Model Based Synchronization of Monitoring and Control Systems

by

Orest Lev Storoshchuk

B.E.E. (Magna Cum Laude), M.E.E, P.Eng.

A Theisis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

McMaster University

Model Based Synchronization of Monitoring and Control

DOCTOR OF PHILOSOPHY (2003)
(Electrical and Computer Engineering)

McMaster University
Hamilton Ontario

TITLE: Model Based Synchronization of Monitoring and Control Systems

AUTHOR: Orest Lev Storoshchuk, B.E.E. (Magna Cum Laude) (General Motors Institute), M.E.E. (McMaster University), P.Eng.

SUPERVISOR: Professor W.F.S. Poehlman,

NUMBER OF PAGES: xiii, 383

Thesis Abstract

This research identifies and provides novel solutions to challenges that arise when implementing an Agile Manufacturing Information Systems (AMIS). This research constitutes a body of original work in that it has successfully integrated aspects of real-time systems control, computer science, electrical engineering, knowledge capture and technology insertion to address the issues faced when developing an AMIS. They include:

1. The divergence between manufacturing control and monitoring systems, addressed by complimenting the current University of Michigan research on domain models and developing a novel mapping from the model to PLC Ladder Logic.

2. Development of the code, heretofore lacking sufficient application of scientific methodology, by providing a method for integrating components into a control plan, capturing and representing a control engineer's knowledge in a generalized abstract model and removing the need for the specialized knowledge.

3. The lack of systems context provided by current monitoring systems enhanced by developing an application generator, based on the generalized abstract model, which automatically creates synchronized control code and information needed by the monitoring system.

4. An expressed lack of experimentation in the software community to transition technology from the theoretical domain to practice in industry is addressed through an experimental effort with industry, where the theoretical methodology was successfully transferred to practice in the manufacturing community. An experimental environment was developed to allow other researchers to replicate the experiments and to continue to extend the research.

This thesis is dedicated

to my parents Myroslava and Roman

who encouraged me in this endeavour,

to my wife Aileen,

my support and steadfast companion,

to my daughter Natalka,

the source of joy in my life.

Acknowledgments

Many people have my deepest gratitude for their assistance and support during the years I endeavoured to complete this thesis. The order they are listed in is of no significance.

Aileen Storoshchuk for her long-term support, encouragement and tolerance of the time this thesis consumed

Myroslava Storoshchuk for her encouragement and financial support

Dr. Poehlman for formulation of the original GM proposal, mentoring, supervising and overall support.

Brian Hastings and the staff at RLM for their sponsorship, advice, knowledge, providing of equipment and personnel utilized in experiments and participation in the transfer of technology.

Tim King for insight into building industrial model and generosity of his time and material.

Shige Wang for review of thesis and discussion of research ideas.

Dr. Shin for encouragement, guidance, advice on the academic process and access to research resources.

Sushil Birla for assistance in formulation of research topic and advice on the academic process.

Jim Nilsson for the time spent as an experimental subject and insight into designing software targeted for use by trades-people.

Dr. Chapman for formulation of the original GM proposal, serving on my committee, providing advice and guidance.

Craig Brown for flexibility and support that accommodated my thesis work with my work schedule.

Dan Nicholson for support for my thesis at my place of employment.

Michael Masleid for technical expertise on a wide range of issues, participation and support in an earlier IEEE 802.11 thesis topic

Larry Van Der Jagt for participation and support in an earlier IEEE 802.11 thesis topic.

Dr. Bone, Dr. Szymanski and Dr. Davidson, members of my thesis committee for their helpful participation.

Table of Contents

List of Figures

List of Figures (continued)

List of Figures (continued)

List of Figures (continued)

List of Tables

Chapter 1

Introduction

In today's global economy of frequent and unpredictable change, manufacturers must devise methods that keep them competitive. The Agile Manufacturing paradigm [War94] has been devised to address the challenges of such an environment. A suitable information system [ACM96] becomes a key component needed for implementing Agile Manufacturing. Such an information system is faced with challenges that arise from Agility. This body of research addresses four of these major challenges: 1) the divergence between manufacturing control and monitoring systems; 2) the lack of systems context provided by current monitoring systems; 3) development of the code lacking application of scientific methodology and; 4) insufficient experimentation in the software community to transition technology from the theoretical domain to practice in industry.

1.1 Background

Various approaches have been formulated to adapt manufacturing to the pressures of the global economy. The goal of Flexible Manufacturing is the ability to respond in a cost effective and timely manner to planned and unanticipated change in external and internal environments [Rol92]. Time based strategies such as Quick Response Manufacturing (QRM) [Sur95], Time Based Competition (TBC) [Buz95]; Lean [Hol97]

and Synchronous Manufacturing [Ste94] have sought to reduce time-to-market, to streamline product development processes and minimize cost. The more recent Agile Manufacturing paradigm [War94], not only encompasses the previous objectives, but also considers many other aspects of change. Schmoyer [Sch94] states that agile manufacturing provides the ability to thrive and prosper in a competitive environment of continuous and unanticipated change and to respond rapidly to changing markets. What sets agility apart from previous approaches is the recognition of the need to empower employees at all levels of the organization, and the need to support employees in discharging the additional responsibilities that empowerment brings [Lay94, OCo94, Tra94a, Tra94b, Ise95, She93].

To achieve agility requires an increased dependency on human judgment [Lay94, Lin94]. Workers need intellectual tools to make informed decisions. Runkle [Run93], highlights the need for timely delivery of accurate information within the agile organization, when he states:

> Just as mass-production leveraged people's physical power and dexterity, agile manufacturing will leverage their capabilities to handle information and make decisions. Agile manufacturing will increase the competitive impact of a person's intellectual power.

Information must be provided in a timely manner, in a way that acknowledges the mental model of the user and minimizes the impact of new technology insertion. A performance support system is needed to serve employees at all levels within the organization.

The vehicle for providing intellectual support for the workforce is the information system. The information system that supports agility has special demands placed upon it

[ACM96]. In particular, methodologies are required for incorporating the frequent changes that characterize an agile environment into the stream of data captured and distributed by the information system. The specific aspect of the information system addressed by this thesis is the synchronization of manufacturing plant floor monitoring and control software code. It addresses the volume and pace of change that are characteristic of agile manufacturing.

1.2 Motivation

To gain a better understanding of the issues in designing, deploying and integrating an Agile Manufacturing Information System (AMIS) into a Manufacturing Organization, a research proposal to pilot such a system was presented to General Motors [Poe95] by McMaster University, the University of Guelph and the Oshawa Car Assembly Plant. General Motors assigned it as a Portfolio Project [Vaz96]. As Coronado *et al.* [Cor02], report, "despite the growing importance of information systems in manufacturing operations, few examples in the literature have investigated the requirements of information systems to support agility". The project was to be a collaborative effort coordinated by the National Centre for Manufacturing Sciences (NCMS) and funded jointly by GM and other collaborators involved in the NCMS research [NCM96]. However, a change in General Motors management precluded conducting the research. This thesis implements a subset of the original Portfolio Project.

The proposed AMIS implementation [Poe95] was guided by the following principles:

> (i) A growing number of companies are empowering lower-level employees to make the kind of decisions that must be made quickly in an agile environment. Agility requires empowerment, which in turn requires the timely delivery of information and knowledge. It is incumbent on the organization to provide performance support, to aid employees in meeting the increased responsibility that accompanies their empowerment.
>
> (ii) Within an organization, collecting data represents the first step in the evolution of understanding. The evolutionary path leads from data to information, from information to knowledge, and finally from knowledge to intelligence. Distinct organizational activities require different evolutionary levels of understanding. It is therefore necessary to determine the current mental model of the user, and to deliver information in the context of the users mental model and problem solving world. Information, enriched by this environmental context, becomes knowledge.
>
> (iii) The initial impact of technology insertion should be minimized. On an on-going basis, all personnel should be provided with a smooth migration path to a higher level of operational abstraction.
>
> (iv) Monitored data is of diminished worth unless it is informed by an awareness of the system context. We refer to data that are enriched by the attachment of modifiers that describe the state of the system at time of data capture, as context-enhanced data (CED). An AMIS requires context-enhanced data. One of the central thrusts of this proposal is to develop a methodology to ensure a supply of context-enhanced data to AMIS, without which it cannot function.

Within an organization, the evolutionary path leads from data to information, from information to knowledge, and finally from knowledge to intelligence. Distinct organizational activities require different evolutionary levels of understanding. Only recently, has there been an appreciation of the success of systems that employ learner-centred design [ACM96], which is appropriate to distribute digital material enabling

multi-level understanding. It is therefore necessary to determine the current mental model of the user [Ras86], and to deliver only the appropriate (i.e. filtered) context-enhanced data (CED, now termed information) in terms of the user's mental model and problem solving world. Information, enriched by this environmental context, becomes knowledge to be processed at other enterprise levels.

Figure 1-1 provides a conceptual schematic of a generic agile-implemented system where data from the shop floor monitoring programs are merged with configuration (state of the system) data (derived from the control programs) to create CED. This serves as input to data filters and automated analyzers, which convert it into information. At the next level, this information is merged with the analytic context (provided by Analytic Context Generators) and generates the knowledge. Analytic context in this instance refers to the original information categorized to a higher level of classification. This knowledge can be injected into any enterprise-wide information system through the organization's intranet. At the same time, such information can be used in-plant to aid human-based investigations (e.g. industrial engineer activities) for process optimization and analysis. Because system metrics can be automatically extracted from the appropriate knowledge stream and combined with this information, the delivery system can greatly empower or assist in an individual's performance. A system with these capabilities is commonly termed, a Performance Support System (PSS). This PSS is extremely agile itself, adapting contexts quickly, depending on the user's problem-solving goals for that session. The analysis to obtain the goals utilizes context-enhanced information and hence becomes a knowledge-based system.

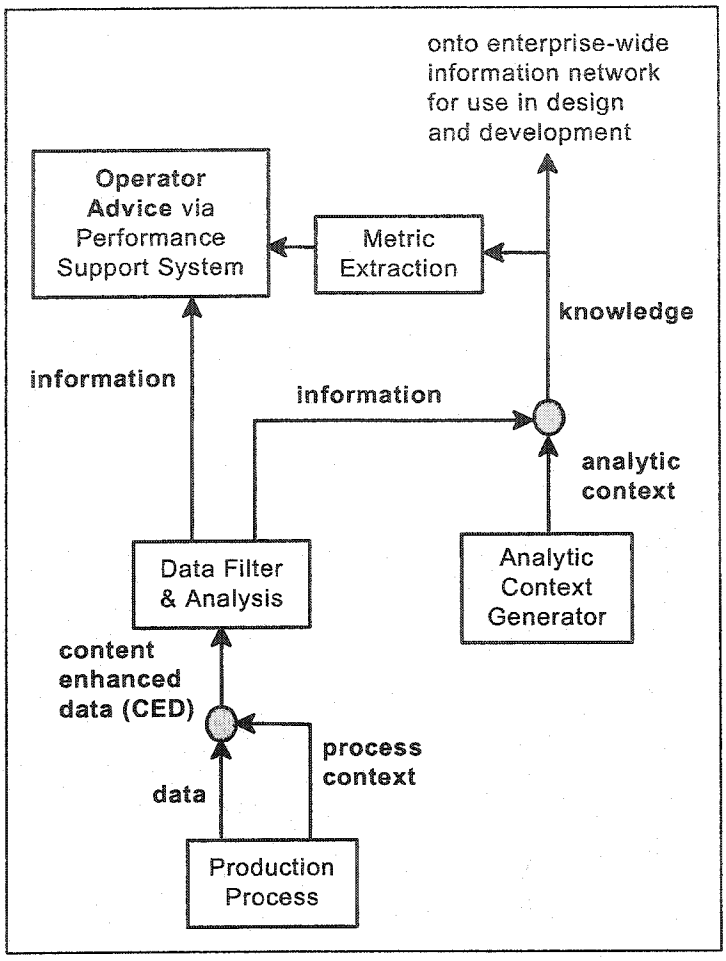


Figure 1-1: Data/Information/Knowledge Flows and Conversions
Taken from [Sto03a]

There is a large body of literature pertaining to principles and methods of software system development, which theoretically could be applied to implement such a system. However, there has been little penetration of these methods into industry practice [Bir98, Bir03, Lip03, End01]. Within software literature [Kit97, Zel97, Tic98, Lin97] there has been expressed, the need for a higher degree of experimentation than is currently the norm. One aspect of this research focuses on transitioning theoretical methods from the academic to the industrial communities through experimental validation in the industrial

setting and the development of a replicable experimental platform for use by other researchers.

