ground between physical disablement and query/response is blanket feedback. As the supervisor changes state it communicates a feedback map to the

plant. The plant is then responsible to not generate any events disabled by the new feedback map. Disregarding the possibility of physical disablement, this requires that all controllable events are controllably generated, meaning that once initiated, an event can be optionally forced to not occur. With an appropriately defined event space, nearly all systems can be described with controllably generated events, although this does restrict the expression of a given system.

Consider a chemical plant that contains a tank with a varying amount of liquid. It may be advantageous to model *"filled to eighty liters"* as a controllable event. If the system changes state such that this event should be disabled, the input that fills the tank can be forced closed. This is control by disablement and corresponds to the classical theory. Since the liquid level could be arbitrarily near the eighty liter mark when the system changes state, it may be too late to disable the event. Such an event is uncontrollably generated but controllably disabled, and cannot be modeled as a controllable event unless it is guaranteed that the plant will always have sufficient warning before the event occurs.

Query/Response

Among the systems examined, this is the most popular and most efficient ideology. Instead of realizing the classic closed loop, one may adopt a query/response system where the plant asks permission to generate an event and the supervisor responds with "yes" or "no". Similarly the plant may ask the supervisor for a set of enabled events, of which it will generate one. Both of these strategies were employed in the DESCO examples. The fundamental property of this ideology is that controllable events are controllably generated, not controllably disabled. This generally requires

obedient components within the system and an existing but imperfect control entity that essentially intermittently asks the DES supervisor for advice.

Consider, for example, the cat and mouse maze. Because the mouse may decide to sleep in a doorway, it is difficult to impose control by physical disablement. Fortunately the system is a metaphor for resource management which could, for example, be implemented among two obedient software processes. In such a system, the "mouse" software process will first desire access to a resource, second receive permission and finally possess the resource. The supervisor (also a software process) may simply check its feedback array indexed by its current state. In fact, in a system with shared data, communication can be greatly reduced. The mouse process may read the supervisor's feedback array indexed by the supervisor's current state; thereby eliminating the query/response. In a sense this is a blanket feedback system where the plant possesses a copy of the supervisor's feedback array, and only the supervisor's current state need be communicated.

8.3 Design Methodologies

8.3.1 Standard

The standard methodology is best for systems with a **complete** or **decoupled** plant and assumes, but does not require, an **immutable** plant. This methodology is very difficult to use on an **incomplete** or **coupled** plant because control requires event generation as well as disablement. If the system has an **augmentable** plant, efficiency advantages may be gained by employing alternate methodologies such as integration. In all cases, the gains achieved by DES control theory are most pronounced when

there is not an **implied solution**, and this is to some degree an assumption of the standard methodology. The cat and mouse maze is an excellent candidate for this methodology, and the work with DESCO is a clear example of its application.

Method

The plant is modeled as an automaton or a set of modular automata. In the case of an **immutable plant** this is a straightforward representation of existing behaviour. The legal requirements are modeled as a set of automata. The definition of the legal specification is the most error-prone phase of this method. A monolithic supervisor, or a set of modular supervisors are automatically generated using software such as is described in Chapter 9. Since the plant exists independent of supervisory control, a minimal-state supervisor is most efficient. Consequently, a supervisor reduction algorithm should be employed. Finally, feedback may be facilitated by whichever ideology is permitted by the plant. With this method, the reduced-state supervisor (or set of modular reduced-state supervisors) is, in fact, the control solution. Given a specification for its closed-loop interface to the plant, concrete implementations could be automatically generated for various programming languages.

8.3.2 Single Path

This is the methodology proposed by [13] and examined in Chapter 5. It can handle an **incomplete** or **coupled** plant and requires an **implied solution**. Supervisory control is not employed. Instead a single control path is achieved which aids a programmer in writing an implementation of a fixed system. In some cases the process is unreasonably complex and the generated solution is no less error-prone than one

generated by ad hoc methods. Without an **implied solution**, some requirements (such as "let the cat and mouse roam as freely as possible") cannot be achieved. By this method, an arbitrary single controllable path would be chosen from the generated supervisor, meaning that when the mouse enters a room in which multiple controllable doors can safely be opened, only one will be permitted. The single path follows directly from the **implied solution**. Furthermore, in many systems the ad hoc process of selecting the single path can become arbitrarily complex. Despite these drawbacks, this methodology can perform very well for certain systems, such as fixed-path manufacturing plants. Essentially the method helps discover which sequence of outputs is necessary to achieve a desired sequence of inputs.

Method

The plant is modeled as an automaton or a set of modular automata. It is assumed that the system does not exist prior to the generation of a control solution. The event space must be defined in accordance with the inputs and outputs of the implementation device. A moderate degree of event abstraction is employed. It is required that individual events roughly map to individual actions or conditional occurrences in the language of the implementation device. The plant is consequently defined as all the behaviour the implementation device is capable of observing and causing, limited by the definition of the event space. The event space can be seen as the interface between software and hardware, thereby reducing the scope from all possible programs (with which the implementation device could be loaded) to only those that can be described as sequences from the event space. While this could result in a single state plant, the actual behaviour may be further limited by causality relationships between certain

events. Specifically, because of the interconnections of the hardware, an input event may be the direct result of a series of output events, and therefore may only occur after certain sequences. The event space definition and analysis of possible sequences leads to the plant model(s).

The legal requirement is first modeled as a set of safety requirements which only remove destructive behaviour. This can be accomplished by examining the plant model(s) and deleting states or transitions that damage the system. These very likely exist because the plant as defined serves no purpose. It only describes all possible programs (restricted by the interface of the event space) that could be loaded onto the implementation device. This property follows from the assumption that the system does not exist previous to the generation of a control solution. In consequence, the plant restricted by the safety specification still does not serve any purpose. This is solved by the generation of a progress specification, which communicates the job that the system should perform. This can be modeled as a single automaton that connects a sequence of uncontrollable input events separated by states with all controllable output events in self-loop. This requires that the human designer knows the specific string of input events that indicates the proper functioning of the machine.

With the plant and legal models defined, automatic computation of a supervisor can be performed. This can be achieved by first computing a monolithic plant and monolithic legal specification, but this may represent too vast a computational task. A method for achieving a supervisor without computing the monolithic models is detailed in [13]. Once the supervisor has been obtained, the human designer must (by an ad hoc algorithm) remove controllable events such that no state has more than one outgoing controllable event. This process may be arbitrarily complex. Next, the

human designer must (by an ad hoc algorithm) translate the remaining structure into the control language of the implementation device. Because the events were made to roughly correspond to single actions or conditions within the implementation device, this last step should not be overly complex.

8.3.3 Initiated-Event

All three variants of this methodology were demonstrated with the vending machine system in Chapter 7. This methodology and its variants were developed specifically to handle an **incomplete** or **coupled** plant and to maintain the usefulness of DES control theory even in the face of an **implied solution**. Based on these system properties, it is best for low-level control. Furthermore, while it is possible to use these methods without an **implied solution** or **coupled** plant, they should be expected to perform poorly. It would not make sense to employ these methodologies without an **incomplete** plant. All three variants demonstrate automatic code generation for the PIC16F84 microcontroller. It is believed that these methods could be used to automatically produce solutions for any implementation device. The integrated variant further helps to automatically generate solutions that compete with correct ad hoc solutions in terms of data requirement and runtime efficiency. The integrated variant was demonstrated on three systems in Chapter 7.

All three methods use the same core, while "modular" and "integrated" attempt to increase the efficiency of the solution. In a system with a large event space and a large number of rules "modular" can be expected to outperform "monolithic" in data requirement at a slight cost of runtime efficiency. For a restricted set of systems, and increased design time, "integrated" should significantly outperform both of the

others.

The fundamental strategy of these methodologies is in the event-space definition and employs the query/response ideology. Inputs, outputs and causality are abstracted into single events. All controllable events must be associated with some initiating input and some generated output. Generally output will be required regardless of whether an event is enabled or disabled. The enablement decision simply indicates which branch of output to process. Inputs not directly associated with outputs should be modeled as uncontrollable events. This abstraction defers complexity to ad-hoc components of the final solution, but does not represent a reduction of automation any greater than that of comparable methods. With the event-space defined in this manner, the plant may often be limited to a single-state automaton. It may be given structure based on assumptions applied to the inputs. Often this can only be achieved by placing restricting assumptions on other entities that interact with the system.

Monolithic Method

First the event-space must be defined according to the "initiated-event" ideology. This is the most important and most error-prone phase of the process. Differing perspectives in this phase yield results differing widely in complexity and re-usability. With the event space defined, the plant automaton should be forthcoming. It is important to not unwittingly include legal requirements in the plant. In the "initiated-event" ideology, all events are initiated by uncontrollable inputs; consequently the possible behaviour (i.e., the plant) can only be restricted by assumptions placed on those inputs (i.e., assumptions in the event definitions). A sequence of events is only

impossible if the event definitions make it impossible, regardless of its irrelevance to the system objectives.

Second, the legal requirement should be specified as a set of low-level rules. By creating modular automata that each express only a single component of the legal requirement, the automation of DES control theory can be maximized. If one instead completely models the **implied solution** (which can be a considerably taxing job) then too much complexity is consumed by its ad hoc definition, and the gains of DES control theory may be lost.

With the event-space, the plant and the legal modules fully defined, generation of a solution can commence. In the monolithic method, a supervisor is generated from the synchronous product of the legal modules. The automatic code generation creates a representation of this supervisor and a subroutine for each event in the event space. The plant code simply calls each event routine in sequence. Each event routine tests for its initiating conditions (which requires ad hoc code). If initiated, an event queries the supervisor. Based on the response, it processes the "yes" or "no" branch of its routine (which have ad hoc contents).

The only ad hoc components of the automatically generated final solution are exactly the specific realizations of the abstract event definitions. It has not been investigated, but it is believed that using some variant of hierarchical DES control theory, these definitions may themselves be automatically generated.

Modular Method

The modular method provides a decreased data requirement at a slight cost of run-time efficiency. It is in almost every respect identical to the monolithic method.

When computing the supervisor, instead of taking the synchronous product of the legal automata, each is transformed into a modular supervisor. This slightly impacts the implementation of the query/response ideology and increases the runtime proportional to the number of supervisors. The decrease in data requirement can be significant, namely the sum of the state-sizes of the rules versus the product of the state-sizes of the rules.

Integrated Method

The integrated method attempts to compete with ad hoc methods by dissolving the distinction between plant and supervisor. This method requires extra information at design time and restricts the possible legal requirements. Integer system variables must be defined, along with their initial values and each event's impact on each variable. This implies that regardless of system state, the occurrence of an event must always affect each system variable in the same way. Furthermore, it is required that the state structure of each rule correspond to exactly one system variable, and that variable must be noted.

With this additional information (and the event-space, plant, and legal modules as input), an algorithm has been proposed to automatically compute a code solution. The solution takes the plant/event form of the previous solutions but integrates the supervisor logic into the event headers. Specifically, whenever an event is initiated, it tests various system variables according to the relevant rules defined by the legal modules. If the "yes" branch is taken it also updates relevant system variables. Consequently, the ad hoc component of the "integrated" method is the same as in the previous methods.

The automatic generation algorithm is able to determine which variables each event must test and how to test them. It is further able to determine how the variable should be updated. The suggested algorithm, requires that each state in each rule correspond to exactly one integer value. Furthermore, the rule testing compares against a set of values. This means that in the worst case, testing for an event's enablement/disablement condition could imply a number of actions equal to the sum of the number of states in all rules. It is believed that a more intelligent algorithm using intervals instead of single values could overcome both these difficulties.

8.4 Closing Remarks

With the theory and methodologies here provided, it is clearly possible to advantageously apply DES control theory to many disparate real systems. In particular, the initiated-event methodology (developed in this dissertation) allows for the advantageous application of DES control theory to partially defined programmable systems. It is these systems that have the most non-intuitive interface with the implementation of DES control theory. Despite this resolution, the practical application of DES control theory still cannot be achieved. Software aids are necessary for the specification, design and verification of DES components. Software is further necessary for the required computations and synthesis performed with these components in the generation of a supervisory solution. In the case of programmable systems, software is again needed to maximize automation in translation from the supervisory solution to an actual implementation in the language of the programmable device (such as the code generation algorithms suggested in Chapter 7). For these reasons, various software tools have been developed to help individuals employ DES control theory.

Several tools are reviewed in the next chapter with a focus on the IDES tool which was developed by this author. Currently there is little cooperation between the available DES software tools, and their interfaces are generally difficult to use and intended for experimentation rather than industrial use. A major goal of the IDES tool is the provision of an adaptable interface for the human specification of DES components. It is hoped that in the future the IDES tool will interface with various other DES related software tools.

Chapter 9

Software

9.1 Background

A number of software packages exist to aid researchers in the field of DES control theory. Foremost among these are CTCT [49] and UMDES [27]. These packages focus on the manipulation of automata and provide complex functions necessary in supervisory control. They are robust command-prompt applications and can generally handle arbitrarily large inputs. Both accept inputs in custom text-based formats and produce outputs in the same formats. The UMDES software is paired with a partner program GIDDES [39] which provides a graphical user interface and visualization of the automata. In [50], seven graph and automata manipulation applications (other than CTCT, UMDES and GIDDES) are evaluated in detail and requirements are suggested for an optimal interface for software designed to represent and manipulate automata. These, among others, are briefly reviewed in the following section.

9.1.1 Relevant Packages

CTCT [49]

The CTCT application is essentially a command-prompt tool that does not accept batch commands. Its purpose is to perform manipulations and computations on DES components specified in its own custom text-based format. It is a time-tested tool and is used by many researchers in the field. The authors of [20] provide an evaluation of the CTCT tool. They state that "the lack of a graphical metaphor in the representation of a DES ... is an unfortunate oversight ..." but confirm that "it is still a powerful and useful tool in the modeling of a DES." They further remark that the "product's usability and effectiveness are reduced due to a cluster of interface design problems" and claim that "a system is only as good as the user's ability to use it."

The CTCT application is a good engine for the manipulation of DES components, but it lacks any means of visualization, has a difficult user interface and does not provide any means to partner with a front-end display engine.

UMDES [27]

The UMDES application is a command-prompt tool that does accept batch commands. Its purpose is to perform manipulations and computations on DES components specified in its own custom text-based format. A graphical representation of the DES components given as input and generated as output is provided by GIDDES. The UMDES application is a time-tested tool and is used by many researchers in the field.

Being a command-prompt tool, UMDES shares some of CTCT's interface problems. While these are alleviated to some degree by GIDDES (a GUI front-end), it is not an ideal solution because input is form-based rather than graphical. Because UMDES does provide a means to partner with a front-end display engine it could be used as a component of any new tool.

GIDDES [39]

The GIDDES application is a GUI front-end for engines that manipulate DES components. The GIDDES tool is currently distributed with UMDES as a bundled package. The GIDDES tool is a Java GUI application that allows form-based creation of UMDES input files. It provides visualization of the specified automata using a component of GraphViz called Grappa [2] (an automatic graph layout program), and it provides access to the UMDES functions.

The GIDDES tool is a good front-end display engine, but does not allow custom modification to visualizations of DES components and requires from-based specification of DES components rather than graphical specification. The GIDDES application is currently tailored to UMDES and therefore does not yet serve as a GUI layer for arbitrary applications.

GraphViz [2]

GraphViz is an open source set of graph drawing tools for Unix and Windows. The mandate of the program is to find efficient drawing algorithms for use on graphs as large as several hundreds of nodes. The tool aims to make very readable drawings, approaching the quality of manual layouts. GraphViz's algorithms attempt to

automatically produce aesthetic graphs by favouring recognition and readability of individual objects; avoiding edge crossings, sharp bends and intersection of unrelated objects; and emphasizing symmetry, parallelism and regularity. The output is of very high quality. Simple, user-friendly installation is provided for multiple platforms and several separate tools are provided under the GraphViz umbrella.

While GraphViz itself cannot directly be used to specify or manipulate DES components, the automatic layout algorithms it provides would be an excellent choice to incorporate into any tool that needs to display the result of an operation (such as synchronous product) on DES components. The GIDDES software is an excellent example of such usage. The automatic layout algorithms should, however, only be used to determine the initial layout of a newly generated DES component. The human user should be able to make custom modifications to the result, and to specify components manually in a graphical, point-and-click manner.

DaVinci [11]

The daVinci [1] application is an interactive graph visualization system. It is different from conventional graph editors in that it is not able to alter the graph structure directly. It was designed to be used as a GUI layer for arbitrary applications. The connected program is exclusively responsible for controlling the graph structure, and the only task of daVinci is to display the graph on the screen. Of particular interest are daVinci's interactive properties such as manual improvement of the graph layout. It is possible to influence the generated graph layout directly at runtime, resulting in immediate feedback on the visualization.

The daVinci application is a good front-end display engine, for systems that have

[1] In February 2005 a new version of DaVinci was released under the name uDraw.

graphs as output. While it merges automatic layout with custom positioning, it does not allow the manual specification of graphs; furthermore its automatic layouts are not demonstrably superior to those of GraphViz. For these reasons, a front-end optimized for manual specification and modification of DES components, partnered with GraphViz for necessary automatic layouts, would provide a superior solution.

Graphlet [44]

The Graphlet application is a toolkit for graph editors and graph algorithms. It features Graphscript, which is an extensible programming language for graph algorithms with user interfaces. Compiled versions for several platforms and the source code itself are available upon request. The toolkit has an impressive GUI, and includes implementations of many standard graph manipulation and testing algorithms.

The Graphlet application is a good toolkit for visualization and automatic layout of arbitrary graphs. Its focus is on graph theory, including issues such as testing planarity and finding the shortest path between two nodes. While such features are important in computing automatic layouts, the tool is specialized in a direction away from that which is immediately applicable to the representation and manipulation of DES components. Again, the greatest contribution this tool could make to the DES effort is in automatic layout of graphs, and its results are not demonstrably superior to those of GraphViz.

VGJ [5]

Visualizing Graphs with Java (VGJ) is a graph drawing and layout tool. It accepts input and generates output in various text-based standards, and supports automatic

layout and organization of the graphs. It also allows graph specification by an intuitive point–and–click GUI. Unlike many other applications this tool allows the representation and visualization of graphs in three dimensional space.

The VGJ tool provides a means of specifying and manipulating the layout of 3D graphs. Its automatic layout capabilities are noticeably inferior to those of GraphViz, as is the quality of its visual output. The manual specification system lacks many features and has defects such as drawing overlapping edges between nodes. This tool would only be a viable option if 3D output was an essential requirement.

XFIG [51]

The Xfig application is a fully-featured, interactive drawing tool. In its vector drawing mode it can produce excellent visual representations of directed graphs that can be exported to various image formats.

The tool is not easy to learn and is not optimized for the specification of DES components. Furthermore there is no means of associating it with a DES manipulation engine such as CTCT or UMDES.

PSTricks [52]

PSTricks is a collection of PostScript-based TEX macros. It allows inline use of the majority of PostScript capabilities and consequently provides specification for the curves and shapes necessary to realize a graph. Like its TEX host, PSTricks is essentially a markup language but has an unforgiving syntax and its compiler errors are technical and unfriendly.

An advantage of this package is the tremendous amount of control the user has

over the display and formatting of the graph, but the learning curve for its use is daunting, and users not already experienced with a TEX package will likely find its complexity prohibitive. Like XFIG, there is no means of associating PSTricks with a DES manipulation engine; however, because of its ease of inclusion in research documents, it would be desirable for a DES software solution to be able to export DES components to the PSTricks format.

Wiese's Little Automata Builder [46]

This tool automatically generates a graphical display of an automaton given its definition as a regular language. It is programmed in Java and allows no user manipulation of the output but does draw high-quality visual output. It also achieves efficient use of space by forcing the nodes to a grid sequence.

This tool does not allow the manual, graphical specification of DES components, nor does it provide any means for an interface with a DES manipulation engine; nevertheless, several of the operations it performs could serve as inspiration in the generation of a fully-featured DES tool.

FSA6 [35]

The FSA6 application is a collection of utilities to construct, manipulate and visualize finite automata. Regular expressions specified by a simple text syntax serve as its input. It is an open-source project, but is difficult to compile and no installation package is provided. It includes an optional GUI only for use on the SICStus Prolog platform. Of particular note, however, is that FSA6 provides interfacing to graphViz, VCG, daVinci and PSTricks.

This tool does not allow the manual, graphical specification of DES components, nor does it provide any means for an interface with a DES manipulation engine; nevertheless, several of the operations it performs could serve as inspiration in the generation of a fully-featured DES tool.

9.2 IDES

After a survey of the relevant software, it is apparent that no fully-featured suite exists for beginning–to–end application of DES control theory. While many of the complex DES computations have been implemented in DES manipulation engines such as CTCT and UMDES, the human interaction with these tools is limited by their interface and hence error-prone. Furthermore, these manipulation engines provide little aid in the initial generation and design of the DES models to be used, and do not aid in the translation of a supervisory solution to any specific implementation. Having established that a need for DES software exists, we propose the following assumptions and requirements.