RLM, a manufacturer of roll forming systems[1], provided a manufacturing environment (with equipment and personnel) for the portion of the Portfolio Project covered by this body of research. The Engineering organization of RLM has been involved in the design of large scale manufacturing automation control systems[2] as well as monitoring system for years. They used similar practices in their organization at the start of the experiments conducted by this research as those that are currently used in the General Motors of Canada Oshawa Car Assembly Plant. The RLM engineering manager Hastings [Has99], indicates that:

> Traditionally controls-engineering is an art more than it needs to be. Typically, a repair technician, who understands the operation and characteristics of a machine, describes how it needs to work. A control engineer tries to get a comprehensive understanding and then goes off to craft the program. With today's focus on agility and continuous improvement, one needs to be able to change controls, by a less skilled person than the control engineer who initially 'crafted' it. Initial craft time is long and non-standard. There is a great need to capture the knowledge of a control engineer so that control code can be developed and later changed, in a disciplined way, by a less skilled person. There is also the need to train support people on floor. A key issue is that the person on the floor needs to have the entire knowledge of the system available. During normal operation systems start to degrade, but cannot be taken out of operation for repair for a long period of time. This gives rise for the need and the ability to modify behaviour to a degraded level to allow continued function. Without the capture of the control engineer's knowledge, such

[1] Design, fabricate and install automatic systems for "Tier 1" automotive suppliers (suppliers who manufacture components used directly by automotive assembly plants).

[2] Several members worked at a the Oshawa Car plant in the capacity of Senior Engineer, Production Superintendent and Manager of Technical training, responsible for the design and installation and operation of many of the current automation systems, which include robotics, automatic guided vehicles automatic monorails and the information system supporting operations.

> modifications become extremely difficult and tend to degrade the quality of the initial control system. Today's technology offers the potential to get all the information needed for the floor people at a lower cost than today. This drives a need for a reduction of different styles of control code. If we can have a single style, we now can educate only once, decrease code generation time, and also have dramatic impacts on start-up and debug time. Information systems will be provided with real data initially during start-up when it is of particular value, and continually as improvements and changes are made, at an affordable price initially, and in the long term. Typically customers find that the information gathering code works only while the original vendor is around.

Lipa [Lip03] indicates that this is still the case for the automotive "Tier 1" suppliers who manufacture components used directly by the automotive manufacturers. The automotive manufacturers have larger and better-trained controls engineering staffs than their direct "Tier 1" suppliers who may have only a single controls technician or engineer and single IT person for several plants. ARC Strategies [ARC02] reports that General Motors North American Operations formed the Control, Robotics and Welding Group (CRW) in 1997, which has been leading a migration to a common controls architecture. ARC states that this is indicative of the need for larger manufacturers to establish their own controls standards rather than rely on the suppliers of their manufacturing systems. A goal for 2002-2003 is common automation software control modules. CRW has developed a proprietary code generator [Bir03, Abr03] that does not employ formal methods as is done in this work. ARC reports CRW recognizes the need to provide an improved interface between plant floor automation systems and other higher-level production information systems. "Creating a link between business systems and PLCs has become a major objective for CRW".

1.3 Organization of the Dissertation

The rest of this dissertation is organized as follows: Chapter 2 presents the current state of the art pertaining to the thesis issues. The cause of divergence between monitoring and control systems and their ad hoc development practice is explored as the cause of poor quality data without system context awareness. Technology transfer principles for methods of software system development that address these issues are proposed. The capture of a control engineer's knowledge in a reusable form is identified as a technology transfer enabler. Current research on discrete event supervisory control is presented and a gradual transition to industry outlined. Chapter 3 presents concisely the multidisciplinary subjects of this research in a manner readily understood by readers who do not have expert knowledge in all of the disciplines involved. A brief overview is provided, the novel contributions to the fields involved are stated and the methodology developed to address issues arising from development of Agile Manufacturing Information Systems is outlined. Chapter 4 describes the tools available to address the issues faced by an AMIS. An object-oriented composition model augmented with the method of finite-state-machines is proposed. PLC Ladder Logic code is utilized for a gradual technology insertion. The capture of a control engineer's expert knowledge in the form of an abstract model and an application generator are presented as the general strategy. Chapter 5 presents the development of an abstract model for the control code. Experimentation with an industry expert control engineer for technology insertion is described. Generalizing from a specific case derives the model, represented with the Object Oriented Unified Modelling Language. A methodology for generating control

code for typical machines based on this model is presented. Chapter 6 outlines the design of an application generator. The artificial intelligence concept of knowledge capture and the Object Oriented techniques of hierarchy, compositional modelling, abstraction and information hiding are applied. An iterative rapid prototyping software development approach is employed to develop, obtain feedback and improve the usability. Chapter 7 describes the design and implementation of a replicable experimental environment. Alternative solutions for implementing a machine simulator are evaluated and a small-scale, functional physical model is put forward as the most suitable approach. Chapter 8 states and interprets the results. Contributions to the state-of-the-art are listed and further areas of study are proposed in Chapter 9.

Chapter 2

Background

The problem of divergence between manufacturing and control systems has been studied by the NCMS as part of an effort to develop agile information systems. It has also been identified as an area of research in Performance Support systems conducted at the GM facility in Oshawa by the "Applied Computersystems Group" (ACsG) of McMaster University and the Department of Mathematics and Statistics at the University of Guelph. The research provided the opportunity to observe the evolution of interaction between the data acquisition and control system development. Lack of system context was identified as an issue with current monitoring systems by the Acsg [Poe95]. The development of methodologies for control software development, suitable for manufacturing systems, has been a focus at the Department of Electrical and Computer Engineering at the University of Toronto (U of T) [CIM03], the Engineering Research Centre (ERC) for Reconfigurable Machining Systems (RMS) [ERC03] at the University of Michigan (UM), and within the ESPRIT III Open Systems Architecture for Controls within Automation systems (OSACA) [Spe97, OSC96] at various European Universities. In addition, Model Based Integration of Embedded Software (MOBIES) [AFR02] research, sponsored by DARPA at several Universities in the United States has explored the use of models.

The ERC includes over 30 industrial companies in a collaborative, technology transfer role. They identify a new manufacturing paradigm named Reconfigurable Manufacturing Systems to address the same issues as Agile Manufacturing:

> Manufacturing companies in the 21st Century will face frequent and unpredictable market changes. These changes include high-frequency introduction of new products, new product demand and mix, new parts for existing products, new government regulations, and new process technology. To stay competitive, manufacturing companies must possess manufacturing systems that are fully and rapidly responsive to all these variables. A responsive system incorporates a production capacity that is adjustable to fluctuations in product demand, is adaptable to new product functions, and is designed to be upgradeable with new process technology to accommodate evolving product specifications and government regulations. Current systems, even so-called flexible manufacturing systems, do not have these characteristics ... The aim of RMS is to design systems, machines, and controls for cost-effective, rapid response to changes in market demand and products.

The ERC identify that the barrier to developing reconfigurable systems is the lack of a scientific approach in designing optimal, scalable configurations of systems that include reconfigurable hardware and reconfigurable software. The ERC is organized into three Clusters of Projects: System-Level Design; Machine/Control Design; and Calibration and Ramp-Up. Within the second cluster there has been development of three methodologies for constructing control software primarily aimed at supporting Reconfigurable Machine Tools [Til99].

2.1 Divergence and Context Issues

A major issue for Agile Manufacturing identified by the National Centre for Manufacturing Sciences (NCMS) [Lip95] is divergence between Manufacturing Executions Systems (MES) [Par98] (software used in manufacturing plants to execute daily operations) collecting real-time data, and the control systems. Lipa [Lip95, Lip03], states that most systems follow a life cycle where:

1) Someone identifies a need and justifies the need for a system.
2) The system is implemented and commissioned.
3) Information technology (IT) evolves independently of the project's control technology.
4) Controls technology evolves independent of the project's IT.
5) The plant spends resources to maintain the systems
6) IT and Controls continue to evolve independently
7) More resources are expended to maintain the systems.
8) Eventually the plant succumbs to increasing system maintenance costs.
9) System becomes dysfunctional.
10) Management turns off the MES portion.
11) Plant identifies a need for the system.
12) The cycle begins again.
13) The software maintenance problem starts again.

Figure 2-1 provides a summary of the diverging cycles.

In automotive manufacturing, the real-time [Shi94] systems, which control process equipment, and those which obtain various real-time information pertaining to the process and equipment, are treated separately. The control code is developed first, with the data acquisition/monitoring code, or information systems (IS), subsequently added as a software maintenance activity. A different group usually writes the monitoring code. The Software Engineering Institute (SEI) has developed the Capability Maturity Model for Software (CMM) [Pau93] for evaluating an organization's

Problem: Chanages and Synchronization become progressively more difficult, costly and time consuming.

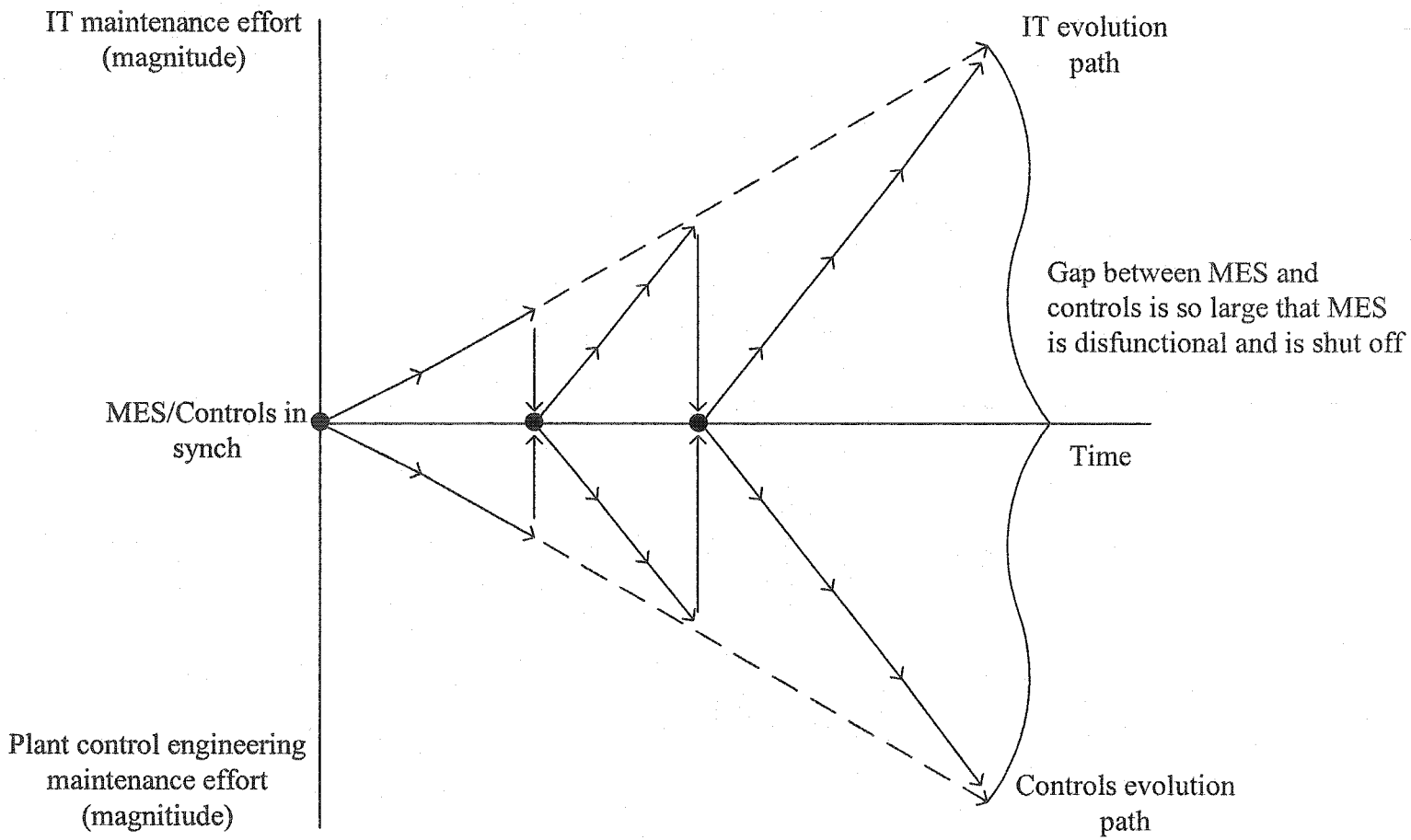


Figure 2-1: Software Maintenance Problem
Adapted From [Lip95]

competence in writing software. According to this model, the higher the number that the organization is assessed, the greater the organization's capability to generate code. Current control code development practice in the automotive industry [Bir03] is at the repeatable level (second lowest competence) is called level 2. At this level basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications. The methods used are primarily those supported by the controller supplier's proprietary tools or by an automation layer built upon the services provided by these tools. The control supplier's tools are PLC family specific and vendor specific. Some abstraction and hierarchy is used but at a low level such as code patterns modularized in subroutines or function blocks provided by the IEC-1131 standard [IEC98]. Higher level, more formal methods based on the Object Oriented approach, Nested Finite State Machines or Perti Nets are not commonly utilized [Lip03, Bir03, Mar01]. Thus, a substantial effort is required to comprehend and generate the initial IS code for new applications, and regular maintenance is needed to keep it synchronized with the continual evolution of the control code. The result is data of poor quality, often without an awareness of the system context [Sto03a].