Assumptions

A1. The majority of humans employing DES control theory will first have a personal understanding of a system and second generate a plant that represents that system.

A2. The majority of humans generating a plant from a personal understanding of a system will draw it as a directed graph before recording it in any other format.

A3. The best software tool for a given human interaction task is one that allows a

human to enter correct inputs in the most comfortable and natural way possible, and that generates correct outputs in a timely manner and in a format easily understood by its intended audience.

Requirements

Based on these assumptions, an ideal modeling tool for DES control theory would provide the following features.

R1. The tool should allow a user to input DES components by drawing them as directed graphs on a medium like paper with a stylus, and to import DES components from various existing data formats.

R2. The tool should allow useful functions to be performed on the specified DES components (possibly generating new components) in a usable and effective manner.

R3. The tool should allow export of DES components to various third party formats.

R4. The tool should directly interface generated solutions with the systems on which they are based, or it should automatically produce software libraries that do the same.

In this context all existing tools fall short. Clearly the requirements are ambitious, albeit possible. The technology detailed in [17] and commercialized by E-Ink Corporation [18] could easily provide such an interface. The development and distribution of such a technology would, of course, be costly, and in the interim a standard GUI application on a personal computer controlled by a keyboard and mouse can suffice.

The challenge is to translate the hand-drawn analogy as smoothly as possible to a keyboard and mouse environment. The Integrated Discrete-Event System (IDES) software package that I have created attempts to partially solve this problem.

The IDES software is a multi-purpose modeling tool that accomplishes two goals. First it functions as an interface for specifying DES components in a manner as analogous as possible to pen and paper drawing. Second it demonstrates the integrated use of DES control theory with custom hardware components for research and pedagogical purposes. It is only a partial solution to the greater DES modeling problem. The primary contribution of the IDES tool is providing an interface analogous to pen and paper drawing.

When a user of DES control theory wishes to use a modeling tool, the input can only exist in one of three formats.

1. The DES components exists beforehand in some pre-existing computer data set, perhaps in a popular format such as that of CTCT or UMDES.

2. The DES components exists beforehand as some paper-based representation, possibly graphical.

3. The DES components exists only in the user's mind, possibly as vaguely defined as an understanding of a system.

The IDES tool was designed with the second and especially the third input possibilities in mind. The first input possibility is really an issue of backwards compatibility. Once an ideal modeling toolkit exists, all other formats would quickly fall out of use. Unfortunately, ideal situations usually only exist in theory, so any robust tool should support the import of other data formats. In the IDES software, this was left

for future work, and it therefore only partially solves the requirement R1.

Several existing packages such as CTCT and UMDES already solve requirement R2, albeit with a clumsy interface. The GIDDES tool already solves the problem of coordinating the use of the functions provided by such toolkits in a usable manner. For these reasons and due to time constraints no attempt was made to satisfy the requirement R2 in the IDES tool. It serves only as a usable and effective means of specifying DES components, and connections to the functions of CTCT and UMDES were left to future work.

The IDES software mainly allows export only to graphical formats; consequently, it only partially satisfies the requirement R3; however, it does interface with custom hardware via the RS232 protocol [19]. It allows monitoring and control of plants that support its communication method as described in the Appendix at Section A.7. In this sense it satisfies the requirement R4, but there is much room for improvement and future work.

9.2.1 Implementation Details and Availability

The IDES tool is a Java application composed of 57 classes and more than 17,000 lines of code. It was developed by Michael Wood and represents more than 500 hours of research and development time. It was produced in the Eclipse IDE [25], and its GUI is based on the SWT widgeting library [25]. While Java is by nature cross platform, because of the use of the SWT, native components are required. While the SWT has been demonstrated to function on a wide variety of platforms, the IDES toolkit only provides an installation package for the Windows platform. Installation packages for other platforms were left as future work. Other than the SWT, the

IDES software utilizes several other third party libraries, namely: Acme.zip (for GIF encoding), comm.jar (for access to the COM1 port), j2d4swt.jar (for use of Java2D inside the SWT), jimi.jar (for PNG encoding), and two custom classes written by Lenko Grigorov for access to GhostScript and MiKTeX.

The primary goal of the IDES software was to facilitate a highly usable interface for the definition of DES components by human users. This requires that the responsiveness of the tool not significantly drop as graph parts are added to the model and also implies an upper bound on reasonable graph sizes. The upper bound may be derived from the fact that graphs are presumed to be entered and manipulated entirely by human users, and at some point it must become unreasonably inefficient for a human to specify an arbitrarily large graph. This is supported by related projects examined in Chapter 3. The author of [29] suggests that a module of more than 300 states is prohibitively large and should be decomposed into smaller sub-models. Similarly in [13] no design module had more than thirty states, and the synthesized control solution had only fifty states. These examples and the continued advancement of modular and hierarchical DES control theory are strong motivation for an interface that is optimized for the specification of small models.

Figures 9.1 and 9.2 display the results of a responsiveness study of the IDES software on a Pentium 4, 3.00 GHz, CPU with 1 GB of RAM. The most complex addition or manipulation operation in the software that increases with graph complexity is the repositioning of a highly connected node. This operation was performed on graphs of increasing sizes and the response rate of the software was recorded. It is displayed in both milliseconds between updates and updates per second. The results show that lag is independent of graph complexity until there are at least 160 objects in

the graph. This implies that the computation required for updates to small graphs is inconsequential in comparison to the overhead of the environment in which they exist.

The average responsiveness of a graph with 500 objects was found to be 10 frames per second. While a frame rate this low does produce visible jitter, it is essentially unnoticeable due to the snap-to-grid feature discussed in Section A.2 of the Appendix. While the software is running, its memory imprint ranges from 15 MB to 25 MB of RAM even when manipulating graphs with up to 4000 objects.

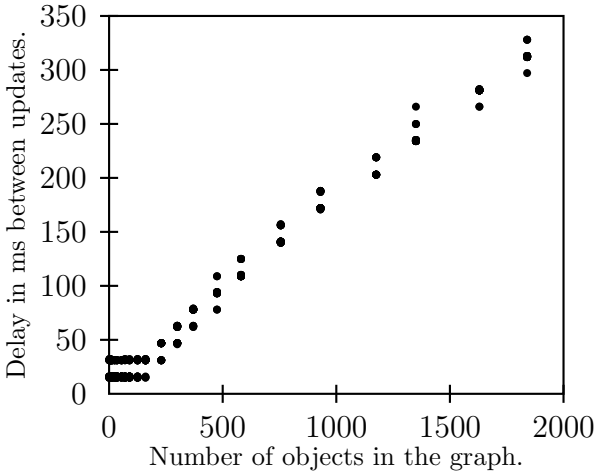

Figure 9.1: Responsiveness versus graph size in ms.

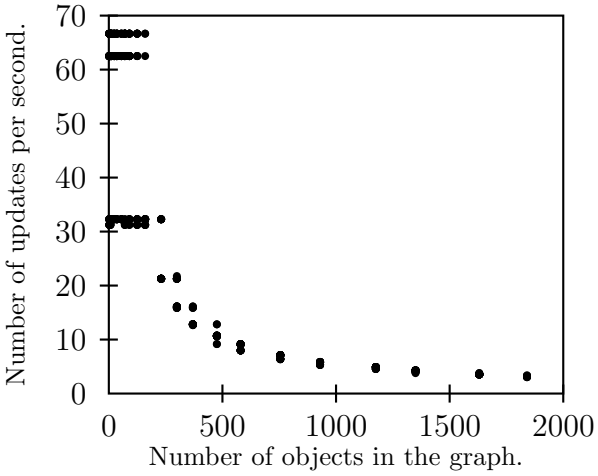

Figure 9.2: Responsiveness versus graph size in frames/s.

The IDES software is considered the property of the DES lab at Queen's University. At the time of the writing of this dissertation, it is available online at http://www.aggressivesoftware.com/research/ides/ and the installation process is very simple. It is password protected and download permission is arbitrarily granted by Queen's DES Lab. Tutorials, JavaDoc and bug reports are also available at the same site. A complete description of the IDES software and all its functions is given in

Appendix A. The IDES software was used extensively in experimentation and modeling during the research and formatting of this thesis. All of the DES figures in this thesis were automatically generated by the IDES software. Several experiments with DES models in real custom hardware were performed in conjunction with the IDES software. The details of one such model are given in Appendix B.

9.2.2 Development Processes

The primary goal of the IDES software was to facilitate a highly usable interface for the definition of DES components by human users, and this goal was reflected in the development process. In order to understand the following discussion, it is helpful to be familiar with the final product as explained in Appendix A; nevertheless, a quick summary is given here. The IDES software lets the user draw a directed graph on a part of the screen arbitrarily called the canvas. The graph is controlled by canvas tools. Canvas tools are selectable from the main menu and from the main toolbar. Only one canvas tool can be selected at a given time and it determines how the mouse is able to manipulate the graph. In this manner, the IDES software is similar to many painting programs. On the canvas a DES is represented as a directed graph realized by circles interconnected by curved arrows. These nodes and edges can be decorated in various ways including text labels. Furthermore, they may be repositioned and the edges may be manipulated as Bezier curves.

The development cycle of the IDES interface iterated on experimentation with human users. Subjects ranged from naive (no knowledge of DES, no formal knowledge of graphs, marginal experience with computers) through intermediate (no knowledge

of DES, moderate knowledge of graphs, moderate experience with computers) to advanced (considerable experience with DES, formal knowledge of graphs, considerable experience with computers). Users were given a simple graph drawn with pen and paper and a workstation with the IDES software in its initial state and were asked to recreate the graph using the IDES software. This provided unbiased analysis of the usability and effectiveness of the interface. Users were also asked to draw arbitrary graphs with the IDES software as a means of gauging their expectations of its functionality. These experiments lead to the following discoveries.

No Logical Relationship Between Graph Components

Naive users consistently persisted in ignoring the relationships between nodes and edges, considering them no more related than lines of ink on paper. This resulted in attempts to create edges before creating nodes. In an early implementation, there were three separate canvas tools for node creation, edge creation and object repositioning and customization. This proved ineffective because users persisted in creating graph components in an arbitrary order. Specifically, in the DES paradigm, and in all graph theory, while it is permissible to have an unconnected node, it is meaningless to have an edge that lacks either source or destination. As a solution to this problem, the node and edge creation tools were merged to a single tool. This solves the problem of unconnected edges. In the new paradigm, a single click on blank space creates a node, while a single click on an existing node initiates an edge from that node. As an accelerator, a double click on blank space both creates a node and initiates an edge from it. Once an edge is initiated it tracks the mouse pointer. A single click on an existing node terminates the edge at that node, while a single click on blank space both

creates a new node and terminates the edge at it. The implementation of this solution greatly accelerated users' ability to create graphs, but also introduced the problem of mistakenly initiated edges. Once initiated, an edge could not be abandoned because a click on blank space created a new node. The most acceptable solution to this was found to be employing right-click as cancel.

Clicking Paradigms

It was found that regardless of user type, different users unpredictably and approximately equally fell into one of two clicking paradigms. Users would attempt to specify an edge either by mousing down, dragging and releasing or by clicking, moving the mouse and clicking again. This problem was solved by implementing both methods. Its only failure is that a quickly specified self-loop in the down, drag, release paradigm can be misinterpreted as a single click in the alternate paradigm leaving the user with a partially initiated edge. A second click solves the problem, finishing the self-loop.

Visual Model Stronger Than Mental Model

Startling to the author, many users (including advanced users) failed to choose the internal area of a node as a clicking target for edge termination. Because the arrowhead of an initiated edge tracks the mouse pointer and because completed edges themselves paint from a point on the circumference of a node to a point on the circumference of another node, users would click outside of (but near) a target destination node, instead of inside it. This conforms with their visual model of the graph they are creating because the arrowhead is approximately in the position they wish to achieve, but is less consistent with an abstract graph model because the target node is not

explicitly specified. This result is particularly interesting because it implies humans tend more toward visual stimulus than abstract mental models. This problem was solved by first checking if a click fell inside any nodes and second checking if it fell near any nodes before concluding that it was a click on blank space.

Maximally Merged Tools

Originally, separate interfaces existed for moving existing objects in the graph and specifying decorations like labels, but users consistently found this frustrating. In the end, a single canvas tool was made responsible for all graph customizations. This was limited only by edge label specification. In a DES, an edge label indicates which event from the event space has occurred. If α joins nodes one and two, and α joins nodes seven and eight, it is the same α. It is one unit of data with a considerable amount of meta-data such as its meaning within the system and its controllability or observability to various supervisors. For this reason edge labels cannot be simply text. The final accepted solution was to create a separate data environment where events could be specified. When a user attempts to label an edge (by double clicking it) if no events are defined they are forwarded to the tab where they may be defined. If some events are defined, a popup chooser appears. This chooser allows the user to toggle which events are associated with the edge. While the learnability of this solution was found to be lower than simple text entry, the increased power of the model and ease of use for educated users was found to be most desirable.

Logical Justification of Results

As stated previously, a DES component such as a plant can be realized as a directed, labeled graph. To achieve an environment analogous to pen and paper drawing, it is necessary that the environment allow the specification of the graph given an amount of input information less than or equal to the information required to realize the same graph on pen and paper. Otherwise, the synthetic environment is more work than the pen and paper environment and hence a failure. The specification of a graph in a pen and paper environment can be decomposed into the following four minimal information actions.

- A node can be created by a single action, namely the specification of its origin.
- An edge can be created by a dual action, namely the specification of its source and destination nodes.
- A label can be created by a dual action, namely the specification of its value and the object to which it pertains.
- The dual action specification of an edge can implicitly create one or both of its associated nodes.

The IDES interface finalized by user experimentation has converged to such a specification. Aside from the restriction of separately specifying the event space, the IDES software provides access to all four minimal information actions with nearly zero learning required. Assuming the user has basic knowledge of a mouse and is familiar with the predominate left-click as primary action convention, then as described in the previous experimentation results, they may specify edges and labels in arbitrary

orders and arbitrary configurations with the same input information used in the pen and paper environment, and the added benefit of automation. The only learnability issue is the requirement of switching to a different canvas tool for label specification, and the use of double-clicking to activate label specification.

9.3 Test Case

Several custom hardware components were developed for experimentation and use with the IDES software. A single test case is described in Appendix B. It contains two microcontrollers, a small LCD and several push buttons and LEDs. These components represent the behaviour of a simple vending machine. This simple test case is included only as an example of hardware that may interface with the IDES software. In this case, the example is fairly contrived. The code on the microcontroller notifies the IDES software of various events and listens for an echo of its notification. Without this echo, it assumes disablement. In the DES model of this component, a single controllable event *"lost pop"* is deemed bad. By running an appropriate supervisor in the IDES software connected to the hardware component, the occurrence of this event (initiated by a human pressing a button) is prevented. Full details of the hardware implementation are included in anticipation of attempts to reproduce this result before advancing to more complicated models.

Chapter 10

Conclusions and Future Work

Discrete-Event Systems control theory provides automated control solutions for systems that are characterized by asynchronous and instantaneous changes of state. Goals expressed in the language of this theory permit the automatic generation of control solutions which guarantee that illegal behaviour will never occur. Despite intensive research on and expansion of the theoretical aspects of this field, a limited amount of research has been reported on its implementation and integration into existing systems. The background theory has been provided and problems arising from the primary assumptions (the plant exists; events are generated by the plant; events occur spontaneously, asynchronously and instantaneously; events are abstract; control is imposed by disablement) have been noted. A survey of related work has been provided along with a wide array of applicable systems. Some of these systems have been examined in detail, their properties noted and methodologies proposed. These properties and methodologies have been reviewed in Chapter 8. Finally, relevant software tools have been analyzed with emphasis on the application developed by this author. The work described herein constitutes some of the first steps in making

the use of Discrete-Event Systems control theory accessible outside of the academic realm.

Certainly it has been confirmed that DES control theory is most amenable to pre-existing systems that function in a non-optimal way and admit control. It has also been confirmed that the best performance is achieved when high-level control is required, and the specification does not imply the solution. Despite the disadvantages associated with systems that do not meet these criteria, it has been shown that the theory can still be advantageously applied. Specifically, several methodologies have been provided for systems in which a portion of the plant and the entire supervisory entity exist within an implementation device such as a microcontroller. Chapter 8 summarized system properties and design methodologies, thereby demonstrating how DES control theory may be applied to a wide variety of systems and how these systems should be approached.

Finally, in analysis of the currently available software tools, it was demonstrated that a real need exists for usable and intuitive software. As a result, the IDES software was developed, which functions as an effective interface for specifying DES components in a manner analogous to pen and paper drawing, and demonstrates the integrated use of DES control theory with custom hardware components for research and pedagogical purposes. To date, other academic tools (such as CTCT, UMDES and GIDDES) have been unconcerned with integrating hardware and software for control and testing purposes. Hence, the IDES software is unique in its attempt to provide an interface for all of modeling, testing and control. Nevertheless, the IDES tool currently serves primarily to aid researchers in visualizing and communicating their work. As can be expected, there is room for future work on this tool, and with

more development time and integration with other tools, it could help advance DES control theory beyond the academic realm.

10.1 Future Work

The work in this dissertation has necessarily covered a wide variety of systems and due to time constraints could not examine large examples in detail. If DES control theory is to come into popular use, it must demonstrate a tangible advantage over alternate methods on real world problems. In order to handle the complexity of larger systems it would likely be necessary to employ more advanced modular and/or hierarchical DES control theory. An investigation into the use of DES control theory to solve an existing complex problem in industry utilizing these advancements would provide a major contribution in increasing the usability of the theory.

The ability to augment a plant provides an avenue for optimization. Specifically, if components of the legal requirement are found to be uncontrollable, ad hoc or formal analysis may be employed to determine what changes to the plant are necessary in order to achieve the requirement, and what cost this entails. There has been little investigation into this topic. It would be very beneficial if during the synthesis of a supervisor, the designers were also given information on how the plant could be modified to achieve a larger subset of the legal specification.

The demonstrated benefit of the "integrated" methodology is derived from the replacement of state structure with integer values. A more general approach to this same optimization is presented in [14] as mentioned in Chapter 7. The continued expansion of FSMs with parameters throughout core DES control theory should greatly increase the applicability of the theory to complex real-world problems.

The algorithms for automatic generation of machine code for the PIC16F84 have only been proposed and not fully implemented. It would be beneficial to have these fully implemented. A software that could specify DES components, then automatically generate machine code, allow the designer to complete the ad hoc components, and finally burn the result onto a chip would greatly aid in convincing researchers and professionals of the applicability of DES control theory.

10.1.1 IDES

The primary goal in the development of the IDES software was to achieve an interface analogous to pen and paper drawing and to export the defined components to various graphical formats. The software solidly achieves these requirements; however, it could be made vastly more powerful. If one considers relatively simple advancements, several features such as rendering of the grid could be re-implemented in a more efficient manner. The quality of inline rendering of LaTeX labels should also be improved. A more standardized file format such as XML should be employed. Also, for increased usability, a complete redesign of the transition specification tab and additional alignment features for edge label placement would be highly desirable.

A very important enhancement would be the ability to define multiple automata tied to the same event-space. Specifying modular or hierarchical relationships between these components would also be desirable. Instead of modeling a single automaton, a single use of the software should model an entire system, including plant modules, legal specifications and computed supervisors. For computation of the supervisors, integration with an existing tool such as CTCT would be desirable. Layouts for the computed results could be provided by third party tools such as GraphViz. These

two advancements (multiple models and connection to computation engines) would accelerate IDES from a visualization tool to a full blown design tool. Further inclusion of implementations of the suggested automatic code generation algorithms would accelerate IDES to a complete end-to-end design solution. Finally, inclusion of modular and hierarchical techniques, both in the input and computation components of the software would make IDES legitimately capable of tackling real world problems in a usable, robust and advantageous manner. The acceptance of any framework hinges on the gain of its use minus the difficulty of its use. A standardized and usable conglomerate design tool is necessary for the application of DES control theory to be feasible.

Bibliography

[1] K. Akesson. Recipe coordination in chemical batch processes. Technical Report 330L, Chalmers University of Technology, 1999. Control and Automation Laboratory.

[2] AT&T. AT&T Labs Research - Grappa: A java graph package. Viewed June 2005, http://www.research.att.com/sw/tools/graphviz/packages/grappa.html.

[3] R. Aveyard. A Boolean model for a class of discrete event systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-4:249–258, 1974.

[4] S. Balemi. Input/output discrete event processes and communication delays. *Discrete Event Systems: Theory and Applications*, 4(1):41–85, 1994.

[5] L. Barowski. VGJ web site. Auburn University, Viewed June 2005, http://www.eng.auburn.edu/department/cse/research/graph_drawing/vgj.html.

[6] B. A. Brandin. *Real-Time Supervisory Control of Automated Manufacturing Systems*. PhD thesis, Department of Electrical Engineering, University of Toronto, 1993. Also appears as Systems Control group technical report # 9302, Department of Electrical Engineering, University of Toronto, Feb 1993.