2.2 Technology Transition Issue

There is a large body of literature pertaining to principles and methods of software system development, which theoretically would address the issues of developing a suitable information system for implementing Agile Manufacturing. However, the application of principles and methodologies to define software systems continues to be

undefined [Bir98]. There is little statistical evidence [Kit97, Zel97, Tic98, Lin97] that applying these principles and methodologies to control software production of the scale found in automotive manufacturing will yield significant improvement. Inadequate scientific guidance in applying these principals can result in an increase in effort, duration and skill requirement. Within software literature [Kit97, Zel97, Tic98, Lin97] there has been expressed, the need for a higher degree of experimentation than is currently the norm. Linkman and Rombach [Lin97], outlined an experimentation and research cycle needed to support the evolution of software engineering as a discipline. The experimental process is needed to successfully transfer software technologies from the theoretical domain into industrial application. The need to transfer knowledge from the academic domain to industrial practice is summarized by Yunus [Yun99], who states,

> If a university is a repository for knowledge, then some of this knowledge should spill over to the neighbouring community. A university must not be an island where academics reach out to higher and higher levels of knowledge without sharing any of their findings.

Within the products offered by manufacturers of discrete event controls, there are a number of products that allow the designer to work at a higher level of abstraction than is the current norm. These products include state machines and Petri nets [Mar01]. According to Object Oriented theory [Boo94], working at a higher level of abstraction would potentially make the initial task of understanding the control algorithms by the monitoring group simpler. However, vendors are finding it difficult to achieve industrial acceptance [Mar01, Lip03]. A more gradual transition is required as outlined in the Portfolio Project proposal [Poe95] is shown in Figure 2-2. The control system code (cntrl

codes) and the monitoring system code (mntrng codes) are shown on the left. The traditional (existing) code and view of the control engineer (C.E.) is Ladder Logic [IEC98] shown at the top. As the C.E. moves to an Agile approach using a more abstract language such as Grafcet [Dav95] in Europe or sequential function charts (SFC) [ABP92] in North America, both the new and equivalent Ladder Logic views are provided. Eventually, the C.E. transitions completely to the Agile approach removing the need to provide the traditional Ladder Logic view. The current methodology is based on Ladder Logic that graphically represents the relay logic used in past decades. The change from physically connecting relays with wires to using specialized computers (known as programmable logic controllers or PLC 's), using Ladder Logic was successful. The success was attributed to being programmed in the same paradigm. Based on the success of transitioning to PLC 's, and the Portfolio Project's proposal for technology insertion advocating this approach, PLC Ladder Logic was chosen as the preferred implementation.

2.3 Control Software Development Methodologies

Supervisory control of discrete systems has been the focus of extensive research. The primary goals have been the avoidance of deadlock, assurance of controllability, and reconfiguration and reuse. However, the issue of support for information systems was not addressed. Transition to industry was given minimal consideration. This body of research utilizes aspects of the previous methods in an innovative approach to address these issues.

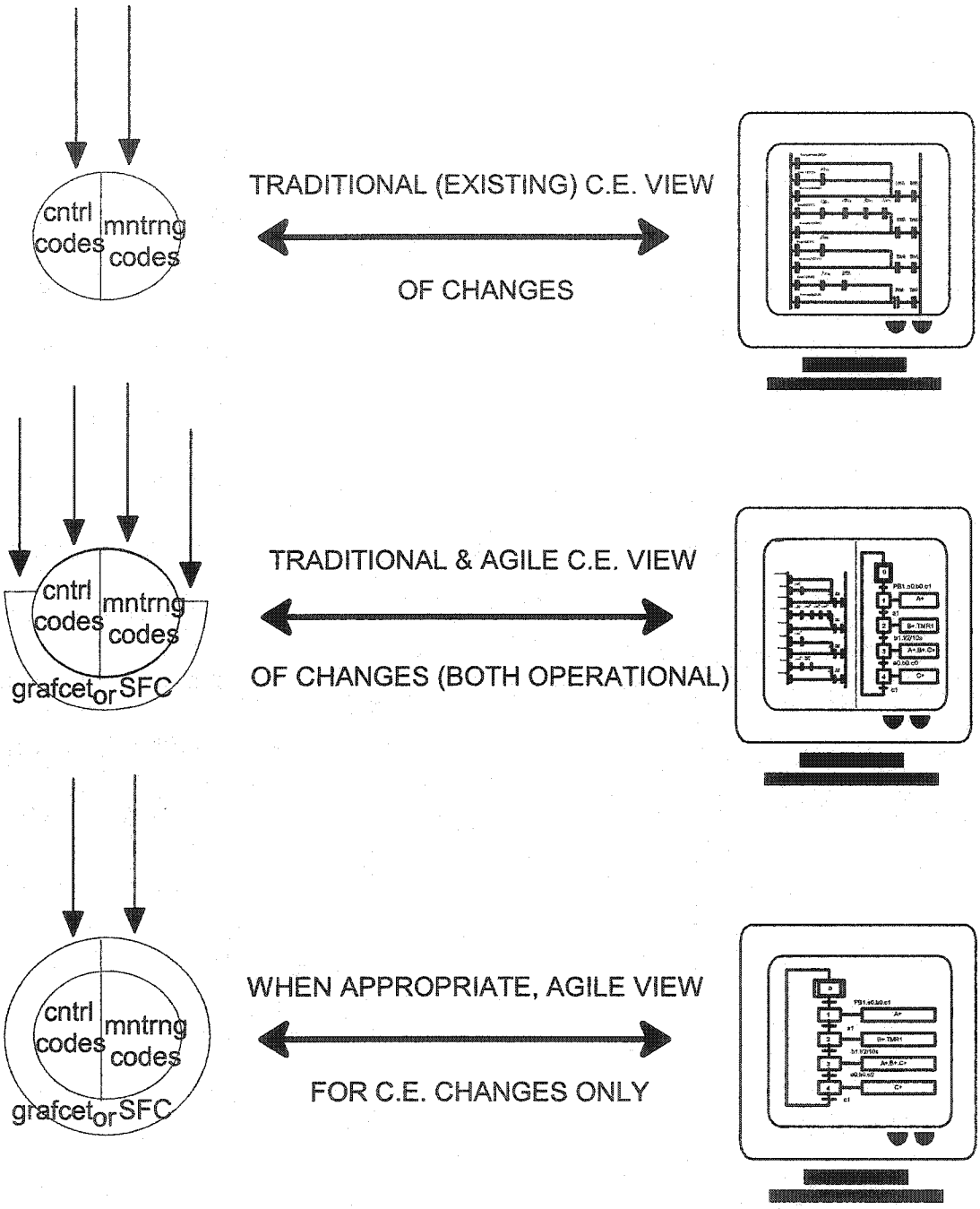


Figure 2-2: Technology Insertion Implementation
Taken From [Poe95]

Ramadge and Wonham [Ram82, Ram86, Ram89] introduced the theory of supervisory control of discrete event systems in the 1980's. There are several different approaches for control software to choose from as categorized by Brandin [Bra96]: knowledge engineering [Mur98, Alb91]; Petri Nets [Hol90, Fre91]; and controlled automata [Ram87, Bra94]. Brandin [Bra92, Bra96], selected controlled automata claiming the following advantages:

- The supervisors and control laws are correct by construction.
- The supervisors and control laws obtained are maximally permissive within the behavioural specifications considered.

Brandon [Bra93, Bra91] demonstrated the feasibility of developing supervisory control systems based on the finite state automata using partial observation [Cie88] to address the state-space explosion for specific applications typically found in manufacturing [Bra96]. Lauzon [Lau97], pointed out the suitability of a Programmable Logic Controller (PLC) as currently the most suitable and widely employed industrial process control technology and presented an automatic translator that used state transition tables and PLC input/output reference tables to produce a text file containing PLC source code. A test bed named Prometheus [Led95a], based on an Allen-Bradley PLC, which physically simulates a manufacturing work-cell, has been used to move from a theoretical supervisor to a physical implementation. To progress to a general approach suitable to encompass the size of control systems typically found in manufacturing, Leduc and Wonham [Led95b], describe an implementation method for DES supervisors using Clocked Moore Synchronous State Machines (CMSSM). They presented a formal model for CMSSM and showed that a CMSSM contains equivalent control information, and discussed

implementation of it on a PLC. This work was extended [Led00, Led02], to deal with the complexity of large-scale systems, by applying decomposition of software into modules (components) that interact via well-defined interfaces. An interface-based hierarchical method is presented, referred to as hierarchical interface-based supervisory control. To deal with the complexity issue of manufacturing control systems Abdelwahed and Wonham [Abd99, Abd00], observed that:

> Many practical systems can be viewed as a set of interacting subsystems or system components. Each of these components performs certain tasks and interacts with other components in a specified way to achieve a global objective. The composition of these subsystems is modeled by relating the behaviour of the overall system to the behaviour of its individual components. This relationship is defined by the way these components interact and communicate which in turn defines what is known as the system architecture… This information provides the main support for encapsulation and abstraction of the system behaviour, ultimately paving the way to contain the complexity of its model. In this regard, hierarchy is known as the most common architectural scheme supporting this purpose. In a hierarchically organized system, basic components are connected (interacting) in a distributed multilevel way.

They [Abd99, Abd00] also presented several approaches for verifying such an architecture that did not require the computation of the synchronous product of the system components.

Endsley, *et al.* [End00], developed an alternate methodology based on a framework using modular finite state machines (MFSM) [Sha02, End01]. They noted that most logic programming in the USA is currently done in Ladder Logic. Although this language is simple to comprehend at a low level, it does not lend itself to modularity or effective re-use. As product lifetimes decrease, manufacturing systems must change

more frequently and quickly, making Ladder Logic inefficient. The proposal is to use modularity and verification to address this problem. The MFSM consists of a system of connected modules shown in Figure 2-3. A module contains a modified version of a finite state machine called Trigger/Response Finite State Machines (T/R FSM) shown in Figure 2-4. Each module receives events from surrounding modules and the environment, changes state based on these events and then sends new events to the surrounding modules and environment.

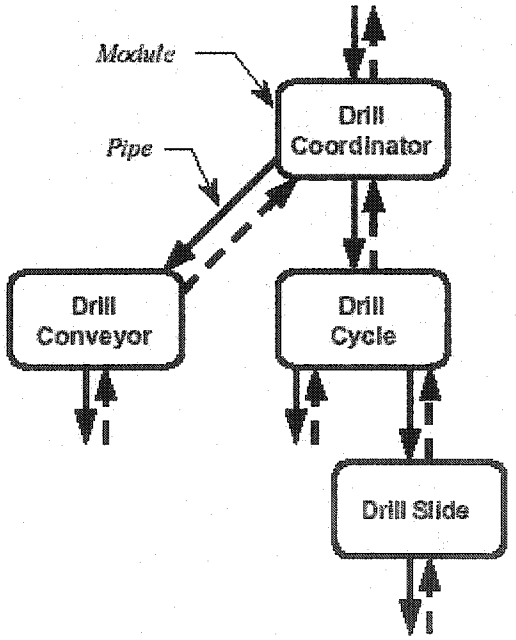


Figure 2-3: Connected System of Modules
Taken From [End01]

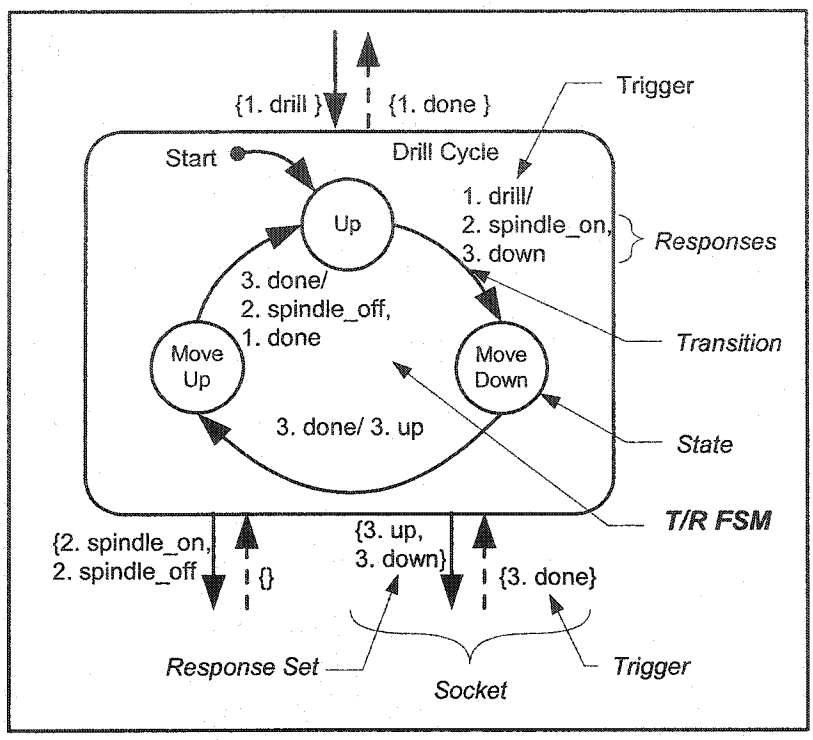


Figure 2-4: Trigger/Response FSM
Taken From [End01]

When describing the motivation for developing the T/R FSM Endsly, *et al.* [End01], indicate that supervisory control theory developed by Ramadge and Wonham [Ram87], controls discrete event systems by preventing undesired events or sequences. This is not the typical way to view the behavior of logic control. Usually one considers the control to cause action that results in events, not preventing them. Golaszewski and Ramadge [Gol87], developed the concept of forced events to address this issue. The T/R FSM uses the concept of forced events to mean events forced to occur when a transition occurs. They also add the concept of multiple events occurring during a transition. These events are not the event that caused the transition to occur but the result of the

transition. To avoid confusion, an event that causes the transition is called a trigger while the event caused by the transition is called a response and may be thought of as a command or a signal to other modules where it is treated as a trigger. Two T/R FSMs are coupled only through a response from one module being sent to another module where it is used as a trigger. It is equivalent to classical parallel FSMs coupled through shared events. A constraint of the methodology is that it does not allow an event to be a trigger in both FSMs or a response in both. Groupings of triggers and responses are referred to as "sockets". These sockets are connected between modules using "pipes". Complex systems are built by combining small modules into a collection inside of a larger module. By placing constrains equivalent to the marked Petri net methodology, the combination can be verified to be deadlock free.