[7] B. A. Brandin and W. M. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 39(2):329–342, 1994.

[8] B. A. Brandin and W. M. Wonham. Final report: Ontario/Rhone-Alpes project on the supervisory control of automated manufacturing systems. Technical report, Department of Electrical and Computer Engineering, University of Toronto, 1995.

[9] B. A. Brandin, W. M. Wonham, and B. Benhabib. Discrete-event systems supervisor control applied to the management of manufacturing workcells. In *7th International Conference on Computer Aided Manufacturing Engineering*, pages 527–536, Cookeville, 1991.

[10] R. D. Brandt, R. Kumar V. Garg, F. Lin, S. I. Marcus, and W. M. Wonham. Formulas for calculating supremal controllable and normal sublanguages. *Systems & Control Letters*, 15(1):111–117, 1990.

[11] Universität Bremen. uDraw web site. Universität Bremen, Viewed June 2005, http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html.

[12] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1999.

[13] V. Chandra, Z. Huang, and R. Kumar. Automated control synthesis for an assembly line using discrete event system control theory. *IEEE Transactions on Systems, Man, and Cybernetics*, 33(2):284–289, 2003.

[14] Y. Chen and F. Lin. Modeling of discrete event systems using finite state machines with parameters. In *Proceedings of the 2000 IEEE International Conference on Control Applications*, pages 941–946, Anchorage, Alaska, USA, September 2000.

[15] S. L. Chung, S. Lafortune, and F. Lin. Limited lookahead policies in supervisory control of discrete event systems. *IEEE Transactions on Automatic Control*, 37(12):1921–1935, 1992.

[16] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, 1988.

[17] B Comiskey, JD Albert, H Yoshizawa, and J Jacobson. An electrophoretic ink for all-printed reflective electronic displays. *Nature*, 394(6690):253–255, 1998.

[18] E Ink Corporation. E - Ink. Viewed June 2005, http://www.eink.com/.

[19] Electronic Industries Association, (http://www.eia.org). *EIA232E - Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*, January 1997.

[20] C. Enright and M. Barbeau. An evaluation of the TCT tool for the synthesis of controllers of discrete event systems. In *Canadian Conference on Electrical and Computer Engineering*, volume 1, pages 241–244, December 1993.

[21] M. Fabian and A. Hellgren. Desco - a tool for education and control of discrete event systems. In R. Boel and G. Stremersch, editors, *Discrete Event Systems, Analysis and Control*, pages 471–472, August 2000.

[22] L. Grigorov. Control of dynamic discrete-event systems. Master's thesis, School of Computing, Queen's University, 2004.

[23] L. Grigorov and K. Rudie. Issues in optimal control of dynamic discrete-event systems. In the *Proceedings of the 16th IFAC World Congress*, Prague, July 2005.

[24] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 2nd edition, 2000.

[25] IBM. Eclipse.org Main Page. Viewed June 2005, http://www.eclipse.org.

[26] Industrial Automation Systems. *Manufacturing Message Specification, International Standard, ISO/IEC 9506*, 1990.

[27] S. Lafortune and D. Teneketzis. University of Michigan DES Group. University of Michigan, Viewed June 2005, http://www.eecs.umich.edu/umdes/toolboxes.html.

[28] S. C. Lauzon, A. K. L. Ma, J. K. Mills, and B. Benhabib. Application of discrete-event-system theory to flexible manufacturing. *IEEE Control Systems*, 16(1):41–48, February 1996.

[29] R. Leduc. PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 1996.

[30] F. Lin. That supervisor is best which supervises least. Workshop on Discrete Event Systems and Artificial Intelligence, Princeton University, Viewed June 2005, http://www.ece.eng.wayne.edu/∼flin/reprints/best.pdf, November 1990.

[31] F. Lin. Analysis and synthesis of discrete event systems using temporal logic. *Joumal of Control-Theory and Advanced Technology*, 9(1):341–350, 1993.

[32] F. Lin and W. M. Wonham. On observability of discrete-event systems. *Information Sciences*, 44(3):173–198, 1988.

[33] F. Lin and W. M. Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions of Automatic Control*, 35(12):1330–1337, 1990.

[34] M. Losito. An architecture for flexible manufacturing systems applied to an assembly cell. Master's thesis, Politecnico di Milano, Italy, October 1999.

[35] G. Van Noord. FSA6 web site. University of Groningen, Viewed June 2005, http://odur.let.rug.nl/∼vannoord/Fsa/.

[36] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

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

[38] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of IEEE*, 77(1):81–98, 1989.

[39] S. L. Ricker. GIDDES web site. Mount Allison University, Viewed June 2005, http://www.mta.ca/giddes/.

[40] K. Rudie, S. Lafortune, and F. Lin. Minimal communication in a distributed discrete-event system. *IEEE Transactions on Automatic Control*, 48(6):957–975, 2003.

[41] K. Rudie, N. Shimkin, and S. D. O'Young. Timed discrete-event systems: A manufacturing application. In *Proceedings of the 1994 Conference on Information Sciences and Systems*, pages 374–381, Princeton, NJ, 1994.

[42] K. Rudie and J. C. Willems. The computational complexity of decentralized discrete-event control problems. *IEEE Transactions of Automatic Control*, 40(7):1313–1319, 1995.

[43] K. Rudie and W. M. Wonham. Think globally, act locally: Decentralized supervisory control. *IEEE Transactions of Automatic Control*, 37(11):1692–1708, 1992.

[44] BRAINSYS Informatic Systems. Graphlet web site. University of Passau, Viewed June 2005, http://www.infosun.fmi.uni-passau.de/Graphlet/.

[45] A. F. Vaz and W. M. Wonham. On supervisor reduction in discrete-event systems. *International Journal of Control*, 44(2):475–491, 1987.

[46] R. Wiese. Wiese's little automata builder web site. University of Tübingen, Viewed June 2005, http://www-ti.informatik.uni-tuebingen.de/∼wiese/Automaton/.

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

[48] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete event systems. *Math. Control Signals Syst.*, 1(1):13–30, 1988.

[49] W.M. Wonham. W.M. Wonham's Home Page. University of Toronto, Viewed June 2005, http://www.control.toronto.edu/people/profs/wonham/.

[50] M. Wood. On the current state of graph drawing software and the construction of small aesthetic graphs. Viewed June 2005, http://www.aggressivesoftware.com/research/files/wood_graph_drawing.doc.

[51] XFIG.org. XFIG web site. Viewed June 2005, http://www.xfig.org/.

[52] T. Van Zandt. PSTricks web site. Viewed June 2005, http://www.tug.org/applications/PSTricks/.

Appendix A

IDES Software User's Guide

A.1 Introduction

The purpose of this guide is to help people use and understand the IDES Software. The first section (which gives an overview of all the toolbar and menu options) is not intended to be read linearly but has been structured like a lookup table. Because of this, information is often repeated in its various sub-sections.

The later sections are written as linear tutorials describing the basic tasks users can undertake. For most users it will be sufficient to read only the **Drawing a Graph** and **Modifying a Graph** sections. These describe all the basic tasks that general users will be interested in performing. The remaining sections document the more advanced features and can be overlooked until needed.

A.2 The Toolbars and Main Menu

A.2.1 New System

The **New System** toolbar item unloads the current system, letting the user start from scratch. If the user has made any changes to the current system, the user will first be prompted to save.

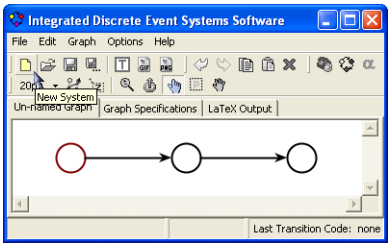

Figure A.1: New System.

A.2.2 Open

The **Open** toolbar item allows the user to load a previously saved system. If the user has made any changes to the current system, the user will first be prompted to save. The system is saved as a simple text file with the extension **gml**. These files can be opened in any simple text editor like Notepad and modified manually.

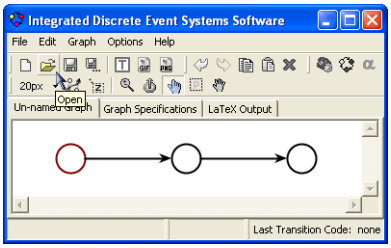

Figure A.2: Open.

A specification for a valid file for the IDES software is given in the **IDES file Specification** section of this guide. The files do not strictly conform to the GML format as explained in that section.

A.2.3 Save

The **Save** toolbar item allows the user to save the current system to a file. The user will only be prompted for a file name if working with a new system, otherwise it will automatically save over the file of the current

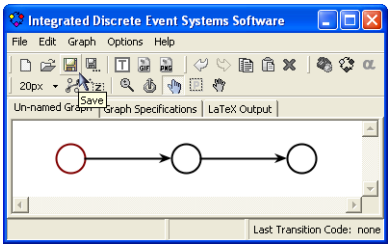

Figure A.3: Save.

system. The system is saved as a simple text file with the extension **gml**. These files can be opened in any simple text editor like Notepad and modified manually. A specification for a valid file for the IDES software is given in the **IDES file Specification** section of this guide. The files do not strictly conform to the GML format as explained in that section.

A.2.4 Save As...

The **Save As...** toolbar item allows the user to save the current system to a new file. The user will be prompted for a new file name. The system is saved as a simple text file with the extension **gml**. These files can be opened in any simple text editor like Notepad and modified manually. A specification for a valid file for the IDES software is given in the **IDES file Specification** section of this guide. The files do not strictly conform to the GML format as explained in that section.

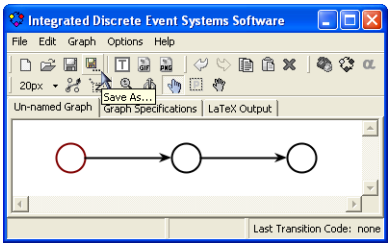

Figure A.4: Save as....

A.2.5 Export To LaTeX

The **Export To LaTeX** toolbar item allows the user to export a representation of the graph to LaTeX. It requires that the user has first specified a **Print Area**, and if one is lacking, the user will be prompted for it. Regardless of the user's system settings, a LaTeX code representation will be printed in the text area inside the **LaTeX Output** tab.

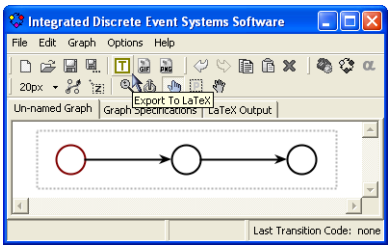

Figure A.5: Export to LaTeX.

If the user has checked the **Export LaTeX directly to EPS** option (from the **Options** menu) then the system will attempt to generate an EPS file which may be included in LaTeX documents. The user will be prompted for a save location for this file. The code used to generate the EPS file is printed in the text area in the **LaTeX Output** tab. Sample code for the inclusion of an EPS file in a LaTeX document is also printed as a remark at the bottom of the code, and is reproduced in Listing A.1.

Listing A.1 Example LaTeX code for the inclusion of any EPS file.

```
\documentclass[12pt]{article}
\usepackage{graphicx}
\begin{document}
  \includegraphics[scale=1]{yourfilename.eps}
\end{document}
```

If the user has checked the **Export LaTeX directly to TeX** option (from the **Options** menu) then the system will attempt to generate a TeX file which contains a representation of the graph as a figure and may be included in LaTeX documents. The user will be prompted for a save location for this file. The contents of the TeX file are printed in the text area in the **LaTeX Output** tab. Sample code for the inclusion of the generated file in a LaTeX document is also printed as a remark at the bottom of the code. Listing A.2 reproduces a generalized version of that information.

Listing A.2 Example LaTeX code for the inclusion of a TeX file containing a figure generated by the IDES software.

```
\documentclass[letterpaper,12pt]{report}
\usepackage{setspace}
\usepackage{pstricks} % only necessary for figures using the pspicture environment
\usepackage{pict2e}   % only necessary for figures using the picture environment
\begin{document}
\psset{unit=1pt}      % only necessary for figures using the pspicture environment
\include{your_tex_file_name_WITHOUT_TEX_EXTENSION} % example: mygraph (instead of mygraph.tex)
\end{document}
```

A.2.6 Export To GIF

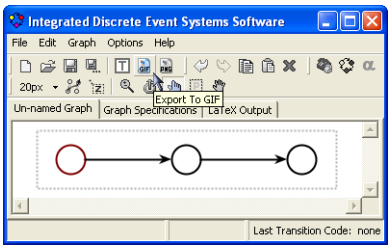

Figure A.6: Export to GIF.

The **Export To GIF** toolbar item allows the user to save a representation of the graph in the GIF format. It requires that the user has first specified a **Print Area**, and if one is lacking, the user will be prompted for it. The export process has some flaws. The export will fail if the selected **Print Area** is not entirely visible on the user's screen (i.e., is hidden by other windows, has scrolled off screen, etcetera) at the instant the user clicks the export button. Also note that in WinXP, the user may need to refresh their Windows Explorer view before opening the exported GIF file in order for it to be readable.

A.2.7 Export To PNG

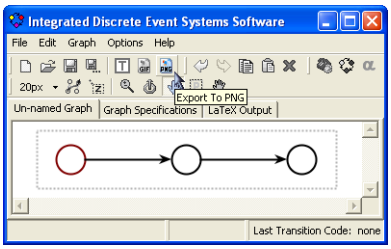

Figure A.7: Export to PNG.

The **Export To PNG** toolbar item allows the user to save a representation of the graph in the PNG format. It requires that the user has first specified a **Print Area**, and if one is lacking, the user will be prompted for it. The export process has some flaws. The export will fail if the selected **Print Area** is not entirely visible on the user's screen (i.e., is hidden by other windows, has scrolled off screen, etcetera) at the instant the user clicks the export button.

A.2.8 Undo

The **Undo** toolbar item was not implemented as a real undo. Every five **actions** (loosely defined) the IDES software remembers a snapshot of the system. These are saved in the IDES/system folder. **Undo** replaces the system with the previous snapshot, and therefore does let the user backup, but not ideally.

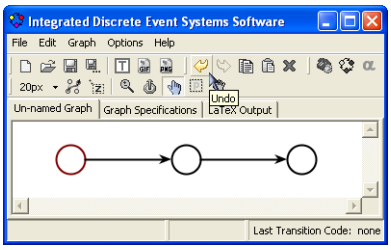

Figure A.8: Undo.

A.2.9 Redo

The **Redo** toolbar item was not implemented as a real redo. Every five **actions** (loosely defined) the IDES software remembers a snapshot of the system. These are saved in the IDES/system folder. **Redo** lets the user move forward through these snapshots when the user has moved backwards through them using **Undo**. Note that if the user activates **Undo**, then makes any changes (even just clicking on a node) the ability to **Redo** can be lost.

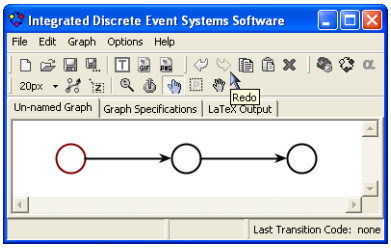

Figure A.9: Redo.

A.2.10 Copy

The **Copy** toolbar item lets the user copy the selected graph elements (which appear drawn in dark red). Note that regardless of the user's group of selected elements, an edge will only be added to the copied group if both of its respective nodes are included in the selection. In

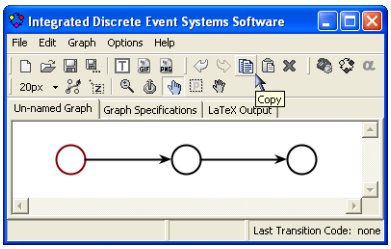

Figure A.10: Copy.

the IDES software, an edge cannot exist without a starting and ending node.

A.2.11 Paste

The **Paste** toolbar item adds copies to the graph of whatever graph elements the user last copied. If paste is used from the main menu, toolbar or CTRL+V shortcut, the new elements will appear centered on the screen. If paste is used from the right-click popup menu, the new elements will appear centered at the mouse-click location.

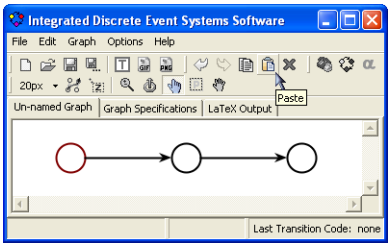

Figure A.11: Paste.

A.2.12 Delete

The **Delete** toolbar item permanently deletes the selected graph elements (which appear drawn in dark red). The user will not be warned or prompted for confirmation. If this tool is used accidentally, the Undo tool may be of assistance, although it will go backwards farther than the user's last action.

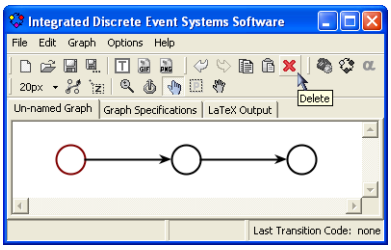

Figure A.12: Delete.

A.2.13 Connect

The **Connect** toolbar item attempts to initialize communication over the COM1 port on the user's computer. This should always succeed, regardless of whether or not there is any external device attached to COM1. When connected, the button is drawn in a **depressed** mode and reads **Disconnect**. The user may click it a second time

to disconnect. The purpose of this feature is to support the **Trace** feature. More information on this feature can be found in the **Animated Trace** section of this guide.

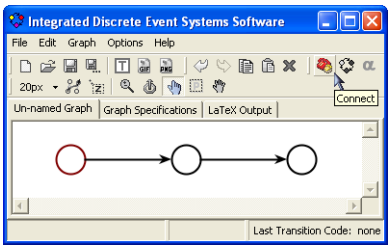

Figure A.13: Connect.

A.2.14 Start Trace

The **Start Trace** toolbar item allows the user to initiate an animated trace of transitions in the graph. The user must be **connected** in order to initiate a trace. Once a trace has been initiated, the button is drawn in a **depressed** mode and reads **Stop Trace**. The user can stop a trace at any time by clicking the trace button a second time. When a trace is started, the initial state of the graph becomes highlighted in blue. The system then listens for transitions. When they occur, the blue highlight animates across that transition to the appropriate state. Transitions can be received from an external device connected to the COM1 port of the user's computer or via the trace field and manual use of the **Alpha** button. More information on this feature can be found in the **Animated Trace** section of this guide.

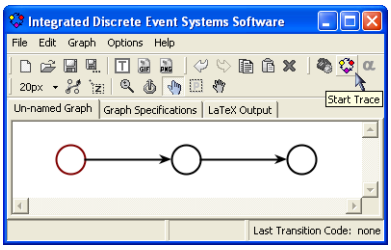

Figure A.14: Trace.

A.2.15 Alpha

The **Alpha** toolbar item is a manual means of feeding custom transition values to the **Trace** system. This allows the user to animate arbitrary event strings. The values sent by the alpha button are specified using the

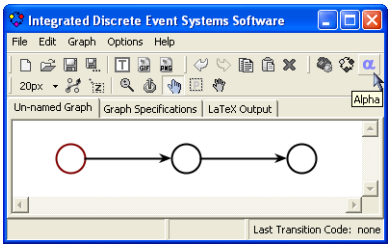

Figure A.15: Alpha.

trace text box at the bottom of the Graph Specification tab. More information on this feature can be found in the **Animated Trace** section of this guide.

A.2.16 Grid Options

The **Grid Options** toolbar item controls the snap-to-grid feature. This applies to the placement of nodes and assists the user in creating nicely aligned graphs. The left half is a toggle button which determines whether the grid should be drawn or not. The right half is a drop down list which determines the scale of the grid. The current grid options are saved and loaded with the user's system. The last-used grid option is remembered for use in any new systems the user creates. Note that some actions run considerably slower when the grid is visible.

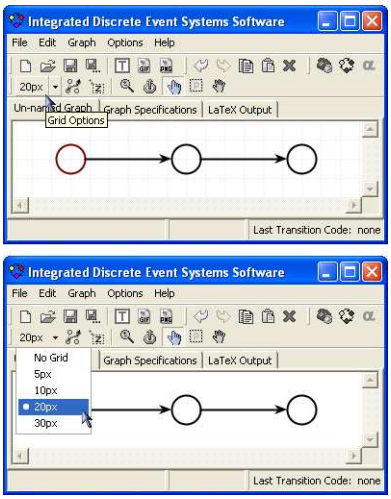

Figure A.16: Grid display and selection.

A.2.17 Show All Edges

The **Show All Edges** toolbar item is a program option that is not saved or loaded with the user's system, but is constant across all systems and is remembered for successive uses of the IDES software. When selected, it causes all edge curves to be drawn in the **modifiable** mode. This is useful when the user intends to customize several edge curves at once, but it can become cluttered in complex graphs.

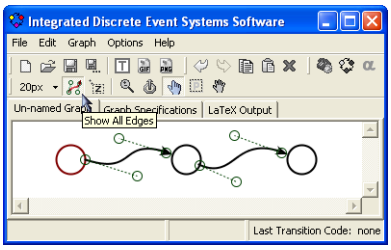

Figure A.17: Show all edges toggle button.