Park, *et al.* [Par99a, Par98], noted that even though there has previous been work on applying Petri nets to design logic controllers for manufacturing systems [Fer92, Zay96, Zho92, Dic93], this approach is not widely utilized in the machining industry. The methodology is not simple, and it requires specialized knowledge about Petri nets. To address the difficulty with general Petri nets, a marked graph [Par99b], was proposed. It provides sufficient power and versatility to cover large classes of machining systems. A manufacturing system is first partitioned into "modules" which correspond to individual stations. Park's approach was to start with a timing bar chart that contains information about the event sequence of each module and the concurrencies of the whole process sequence. Causal dependencies of the sequence are represented using a time axis with dotted arrows correlating sequences that depend on each other. The information

from the timing chart for each module is transformed into a separate marked graph Petri Net. A marked graph is a subclass of Petri nets that has an initial marking and each place has only one input transition and one output transition. The three qualitative measures Park [Par99a], was concerned with are:

- Safeness to guarantee stable behaviour. Only one token is allowed in the net. Also allows direct conversion to Sequential Function Chart to program PLC's
- Liveness to insure absence of deadlocks
- Reversibility to insure cyclic behaviour so the system will perform its function repeatedly and can always recover to its initial state

A set of reduction rules [Par99a], were proposed that preserved liveness, safeness and reversibility. These rules were applied to each module Petri nets to simplify them with a modification to keep:

- Transitions of each module related to transitions of other modules to allow integration of module.
- Initial operation to identify the start of each modules cycle
- Last transition of each module Petri net so that the entire cycle can be clearly identified.

After reducing the module Petri nets, the results are combined into one system Petri net. The PLC programming language Park chose was Sequential Function Chart (SFC), which is one of the IEC1131-3 [Lew95] languages for logic controllers. SFC is based on Grafcet [Dav94, Dav95], which was derived from Petri nets. Park, *et al.* [Par99c], derived a set of conversion rules to transform his Petri net design into SCF.

The basic modular architecture was expanded [Par99c, Par00], to address control modes typically found in logic controllers including exception handling logic. Typically,

automatic machining systems have three control modes: auto, hand, and manual mode. Different control logic is necessary for each control mode. In this modeling methodology, each operation in a logic controller is considered first using Petri net formalism. Two types of operation modules are designed to address fault recovery. A superposition methodology was developed to overcome control logic complexity caused by different control modes. This allowed the control logic for each control mode to be designed separately. A modular logic controller consists of a mode decision module and control modules for stations in a machining system. To provide performance analysis capability, timing specifications were added to the modular controller to make it into a timed modular logic controller [Par99d]. Since operations were modeled by places in their modular logic controller, a place timed Petri net (execution time of the operation represented by the place) was selected. Since a reduction technique is used to hierarchically group places into a single macro place, the time assigned to it is the sum for a series or maximum for parallel configurations.

Birla [Bir98], developed a modeling methodology that combines the object-oriented [Ber95] composition model with the formal method of finite state machines [Har87, Har90, Bra93, Kuu91, Wan99, Shi98]. This methodology is compliant with the OMAC user group specifications [OMA99]. Birla [Bir97], developed a library of reusable building blocks and software organizing principle. The architecture overcomes a number of historic limitations of traditional machine tool controllers:

- scalability
- interaction across controlled objects in close temporal coupling, e.g. interaction of continuous and discrete control objects

- retrofit of sensors for high-speed data acquisition during motion
- retrofit of objects that specify control laws
- variety of kinematic configurations
- modification of motion path during motion.

However, Birla [Bir98], states that even when one has a full set of components, too much effort is spent in composing and reconfiguring applications. He suggests that the *ease of reconfiguration* should be further researched.

It has been noted [Wan99, Sto01, Sto02], that software for manufacturing control systems can be divided into two parts, controller software and application software, as shown in Figure 2-5. Controller software defines the functionality of the system and is platform-dependent. The application software defines the behavior of the system and should be platform-independent. In this architecture, the software of a control system is

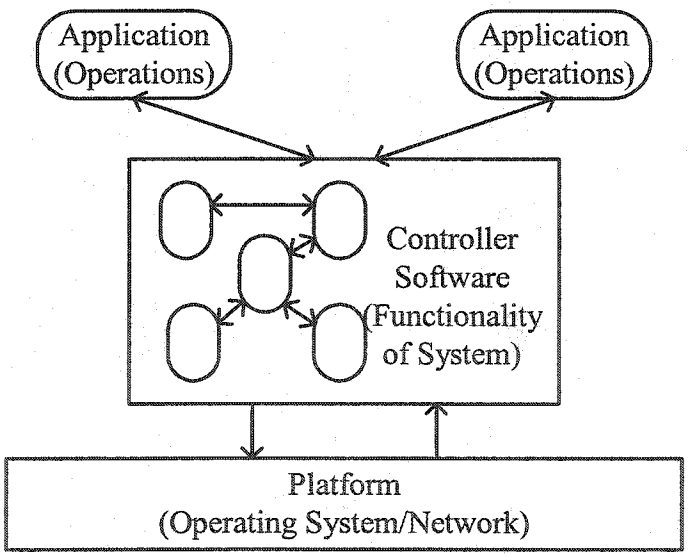


Figure 2-5: Controller Software Architecture
Taken From [Sto02]

modeled as a set of communicating components. Components are designed and

implemented with a set of external and internal interfaces, as well as environmental protocols, as shown in Figure 2-6. Inside each component, there is a control logic driver devised to support separate control logic specifications. There is a mechanism that separates the behavioral information from the functionality of the component. It can be implemented without the control domain knowledge. The environmental protocol enables the component to be used in various platform configurations. Further, the existence of internal/external event mapping enables the implementation of the component independently of its deployment. Therefore, components can be purchased from various vendors, ported from other existing applications, or specifically developed for a particular application.

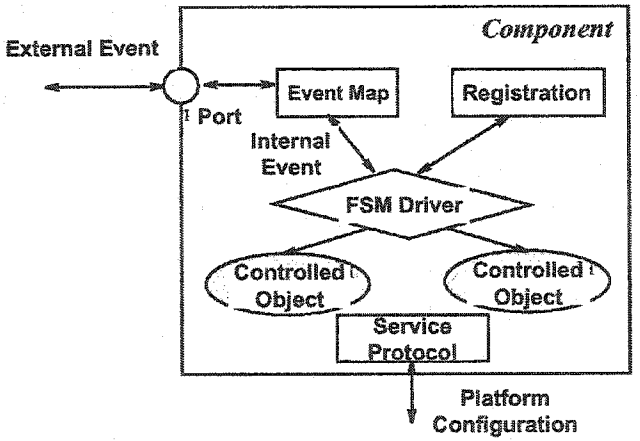


Figure 2-6: Software Component Structure.
Taken From [Sto02]

To support composition with different granularities, components are organized hierarchically. The behaviour of a component can be specified independently of its implementation and the other components' behaviours. The behaviour of each basic component is modeled as a Finite State Machine (FSM), and the behaviour of a

composite component is modeled as a Nested Finite State Machine (NFSM), which synchronizes the FSMs of its member components. The FSM they [Wan00] typically used is a Mealy machine. A more formal definition of the NFSM can be found in [Vil97].

2.4 Summary

This chapter presented the current research addressing the issues of divergence, system context, code development methodology and the lack of sufficient experimentation by the software community for transitioning technology to industry. The next chapter describes the tools used in the design proposed as a solution to these problems.

Chapter 3

Summary of Contributions and Methodology

This chapter is included to address the difficulty in presenting a multidisciplinary research subject in a concise manner that can be readily understood by readers who do not have expert knowledge in all of the disciplines involved. These disciplines include real-time systems control, computer science, electrical engineering, artificial intelligence and technology insertion. This chapter briefly presents an overview, the novel contributions to the fields involved and the methodology developed that address issues arising from development of Agile Manufacturing Information Systems (AMIS).

3.1 Overview

3.1.1 Motivation

To deal with rapid change, which is generally accepted as being "the new normal", industry has adopted the agile manufacturing paradigm. Empowered workers are a key ingredient. Workers are dependent on timely and accurate information to make decisions. Information systems are dependent on monitoring for data and monitoring systems in turn are dependent on the control systems that provide the data. In the new generation, the norm is constant change (from data itself through to subsequent uses), not the exception, therefore the need for an AMIS becomes acute.

The development and subsequent non-disruptive, successful insertion of an AMIS into a business organization requires the use of modern concepts and tools. For example, context enhanced data maps well onto the object-oriented approach where objects consist of both attributes (values) and behaviours (methods) corresponding to data and context. The deployment process of a new knowledge-based system requires not only a minimal disruption of the current organization/operator activities, but also provides a migration path for upgrading the skills and capabilities of personnel. An agile approach is well suited as this matches its primary tenants. These concerns are the focus of modern technology insertion techniques (TI). To achieve a complete AMIS requires a more global approach of deploying Performance Support Systems (PSS) at all operational levels of an industrial enterprise [Sto03a]. McMaster's Applied Computersystems Group (AC_SG) has pioneered an "anthropomorphic" approach [Poe93, Poe94] with agents at several levels (i.e. "managers" at higher levels, "supervisors" at middle levels and "technicians" at lower levels), which serves to realize the implementation of the "Performance Support System" [Wil93]. A growing trend in Component Software Architecture is the use of PSSs where knowledge based rules are used to select the components appropriate for a specific circumstance [Dub02]. The use of such problem solving techniques is becoming prominent in modern systems. This achieves the agility cornerstone of empowering people to function at a higher level of capability and efficiency by providing a consistent set of problem solving tools for a given criteria set. Put another way, suitable information reaches the appropriate person, at the right time, to empower the individual in making informed decisions. Such an environment enables

agile behaviour and allows the employee to cope with constant change and the new normal.

These aspects are examined by this thesis in the pursuit of addressing the very specific problem encountered in the manufacturing arena: process control and monitoring. While the area is finely focused, the implications of the human factors addressed are much broader. In particular, the procedures required to achieve operator effectiveness and capability at an agile level, are significant and require understanding and modifying operator process control methods (the aforementioned TI migration path). The training of new personnel to operate at this higher level of agility must also be addressed. In this sense, knowledge capture from experts and its dissemination during both process operation and operator training is important and explored within this thesis.

3.1.2 Problem

A problem exists in keeping monitoring and control systems synchronized. The problem stems from different groups developing the systems. The control group first writes the control code, followed by the monitoring group analyzing their work to provide the monitoring code. As time goes on, the control code is changed resulting in divergence with the monitoring code. Effort is expended to synchronize them and the cycle repeats. In a generic approach, a single enhanced entry point is developed for both control and monitoring engineers, as well as support trades people, to access and perform their operations. In this way, when control makes a change, the computer system adds

appropriate modifications via tagging this aspect to the new data, or the "context" portion of the "context enhanced data". A more detailed examination of the approach follows.

Current industrial practice for monitoring in manufacturing assembly lines [Sto03a, Lip03, Abr03, Has02, ARC02, Poe95], as a particular example, is collecting data without system context. System context refers to the state of the machine at the time the data was collected. For example, if one is counting the number of parts built, it is important to distinguish between normal automatic operation and a maintenance person shuttling the same part back and forth while setting up the machine.

There is a large body of literature on methods, which would theoretically address this situation but there has been little penetration into use by industry. There is little written on applying these methods to industrial problems and insufficient experimental proof that they are significantly better than the current practice to ignore monitoring data collection in favour of efforts to optimize and eventually change control side configurations.

3.1.3 Contributions

The following contributions address these problems:

- A generic abstract model and framework that can be applied to most roll-forming work-cells,
- An empirically developed methodology,
- Transformation from the model representation to PLC code, in order to aid with technology insertion,
- Knowledge capture in an application generator,

- And an Experimental environment.

3.1.4 State of the Art

The control of discrete event systems has been the focus of extensive research. Brandin [Bra96] has categorized it as knowledge engineering [Mur98, Alb91], Petri nets [Hol90, Fre91], controlled automata [Ram87, Bra94], while Castillo and Smith [Cas02] further expanded the approaches with state charts [Har87, Har87b, Har88, Har90] as well as various formal languages [Ben91, Cas97, Hal92, Hal93, Gue91, Ber91, Bou91] and algebras [Fen96, Mil89, Bol87, Bow98, Hoa85, Met94]. To address the complexity that arises in manufacturing cells, hierarchy, abstraction, modularity and other Object Oriented (OO) techniques have been used. However, the issues of monitoring system support and its transition to industry have not been addressed [Sto03a, Lip03, Abr03, Has02, ARC02, Cor02]. Due to the scarcity of literature on the subject, it would appear that this problem has not yet been systematically attacked from an academic perspective. An attempted remedy to this specific problem in the manufacturing arena is addressed in our forthcoming paper [Sto03a].