A.2.18 Show All Labels

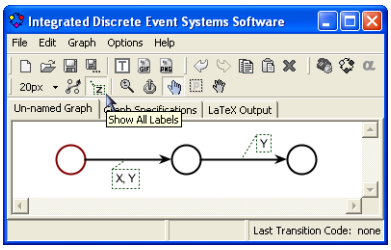

Figure A.18: Show all labels toggle button.

The **Show All Labels** toolbar item is a program option that is not saved or loaded with the user's system, but is constant across all systems and is remembered for successive uses of the IDES software. When selected, it causes all edge labels to be drawn in the **modifiable** mode. This can save the user time if the user intends to customize several edge labels at once. It also clarifies which labels belong to which edges.

A.2.19 Zoom

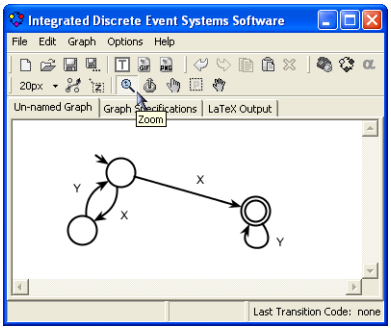

Figure A.19: Zoom canvas tool

The **Zoom** toolbar item is a canvas tool. Only one canvas tool can be selected at a given time, and it determines what the mouse can do to the graph. Zoom allows the user to zoom in and out five steps in either direction. Left clicks on the graph zoom in, and right clicks on the graph zoom out. Note that the zoom is centered around the mouse click. The zoom effect causes a repositioning of the user's graph, the user can adjust this using the scrollbars, or the **Move** canvas tool. Note that the user is still able to perform all functions regardless of the current zooming state. Figure A.20 shows the graph in Figure A.19 zoomed out by one step and Figure A.21 it zoomed in by one step.

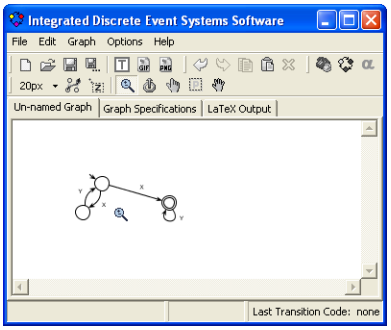

Figure A.20: Zoomed out by one step.

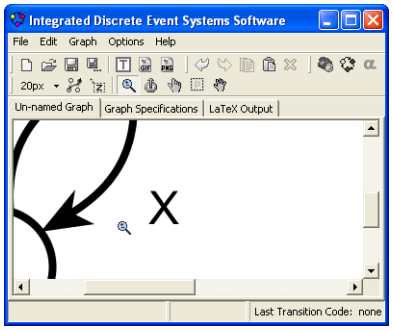

Figure A.21: Zoomed in by one step.

A.2.20 Create Nodes or Edges

The **Create Nodes or Edges** toolbar item is a canvas tool. Only one canvas tool can be selected at a given time, and it determines what the mouse can do to the graph. This tool lets the user draw nodes and edges in the graph, and it is the default selected startup tool. This tool is discussed in the **Drawing a Graph** section of this guide.

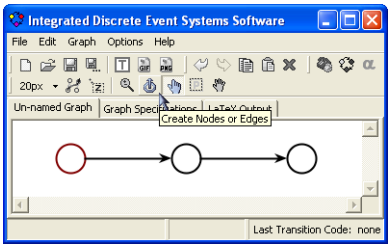

Figure A.22: Create nodes or edges canvas tool.

A.2.21 Modify Nodes, Edges or Labels

The **Modify Nodes, Edges or Labels** toolbar item is a canvas tool. Only one canvas tool can be selected at a given time, and it determines what the mouse can do to the graph. This tool lets the user modify the positions and shapes of nodes and edges and labels in the graph. It also lets the user specify edit and create labels. This tool is discussed in the **Modifying a Graph** section of this guide.

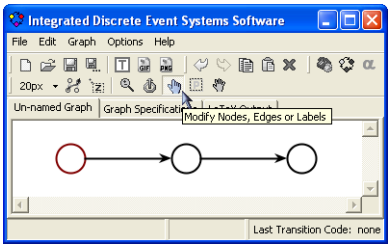

Figure A.23: Modify nodes, edges or labels canvas tool.

A.2.22 Print Area

The **Print Area** toolbar item is a canvas tool. Only one canvas tool can be selected at a given time, and it determines what the mouse can do to the graph. The print area is necessary for export to PNG, GIF and LaTeX. When selected, the user can click and drag on the canvas to specify a bounding box for the print area.

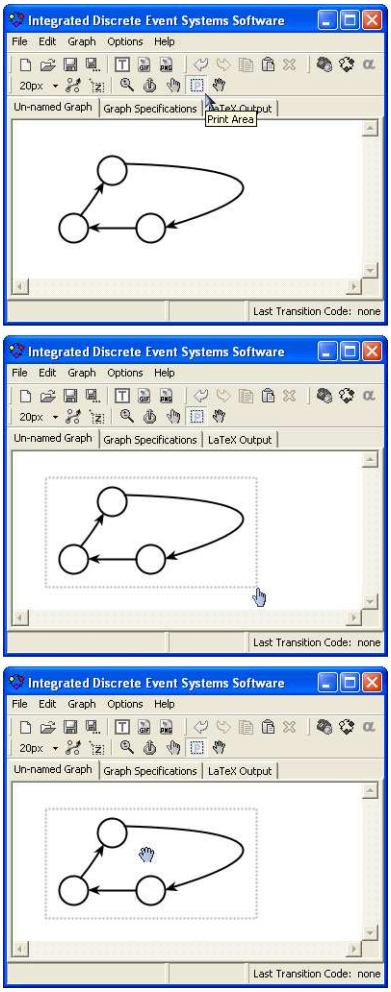

Figure A.24: Print area canvas tool – selecting, defining and doving.

Once an area is specified it can be moved. With the print area tool still selected, when the mouse hovers over the print area, the cursor changes to the **move** cursor, and the user can click and drag to move the print area. No other graph objects are repositioned.

The user can also adjust the bounds of the print area by moving the cursor over the borders. The cursor will change to a directional cursor and the user can click and drag to adjust the borders. Note that to get rid of the print area, the user can click and release outside of the specified area.

Once a print area is specified, the user is able to export to PNG, GIF or LaTeX. These actions are available on the top toolbar and in the File menu under Export.

The **Draw a border when exporting** option affects the user's export. If this option is selected, there will be a black border around the user's exported graph. If it is not selected, there will not be a border. The border would appear exactly where

the print area bounds were specified.

The export to PNG and GIF features have some flaws. The export will fail if the selected **Print Area** is not entirely visible on the user's screen (i.e., is hidden by other windows, has scrolled off screen, etcetera) at the instant the user clicks the export button.

A.2.23 Move Graph

The **Move Graph** toolbar item is a canvas tool. Only one canvas tool can be selected at a given time, and it determines what the mouse can do to the graph. With this tool, the user is able to move the entire graph. It is similar to a permanent scroll bar. The user may simply left-click, drag and release to move the graph. Note that if the user drags the graph off screen, the scrollbars will adjust after the user releases the mouse button. Also note that this tool drags the grid as well as the graph as visible in Figure A.25.

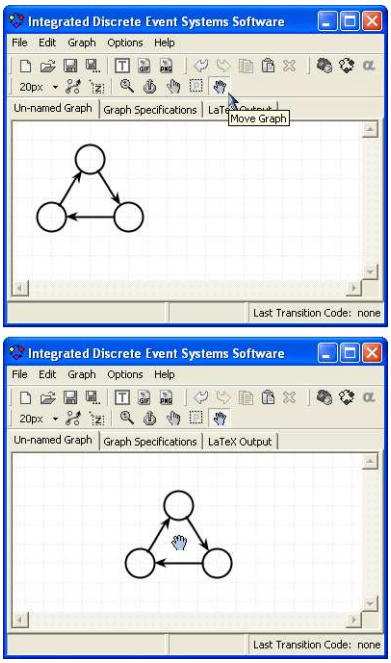

Figure A.25: Move graph canvas tool – selecting and moving.

A.2.24 The File Menu

The **File** Menu contains exactly the same options as the File toolbar, except for Exit which shuts down the IDES software. On exit, either by the File menu or by the X at the top right of the window, if the user has modified the current system, the user will be prompted

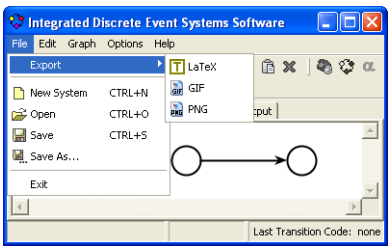

Figure A.26: File menu.

to save the changes. Note that some of the options in the File menu also have keyboard shortcuts.

A.2.25 The Edit Menu

The **Edit** Menu contains exactly the same options as the Edit toolbar. Note that some of the options in the Edit menu are also available in various right-click popup menus on the canvas. Note that some of the options in the Edit menu also have keyboard shortcuts.

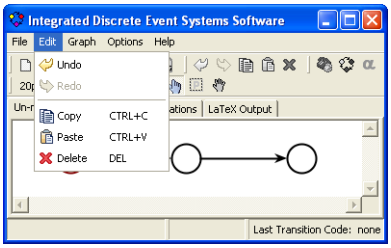

Figure A.27: Edit menu.

A.2.26 The Graph Menu

The **Graph** Menu contains exactly the same options as the Graph toolbar except that the grid options are only available on the toolbar.

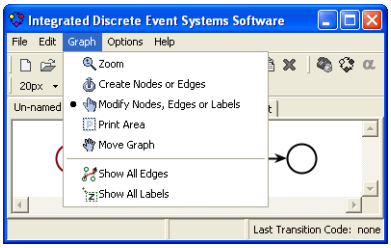

Figure A.28: Graph menu.

A.2.27 The Options Menu

The **Options** Menu contains system settings. Most of these are not saved or loaded with individual graphs, but are constant across the IDES software. They are remembered for successive uses of the IDES software.

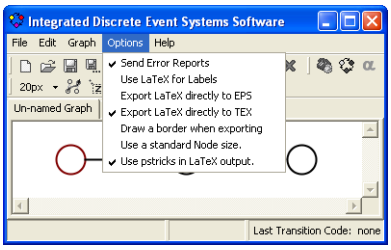

Figure A.29: Options menu

Send Error Reports

The **Send Error Reports** menu item is a part of the Options Menu. Regardless of this option, if a crash occurs the error message is printed to the console (if it

exists), and is displayed in a popup window. If this option is selected, the IDES software will attempt to spawn an internet browser to a bug-report URL, and it will preload the text box there with the appropriate error message. The user can also enter a description of what was being done when the error occurred. All the user has to do to submit the bug report is click the **Add** button at the right hand side of the screen. The user **does not** need to sign in to the webpage to add bug reports. At the time of the writing of this dissertation, the bug-report URL was http://www.aggressivesoftware.com/research/ides/bugs/

Use LaTeX for Labels

The **Use LaTeX for Labels** menu item is a part of the Options Menu. There are two options for drawing Labels: Glyphs and LaTeX. Glyphs are faster, native to Java and look nicer on the IDES software screen and in exported PNGs and GIFs. Glyphs only support regular keyboard symbols. LaTeX is slower, requires that the user has GhostScript and MiKTeX installed, and looks somewhat unattractive on the IDES software screen and in exported PNGs and GIFs. LaTeX supports all LaTeX commands, so the user is able to make much more diverse labels, and it works very well in the export to LaTeX—meaning that even though it may not look attractive in the IDES software, it will look good in the exported LaTeX. The only current bug with LaTeX labels is with line breaks in Node labels. Because line breaks are not supported in the picture environment, the software manually breaks up the code and repositions the fragments, and it does not work perfectly.

Export LaTeX directly to EPS

The **Export LaTeX directly to EPS** menu item is a part of the Options Menu. It is a system option and is not saved or loaded with individual graphs. When this option is selected, use of the **Export to LaTeX** feature causes the system to attempt to generate an EPS file which may be included in LaTeX documents. The user will be prompted for a save location for this file. The code used to generate the EPS file is printed in the text area in the **LaTeX Output** tab. Sample code for the inclusion of an EPS file in a LaTeX document is also printed as a remark at the bottom of the code, and is reproduced in the **Export to LaTeX** section of this guide.

Export LaTeX directly to TeX

The **Export LaTeX directly to TeX** menu item is a part of the Options Menu. It is a system option and is not saved or loaded with individual graphs. When this option is selected, use of the **Export to LaTeX** feature causes the system to attempt to generate a TeX file which contains a representation of the graph as a figure and may be included in LaTeX documents. The user will be prompted for a save location for this file. The contents of the TeX file are printed in the text area in the **LaTeX Output** tab. Sample code for the inclusion of the generated file in a LaTeX document is also printed as a remark at the bottom of the code, and is reproduced in the **Export to LaTeX** section of this guide.

Draw a border when exporting

The **Draw a border when exporting** menu item is a part of the Options Menu. It is a system option and is not saved or loaded with individual graphs. When this

option is selected, a black border is drawn around the user's exported PNG, GIF or LaTeX output in place of the bounding box of the specified print area. When it is not selected, no border is drawn.

Use standard Node size.

The **Use standard Node size** menu item is a part of the Options Menu. It is **not** a system option and is saved and loaded with individual graphs. The default behaviour of nodes is to size to snugly fit their internal labels. When this option is selected, all nodes adjust to the largest node size. This is helpful for symmetry in the graph.

Use pstricks in LaTeX output.

The **Use pstricks in LaTeX output** menu item is a part of the Options Menu. It is a system option and is not saved or loaded with individual graphs. This determines the environment used when generating a LaTeX representation of the graph. **PsTricks** allows the use of the **pspicture** environment. When this option is not selected, the **picture** environment from the **Pict2e** package is used. When the user exports the graph as a TeX file, the output can be manually modified according to the syntax of the chosen environment.

A.2.28 The Help Menu

The **Help** Menu contains a link to a web based tutorial at a help-tutorials URL and a message about the version and authors of the IDES software. At the time of the writing of this dissertation, the help-tutorials URL was http://www.aggressivesoftware.com/research/ides/tutorials/

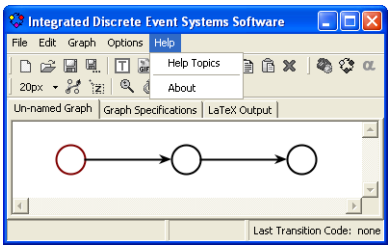

Figure A.30: Help menu

A.3 Drawing a Graph

When the IDES software program is started, the **Create Nodes or Edges** canvas tool is selected by default. When the user clicks on blank space on the canvas with this tool, a node is created roughly centered about the cursor, as shown in Figure A.31. Every action other than this uses the tip of the cursor, but node creation is centered about the cursor as a whole. When a grid is specified (20px is the default), the origin of created nodes snap to the nearest grid location. This can result in the created node seeming to appear in the wrong location, but is actually desirable behaviour.

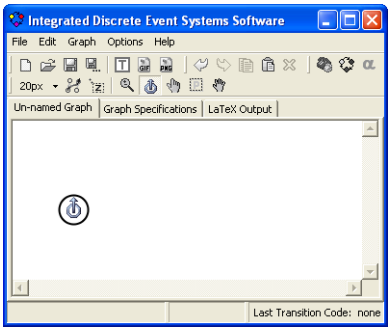

Figure A.31: Creating a new node.

When the user clicks with the pointer of the cursor inside (over top of) a node, edge creation begins. When the user is in the process of creating or modifying graph objects, they are drawn in bright red. When creating edges the user has two options.

The user may click-and-release to start the edge, then move the cursor, then click-and-release to finish the edge. Alternatively the user may mouse-down to start the edge, then move the cursor, then mouse-up to finish the edge. When the user is in mid-creation of an edge, its tail is drawn at the center of the start node, and its head is drawn at the tip of the cursor, as shown in figure A.32.

To finish the edge creation process, the user may simply click on blank space. Another node is created and the edge is joined to it. A new node created in this way is centered at the grid location nearest the tip of the cursor. To speed things up, the user may double click on blank space. This both creates a new node and starts a new edge from it. To join an edge to an existing node, the user may simply click with the cursor's pointer inside (on top of) an existing node.

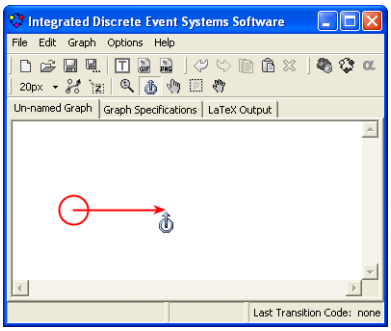

Figure A.32: Creating a new edge.

Edges will attempt to automatically position themselves in a desirable manner. All edges by default travel directly between the centers of their source and destination nodes. When the default behaviour would cause overlap or collision, the edges attempt to reposition themselves accordingly. Notice that in Figure A.33 the two edges between the left and center node have formed into arcs symmetrically. Multiple edges between two nodes will always attempt to display in a symmetrical and reasonable manner. Notice also the arc from the left-most node to the right-most node. Edges will draw in a wide arc such as this when they

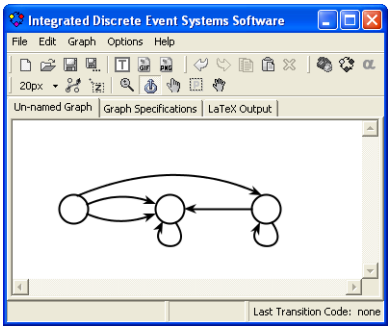

Figure A.33: An example graph.

detect other nodes along their default path. The automatic positioning algorithms are based on a set of common configurations, and can be tricked into undesirable behaviour.

To create a self-loop the user may start an edge in the usual way and choose the same node as the destination. To speed things up, the user may double click on a node to create a self-loop.

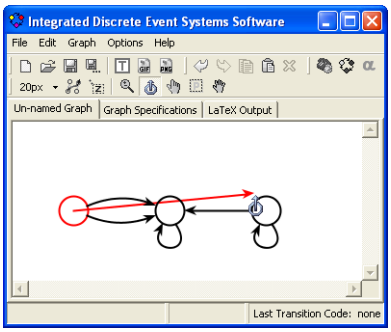

Figure A.34: Disconnecting an edge.

The **Create Nodes or Edges** tool can also be used to disconnect an edge from a node and reconnect it elsewhere. To disconnect an edge, the user must click with the pointer of the cursor on top of the arrowhead of the edge. This causes the edge to revert to the mid-creation state, as shown in Figure A.34. The user cannot disconnect an edge from its tail end; however, the user can use the right-click menu to reverse the direction of the edge and then disconnect it.

A.4 Modifying a Graph

Since the default layout algorithms are not perfect, the user will need to customize the position of elements in the graph. The **Modify Nodes, Edges or Labels** canvas tool facilitates this. The user can mouse-down on a node, drag it around, and mouse-up to reposition it. As shown in Figure A.35, all the edges connected to the node will adjust with the user's movement. Note that all elements being modified will paint in bright red.

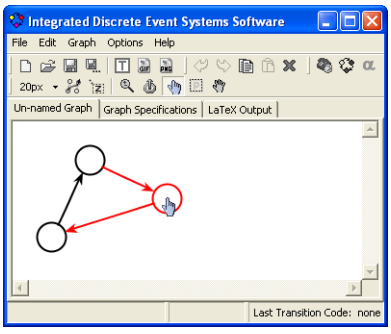

Figure A.35: Respositioning a node.

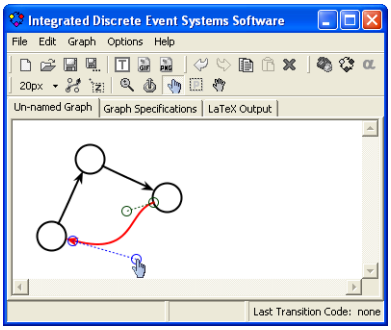

Figure A.36: Customizing an edge.

To adjust edges the user must first click on their arrowheads. This selects the edge and causes it to draw its anchors (four green circles). The user may click on and drag the anchors to reposition them. The edge will update accordingly, as shown in Figure A.36. When the user moves one anchor, its attached anchor may also update. In most cases this makes adjustments easier and faster. In some cases this behaviour can be frustrating, so the feature can be disabled by holding down the **CTRL** button while modifying individual anchors. Once the user has customized the position of an edge, later movements of its associated nodes will attempt to maintain the user's customizations. In some cases this is undesirable. The right-click menu can be used to **reset configuration** of an edge or a group of edges.

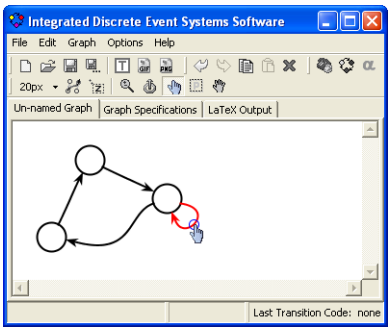

Figure A.37: Customizing a self loop.

Self-loop edges are not fully customizable. As shown in Figure A.37, the user can click on its arrowhead or single anchor point to rotate it about its node, nothing more.

The modify tool can also be used to group objects. Grouped objects paint in dark red. Whenever the user clicks on an object, it becomes the active object and paints in dark red. Such an action creates a group of only one element. The user can hold the CTRL button and click on other objects to add or remove them from the current grouping. The user can also mouse-down on blank space and drag the mouse to specify a new grouping, as shown in Figure A.38. When the user releases

the mouse button, the bounding box of the group area will shrink to encompass the selected nodes.

When the user has created a grouping with a bounding box (rather than creating the group via CTRL+click), the user can drag the grouped elements. When the user moves the mouse over the grouped area the cursor changes to the **move** cursor. The user can then mouse-down, drag and release to move all the objects in the selected group, as shown in Figure A.39. Note that this is very different than using the **move** tool, and the grouped objects still snap to the grid if one is specified.

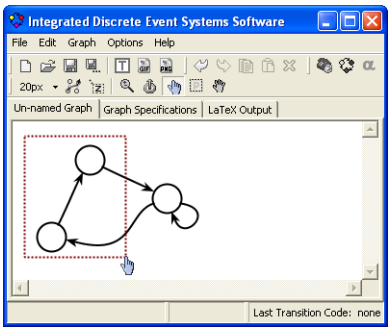

Figure A.38: Selecting a group.

When moving the group, all affected objects paint in bright red. Note that a grouped self-loop will not paint as modified because it is defined at an angle from its source node which is not being changed. External edges connected to the group will update in the usual way, attempting to maintain their specified configurations. Note that the user can use CTRL+click to add more nodes and edges to the group, but the bounding box does not change shape because unselected nodes may exist in the intermediate space. Movement of the group will include all grouped elements, even those outside the bounding box. If the user has added an edge to the group without adding either of its nodes, then the edge will not move when the user moves the group.

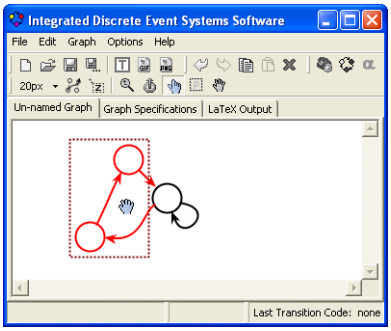

Figure A.39: Moving a group.

Left clicking on empty space, or on a graph object causes the current grouping

to be abandoned and a new one (perhaps empty) to be formed. Note that right clicking does not affect the groupings. The grouping behaviour just described plays an important role with the edit functions as described in the **Right-Click Popup Menus** section of this guide. The **Modify Nodes, Edges or Labels** tool can also be used to specify and modify edge and node labels as described in the **Labels** section of this guide.

A.5 Right-Click Popup Menus

When the user right-clicks on the canvas with any tool **except** the zoom tool, one of four popup menus is generated depending on whether the click was on an edge's arrowhead, inside a node, on blank space inside the bounding box of a grouped area, or on blank space outside the bounding box of a grouped area.

A.5.1 Inside the Bounding Box

When the user right-clicks on blank space inside the bounding box of a grouped area, a popup menu is generated with actions that affect everything within the group.

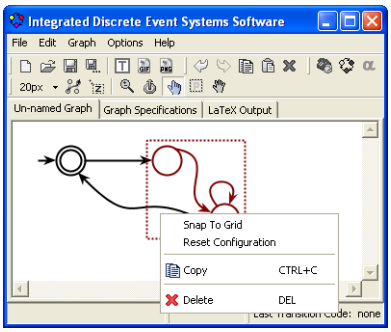

Figure A.40: Inside the bounding box.

Snap To Grid

The **Snap To Grid** option causes all the grouped nodes to snap to the specified grid, which, in turn, may cause reconfiguration of attached edges.

Reset Configuration

The **Reset Configuration** option causes all edges attached to grouped nodes to revert to their default configurations based on the automatic layout algorithms.

Copy

The **Copy** option copies all grouped objects and stores them in the edit buffer. Edges will only be copied if both their start and destination nodes are also in the group. Edges cannot exist in this software without both associated nodes.

Delete

The **Delete** option deletes all grouped objects. The user will not be warned or asked for confirmation. If the user makes a mistake, the Undo action can be helpful, but it will go backwards several steps. Note that if the user deletes a node, all attached edges will also be deleted. Edges cannot exist in this software without both associated nodes.

A.5.2 Outside a Bounding Box

When the user right-clicks on blank space not inside the bounding box of a grouped area, a popup menu is generated with all applicable edit options.

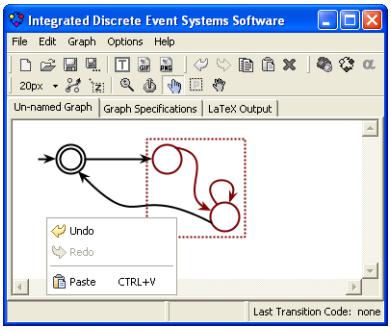

Figure A.41: Outside the bounding box.

Undo

The **Undo** option was not implemented as a real undo. Every five **actions** (loosely defined) the IDES software

remembers a snapshot of the system. These are saved in the IDES/system folder. Undo therefore does let the user back-up, but not ideally.

Redo

The **Redo** option was not implemented as a real redo. Every five **actions** (loosely defined) the IDES software remembers a snapshot of the system. These are saved in the IDES/system folder. Redo lets the user move forward through these snapshots when the user has moved backwards through them using Undo. Note that if the user activates Undo, then makes any changes (even just clicking on a node) the ability to Redo can be lost.

Paste

The **Paste** option adds copies to the graph of whatever graph elements the user last copied. The new elements will appear centered at the user's click location.

A.5.3 Edges

When the user right-clicks on a graph object regardless of whether it is inside a bounding box or not, a popup menu is generated that applies to the selected object only. The popup menu for an edge may be activated by right clicking on its arrowhead. When the edge is drawing its anchors and/or label, they may also be used as valid right click targets.

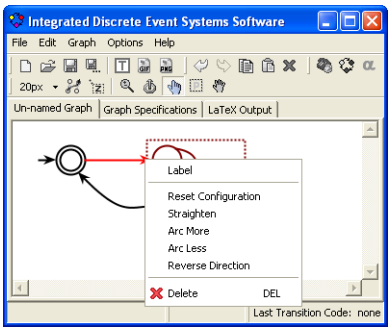

Figure A.42: Edge right-click popup menu.

Label

The **Label** option activates the label chooser which can also be activated by double-clicking on the arrowhead or label of the edge. If no edge transitions have been specified, then this menu forwards the user to the Graph Specification tab. Labels are described in the **Labels** section of this guide.

Reset Configuration

The **Reset Configuration** option causes the edge to revert to its default configuration based on the automatic layout algorithms.

Straighten

The **Straighten** option causes the edge to reconfigure as a straight line.

Arc More

The **Arc More** option causes the edge to increase its arc. It is a primitive feature and does not work very well.

Arc Less

The **Arc Less** option causes the edge to decrease its arc. It is a primitive feature and does not work very well.

Reverse Direction

The **Reverse Direction** option causes the edge to reverse direction.

Delete

The **Delete** option removes the edge from the graph. The user will not be warned or asked for confirmation.

A.5.4 Nodes

When the user right-clicks on a graph object regardless of whether it is inside a bounding box or not, a popup menu is generated that applies to the selected object only. The popup menu for a node may be activated by right clicking inside the node.

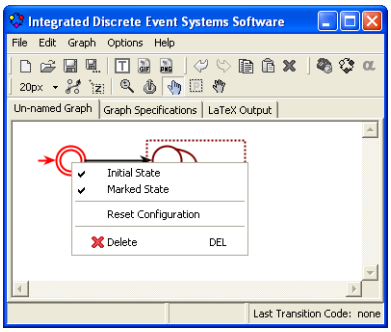

Figure A.43: Node right-click popup menu.

Initial State

The **Initial State** option sets the node as the initial state of the graph. Only one node can be the initial state, if another was previously set as the initial state, its setting is revoked. The initial state is designated by an incoming arrow symbol. This symbol can be rotated about the node by clicking and dragging it with the modify tool. The user can also use this menu to uncheck this option.

Marked State

The **Marked State** option sets or unsets the node as marked. Marked nodes draw with a double circle.

Reset Configuration

The **Reset Configuration** option causes all edges attached to the node to revert to their default configurations based on the automatic layout algorithms.

Delete

The **Delete** option deletes the node and all attached edges. The user will not be warned or asked for confirmation. If the user makes a mistake, the Undo action can be helpful, but it will go backwards several steps. Edges cannot exist in this software without both associated nodes.

A.6 Labels

There are two options for drawing Labels: Glyphs and LaTeX. Glyphs are faster, native to Java and look nicer on the IDES software screen and in exported PNGs and GIFs. Glyphs only support regular keyboard symbols. LaTeX is slower, requires that the user has GhostScript and MiKTeX installed, and looks somewhat unattractive on the IDES software screen and in exported PNGs and GIFs. LaTeX supports all LaTeX commands, so the user is able to make much more diverse labels, and it works very well in the export to LaTeX—meaning that even though it may not look attractive in the IDES software, it will look good in the exported LaTeX. The only current bug with LaTeX labels is with line breaks in Node labels. Because line breaks are not supported in the picture environment, the software manually breaks up the code and repositions the fragments, and it does not work perfectly.

When the **Use LaTeX for Labels** option is selected, the software will attempt

to find the locations of MiKTeX and GhostScript. If it can not find them, it will ask the user for their locations. If the user cancels out of the dialogue, or gives false information, **Use LaTeX for Labels** will be automatically deselected. The software cannot embed LaTeX in the graph without MiKTeX and GhostScript.

A.6.1 Node Labels

To add or edit the label for a node, the user may simply double-click the node when the **modify tool** is selected. This opens a popup window where the user may type the text for the node. When the user is done, they may simply click back anywhere on the main window, or hit ENTER. It was the assumption that most labels would be a single line. If the user wants multiple lines, they may hold CTRL when the hitting the ENTER key and a new line will be generated instead of closing the popup. While the popup is open it may be resized. The last used size is stored as a system variable and remembered for successive uses of the software. Figure A.44 demonstrates the input of information into the popup text box.

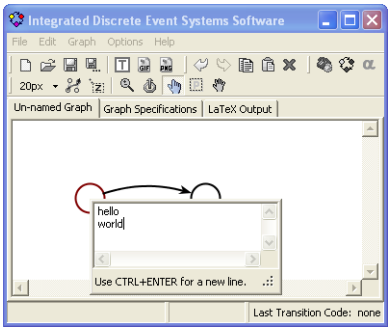

Figure A.44: Adding a label to a node.

When **Use LaTeX for Labels** is not selected the system is using glyphs. This means that exactly what the user puts in the popup text box will appear centered in the node, as shown in Figure A.45. Note that the only difference between the input and the result is that the input text box is left aligned, while the rendered

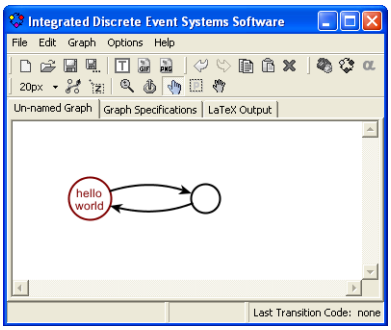

Figure A.45: The resulting Glyph label.

glyphs are centered. When **Use LaTeX for Labels** is selected, the system is using LaTeX. This means that the contents of the popup text box is the LaTeX code, while what appears in the graph is the rendered LaTeX.

When switching between glyph and LaTeX labels, if no value has been specified for the destination format, then the value specified in the current format (if any) is used. Glyph and LaTeX values are stored separately and are remembered as the user switches back and forth between them. Note that the user must use \\ in order to achieve a new line in the rendered LaTeX, but CTRL+ENTER still functions for a new line in the text box. Figure A.46 demonstrates the result of switching to LaTeX. Because no LaTeX label was previously defined for the node, it assumes the previously specified glyph value. Since LaTeX code ignores the carriage return, it renders the text as a single line.

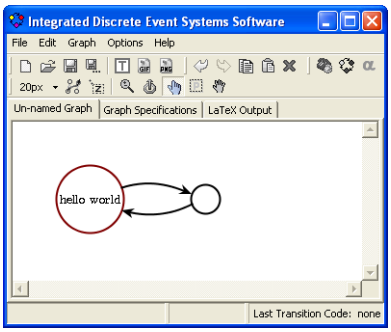

Figure A.46: The resulting LaTeX label.

Figure A.47 demonstrates a more complicated LaTeX label. Note that the poor resolution of the rendered label is due to a flaw in the implementation of the IDES software. In graphs Exported to LaTeX for use in LaTeX documents, the resolution is of high quality. Also note the careful use of the **$** symbol. If it is not use appropriately, the rendered result can be quite undesirable.

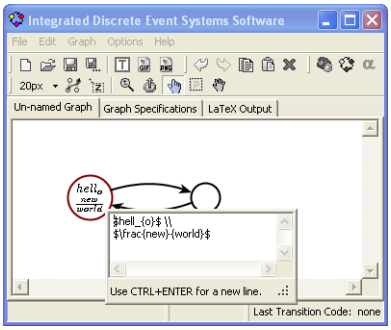

Figure A.47: A complex LaTeX example.

A.6.2 Edge Labels

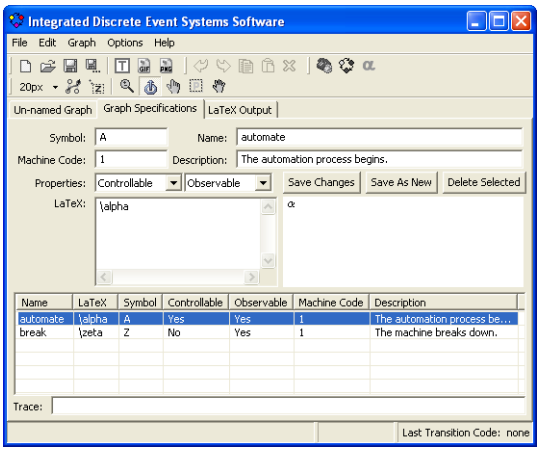

Figure A.48: The graph specifications tab.

Edge labels are very different from node labels. The user still has the same glyph/LaTeX options and behaviour, but the user must specify transitions in the **Graph Specifications** tab (shown in Figure A.48) before the labels can be added to edges. Most of the fields such as **Name**, **Description**, **Machine Code** and **Properties** are optional meta-data. **Symbol** is the value used for glyph labels, and **LaTeX** is the value used for the code for the LaTeX labels. When using LaTeX labels, the rendered version of the code appears to the right of the LaTeX specification, otherwise that area states that LaTeX rendering is disabled.

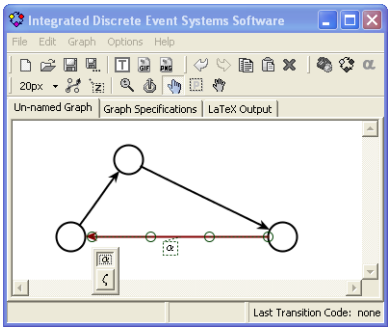

Figure A.49: The edge label chooser in LaTeX mode.

To specify transitions, the user may simply enter values into any of the boxes and click **Save As New**. To edit existing transitions, the user must click on a row in the table. This populates the top edit area with the selected data. After making the required changes, the user must click **Save Changes**. Once the user has specified transitions, it is very easy to specify edge labels. The user may use the label option from the edge's right-click menu, or with the **modify tool** selected, the user may double-click on the arrowhead or on an existing label. All these actions activate the transition chooser popup.

The transition chooser popup shown in Figure A.49 is simply an array of toggle

buttons, as the user switches them on and off, the respective values appear or disappear near the edge. When the user is finished, they may simply click anywhere off of the popup chooser. The transition chooser contains all specified transitions, and any combination of these may be associated with any edge. They appear as a comma delimited list in the order in which they are added to the edge. As demonstrated in Figure A.50, any edge that is associated with at least one uncontrollable transition is drawn as a dashed rather than a solid line. Controllability is specified by the **properties** field of the transition. Unfortunately the LaTeX **picture** environment doesn't support dashed lines, so they only appear in exported LaTeX when the **Use pstricks in LaTeX output** option is selected.

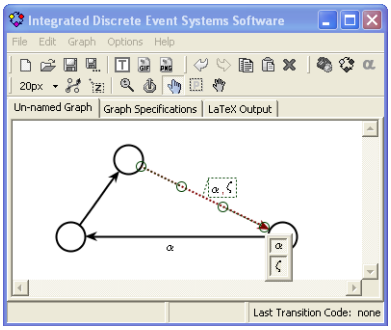

Figure A.50: An uncontrollable transition.

When switching back to glyphs, as demonstrated in Figure A.51, note that the selections in the chooser do not change. The only difference is that the displayed values are derived from the **symbol** field in the specifications table, instead of the **LaTeX** field. In the case were a transition is defined without either a value in the **symbol** field or in the **LaTeX** field, the corresponding button in the transition chooser is drawn with an empty value.

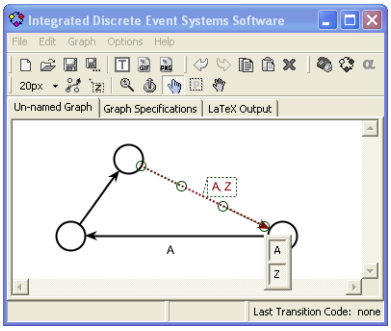

Figure A.51: The edge label chooser in glyph mode.

Note that the user can reposition the label with the **modify tool** by clicking on it and dragging it. It can be moved to any arbitrary location, but cannot be

rotated from the horizontal. Advanced users who export to LaTeX using the **pspicture** environment and the **Export LaTeX directly to TeX** option can manually modify the result to rotate labels by substituting the \put(x,y){example text} command with the \rput{angle}(x,y){example text} command.

A.7 Animated Trace

The Trace feature allows the user to initiate an animated trace of transitions in the graph. Note that in the **Graph Specifications tab** (shown in Figure A.52), each transition may be assigned a value in the **machine code** field. This feature was developed for use with custom external hardware representing a real **plant**. The external hardware was designed to transmit single bytes representing events as they occurred in the plant. The bytes were transmitted over a serial cable using the rs232 protocol. The **machine code** field specifies which bytes map to which transitions in the graph model.

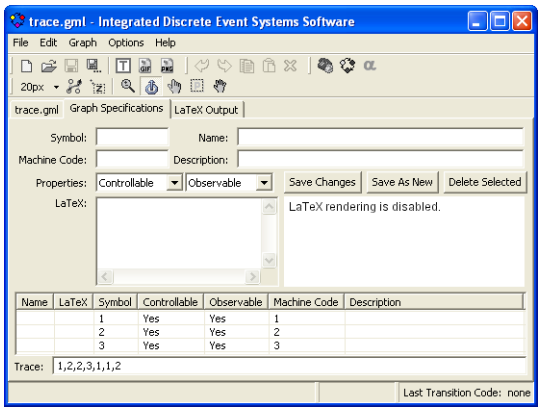

Figure A.52: Graph specifications tab containing a manual trace value.

Because the Trace feature was designed for use with external hardware, it requires that the **Connect** toolbar item first be used to establish that hardware communication is possible before starting a trace. The connect button attempts to establish that (9600 baud, 8 data bits, no parity, 1 stop bit, no flow control) communication over the COM1 port on the user's computer is possible. This should always succeed, regardless of whether or not there is any external device attached to COM1. When

connected, the button is drawn in a **depressed** mode and reads **Disconnect**. The user may click it a second time to disconnect, which would also terminate any current trace.

Once connected, a trace can be initiated by clicking the **Start Trace** toolbar button which then is drawn in a **depressed** mode and reads **Stop Trace**. The user can stop a trace at any time by clicking the trace button a second time. When a trace is started, the initial state of the graph becomes highlighted in blue, which is visible in colour versions of Figure A.53. The system then listens for transitions. When they occur, the blue highlight animates across that transition to the appropriate state, which is visible in colour versions of Figure A.54. The label area at the very bottom of the software window reports information on the current state of the trace. It displays messages when the user connects, disconnects and starts or stops a trace. It also displays any machine codes as they are received.

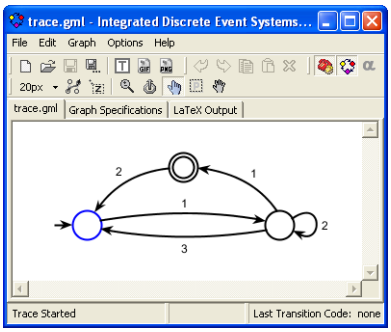

Figure A.53: An initiated trace.