3.1.5 Foundation

Existing methodologies, outlined in chapters 2 and 4, were first surveyed to provide a basis for this research. Simplicity of method, acceptability by target users, simplicity of tools and the ability to automate, were the criteria on which the selection was based. Thus, Birla's state based object oriented methodology serves as the

foundation for our work. The OO methodology involves encapsulating objects (static attributes) with their behaviour (methods). This is ideal for keeping data and context together (just like encapsulated objects). In addition, this provides a method of organizing associated information such a video capture of sensor and actuator locations that can then be automatically retrieved. For example, an object representing a state transition can encapsulate the video of the sensors that signal the event which allows the transition to occur. The OO methodology provides an approach for composing software by selecting and configuring components [Dub02] as presented in [Sto03a] where software selects the appropriate tool from a toolbox according to the scope of the task at hand.

Birla [Bir98] notes a composition problem with his methodology. When one has a full set of components in a library, too much effort is required to understand what the components do and how to put them together. Our analysis indicates that this arises from the numerous different methods available to construct control code.

3.2 Generic Model, Framework and Application Generator

To deal with the composition problem, we developed a generic OO class model and framework that constrains the user to a single standard approach and provides an application generator to assist the user. Such a model and framework define all of the kinds of objects that can be used as well as how they interconnect and interact. The user is constrained to simply selecting an appropriate set that can only be configured in the standard manner. An application generator is employed by the user to map the behaviour

requirements of a specific work-cell (how it is to operate) to the appropriate generic classes, which are then configured and instantiated into an object model from which control code is generated. The software comprising the application generator is knowledge-based representing the method used by the controls expert. It provides an agile tool for the non-expert user for generating control code as well as the information needed by the monitoring system in the same manner as the expert. The application generator currently allows the user to configure objects that result in Boolean Logic. The abstract model encompasses conditions containing timers and counters, however, the application generator utilizes timers only for a lubricator design pattern. The inclusion of user assigned timers and counters in various conditions is reserved for future work. The following sequence of steps was utilized during this research:

- A simple test case was used to begin developing the model, framework and methodology and test its applicability. For this test case, control of a single station was developed using a state-based approach and a class model was derived.
- Having proven the value of the technique, a more complex case was used to develop a generic OO model. A work cell containing examples of the different kinds of stations used in roll forming served as a representative system.
- A manufacturer of roll-forming work-cells has successfully used the model and methodology for three years.
- A controls expert on these actual control systems was consulted as critic to verify that all of the configurations in use was included (as far as is known) in an effort to insure that the model was generic.
- A standard transformation from the model to PLC code was developed to aid with the technology insertion and to prevent the proliferation of different coding implementations.

- We refined and adapted the models for these systems in an iterative fashion to address the requirements that were identified through the iterations.
- Generalization from these specific cases was used to derive a generic abstract class model.
- The control engineer's knowledge was captured in the class model.
- A three level hierarchy was used to deal with the complexity of the systems.
- A limited number of classes was used to keep the model understandable. This is in keeping with Miller's research [Shi94, Bad94, Mil56] on the capacity of human information processing.
- An application generator was written to automate the process of assembling the object model and generating the control code.
- A training course was developed to educate and transition trades people from current practice to the use of the methodology via the application generator. The time restrictions imposed by the thesis limited the review of the course to one tradesman. However, it did illustrate the feasibility of upgrading human skills to the new level required by the new methodology TI.
- Experimentation with a Pipe Fitter tradesman was used to evaluate and refine the transition path of training material and the application generator interface.

3.3 Methodology

A single point of entry is the key attribute of our methodology that addresses the divergence between monitoring and control. All activity occurs at the abstract model level where the user maps the requirements of the system onto the classes that implement them. This mapping results in the specific object model for the work-cell at hand. From this object model, code is either automatically generated by an application generator, or manually written by employing standard transformations. The same code topologies always result from applying the transformations. The object model also provides all of the information that a monitoring engineer requires for monitoring systems by identifying

and locating the required data along with the associated context (state of the various objects). Similarly, when changes or maintenance are required, the user works at the top abstract model level to modify the object model and then generates the code again. Thus the control and monitoring systems are always in synchronization, being derived from a single object model constrained to be instantiated from a generic but single standard class model.

In order to have one entry point to solve the problem, all levels of people, both skilled and unskilled will need to effectively access the system of interest through this point. Everyone in the organization, with different job functions and expertise, needs to operate at this high level of abstraction. To transition an organization to this point requires:

1 A migration path from existing operations to that higher level of access.

2 Accommodation of different perspectives and skill levels:
   (a) Control types including development engineers, troubleshooting technicians and maintenance electricians (decreasing skill)-- our main customers.
   (b) Monitoring types -- less skilled at this level.

This research utilized an experimental iterative approach with individuals representing various levels of skills and perspectives to develop and refine the methodology, migration path, application generator user interface, training approach and material. The foundation was provided by the author's research (both as part of the curriculum and as an employee of General Motors) in manufacturing control systems [Sza86, Sto01, Sto02, Sto03a, Sto03b], monitoring systems [Sto86] and communication

architecture for support of monitoring systems [Sto83, Sto90, Hod91, Hod92, Hod93, Gra94].

3.4 Summary

This chapter presented concisely the multidisciplinary subjects of this research in a manner readily understood by readers who do not have expert knowledge in all of the disciplines involved. A brief overview was provided, the novel contributions to the fields involved were stated and the methodology developed to address issues arising from development of Agile Manufacturing Information Systems was outlined. The following chapters provide much more detail on the approaches taken and the rationale for those approaches. A stepwise overview of the methodology is provided in Appendix A. Results, discussions and conclusions round out the rest of the thesis.

Chapter 4

Tools For Synchronizing Monitoring and Control Systems

To address issues arising from development of Agile Manufacturing Information systems requires integration of aspects of many disciplines. These disciplines include real-time systems control, computer science, and electrical engineering, artificial intelligence and technology insertion. This chapter reviews tools from these disciplines that were applied in the design of the solution. Modeling of real-time systems is used for synchronization of monitoring and control codes. Artificial Intelligence is utilized within an application generator to provide a method of configuring reusable components into a control plan and to address the lack of context in the collected data. Object Oriented methodology supplemented with extended finite state machines addresses the lack of application of scientific methodology in current control code development. Technology insertion and experimentation is applied to transition the technology needed to solve these problems from the theoretical domain to practice in industry.

4.1 Real-Time System Modelling

The GM Portfolio Project proposal [Poe95] to pilot an AMIS in Oshawa identified the need to move the control engineer's practice of generating control code to a higher level of abstraction and discipline than working at the Ladder Logic level. Poehlman, *et al.* proposed Petri Nets as the underlying methodology and Graphcet as the

implementation. However, Wang [Wan00b], pointed out that the state machine paradigm, though not as powerful as Petri nets, was sufficient with less complexity, making it easer to comprehend and thus more likely to be accepted by the targeted user. It has been shown [Tan81] that discrete systems can be modeled by either approach. State machines lacked the utility of multiple tokens as well as their use for synchronization, but this lack was also an advantage. Simple tools such as spreadsheets could be used to express a state machine. However, there was no simple manner to represent the functions of tokens. Since the targeted user was working directly at the code level, the smallest change required that would address the synchronization problem would provide the highest probability of acceptance. As mentioned by Endsley, *et al.* [End01], academic researchers have used formal discrete-event system frameworks (such as Petri nets and finite state automata) for control problems but little of this research has penetrated into industry. Muro-Medrano, *et al.* [Mur98], indicate that Petri nets are a suitable means to describe simple systems. However, the use of ordinary Petri nets in modeling of large complex systems can lead to models of unmanageable size. This drawback is reduced by using high-level Petri nets (e.g. coloured Petri nets or predicate/transition nets) which provide more compact and manageable descriptions. The high-level Petri net formalism still lacks appropriate features to support modern systems engineering concepts such as modularity, encapsulation, top-down and bottom-up designs, data abstraction, specialization, and inheritance. In addition, they are not powerful enough to describe complex systems (such as information or manufacturing systems). Muro-Medrano, *et al.* [Mur98], proposed to address these issues by integrating

high-level Petri nets with the frame/object-oriented paradigm. Birla's research [Bir97] showed that the use of the Object Oriented (OO) paradigm could provide a library of reusable components that would provide the missing Petri Net functions. Poehlman observed [Poe02] that the problem with many of the formal methods is their insistence on strict adherence to the formality. The difficulty of the targeted user to comprehend and apply the methods was not adequately addressed, resulting in the lack of adoption of the method. This research is the launch of a technology insertion and transfer process. The objective is to eventually migrate industry from their current state of the art to the highest level attainable. The use of a less formal method is simpler to comprehend and thus more likely to be adopted by the targeted user. As the benefits are realized, industry should become more receptive to more complex methods. Thus, this research chose to compliment Birla's work [Bir97] as the simplest, most acceptable method for industrial users. The foundation of his domain model is the object-oriented paradigm [OMG95] with an extended finite state model [Har87] for specifying the dynamics of behaviour (flow of control). Birla [Bir97], summarizes the controversy over this form of modelling for this research application domain, particularly for real-time controls:

> Criteria at one end of the spectrum, are logical unambiguity and precision, driving toward a formal model, and criteria at the other end of the spectrum are ease of comprehension and implementation, requiring a seamless transformation from a model to an implementation. There are no commercial tools available for meeting these criteria economically in this application domain. Extensibility of a specification is another important criterion for adaptability to changes in requirements. Portability of the specification, or independence from an implementation language, or freedom to select different implementation languages for different modules is also a consideration.

Birla [Bir97], makes the following observation on formal specification languages:

> Formal specification languages leave semantic gaps between the real-world requirements and the formal model and between the formal model and the implementation language. In the worst case, there is a loss of information and uncertainty in the transformation from human expression to the formal language, because a formal specification language, by itself, does not make it easier to capture and express concepts natural to the application domain. It is difficult to validate the formal specification against the real requirements of the application domain, without a number of implementation cycles. In the second case, uncertainty is introduced in the transformation from the formal model to an implementation language. It is difficult to establish that the transformation tool is more correct than the more popular language compilers in commercial use, particularly, in the implementation-specific aspects of the tool. With its much lower usage volume than the leading language compilers, the transformation tool will inherently have a longer debug interval and higher amortized cost.

On the question of formality of the object model Birla [Bir97], states :

> One weakness of the OO model is its lack of a sound mathematical foundation. We assess the relevance of this weakness by comparing our domain modeling paradigm with the Z notation [Spi89] - one of the more commonly accepted formal specification languages for real-time software. The Z notation is based on first-order logic and set theory. It relies on a combination of mathematical notation and prose. The notation is used to express types, including a special type called schema, predicates on these types, a schema calculus for defining schema expressions, and a description of state machines. The Z schema corresponds to the C++ classes used to build our domain model. The Z schema inclusion corresponds to specialization in the object model. The Z type is an extensional concept (set membership of included elements), in contrast to the more compact and "pure" intentional concept in the object model. Our domain model specification of the dynamics of behavior (flow of control) follows an extended finite state machine mode (FSM). It identifies the state of an object symbolically, but we do not have as compact a notation for the value of an object after an operation. The object model also lacks as compact a notation as Z for expressing predicates. Thus, considering the criterion of disambiguating requirement specifications, the foundation of our domain model is comparable to the Z notation. Therefore, we do not consider other formal modeling alternatives in further evaluation.

4.2 Technology Insertion

The Portfolio Project proposal [Poe95] recommended placing a computerized system between the engineer developing control code and the controller itself. The engineer would interact with the system which would create the code for the controller on his behalf. To a large extent this is analogous to computerization of human tended machines. The same issues that arise in that form of automation identified by Lind [Lin97b], will also be present in the AMIS effort and the same metrics of reliability, flexibility, low-cost and operator-friendliness. Lind indicates that mature human-machine interaction is both highly complex and highly effective due to the many work-hours that human operators interact with the machine. Any attempt to insert a *computer-based-intermediary* between a human and the machine requires great care to prevent loss of effectiveness. This creates a *technology insertion gap* between the user and the machine. Lind illustrated this process which he adapted from Holnagel [Hol91] as shown in Figure 4-1. After the computer intermediary is in place, the new composite system undergoes what Hollnagel refers to as post-insertion development shown in Figure 4-2 adapted by Lind. The first stage is one of disparate relation between the human, computer intermediary and machine. In this stage the human is disoriented by the change in how he must now interact with the machine. In many cases productivity is reduced to the point where machine operation is disrupted. The familiar direct manipulation and feedback have been replaced with an intermediary that behaves in a different manner. The first milestone in the process occurs when the human sufficiently adapts to and accepts the computer intermediary so that machine operation can resume. Hollnagel

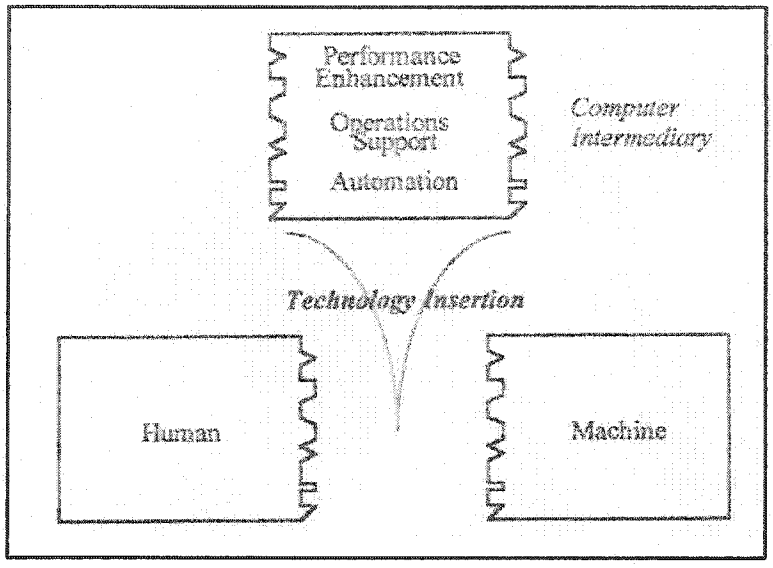


Figure 4-1: Technology Insertion Gap
Taken From [Lyn97b]

names this state the embodiment (amplificatory) relation. The computer intermediary transforms the experience and adapts it to the user. The user perceives the computer as separate from the machine. The computer is seen as a tool through which machine and user interaction occurs. As the user gains familiarity and better understanding of the computer intermediary, there is a shift in perception of relationship. The user now sees the computer as an integral part of the machine and interacts with it rather then through it. The final stage of the insertion process is when the user has reached a degree of comfort with the computer and considers it as the preferred machine interface.