If no initial state is specified, then starting a trace has no effect. If a machine code is received and no outgoing transition from the current node is associated with the received machine code then it is simply ignored. If a machine code is received and more than one outgoing transition from the current node is associated with the received machine code then one of them is arbitrarily chosen (but not randomly chosen). The

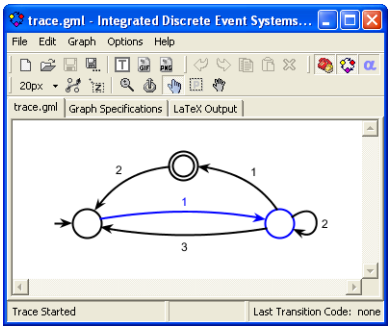

Figure A.54: A trace in mid animation.

trace continues until the **Stop Trace** toolbar button is clicked. While a trace is in progress, it is still possible to edit all data, both in the graph and in the graph specifications.

The software/machine communication is based on single byte packets which are interpreted as integer values between zero and 255. This limits the effective event space to 256 elements, but was deemed sufficient for simple modeling purposes. As an added feature, the system can also be used as a means of control. When a byte is received from COM1 and an associated outgoing transition is found, the software sends the same byte back out COM1, but if no matching transition is found, it does not echo the byte. Custom hardware can interpret this as control. If the custom hardware consistently sends controllable event codes before the events occur, it can interpret the lack of an echo as indication that the event should be disabled.

As an added feature, the **Alpha** toolbar item is a manual means of feeding custom transition values to the **Trace** system. This allows the user to animate arbitrary event strings. As with the hardware trace, the alpha mode requires that all specified machine codes be integers, but the alpha mode accepts all positive integers; whereas, the hardware mode is limited by the single byte packet size. In the alpha mode, noninteger machine codes are simply ignored.

The values to be sent by the alpha button are specified using the **Trace** text box at the bottom of the **Graph Specification tab**. This text box functions as an integer array. Any sequence of integers may be specified in it as a comma-delimited list with no spaces. When a trace is started, a pointer is initialized to the first element of that array. Each time the user clicks the Alpha button the current element of the array is transmitted to the trace system and the pointer in incremented. When the

end of the array is reached, subsequent pushes of the Alpha button are ignored. To start the animation from the beginning, the user must stop and restart the trace.

In Figure A.52 the value in the trace text box at the very bottom of the window is **1,2,2,3,1,1,2**. Note that in this case the event labels are equal to their respective machine codes, but in general they are permitted to differ. When a trace is started, the highlight goes to the initial state as shown in Figure A.53. When the Alpha button is pressed, the trace system receives a **1** and the graph animates across that transition to the next state as shown in Figure A.54. A second press of the Alpha button would cause the graph to animate across the self-loop with machine code **2**.

A.8 IDES file Specification

It was originally the intention that the IDES software utilize the GML format, but a custom format was ultimately used instead. An example of such a file is given in Listing A.3 below. No tabs characters are allowed. No blank lines are allowed. Each system variable and data name must be preceded by exactly four spaces and must be separated from its value by exactly one space. For any string value that could include a line break, the escape value **&#13** must be used instead.

The first five lines contain system variables. It is acceptable to **not** include any or all of these lines; however, including some of the variable names with empty or invalid values will cause the file to be unreadable by the IDES software. The five parameters are as follows:

grid : The snap-to-grid value for this graph (acceptable values are 0, 5, 10, 20, 30).

grid_displacement : The displacement from (0,0) that the grid and entire graph has undergone due to the move tool. Manually modifying this value will considerably

damage the graph.

scale : The current zoom state. Manually modifying this value will considerably damage the graph.

trace : The string saved in the trace text box at the bottom of the graph specifications tab.

standard_node : The boolean (1 or 0) value for the **Use standard node sizes** option.

print_area : The coordinates of the print area in the form top_left_x, top_left_y, bottom_right_x, bottom_right_y.

The second section contains the graph specifications data. The parameter names correspond exactly to the field names in the graph specifications tab of the IDES software. In the file, data sets must be numbered $[0 \ldots n-1]$ in increasing order. This ID is later used by edges to indicate which data set they include. Each of the data sets' seven parameters may be left blank except for parameters controllable and observable which are boolean values and must be valued either 0 or 1. All other parameters are strings and may be left blank.

The third section contains the nodes. Their parameters are as follows:

id : Values $[0 \ldots n-1]$ in ascending order are used by the edges to indicate to which nodes they are connected.

x : The x coordinate of the node's origin.

y : The y coordinate of the node's origin.

r : The radius of the node.

a : The attributes of the node represented as a bitwise integer, see the JavaDoc for more details.

dx : The x component of the direction vector of the initial state arrow, if it exists.

dy : The y component of the direction vector of the initial state arrow, if it exists.

l : The glyph label for the node.

c : The code for generating the LaTeX label for the node.

The final section contains the edges. Their parameters are as follows:

source : The id of the node at which this edge originates.

target : The id of the node at which this edge terminates.

x1 : The x coordinate of the tail of the Bezier curve.

y1 : The y coordinate of the tail of the Bezier curve.

ctrlx1 : The x coordinate of the control point for the tail of the Bezier curve.

ctrly1 : The y coordinate of the control point for the tail of the Bezier curve.

ctrlx2 : The x coordinate of the control point for the head of the Bezier curve.

ctrly2 : The y coordinate of the control point for the head of the Bezier curve.

x2 : The x coordinate of the head of the Bezier curve.

y2 : The y coordinate of the head of the Bezier curve.

dx : The x component of the direction vector of a self-loop curve.

dy : The y component of the direction vector of a self-loop curve.

tdi : A comma-delimited list of data IDs associated with this edge.

gtx : The x component of the displacement of the top left corner of the label from the midpoint of the curve.

gty : The y component of the displacement of the top left corner of the label from the midpoint of the curve.

a : The attributes of the edge represented as a bitwise integer, see the JavaDoc for more details.

Listing A.3 Example of the IDES file format

```
graph [
    grid 20
    grid_displacement (-36,-38)
    scale 1.0
    trace
    standard_node 0
    print_area 14,13,375,371
    data0.NAME Want
    data0.LATEX $w_{1}$
    data0.SYMBOL w1
    data0.CONTROLLABLE 1
    data0.OBSERVABLE 0
    data0.MACHINE_CODE
    data0.DESCRIPTION Suddenly wants the resource.
    data1.NAME Grant
    data1.LATEX $g_{1}$
    data1.SYMBOL g1
    data1.CONTROLLABLE 0
    data1.OBSERVABLE 0
    data1.MACHINE_CODE
    data1.DESCRIPTION Suddenly is granted the resource.
    node [
        id 0
        x 64
        y 62
        r 19
        a 3
        dx 0.77724487
        dy 0.6291982
        l I,I
        c I I
    ]
    node [
        id 1
        x 204
        y 62
        r 19
        a 0
        l R,I
        c R I
    ]
    edge [
        source 0
        target 1
        x1 83.0
        y1 62.0
        ctrlx1 102.0
        ctrly1 62.0
        ctrlx2 146.0
        ctrly2 62.0
        x2 178.0
        y2 62.0
        dx 0.0
        dy 0.0
        tdi 0
        gtx 5
        gty 5
        a 0
    ]
]
```

Appendix B

A Vending Machine

B.1 Summary

A pop vending machine is the classic textbook example of a finite-state machine. It was, therefore, an obvious choice for a hardware model for the purpose of demonstrating the implementation of DES control theory. Immediately, however, two problems arise. The first problem involves the definition of the plant. It is unclear if the plant should be equal to all definable behaviour—every sequence of every component of the machine acting randomly, or if the plant should be defined as the legal specification—which greatly reduces the usefulness of DES control theory. The second problem involves generation of events. In a hardware example such as a pop vending machine, it is clear that all inputs are uncontrollable events, and all outputs are controllable events. There is also a clear causality relationship between inputs and outputs not neatly captured by standard DES control theory. A naïve implementation might assume that the plant should continuously and randomly generate controllable events while the structure of the supervisor maintains the necessary causality relationship

between inputs and outputs.

B.2 Circuit

The vending machine model was developed for the purpose of investigating what issues may arise in the application of DES control theory to the design of a simple machine and for the purpose of demonstrating interaction of the IDES software with a real plant. A photo of the implementation is shown in Figure B.1.

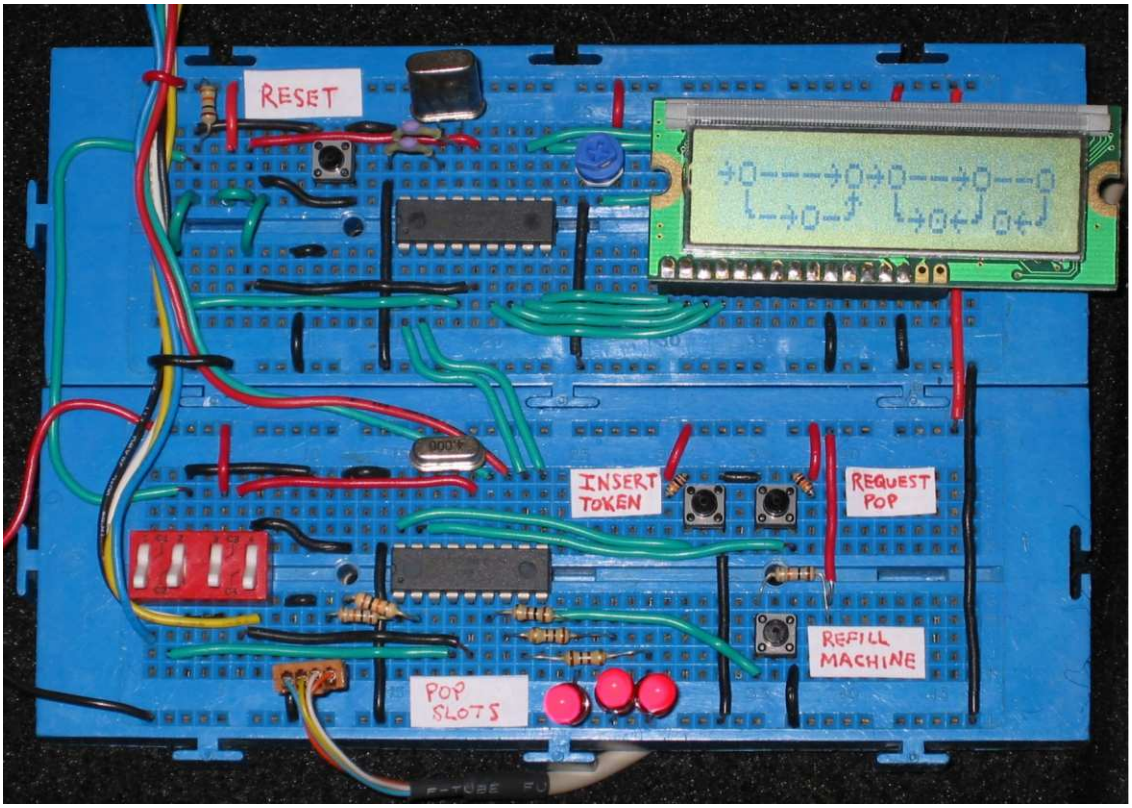

Figure B.1: A photograph of the vending machine model.

This hardware and several variations of it were used in investigation into the issues and principles involved in implementation of DES control theory. It is composed of two PIC16F84 microcontrollers, an LCD, some push buttons and some LEDs. The switches and wiring to the left of the microcontrollers are necessary for programming the chips and are not part of the circuit. Two microcontrollers are necessary in this model because of the limited number of IO pins on the PIC16F84. The upper microcontroller manages the LCD and functions as a slave unit to the lower microcontroller. The master microcontroller delegates simple commands to the slave microcontroller, allowing alternate views of this system as one of distributed control. The master microcontroller is also connected to the COM1 port of a PC for integration with the IDES software. A clean circuit diagram is provided in Figure B.2.

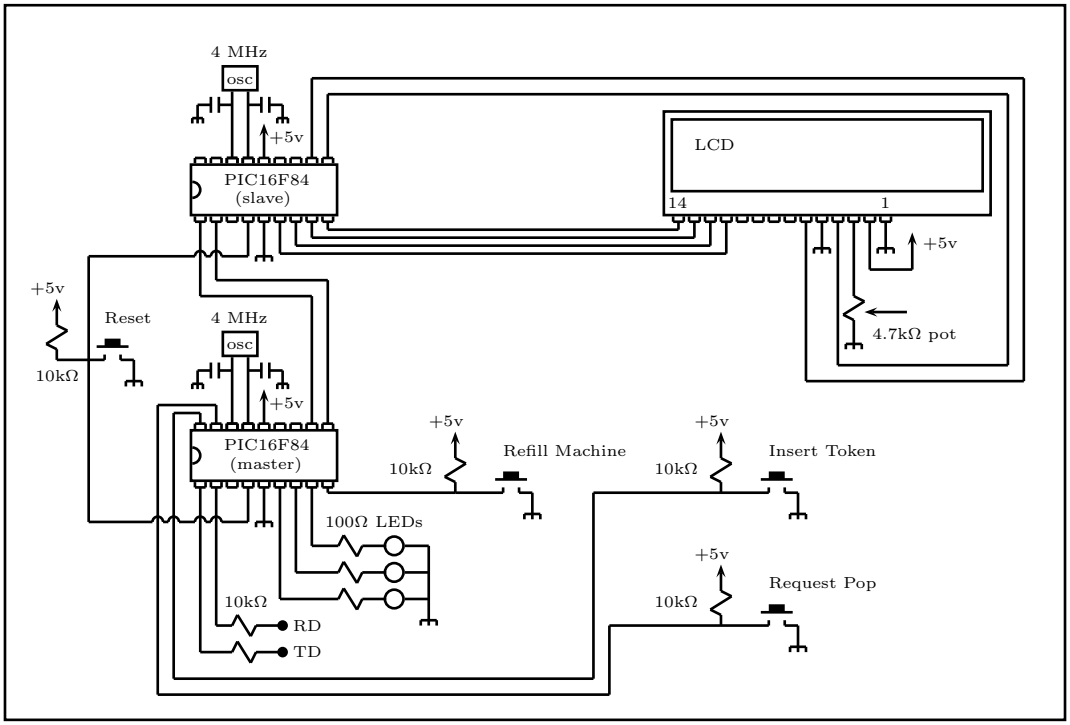

Figure B.2: The circuit diagram for the vending machine model.

The machine works as follows. The pushbuttons allow input from a human user. The LCD allows output to the human user in the form of text messages such as the current number of credits in the machine or whether or not a requested type of pop is available. The LEDs represent pop in the vending machine. When they are lit, they represent individual units of pop in the machine. The act of an LED becoming unlit represents the delivery of a pop from the machine to a human user.

B.3 Model

This particular incarnation of the model functions according to the following specification. The machine will accept only one form of currency, which we will call a token. The machine will dispense only one form of pop which we will call pop. The machine can contain zero, one, two, or three pops. A technician can refill the machine, bringing the current number of pops to three. The machine has memory and keeps track of the current number of credits. Credits are tokens that have been inserted into the machine, but not spent. When a pop is dispensed by the machine, the current number of credits is decremented by three unless the current number of credits was less than three in which case the credit total is unchanged. This undesirable behaviour (pop dispensed, but credits not decremented) is called lost pop. Finally, the machine can be reset, which sets the current number of credits to zero and the current number of pops to three.

Symbol	Description
T	A token was accepted bringing the current credit total to at least the cost of one pop.
P+	A pop was dispensed by the machine and the credit total was decremented by three, resulting in a credit total greater than or equal to the cost of one pop.
R	The pop machine was refilled by the technician.
L	A pop was dispensed even though the credit total was less than the cost of one pop.
P	A pop was dispensed by the machine and the credit total was decremented by three, resulting in a credit total less than the cost of one pop.

Table B.1: The event space of the plant.

Figure B.3 provides an abstract plant representation of the pop machine. This plant has no knowledge of the exact number of credits in the machine and provides a simple event-based view of the system. The event space of the plant is given in Table B.1. This abstracted view of the machine follows from the structure of the program on the microcontroller. The source code for the master microcontroller as given in Listing C.1 shows that when certain logical events occur within the system it sends a byte out the serial cable to the COM1 port of an attached PC. Specifically, it uses the values 7, 8, 9, 10 and 11 to represent the five events within the plant. Consequently, when the IDES software has the plant model of Figure B.3 loaded, and is connected to the hardware model via the trace feature, then as events actually occur within the plant (due to human interaction) the current highlighted state in the IDES software updates appropriately.

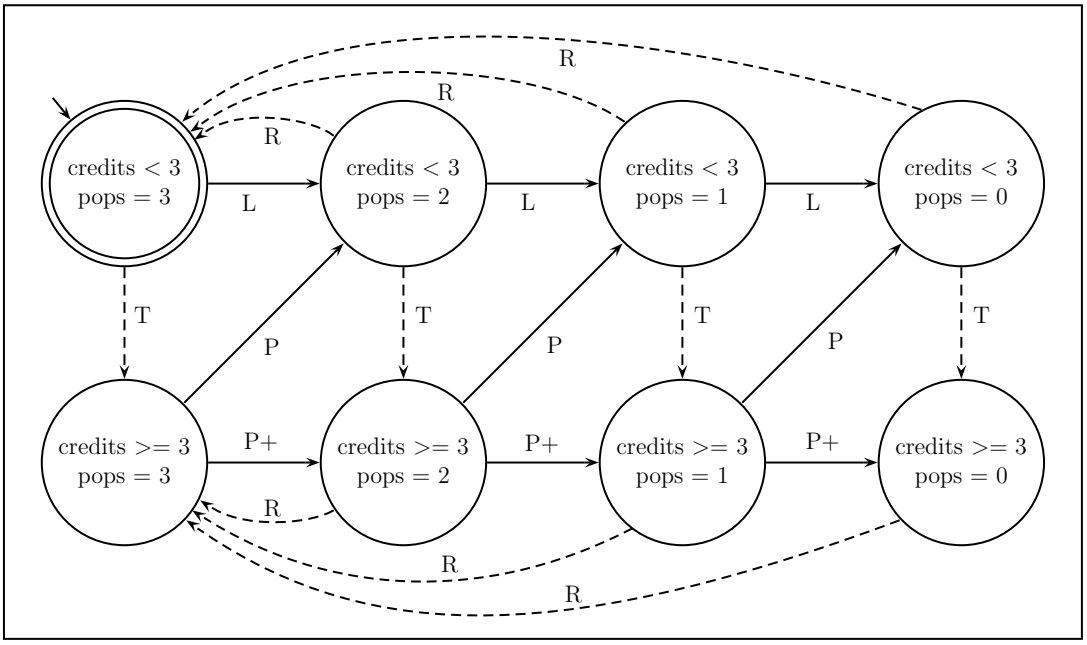

Figure B.3: The plant.

For a controllable event such as L, notification is sent to the PC before the event occurs. If the PC does not echo the event code, the microcontroller assumes that the event has been disabled and does not carry through with the event sequence. The legal specification as shown in Figure B.4 is controllable with respect to the plant. This means it can serve as an implicit supervisor. Consequently, when the IDES software has the legal model of Figure B.4 loaded, and is connected to the hardware model via the trace feature, then as events actually occur within the plant (due to human interaction) and the highlight moves within the legal model, the L event is prevented from occurring because it is not found in the model and therefore not echoed back to the microcontroller. This means that when the credits are less than three and a human user presses the **Request Pop** button in the controlled mode, the number of pops are not decremented (lost pop is disabled); whereas, they would

be decremented in the uncontrolled mode.

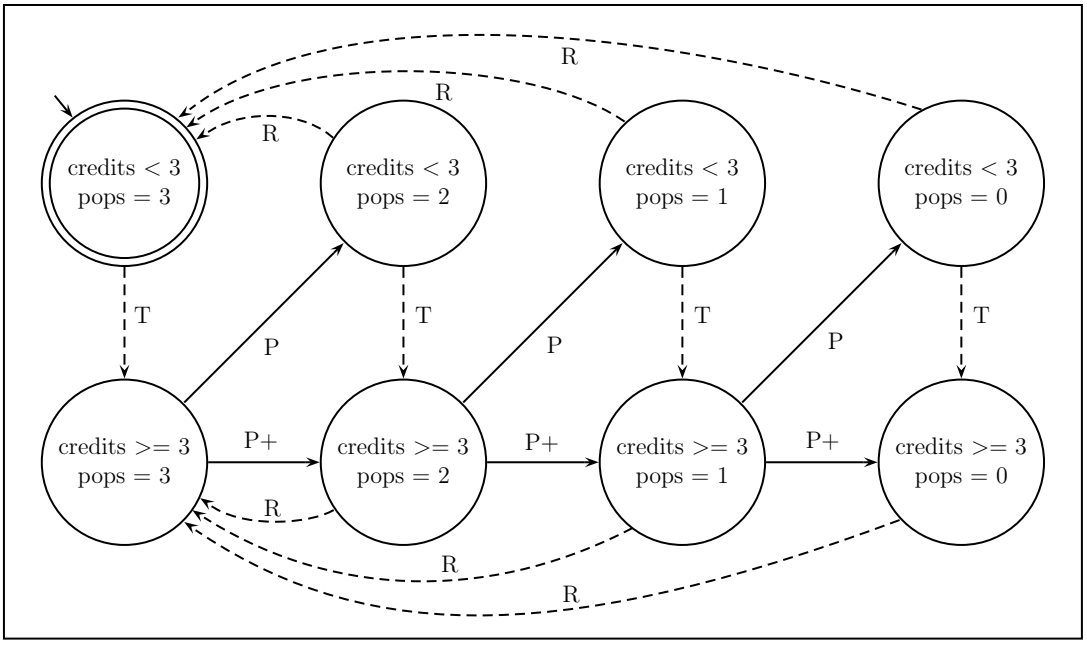

Figure B.4: The legal language.