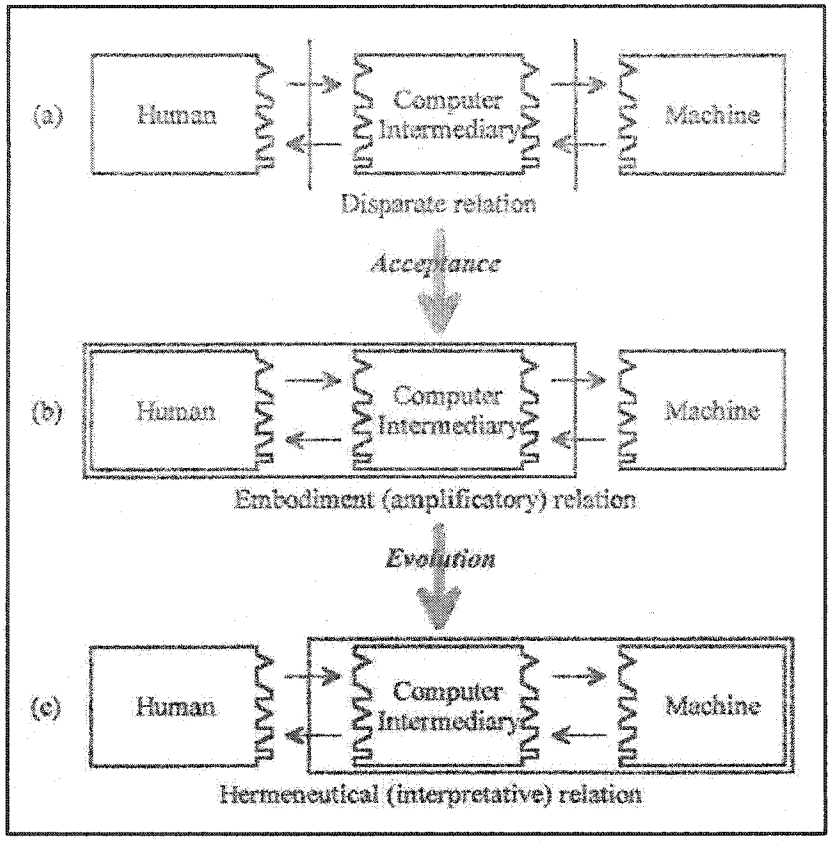


Figure 4-2: Three Stages of Post-Insertion Development
Taken From [Lyn97b]

4.3 Application Generator

An application generator can be used to address the issue of selecting and configuring reusable components into a control plan [Bir98]. Horowitz and Munson [Hor84] describes an application generator as:

> A software package which is designed to help end-users build applications in a given domain, such as financial ledger systems. Compared with the previous category (reusable design), these packages have very specific knowledge about the area of application in which programs are to be written. This permits the specification language to be very precise. Also they are designed with a naïve end-user in mind. One could view an application

> generator as the output of a domain analysis. As the domain becomes more tightly constrained, the efficiency and use of these systems is easier to achieve.

They state that a nonprocedural language which is designed for the non-technical person to use is of primary importance. The end user selects items from a menu. A second basic element is built-in knowledge about the details of program creation. Long debugging periods associated with the writing of programs in a conventional manner are avoided because the code has already been debugged. Such an interface can be readily implemented using standard graphical methods for item selection available in windows based system such as Microsoft Windows, MAC OS and others. An expert system can be built to provide the detailed knowledge of how to write the control code. Giarratano and Riley [Gia89], assert that expert systems are commonly used when an inadequate algorithm or no algorithmic solution exist as is the case with the current methods for creating control code.

The application generator concept can be augmented with Coyne, *et al's.*[Coy90] view of knowledge based design systems where they consider a design to be a description of a class of artefacts the instances of which is the design. The concept of a frame is utilized when creating a design. They state that class frames and instance frames are needed. Initially, the information available to a designer may be only of the *class* frame variety – that is, information about classes and class properties. Any one design is represented in terms of *instance frames*. Each instance frame is an instantiation of the class frame to which it is connected. The instance frame inherits all of the slots

(locations for specific attributes) of the class frame to which it is connected. This copying of frames provides a concise and well-structured system for organizing information about design objects. Thus to create an application generator one needs a description of classes that describes the hierarchy and relationship of the parts that a design can be constructed from. This is an abstract model. To create a design, the particular parts that are needed are selected in the application generator, which creates a copy of the class frames representing the parts, and provides a means for the user to fill in the values of the slots or attributes that uniquely configure the instantiated part. This creates the instance frames. Using the information in the instance frame, a simple rule based forward chaining approach can be used to step through a sequence of creating logic to implement the objects and their behaviour and then cycling through the created logic to map it into appropriate Ladder Logic code segments.

Coyne, *et al.* [Coy90], suggested that certain tasks that are difficult to model mathematically could be accomplished more easily using knowledge-based systems. They further state that design is a process of instantiation, proceeding from a class description to a description of an instance. Reasoning takes place by matching the patterns of a new situation with those of past experiences. This approach forms the foundation strategy for the application generator. The patterns that can generically solve the control needs of typical systems are captured from a control engineer in the form of the class frame. An instruction process familiarizes the user with a vocabulary of parts that describes all of the parts in the class frame. The application generator provides a means for the user to map the control requirements of the system to a selection of parts

required. An instance frame is then generated and the user supplies the information for the slots. Additional parts are generated based on the hierarchy of the model, parts already selected and on further user selection. Eventually, the instance frame is complete and the generation of logic can commence by applying rules for identifying the components in the required sequence, selecting and configuring the appropriate logic patterns previously defined by the controls expert. The same process is then applied to the logic components that have been configured into an instance frame to finally generate PLC Ladder Logic control code from patterns previously defined.

4.4 Technology Transfer

One of the obstacles of transferring software technology to industry is substantiating that the new method is better than the current state of the art. There is little statistical evidence that applying these software principles and methodologies to practical scale control software production will yield significant improvement. Linkman, *et al.* [Lin97], suggests the use of experimentation as a vehicle for software technology transfer. Fowler and Levine [Fow93], provide a conceptual framework for software technology transition shown in Figure 4-3. They state that a broker of some kind usually

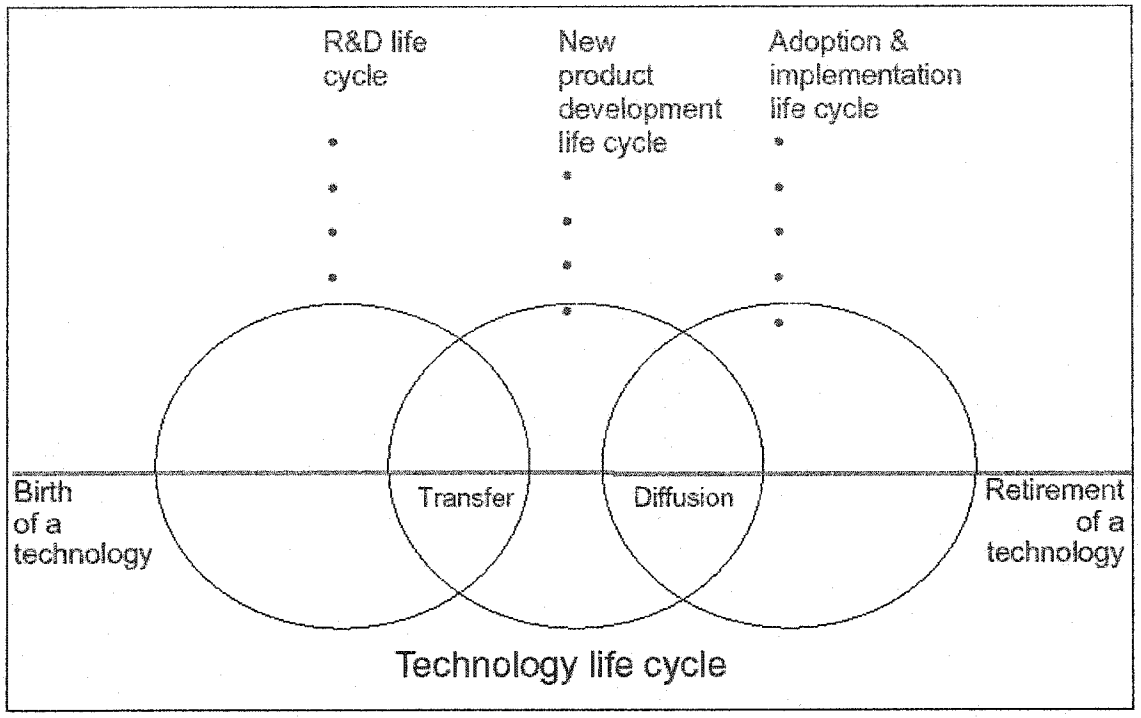


Figure 4-3: Conceptual Framework For Technology Transition
Taken From [Fow93]

facilitates the transference of technology between technology producers and technology consumers. Generally the broker is an advocate on behalf of the producer, not the consumer. The **receptor function** is the complementary role acting on behalf of the consumer. The concept of **mutual adaptation** offers a model for how the receptor function, as a recipient of technology for downstream organizations, adds value. Mutual adaptation means that adjustments are made to both the receiving organization and the technology in order for adoption and use to occur. Receptors act as co-inventors or at least co-developers, translating the technology on behalf of the context they represent. This relation is shown in Figure 4-4 and 4-5. Engaging an industrial user in an

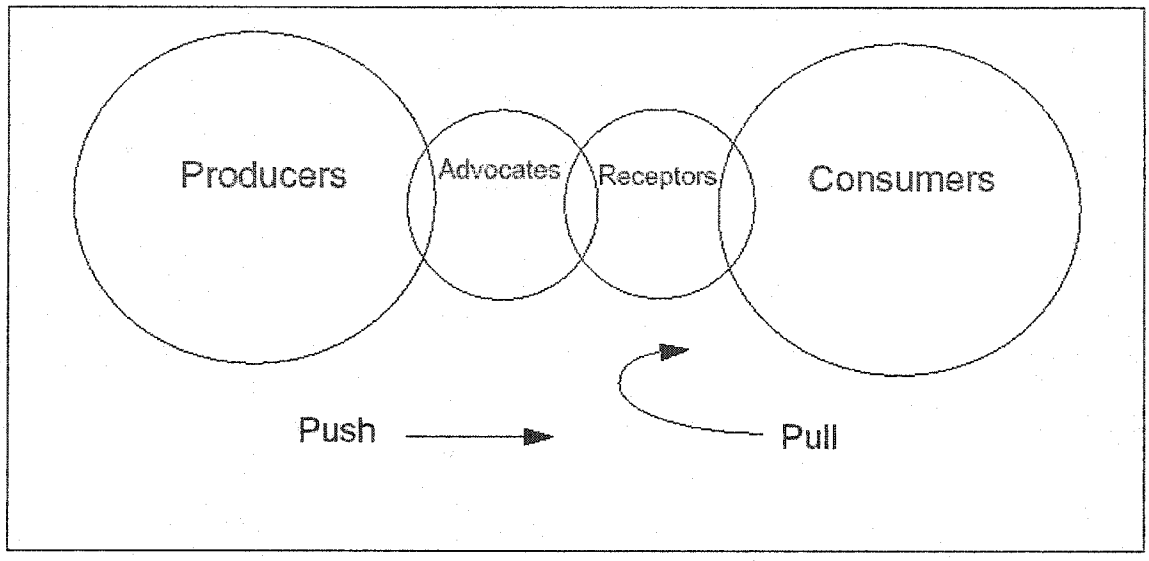


Figure 4-4: Transactions Between Intermediaries
Taken From [Fow93]

experimental development effort and developing control code for actual machines using the new methodology uses the two tools of experimentation and mutual adaptation. This approach should increase the chance of acceptance. The end user has input into the design, provides their insights and thereby gains a sense of ownership. The system used for the experiment is the end-user's system. This addresses the potential mismatch between the types of problems that have been posed in the academic community and the types of problems seen in the manufacturing industry.

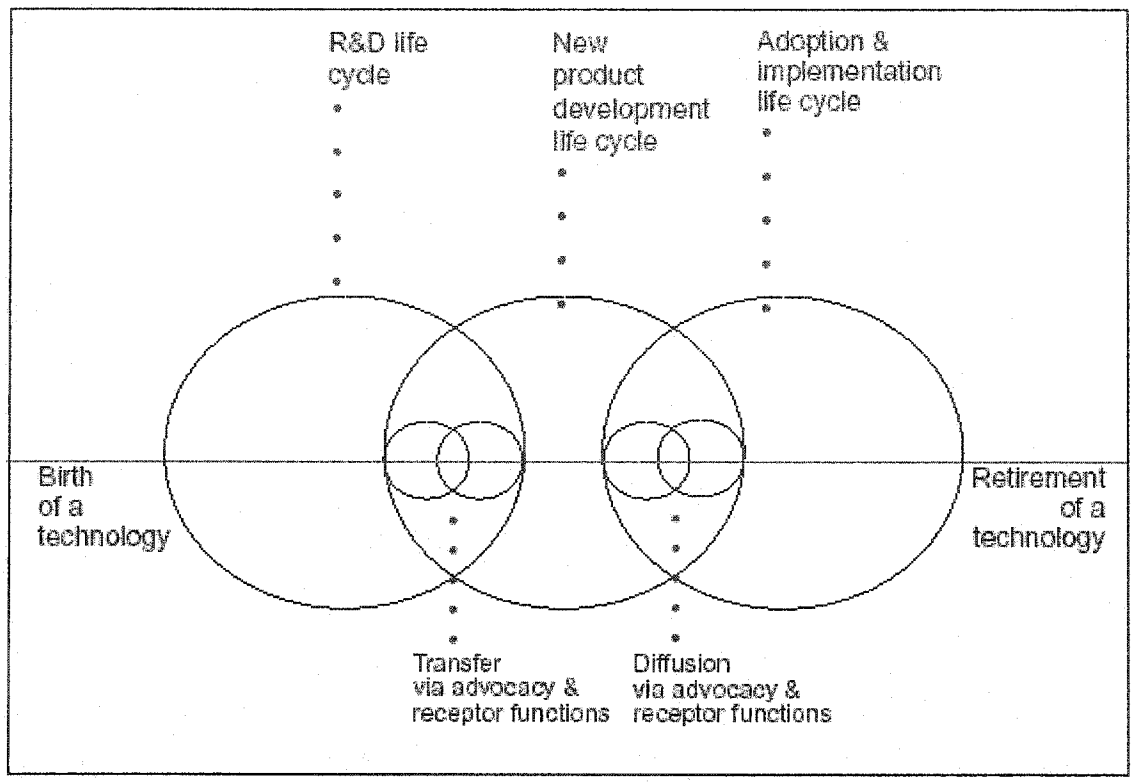


Figure 4-5: Conceptual Framework with Transactions Between Intermediaries. Taken From [Fow93]

Since the experiment requires an expenditure of resources by the industrial receptor that needs to be justified, an iterative approach is used. The strategy is employed to reduce commitment of effort of the industrial participant for the first iteration as much as possible. The objective is to select a major architectural component and prove its value. This provides a cost justification to commit substantially more resources for the next iteration and shows that the degree of risk incurred in addressing a larger, representative scale system is acceptable.

4.5 Human Computer Interface

The human-computer interface is where interaction between the computer systems and the human occur. It is commonly referred to as the user interface. This includes the inputs and outputs of the computer such as the video display, keyboard and mouse and the human's ability to sense and manipulate consisting primarily sight and hand motion. Nickerson [Nic86], identifies two aspects of the human-computer interface. The physical interface encompasses the mechanisms of the interaction consisting of the computer interface devices and the human's sensory-motor system. The cognitive interface pertains to the form and content of the information exchanged. The cognitive interface is based on models of how humans think and solve problems. Ziegler and Bullinger [Zie91], as quoted by Lind [Lin97b] state that it is desirable to integrate these two views into one common representation.

There are different approaches or styles available to implementing the interface. Lind, provides the following summary of five primary styles identified by Shneiderman [Shn91], with their advantages and disadvantages:

- Menu Selection: The user selects items from a list displayed by the computer. This style can shorten training time, reduce the number of keystrokes, promote structured decision-making, and enable use of dialogs and support error handling. However, it may slow down frequent users, consume display space and tend to become complex and difficult to navigate if many submenus are needed.

- Form Fill-In: The user supplies information by "filling in" a set of labelled fields comprising a "form". This style simplifies data entry, requires modest training, provides convenient assistance and context for activity, permits use of form management tools, and enables built-in

automatic data validation and error handling mechanisms. However, forms consume screen space and require typing skills.

- Command Language: The user specifies commands using well-structured syntactical formalism, permitting flexibility and speed, and encouraging user initiative. A command language is appealing to experienced users who are able to formulate command sequences mentally with little reliance on the visual feedback associated with menus or forms. To their disadvantage, command languages require substantial training and memorization, may be difficult to remember, and can suffer from poor error handling mechanisms.

- Natural Language: the user communicates with the computer using the user's natural language, thereby relieving some of the burdens associated with formalized command languages. Natural language for computer interaction suffers from ambiguity necessitating clarification, and may require more typing than command languages.

- Direct Manipulation: The user interacts with the computer system in a highly visual manner, employing a graphics screen and pointing device (such as a mouse, light pen or touch sensitive screen). Instead of typing in information or specifying commands, the user selects and manipulates objects to perform tasks. In this interaction style, the task can be visually represented making it easy to learn and retain system operation, encouraging user exploration, and reducing errors. Direct manipulation requires specialized hardware such as graphics screens and pointing devices, and a high level of programming.

To aid in selection of the appropriate style for the task and user, Shneiderman developed a set of guidelines. These are presented as a set of three rule bases. The first guideline matches the interaction style with task related factors as shown in Figure 4-6. The second considers the user skill level shown in Figure 4-7 and the third set of rules determines a variety of interface design aspects based on the type of user shown in Figure 4-8. Even with guidelines available, researchers report [Nic86] that developers typically underestimate the difficulty that users will encounter. Consequently, they recommend an architecture where the user interface can be cleanly separated from the rest of the system.

This allows improvements to the interface to be made without requiring major revisions to the underlying system.

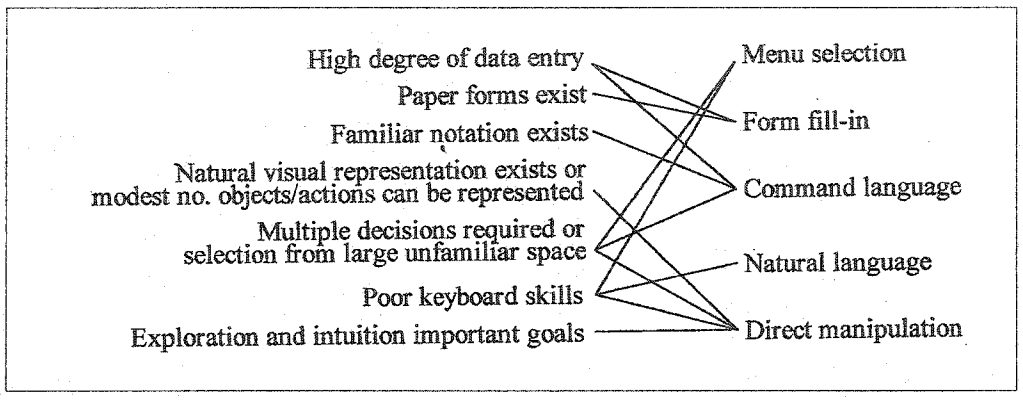


Figure 4-6: Task Factors as Determinants of Interaction Styles
Taken From [Lyn97b]

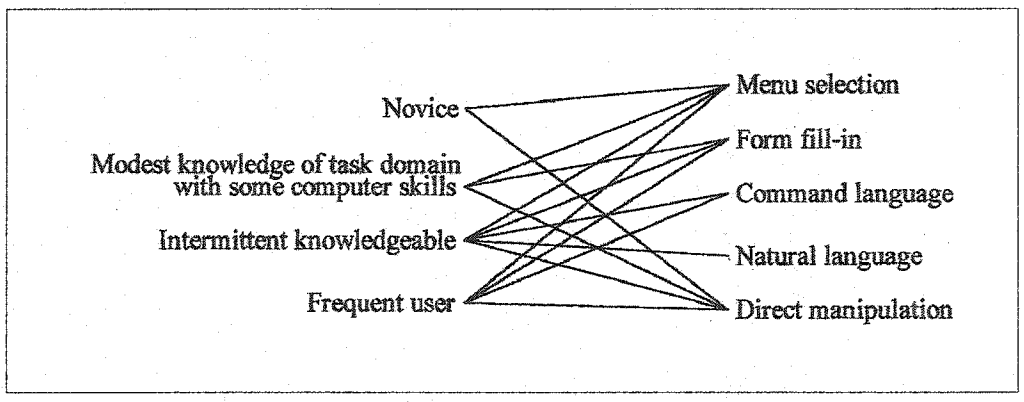


Figure 4-7: User Skill Factors as Determinants of Interaction Styles
Taken From [Lyn97b]

	Novice	*Knowledgeable Intermittent*	*Frequent User*
Informative feedback	High density	Modest	Short, sparse or none
Pace	Slower	Moderate	Faster
Other	Introductory tutorial/demo. Limited subset of actions and functionality.	Online help. Chance to move up to more powerful actions, but protection from danger.	Online reference with elaborate search mechanisms. Abbreviations, shortcuts, user-defined macros, access to system internals.

Figure 4-8: User Skill Levels for Determining Various Aspects of Interaction Design Taken From [Lyn97b]

Initially, the user interface was considered to be of secondary and lower importance than the other aspects of this research. The application generator was designed such that the user interface was a separate tier in a three tiered architecture [Rey02] to allow later improvement should they be required. The targeted user was a tradesman not familiar with control code generation nor familiar with the use of computers for design purposes. Thus the novice skill level was assigned. Poor keyboard skills, multiple decision requirements and selection from a large unfamiliar space were selected for the task factors. Using this as the characteristics expected, the guidelines described indicated that the direct manipulation style was appropriate. In addition, an introductory tutorial and demo with a slower pace with a high density of feedback was suggested. These were incorporated into the application generator user interface.

4.6 Iterative System Design

A clear direction did not exist for obtaining a solution to the problems addressed by this research. Software engineering suggested using an iterative approach to system design when faced with this situation. Booch [Boo94], indicates that the requirements of a software system often change during its development. The very existence of a software development project alters the perception of the problem. When users start to see design documents, early prototypes and begin using preliminary systems, they develop a deeper understanding and can better articulate their real needs. Yourdon [You93], presents an excellent analysis of this iterative process. He starts with the traditional waterfall life cycle shown in Figure 4-9, and provides a definition of methodology, life cycle and method. He refers to methodology as a step-by-step battle plan or cookbook, for

achieving some desired result. A software methodology usually identifies the major activities (i.e. analysis, design, coding, and testing) to be carried out and indicates which people should be involved in each activity and what role they play. Life cycle is synonymous to methodology, while method is the step-by-step approach for performing one or more of the major activities contained in the overall methodology. "Structured" as opposed to "object-oriented" are mostly differences in method.

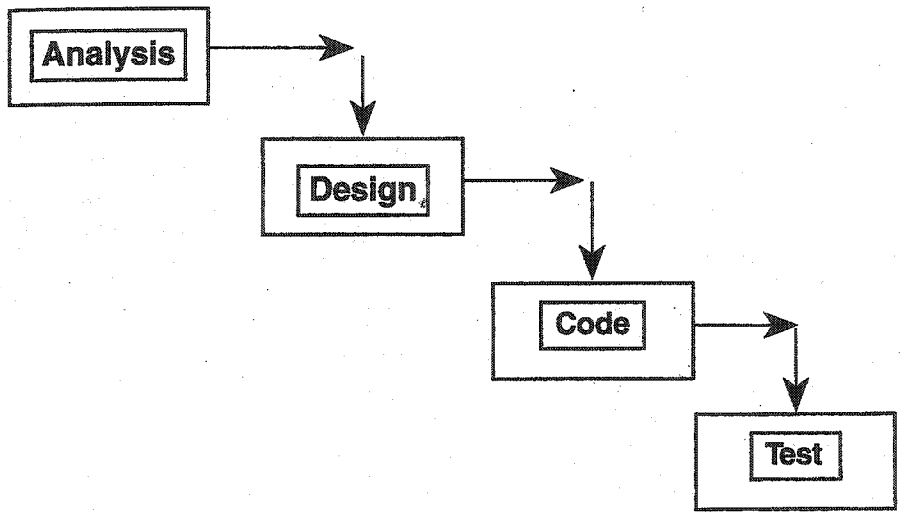


Figure 4-9: The Waterfall Life Cycle, Taken From [You93]

Yourdon points out the problems associated with the waterfall approach where one follows the sequence rigidly:

- It takes too long to see results. Nothing demonstrable and readily understood by the end user is available until code is produced. By then the earlier steps have been completed and much time has gone by.
- It is very dependent on stable, correct requirements. The code quality depends on the quality of the design, and the quality of the design depends on the quality of the analysis effort. Any changes, misinterpretations or misunderstanding of the user requirements can lead to a brilliant solution to the wrong problem.