This model therefore functions as a real implementation of DES control theory and demonstrates the effectiveness of the IDES software as a tool both for understanding a plant's behaviour (trace in the uncontrolled mode) and functioning as a supervisor (trace in the controlled mode). Of course, this system is somewhat contrived and it is clear that the separation of the plant logic (in the microcontrollers) and the supervisor logic (on the PC) is considerably more work than an integrated solution where the microcontrollers are programmed with correct logic and do not require the aid of an external decision-making entity.

The source code for the master and slave microcontrollers are given in Listings C.1, C.2, C.3 and C.4. The most interesting logic is contained in the source code for the master in Listing C.1. This is written in assembly language and contains all

the logic for encapsulating the plant model and interfacing with the IDES software on a PC. The other three listings contain the source code for the slave unit and are written in C. The C was compiled using the PICC LITE compiler and three simple files delay.c, delay.h, and pic.h were omitted.

Listing B.1 PIC16F84 assembly language code for the master microcontroller.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  notes  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; - interrupt flags are set when the interrupt condition occurs, regardless of the interrupt enable
;   bit; therefore, clear the flags before you enable the interrupts.
; - in order to access the TRIS registers, bank 1 must be selected

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  directives  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                LIST P=16F84
                INCLUDE "p16f84.inc"
                ERRORLEVEL -224                         ;  supress irrelevant error messages
                __CONFIG _PWRTE_ON & _XT_OSC & _WDT_OFF  ;  choose the configuration fuses

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  equates  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

START_ADDRESS  EQU  h'0000'      ;  where the program starts after a reset
ISR_ADDRESS    EQU  h'0004'      ;  where the machine goes when an interrupt is serviced

PC_PORT        EQU  h'0005'      ;  PORTA, the rs232 lines for pic-pc communication are here
PC_TD          EQU  d'2'
PC_RD          EQU  d'3'

MC_PORT        EQU  h'0006'      ;  PORTB, the rs232 lines for pic-pic communication are here
MC_TD          EQU  d'4'
MC_RD          EQU  d'5'

LED_PORT       EQU  h'0006'      ;  PORTB, the port where the pop led indicators are connected
LED            EQU  d'0'
NON_LEDS       EQU  b'11111000'  ;  the pins on the LED_PORT that aren't for LEDs

TOKEN_PORT     EQU  h'0005'      ;  PORTA, the port where the "Insert Token" button is connected
TOKEN          EQU  d'1'

POP_PORT       EQU  h'0005'      ;  PORTA, the port where the "Request Pop" button is connected
POP            EQU  d'0'

RFILL_PORT     EQU  h'0006'      ;  PORTB, the port where the "Refill Pop" button is connected
RFILL          EQU  d'3'

FLAG_T_R       EQU  d'0'         ;  0th bit of flags :: 0 -> use transmit isr :: 1 -> use receive isr
FLAG_PC_MC     EQU  d'1'         ;  1th bit of flags :: 0 -> use PC channel   :: 1 -> use MC channel
FLAG_TOKEN     EQU  d'2'         ;  2th bit of flags :: last known state of token button (1=unpushed)
FLAG_POP       EQU  d'3'         ;  3th bit of flags :: last known state of pop button (1=unpushed)
FLAG_RFILL     EQU  d'4'         ;  4th bit of flags :: last known state of refill button (1=unpushed)

OP_CLEAR       EQU  d'0'         ;  LCD opcodes
OP_LOGO        EQU  d'1'
OP_DISPENSE    EQU  d'2'
OP_TOKENS      EQU  d'3'
```

```
OP_TOKEN_POP   EQU  d'4'
OP_TOKEN_MAX   EQU  d'5'
OP_ERROR       EQU  d'6'
OP_REFILL      EQU  d'7'        ;  pc code, machine refilled
OP_SUFFICIENT  EQU  d'8'        ;  pc code, there is now enough cash for a pop
OP_POP         EQU  d'9'        ;  pc code, pop delivered, not enough cash for another
OP_POP_PLUS    EQU  d'10'       ;  pc code, pop delivered, remains enough cash for another
OP_LOST        EQU  d'11'       ;  pc code, pop delivered, but not paid for
OP_RESET       EQU  d'255'      ;  pc code, hard reset

NULL           EQU  d'0'
MAX_TOKENS     EQU  d'6'        ;  the total #of current credits the machine is willing to accept
POP_COST       EQU  d'3'        ;  the number of tokens required for a pop
FULL_STATE     EQU  b'00000111'  ;  the state of the pops variable when the machine is full

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  data  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                CBLOCK h'0C'    ;  set up general registers starting at address 12
                  counter       ;  a general counting register
                  big_count     ;  a second general counting register
                  huge_count    ;  a third general counting register
                  character     ;  holds a character either sent or received by the rs232 protocol
                  char_count    ;  keeps track of which bit should be sent next in the rs232 protocol
                  temp_char     ;  for rs232 sends, allows send without destroying value in character
                  flags         ;  flags for various control decisions
                  temp          ;  a general temp register
                  w_temp        ;  to save state of W while in an isr
                  status_temp   ;  to save state of STATUS while in an isr
                  tokens        ;  total number of credits
                  pops          ;  pop bit flags xxxxx111, 1 implies pop is in machine and led is lit
                ENDC

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  start  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                ORG START_ADDRESS     ;  the fixed start address
                goto initialization

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  interrupt service routine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                ORG ISR_ADDRESS       ;  the fixed interrupt vector

                movwf w_temp          ;  save the machine state
                swapf STATUS, W
                movwf status_temp

                btfss flags, FLAG_T_R ;  decide which isr handler to use
                call rs232_transmit
                btfsc flags, FLAG_T_R
                call rs232_receive

                bcf INTCON, TOIF      ;  clear TOIF to prevent infinite interrupt loop

                swapf status_temp, W  ;  restore the machine state
                movwf STATUS
                swapf w_temp, F
```

```
                swapf w_temp, W

                retfie                ;  this automatically sets GIE
                                      ;  (which was automatically cleared when the interrupt occurred)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  rs232 transmit isr subroutine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rs232_transmit  movlw d'161'          ;  every time the interrupt occurs,
                                      ;  load TMR0 with 161, so the count is 161-256
                                      ;  (i.e., 95 steps = 95us)
                                      ;  recall that the isr latency and preceeding code consumes time
                movwf TMR0

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rt_start_bit    movlw d'10'           ;  check if this is the start bit
                subwf char_count, W   ;  result is stored in W, status register is modified
                btfss STATUS, Z       ;  zero bit is zero only when result is NOT zero
                goto rt_end_bit

                movf character, W
                movwf temp_char

                btfsc flags, FLAG_PC_MC  ;  choose channel, start bit is a one
                goto $+3
                bsf PC_PORT, PC_TD
                goto $+2
                bsf MC_PORT, MC_TD

                goto rt_done

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rt_end_bit      movlw d'1'               ;  check if this is the end bit
                subwf char_count, W      ;  result is stored in W, status register is modified
                btfss STATUS, Z          ;  zero bit is zero only when result is NOT zero
                goto rt_data_bit

                btfsc flags, FLAG_PC_MC  ;  choose channel, end bit is a zero
                goto $+3
                bcf PC_PORT, PC_TD
                goto $+2
                bcf MC_PORT, MC_TD

                goto rt_done

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rt_data_bit     btfsc temp_char, 0       ;  for data bits, write the negative of the current bit
                                         ;  of character onto the TD pin
                goto rt_clear_bit

rt_set_bit      btfsc flags, FLAG_PC_MC  ;  choose channel
                goto $+3
                bsf PC_PORT, PC_TD
                goto $+2
                bsf MC_PORT, MC_TD

                goto rt_rotate_bit
```

```
rt_clear_bit    btfsc flags, FLAG_PC_MC  ;  choose channel
                goto $+3
                bcf PC_PORT, PC_TD
                goto $+2
                bcf MC_PORT, MC_TD

rt_rotate_bit   rrf temp_char, F         ;  advance to the next bit

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rt_done         decfsz char_count, F     ;  if all bits have been sent, disable the timer0 interrupt
                goto $+2
                bcf INTCON, T0IE

                return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  rs232 receive isr subroutine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rs232_receive   movlw d'161'          ;  every time the interrupt occurs,
                                      ;  load TMR0 with 161, so the count is 161-256
                                      ;  (i.e., 95 steps = 95us)
                                      ;  recall that the isr latency and preceeding code consumes time
                movwf TMR0

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rd_end_bit      movlw d'1'               ;  check if this is the end bit
                subwf char_count, W      ;  result is stored in W, status register is modified
                btfsc STATUS, Z          ;  zero bit is zero only when result is NOT zero
                goto rd_done             ;  do nothing for the end bit

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rd_data_bit     rrf character, F         ;  advance to the next bit

                btfsc flags, FLAG_PC_MC  ;  choose channel
                goto $+4
                btfsc PC_PORT, PC_RD     ;  write the negative of the current bit
                                         ;  on the RD pin into the character register
                goto rd_clear_bit
                goto rd_set_bit
                btfsc MC_PORT, MC_RD     ;  write the negative of the current bit
                                         ;  on the RD pin into the character register
                goto rd_clear_bit
                goto rd_set_bit

rd_set_bit      bsf character, 7
                goto $+2
rd_clear_bit    bcf character, 7

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rd_done         decfsz char_count, F  ;  if all bits have been received, disable the timer0 interrupt
                goto $+2
                bcf INTCON, T0IE

                return
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  rs232 transmit initializer  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

send_char       bcf flags, FLAG_T_R

                movlw d'10'               ;  set count_char = #of bits to be sent in the rs232 protocol
                movwf char_count

                bsf INTCON, T0IF          ;  we wish to start the interrupt sequence right away

                bsf INTCON, T0IE          ;  enable the timer overflow interrupt

dead_send       btfsc INTCON, T0IE        ;  dead loop until the transaction is complete
                goto dead_send            ;  and the isr routine has disabled T0IE

                movlw h'0f'               ;  kill some time just to be safe
                movwf counter
                decfsz counter, F
                goto $-1

                return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  rs232 receive initializer  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

get_char        btfsc flags, FLAG_PC_MC   ;  choose channel
                goto gc_mc_poll
gc_pc_poll      btfss PC_PORT, PC_RD      ;  poll for the start bit
                goto gc_pc_poll
                goto $+3
gc_mc_poll      btfss MC_PORT, MC_RD      ;  poll for the start bit
                goto gc_mc_poll

                bsf flags, FLAG_T_R

                movlw d'9'                ;  set count_char = #of to be received in the rs232 protocol
                movwf char_count

                movlw d'146'              ;  load TMR0 with 146, so the count is 146-256 (i.e. 110us)
                movwf TMR0
                bcf INTCON, T0IF          ;  because it may already be set

                bsf INTCON, T0IE          ;  enable the timer overflow interrupt

dead_get        btfsc INTCON, T0IE        ;  dead loop until the transaction is complete
                goto dead_get             ;  and the isr routine has disabled T0IE

                return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  initialization  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

initialization  bsf STATUS, RP0           ;  select bank 1

                movlw b'00001011'         ;  specify the io mask for PORTA (1=input, 0=output)
                movwf TRISA

                movlw b'00101000'         ;  specify the io mask for PORTB (1=input, 0=output)
```

```
                movwf TRISB

                movlw b'00001000'        ;  initialize the option register
                movwf OPTION_REG         ;  (TMR0 based on internal clock with 1:1 prescalar)

                bcf STATUS, RP0          ;  select bank 0

                movlw b'00000000'        ;  start with the interrupt control register zeroed
                movwf INTCON             ;  (everything disabled)

                clrf PORTA               ;  start with zero in PORTA (note TD pin must = 0)
                clrf PORTB               ;  start with zero in PORTB

                bsf flags, FLAG_TOKEN    ;  last known state of the token button is un-pushed = 1
                bsf flags, FLAG_POP      ;  last known state of the pop button is un-pushed = 1
                bsf flags, FLAG_RFILL    ;  last known state of the refill button is un-pushed = 1

                clrf tokens              ;  start with zero tokens in the system

                movlw FULL_STATE         ;  initialize the machine to be full of pops
                movwf pops
                iorwf LED_PORT, F

                bsf INTCON, GIE          ;  globally enable interrupts

                call short_wait          ;  for the benefit of the rs232 start bit

                movlw OP_LOGO            ;  notify mc of the reset event
                movwf character
                bsf flags, FLAG_PC_MC
                call send_char           ;  mc opcode
                call send_char           ;  mc null data

                movlw OP_RESET           ;  notify pc of the reset event
                movwf character
                bcf flags, FLAG_PC_MC
                call send_char           ;  pc opcode

                call busy_wait           ;  for the benefit of the slave startup screen

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  main  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

main            nop

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

token_test      btfsc flags, FLAG_TOKEN  ;  test the last known state
                goto tt_unpushed
                goto tt_pushed

tt_unpushed     btfss TOKEN_PORT, TOKEN  ;  last known state was un-pushed (i.e. 1)
                bcf flags, FLAG_TOKEN    ;  update last know state
                goto tt_done

tt_pushed       btfss TOKEN_PORT, TOKEN  ;  last known state was pushed (i.e. 0)
                goto tt_done
                bsf flags, FLAG_TOKEN    ;  update last known state
                call insert_token
```

```
tt_done         nop

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

pop_test        btfsc flags, FLAG_POP    ;  test the last known state
                goto pt_unpushed
                goto pt_pushed

pt_unpushed     btfss POP_PORT, POP      ;  last known state was un-pushed (i.e. 1)
                bcf flags, FLAG_POP      ;  update last know state
                goto pt_done

pt_pushed       btfss POP_PORT, POP      ;  last known state was pushed (i.e. 0)
                goto pt_done
                bsf flags, FLAG_POP      ;  update last know state
                call request_pop

pt_done         nop

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

fill_test       btfsc flags, FLAG_RFILL  ;  test the last known state
                goto ft_unpushed
                goto ft_pushed

ft_unpushed     btfss RFILL_PORT, RFILL  ;  last known state was un-pushed (i.e. 1)
                bcf flags, FLAG_RFILL    ;  update last know state
                goto ft_done

ft_pushed       btfss RFILL_PORT, RFILL  ;  last known state was pushed (i.e. 0)
                goto ft_done
                bsf flags, FLAG_RFILL    ;  update last known state
                call rfill_pop

ft_done         nop

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                goto main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  insert token routine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

insert_token    movlw MAX_TOKENS      ;  test if we are willing to accept more tokens
                subwf tokens, W
                btfsc STATUS, Z       ;  zero bit is zero only when result is NOT zero
                goto done_token       ;  when at max, just ignore the input

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                incf tokens, F        ;  increment the number of tokens

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                bsf flags, FLAG_PC_MC ;  notify the mc of the new total
                movlw OP_TOKENS
                movwf character
                call send_char        ;  mc opcode
                movf tokens, W
                movwf character
```

```
                call send_char       ;  mc data

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                movlw POP_COST       ;  test if we just crossed the cost threshold
                subwf tokens, W
                btfss STATUS, Z      ;  zero bit is zero only when result is NOT zero
                goto done_token      ;  when at max, just ignore the input

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                bcf flags, FLAG_PC_MC ;  notify the pc that there are now enough tokens to buy a pop.
                movlw OP_SUFFICIENT
                movwf character
                call send_char

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

done_token      return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  request_pop routine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

request_pop     btfss pops, 0        ;  test if there are any pops in the machine (0th bit last pop)
                goto done_pop        ;  when there are not enough pops, simply ignore the request

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                movlw POP_COST       ;  test if there are enough tokens to purchase a pop
                subwf tokens, W
                btfss STATUS, C      ;  carry bit is a one only when the result is non-negative
                goto lost_pop        ;  when there are not enough tokens, generate the lost pop event

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                rrf pops, F          ;  remove the pop
                movlw NON_LEDS
                iorwf pops, W
                andwf LED_PORT, F

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                movlw POP_COST       ;  pay for pop
                subwf tokens, F

                movlw OP_TOKENS      ;  tell the mc to display new token total
                movwf character
                bsf flags, FLAG_PC_MC
                call send_char       ;  mc opcode
                movf tokens, W
                movwf character
                call send_char       ;  mc data

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                movlw POP_COST       ;  test if there remains enough credits to buy another pop
                subwf tokens, W
                btfss STATUS, C      ;  carry bit is a one only when the result is non-negative
                goto below_pop
```

```
above_pop       movlw OP_POP_PLUS     ;  tell the pc that a pop has just been delivered,
                movwf character       ;  and there are enough credits for another
                bcf flags, FLAG_PC_MC
                call send_char        ;  pc opcode
                goto done_pop

below_pop       movlw OP_POP          ;  tell the pc that a pop has just been delivered,
                movwf character       ;  and there are not enough credits for another
                bcf flags, FLAG_PC_MC
                call send_char        ;  pc opcode
                goto done_pop

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

lost_pop        movlw OP_LOST         ;  tell the pc that a pop has just been delivered, but not paid
                movwf character
                bcf flags, FLAG_PC_MC
                call send_char        ;  pc opcode

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                call get_char
                movlw OP_LOST
                subwf character, W
                btfss STATUS, Z       ;  zero bit is zero only when result is NOT zero
                goto done_pop         ;  only remove the pop if the controller allows this event
                rrf pops, F           ;  remove the pop
                movlw NON_LEDS
                iorwf pops, W
                andwf LED_PORT, F

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

done_pop        return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  rfill_pop routine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

rfill_pop       movlw FULL_STATE      ;  set the machine to be full of pops
                movwf pops
                iorwf LED_PORT, F

                movlw OP_REFILL       ;  tell the pc that a pop has just been delivered
                movwf character
                bcf flags, FLAG_PC_MC
                call send_char        ;  pc opcode

                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  simple busy wait routine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

busy_wait       movlw d'3'
                movwf huge_count
                movlw h'ff'
                movwf big_count
                movlw h'ff'
```

```
                movwf counter
                decfsz counter, F
                goto $-1
                decfsz big_count, F
                goto $-5
                decfsz huge_count, F
                goto $-9
                return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  short busy wait routine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

short_wait      movlw h'ff'
                movwf big_count
                movlw h'ff'
                movwf counter
                decfsz counter, F
                goto $-1
                decfsz big_count, F
                goto $-5
                return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  end  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                end                     ; end of program
```

Listing B.2 C language code for the slave microcontroller (main.c)

```
#include <pic.h>
#include "delay.c"
#include "lcd.c"
#include "rs232.c"

__CONFIG(WDTDIS & XT & UNPROTECT);

/*****************************************************************************************
 * Print the total tokens to the LCD.
 */
void print_tokens(unsigned char tokens)
{
  unsigned char high_digit;
  unsigned char low_digit;

  high_digit = tokens/10;
  low_digit = tokens-(high_digit*10);

  lcd_overwrite_string("Total Tokens: ");
  lcd_write_char(high_digit + 0x30);
  lcd_write_char(low_digit + 0x30);
}

/*****************************************************************************************
 * Main Loop.
 * Initailze the system while displaying "Resetting: ..." on the LCD.
 * After the initialization delay, display the custom logo on the LCD.
 * Loop, listening for command/data byte pairs and executing accordingly.
 *
 * command 0 -> Clear the LCD
 * command 1 -> Draw the logo on the LCD
 * command 2 -> Display pop attempt message
 * command 3 -> Display tokens total (from received data)
 * command 4 -> Display pop delivered message
 * command 5 -> Display token rejected message
 */
main(void)
{
  unsigned char command;
  unsigned char data;

  TRISA = 0b00001000;
  TRISB = 0b00000000;

  rs232_initialize();                  // initalize the devices
  lcd_initialize();

  lcd_overwrite_string("Resetting: ");  // initalization delay and prompt
  delay_ms(250);
  lcd_write_char('.');
  delay_ms(250);
  lcd_write_char('.');
  delay_ms(250);
  lcd_write_char('.');
  delay_ms(250);
```

```
  lcd_draw_logo();                      // draw the startup logo

  while(1==1)
  {
    command = rs232_get_char();         // receive a command byte
    data = rs232_get_char();            // receive a data byte

    if (command == 0) { lcd_clear_and_home(); }  // clear the LCD

    if (command == 1) { lcd_draw_logo(); }       // Draw the logo on the LCD

    else if (command == 2)                       // Display pop attempt message
    {
      lcd_overwrite_string("Now attempting");
      lcd_write_string_at_second_line("to deliver pop.");
    }

    else if (command == 3) { print_tokens(data); }  // Display tokens total (from received data)

    else if (command == 4)                       // Display pop delivered message
    {
      print_tokens(data);
      lcd_write_string_at_second_line("Pop Delivered!");
    }

    else if (command == 5)                       // Display token rejected message
    {
      print_tokens(data);
      lcd_write_string_at_second_line("Token Rejected!");
    }

    else                                         // Display an error message and loop infinitely
    {
      lcd_overwrite_string("Fatal Error:");
      lcd_write_string_at_second_line("Requires Reset.");
      while(1==1) { continue; }
    }
  }
}
```

Listing B.3 C language code for the slave microcontroller (lcd.c)

```
static bit LCD_RS  @ ((unsigned)&PORTB*8+4);  // Register select
static bit LCD_EN  @ ((unsigned)&PORTB*8+5);  // Enable

#define LCD_STROBE  ((LCD_EN = 1),(LCD_EN=0))

/*****************************************************************************************
 * Write one byte to the LCD.
 */
void lcd_write_byte(unsigned char c)
{
  PORTB = (PORTB & 0xF0) | (c >> 4);
  LCD_STROBE;
  PORTB = (PORTB & 0xF0) | (c & 0x0F);
  LCD_STROBE;
  delay_us(40);
}

/*****************************************************************************************
 * Clear and home the LCD.
 */
void lcd_clear_and_home(void)
{
  LCD_RS = 0;  // control
  lcd_write_byte(0x01);
  delay_ms(2);
}

/*****************************************************************************************
 * Go to the specified position.
 */
void lcd_goto(unsigned char pos)
{
  LCD_RS = 0;  // control
  lcd_write_byte(0x80+pos);
}

/*****************************************************************************************
 * Write one char to the LCD.
 */
void lcd_write_char(char c)
{
  LCD_RS = 1;  // characters
  lcd_write_byte(c);
}

/*****************************************************************************************
 * Write a string of chars to the LCD.
 */
void lcd_write_string(const char * s)
{
  LCD_RS = 1;  // characters
  while(*s)
  { lcd_write_byte(*s++); }
}
```

```
/*********************************************************************************************
 * Clear and home the LCD,
 * then write a string of chars to the LCD.
 */
void lcd_overwrite_string(const char * s)
{
  lcd_clear_and_home();
  LCD_RS = 1;  // characters
  while(*s)
  { lcd_write_byte(*s++); }
}

/*********************************************************************************************
 * Write a string of chars to the LCD
 * starting at the beginning of the second line of the display.
 */
void lcd_write_string_at_second_line(const char * s)
{
  lcd_goto(64);
  LCD_RS = 1;  // characters
  while(*s)
  { lcd_write_byte(*s++); }
}

/*********************************************************************************************
 * Load the custom characters into cgram.
 * Also performs Clear and Home.
 */
void lcd_load_custom_characters(void)
{
  LCD_RS = 0;  // control
  lcd_write_byte(0b01000000);  // cgram address #0
  LCD_RS = 1;  // characters

  // pattern is xxxbbbbb

  // arrow right
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00000010);
  lcd_write_byte(0b00011111);
  lcd_write_byte(0b00000010);
  lcd_write_byte(0b00000100);

  // arrow left
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00001000);
  lcd_write_byte(0b00011111);
  lcd_write_byte(0b00001000);
  lcd_write_byte(0b00000100);

  // arrow up
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00001110);
  lcd_write_byte(0b00010101);
  lcd_write_byte(0b00000100);
```

```
  lcd_write_byte(0b00001100);
  lcd_write_byte(0b00011000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);

  // node
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00001110);
  lcd_write_byte(0b00010001);
  lcd_write_byte(0b00010001);
  lcd_write_byte(0b00010001);
  lcd_write_byte(0b00001110);

  // marked node
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00001110);
  lcd_write_byte(0b00010001);
  lcd_write_byte(0b00010101);
  lcd_write_byte(0b00010001);
  lcd_write_byte(0b00001110);

  // line
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00011111);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);

  // corner bottom left
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00000110);
  lcd_write_byte(0b00000011);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);

  // corner bottom right
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00000100);
  lcd_write_byte(0b00001100);
  lcd_write_byte(0b00011000);
  lcd_write_byte(0b00000000);
  lcd_write_byte(0b00000000);

  lcd_clear_and_home();
}

/*****************************************************************************************
 * Initialize the LCD.
 */
```

```
void lcd_initialize(void)
{
  LCD_RS = 0;  // control
  delay_us(100);
  lcd_write_byte(0x28);  // 4 bit mode, 1/16 duty (2 lines), 5x8 dots
  lcd_write_byte(0x0C);  // display on, blink off, cursor off
//lcd_write_byte(0x0F);  // display on, blink on, cursor on
  lcd_write_byte(0x06);  // entry mode

  lcd_load_custom_characters();
}

/*********************************************************************************************
 * Display a pattern of custom characters from cgram.
 * Also performs Clear and Home.
 */
void lcd_draw_logo(void)
{
   // 0 = arrow right     1 = arrow left      2 = arrow up              3 = node
   // 4 = marked node     5 = line            6 = corner bottom left    7 = corner bottom right

   lcd_clear_and_home();

   lcd_write_char(0);
   lcd_write_char(3);
   lcd_write_char(5);
   lcd_write_char(5);
   lcd_write_char(5);
   lcd_write_char(0);
   lcd_write_char(3);
   lcd_write_char(0);
   lcd_write_char(3);
   lcd_write_char(5);
   lcd_write_char(5);
   lcd_write_char(0);
   lcd_write_char(3);
   lcd_write_char(5);
   lcd_write_char(5);
   lcd_write_char(3);

   lcd_goto(64);

   lcd_write_char(' ');
   lcd_write_char(6);
   lcd_write_char(5);
   lcd_write_char(0);
   lcd_write_char(3);
   lcd_write_char(5);
   lcd_write_char(2);
   lcd_write_char(' ');
   lcd_write_char(6);
   lcd_write_char(0);
   lcd_write_char(4);
   lcd_write_char(1);
   lcd_write_char(7);
   lcd_write_char(4);
   lcd_write_char(1);
   lcd_write_char(7);
}
```

Listing B.4 C language code for the slave microcontroller (rs232.c)

```
static volatile bit RS232_TD @ (unsigned)&PORTA*8+2;
static volatile bit RS232_RD @ (unsigned)&PORTA*8+3;

/*****************************************************************************************
 * Initialize the transmit pin,
 * by ensuring it is low for an entire message cycle.
 */
void rs232_initialize()
{
  RS232_TD = 0;
  delay_ms(1);
}

/*****************************************************************************************
 * Send a single char out the transmit pin.
 */
void rs232_send_char(unsigned char c)
{
  c = ~c;

  RS232_TD = 1;             // start bit
  delay_us(100);

  RS232_TD = (c & 1);       // 8 data bits
  c = c >> 1;
  delay_us(98);

  RS232_TD = (c & 1);
  c = c >> 1;
  delay_us(98);

  RS232_TD = (c & 1);
  c = c >> 1;
  delay_us(98);

  RS232_TD = (c & 1);
  c = c >> 1;
  delay_us(98);

  RS232_TD = (c & 1);
  c = c >> 1;
  delay_us(98);

  RS232_TD = (c & 1);
  c = c >> 1;
  delay_us(98);

  RS232_TD = (c & 1);
  c = c >> 1;
  delay_us(98);

  RS232_TD = (c & 1);
  c = c >> 1;
  delay_us(98);
```

```
  RS232_TD = 0;             // stop bit
  delay_us(100);
}

/*********************************************************************************************
 * Send a string of chars out the transmit pin.
 */
void rs232_send_string(const char * s)
{
  while(*s)
  { rs232_send_char(*s++); }
}

/*********************************************************************************************
 * Get a single char from the receive pin.
 */
unsigned char rs232_get_char()
{
  unsigned char c = 0;

  while (RS232_RD == 0) { continue; }             // start bit
  delay_us(150);

  if (RS232_RD == 0) { c = c | 0b00000001; }      // 8 data bits
  delay_us(97);

  if (RS232_RD == 0) { c = c | 0b00000010; }
  delay_us(97);

  if (RS232_RD == 0) { c = c | 0b00000100; }
  delay_us(97);

  if (RS232_RD == 0) { c = c | 0b00001000; }
  delay_us(97);

  if (RS232_RD == 0) { c = c | 0b00010000; }
  delay_us(97);

  if (RS232_RD == 0) { c = c | 0b00100000; }
  delay_us(97);

  if (RS232_RD == 0) { c = c | 0b01000000; }
  delay_us(97);

  if (RS232_RD == 0) { c = c | 0b10000000; }
  delay_us(97);

  delay_us(100);                                  // stop bit

  return c;
}
```

Appendix C

Abstract Vending Machine

Listing C.1 An implementation in PIC16F84 assembly code of the portion of the control solution for an abstract vending machine that can be automatically generated from DES control theory.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  equates  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
TOKEN_EVENT  EQU d'0'        ;  offset of token  in the sdata array
POP_EVENT    EQU d'1'        ;  offset of pop    in the sdata array
REFILL_EVENT EQU d'2'        ;  offset of refill in the sdata array
ENABLED      EQU d'0'        ;  zero indicates enablement
DISABLED     EQU h'f'        ;  fifteen indicates disablement

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  data  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             CBLOCK h'0C'    ;  set up general registers starting at address 12
               event         ;  an event that is about to occur in the plant
               decision      ;  the enablement decision of the supervisor
               sstate        ;  the state of the supervisor
               temp          ;  a general temp register
             ENDC

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  initialization  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             org h'0000'
             ;...            ;  do general initialization
             clrf sstate     ;  initial supervisor state is zero

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  main  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
main         call doToken
             call doPop
             call doRefill
             goto main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  event routines  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
doToken      ;...            ;  test the sensor
             goto token_exit ;  if the event hasn't been initiated then quit

             movlw TOKEN_EVENT
             movwf event
             call supervisor ;  ask the supervisor
             btfsc decision, 0
             goto token_no

token_yes    ;...            ;  do the work for the yes case
             goto token_exit
token_no     ;...            ;  do the work for the no case
token_exit   return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
doPop        ;...           ;  test the sensor
             goto pop_exit   ;  if the event hasn't been initiated then quit

             movlw POP_EVENT
             movwf event
             call supervisor ;  ask the supervisor
             btfsc decision, 0
             goto pop_no

pop_yes      ;...           ;  do the work for the yes case
             goto pop_exit
pop_no       ;...           ;  do the work for the no case
pop_exit     return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

doRefill     ;...           ;  test the sensor
             goto refill_exit ; if the event hasn't been initiated then quit

             movlw REFILL_EVENT
             movwf event
             call supervisor ;  notify the supervisor

refill_yes   ;...           ;  do the work
refill_exit  return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  supervisor subroutine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  input:  assumes event = event currently being initiated
;;          i.e. TOKEN_EVENT, POP_EVENT or REFILL_EVENT
;;  output: sets decision = ENABLED or DISABLED
;;          also updates its state accordingly (in sstate)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
supervisor   movf sstate, W  ;  used determine which data to fetch from the sdata array
             movwf temp

             movlw sdata     ;  the start of the sdata array

             movf sstate, F  ;  in state zero we want to skip the loop
             btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
             goto $+4

             addlw d'3'      ;  adds to W the # to skip ahead to get the data for the current state
             decfsz temp, F
             goto $-2

             addwf event, W  ;  adds to W the # to skip ahead to get the data for the current event

             movwf FSR       ;  the address of the next_state data (INDF will now access the correct data)
             movlw DISABLED
             subwf INDF, W   ;  test if the transition is disabled
             btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
             goto disable

enable       movf INDF, W    ;  update the next state
             movwf sstate
             movlw ENABLED   ;  mark as enabled
             movwf decision
             goto ssExit
```

```
disable      movlw DISABLED   ;  mark as disabled
             movwf decision

ssExit       return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  supervisor data  (h'f' indicates disablement)  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

sdata        dw h'1', h'f', h'0'  ;  outgoing (token, pop, refill) from state 0
             dw h'2', h'f', h'1'  ;  outgoing (token, pop, refill) from state 1
             dw h'f', h'3', h'2'  ;  outgoing (token, pop, refill) from state 2
             dw h'4', h'f', h'0'  ;  outgoing (token, pop, refill) from state 3
             dw h'5', h'f', h'1'  ;  outgoing (token, pop, refill) from state 4
             dw h'f', h'6', h'2'  ;  outgoing (token, pop, refill) from state 5
             dw h'7', h'f', h'0'  ;  outgoing (token, pop, refill) from state 6
             dw h'8', h'f', h'1'  ;  outgoing (token, pop, refill) from state 7
             dw h'f', h'9', h'2'  ;  outgoing (token, pop, refill) from state 8
             dw h'f', h'f', h'0'  ;  outgoing (token, pop, refill) from state 9

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  end  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             end              ; end of program
```

Listing C.2 An alternative implementation using distributed control theory.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  equates  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
TOKEN_EVENT  EQU d'0'        ;  offset of token  in the sdata array
POP_EVENT    EQU d'1'        ;  offset of pop    in the sdata array
REFILL_EVENT EQU d'2'        ;  offset of refill in the sdata array
ENABLED      EQU d'0'        ;  zero indicates enablement
DISABLED     EQU h'f'        ;  fifteen indicates disablement

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  data  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             CBLOCK h'0C'    ;  set up general registers starting at address 12
               event         ;  an event that is about to occur in the plant
               decision      ;  the enablement decision of the last asked supervisor
               s1state       ;  the state of the supervisor1
               s2state       ;  the state of the supervisor2
               n1state       ;  the next state for supervisor1 if no one else disables the event
               n2state       ;  the next state for supervisor2 if no one else disables the event
               temp          ;  a general temp register
             ENDC

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  initialization  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             org h'0000'
             ;...            ;  do general initialization
             clrf s1state    ;  initial supervisor state is zero
             clrf s2state    ;  initial supervisor state is zero

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  main  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
main         call doToken
             call doPop
             call doRefill
             goto main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  event routines  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
doToken      ;...            ;  test the sensor
             goto token_exit ;  if the event hasn't been initiated then quit

             movlw TOKEN_EVENT
             movwf event

             call supervisor1 ; if any supervisor disables then take the no branch
             btfsc decision, 0
             goto token_no

             call supervisor2 ; if any supervisor disables then take the no branch
             btfsc decision, 0
             goto token_no
```

```
token_yes    call super_commit
             ;...           ;  do the work for the yes case
             goto token_exit
token_no     ;...           ;  do the work for the no case
token_exit   return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

doPop        ;...           ;  test the sensor
             goto pop_exit   ;  if the event hasn't been initiated then quit

             movlw POP_EVENT
             movwf event

             call supervisor1 ; if any supervisor disables then take the no branch
             btfsc decision, 0
             goto pop_no

             call supervisor2 ; if any supervisor disables then take the no branch
             btfsc decision, 0
             goto pop_no

pop_yes      call super_commit
             ;...           ;  do the work for the yes case
             goto pop_exit
pop_no       ;...           ;  do the work for the no case
pop_exit     return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

doRefill     ;...           ;  test the sensor
             goto refill_exit ; if the event hasn't been initiated then quit

             movlw REFILL_EVENT
             movwf event

             call supervisor1 ; all supervisors must be notified
             call supervisor2
             call super_commit

refill_yes   ;...           ;  do the work
refill_exit  return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  supervisor1 subroutine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  input:  assumes event = event currently being initiated
;;          i.e. TOKEN_EVENT, POP_EVENT or REFILL_EVENT
;;  output: sets decision = ENABLED or DISABLED
;;          also updates its state accordingly (in s1state)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
supervisor1  movf s1state, W ;  used determine which data to fetch from the s1data array
             movwf temp

             movlw s1data    ;  the start of the s1data array

             movf s1state, F ;  in state zero we want to skip the loop
             btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
             goto $+4

             addlw d'3'      ;  adds to W the # to skip ahead to get the data for the current state
```

```
            decfsz temp, F
            goto $-2

            addwf event, W  ;  adds to W the # to skip ahead to get the data for the current event

            movwf FSR       ;  the address of the next_state data (INDF will now access the correct data)
            movlw DISABLED
            subwf INDF, W   ;  test if the transition is disabled
            btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
            goto s1Disable

s1Enable    movf INDF, W    ;  update the next state
            movwf n1state
            movlw ENABLED   ;  mark as enabled
            movwf decision
            goto s1Exit

s1Disable   movlw DISABLED   ;  mark as disabled
            movwf decision

s1Exit      return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  supervisor2 subroutine  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  input:  assumes event = event currently being initiated
;;          i.e. TOKEN_EVENT, POP_EVENT or REFILL_EVENT
;;  output: sets decision = ENABLED or DISABLED
;;          also updates its state accordingly (in s2state)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
supervisor2  movf s2state, W ;  used determine which data to fetch from the s2data array
            movwf temp

            movlw s2data    ;  the start of the s2data array

            movf s2state, F ;  in state zero we want to skip the loop
            btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
            goto $+4

            addlw d'3'      ;  adds to W the # to skip ahead to get the data for the current state
            decfsz temp, F
            goto $-2

            addwf event, W  ;  adds to W the # to skip ahead to get the data for the current event

            movwf FSR       ;  the address of the next_state data (INDF will now access the correct data)
            movlw DISABLED
            subwf INDF, W   ;  test if the transition is disabled
            btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
            goto s2Disable

s2Enable    movf INDF, W    ;  update the next state
            movwf n2state
            movlw ENABLED   ;  mark as enabled
            movwf decision
            goto s2Exit

s2Disable   movlw DISABLED   ;  mark as disabled
            movwf decision

s2Exit      return
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  commit all supervisors to the computed change of state  ;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
super_commit movf n1state, W
             movwf s1state
             movf n2state, W
             movwf s2state
             return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  supervisor data  (h'f' indicates disablement)  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

s1data       dw h'1', h'f', h'0'  ;  outgoing (token, pop, refill) from state 0
             dw h'2', h'f', h'1'  ;  outgoing (token, pop, refill) from state 1
             dw h'f', h'0', h'2'  ;  outgoing (token, pop, refill) from state 2

s2data       dw h'0', h'1', h'0'  ;  outgoing (token, pop, refill) from state 0
             dw h'1', h'2', h'0'  ;  outgoing (token, pop, refill) from state 1
             dw h'2', h'3', h'0'  ;  outgoing (token, pop, refill) from state 2
             dw h'f', h'f', h'0'  ;  outgoing (token, pop, refill) from state 3

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  end  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             end             ; end of program
```

Listing C.3 An ad hoc solution functionally equivalent to the solutions automatically generated by DES control theory.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  equates  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
MAX_POP  EQU d'3'           ;  The maximum number of pop the machine can contain
POP_COST EQU d'2'           ;  The cost of a pop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  data  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             CBLOCK h'0C'    ;  set up general registers starting at address 12
               tokens        ;  the current number of un-spent tokens
               pops          ;  the current number of available pop
               temp          ;  a general temp register
             ENDC

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  initialization  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             org h'0000'
             ;...            ;  do general initialization
             clrf tokens     ;  start with no un-spent tokens
             movlw MAX_POP   ;  start with full pop
             movwf pops

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  main  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
main         call doToken
             call doPop
             call doRefill
             goto main

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  event routines  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
doToken      ;...            ;  test the sensor
             goto token_exit ;  if the event hasn't been initiated then quit

rule_2_1     movf pops, F    ;  don't accept tokens when pops is zero
             btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
             goto token_no

rule_1_2     movlw POP_COST  ;  don't accept more than POP_COST tokens
             subwf tokens, W
             btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
             goto token_no

token_yes    incf tokens     ;  update the system variables
             ;...            ;  do the work for the yes case
             goto token_exit
token_no     ;...            ;  do the work for the no case
token_exit   return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
doPop        ;...            ;  test the sensor
             goto pop_exit   ;  if the event hasn't been initiated then quit

rule_2_2     movf pops, F    ;  don't deliver pop when pops is zero
             btfsc STATUS, Z ;  zero bit is clear only when result is NOT zero
             goto pop_no

rule_1_1     movlw POP_COST  ;  don't deliver pop unless tokens = POP_COST
             subwf tokens, W
             btfss STATUS, Z ;  zero bit is clear only when result is NOT zero
             goto pop_no

pop_yes      clrf tokens     ;  update the system variables
             decf pops
             ;...            ;  do the work for the yes case
             goto pop_exit
pop_no       ;...            ;  do the work for the no case
pop_exit     return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

doRefill     ;...            ;  test the sensor
             goto refill_exit ; if the event hasn't been initiated then quit

             movlw MAX_POP   ;  update the system variables
             movwf pops
refill_exit  return

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  end  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
             end             ; end of program
```