- It is difficult to trace requirements to code due the sequential nature of the process. The requirements pass through a sequence of different teams of people over a lengthy period of time. This often leads to information being lost, garbled and embellished.

- It delays the detection of errors until the end. Error detection is reserved to the formal testing phase towards the end of the project. By then it is often too late. There is often enormous pressure to put the system into operation. Errors in analysis and design are extremely difficult and expensive to correct. Automated tools can lessen this problem detecting logical errors in the specification and design. However, no tool can spot the error of an analyst fundamentally misunderstanding an aspect of what the user wanted.

Yourdon points out that even though most implementations of the waterfall life cycle presume that one activity must be finished before the next begins, there is no reason to prevent some of the activities from overlapping. He presents a continuum of range of choice. At one end, labelled "conservative" all of activity N is completed before activity N+1 begins. At the other end labelled "radical" all of the activities in the waterfall life cycle would be taking place simultaneously. A project can be developed at any level in between these extremes. One could complete 75 percent of analysis, followed by the completion of 75 percent of design and code to provide a reasonably complete skeleton version of a system whose details could then be refined by a second pass through the entire project life cycle. Or one may finish all of the analysis followed by a much smaller portion of design and implementation. The range is infinite. The decision is normally influenced by the following factors:

- How fickle is the user?
- What pressures are present to produce immediate, tangible results?

- What pressures are present to produce an accurate schedule, budget, and estimate of personnel and other resources?
- What are the dangers of making a major technical blunder?

Yourdon states that these questions do not have clear answers. They are subjective decisions based on the past experience of the project manager. They are based on evaluating the personality and experience level of the users. The politics of the user environment as well as ones own company must be assessed. Projects in the agile manufacturing environment are continually trying new technologies, or new applications of technologies in combinations, or at a scale that has not previously been done. The only information that is often available is the vendor's manuals describing their products used in isolation. It is not known if they will work together, if the throughput promised by one vendor might be destroyed by the system resource use of another vendor's products. The radical approach is best suited to projects with a large research and development component. The conservative approach is better suited to larger projects where large amounts of money are being spent, and for which careful analysis and design are required. Every project requires its own special blend of radical and conservative implementation. To deal with the individual nature of any project, particularly in the ever-present change domain of agility, one must be prepared to modify one's approach midstream if necessary.

DeGrace and Stahl [DeG90], referenced in Yourdon [You93], illustrate in Figure 4-10 an adaptation to the traditional waterfall suggested by Boehm [Boe85]. The project could be managed as a series of increments, resulting in a series of waterfall life cycles,

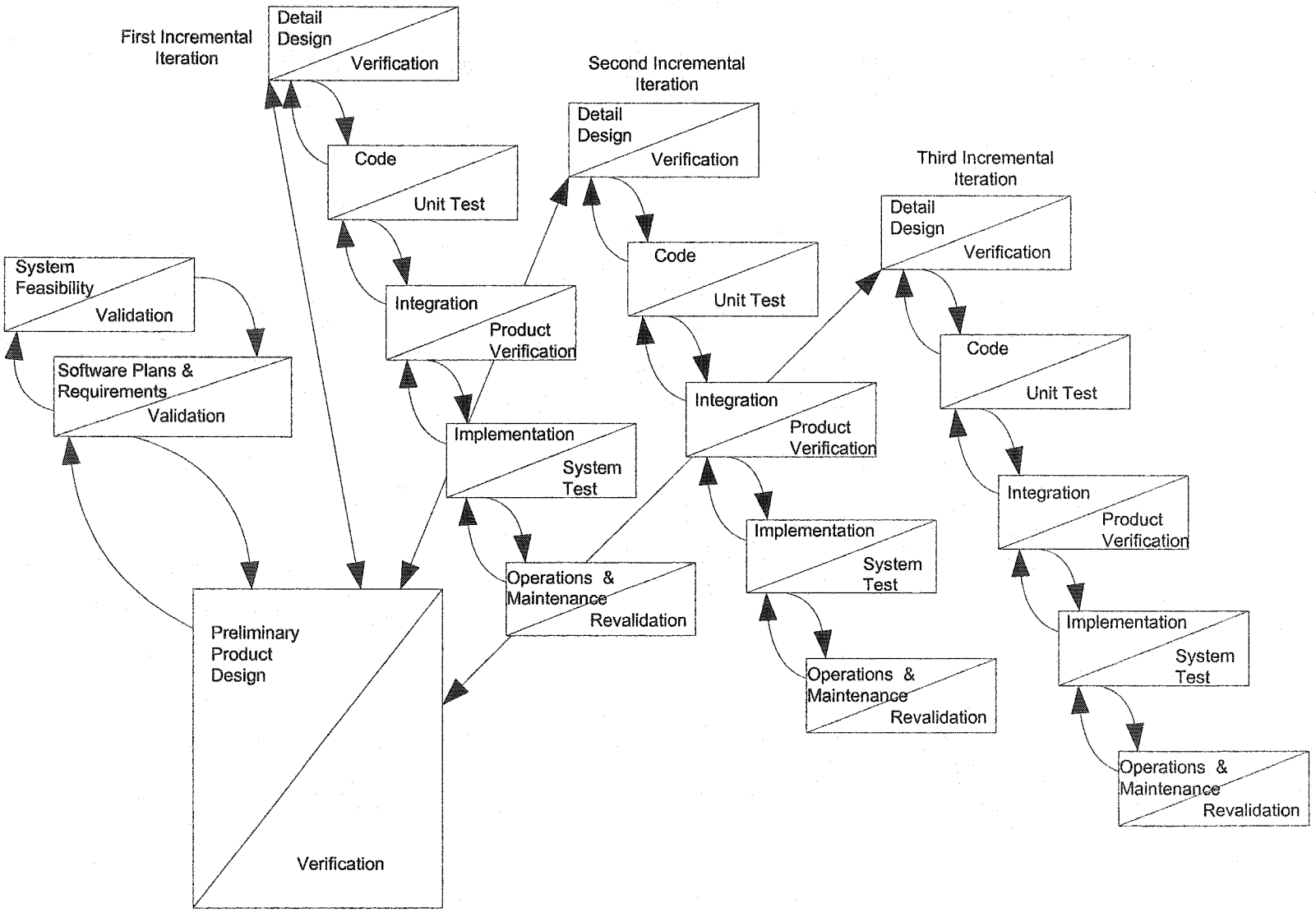


Figure 4-10: The Incremental Life Cycle - Adapted from [You93]

one for each increment. Boehm [Boe85], has suggested that software development could also be considered as a series of spirals illustrated by DeGrace and Stahl [DeG90], in Figure 4-11 referenced by Yourdon. One of the key elements is the use of prototyping. This research effort has a high degree of uncertainty associated with it. The ability to rapidly iterate through the phases should minimize the amount of effort spent discovering approaches, which cannot be used due to some limitation or issue that was not initially apparent.

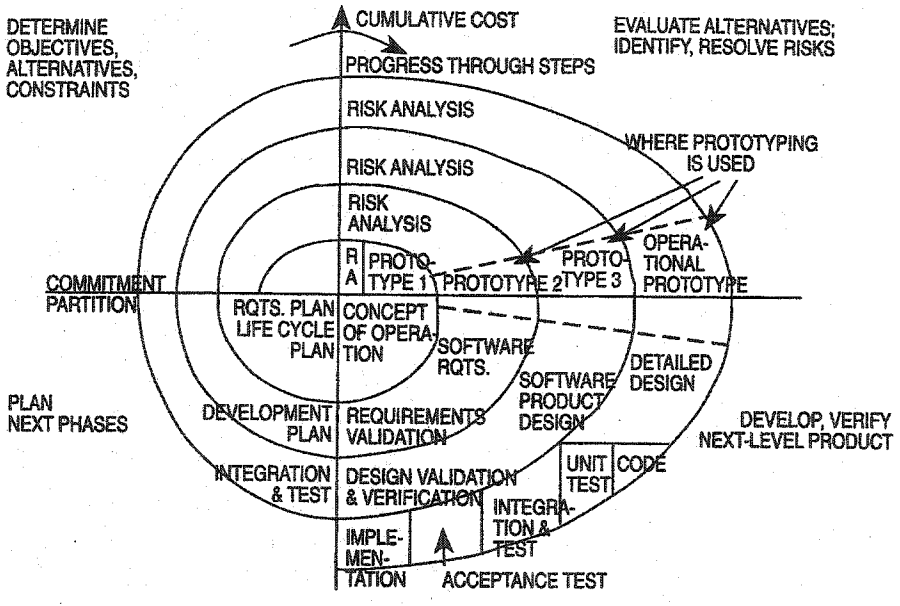


Figure 4-11: The Spiral Life Cycle, Taken From [You93]

Prototyping has been practiced informally since the first computer program was developed. Boar [Boa84] quoted in Yourdon was the first to champion the use of prototyping as a legitimate way of discovering user requirement. His vision of the prototyping life cycle is shown in Figure 4-12. Prototyping has been found to be

particularly effective for exploring alternative human interfaces for a system. It may sometimes be the only way to eliciting the requirements from users who are unsure of the very nature of the complex systems they want built. The diagram shows that some projects are not good candidates for prototyping. An obvious example is the scenario where there is a lack of good prototyping tools. It should be noted that the radical approach to the waterfall life cycle, is really a form of prototyping. A skeleton version of a system can be constructed from a first pass at the top levels of analysis and design. If the skeleton proves to be unacceptable to the user, the analysis and design can be changed and a new skeleton quickly generated to check the validity of the refinement.

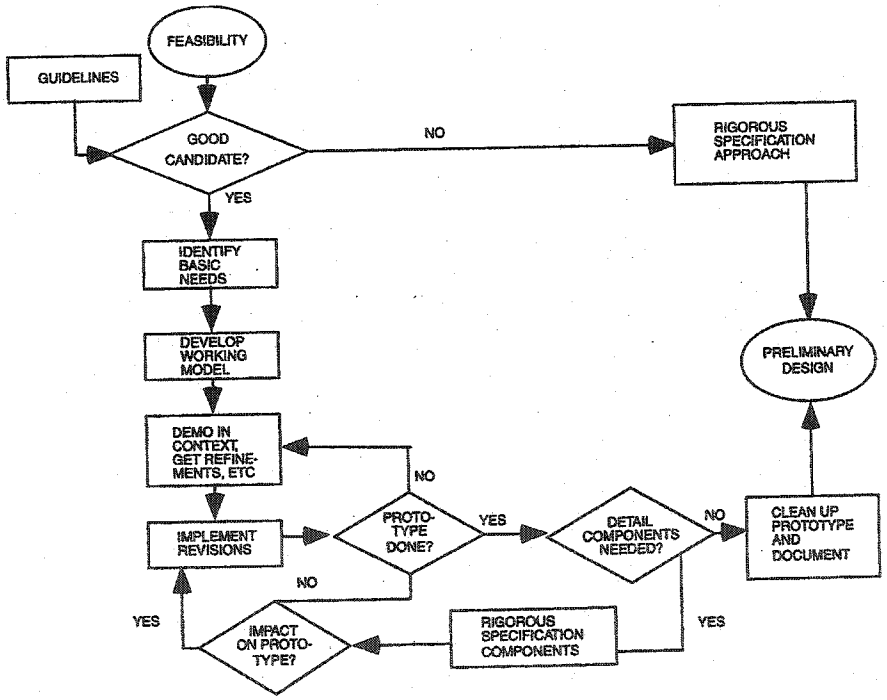


Figure 4-12: The Prototyping Life Cycle, Taken From [You93]

It is not necessary to make a binary choice between a waterfall approach and a prototyping approach. One can combine the best features of both. The trick is to establish an appropriate balance between the two. The key point to remember is that prototyping and waterfall life cycles can be used together. The more techniques that one is aware of, the more options one has to resolve a particular situation. This is analogous to having many different tools in a toolbox that can be mixed and matched according to need. Prototyping is becoming more popular today, mainly because there are powerful tools that support such an approach which were not previously available.

4.7 Addressing Complexity through Object Oriented Philosophy

Booch [Boo94], suggests that the highly complex systems can be approached by studying how complex systems in other disciplines are organized as well as looking into the architecture of even more complex systems found in nature, such as the human circulatory system or the structure of a plant. The structure of a personal computer is based on major elements such as the motherboard, monitor, keyboard, secondary storage, etc. Any of these parts can be further decomposed into simpler elements, which in turn can be decomposed into even more primitive elements. This forms a hierarchic nature in complex systems where the whole functions properly only due to the collective collaboration between its major parts. These levels represent different levels of abstraction built upon each other and each understandable by itself. At each level is a collection of collaborating devices, which provide services to higher layers. The Object Oriented philosophy closely follows the layered abstraction found in nature. This is a

contrast to the structured decomposition approach where the focus is on process and data flows and the decomposition into sub-processes.

Booch [Boo94], draws on the work of Simon [Sim82], to summarize five attributes of complex systems:

1) Frequently, complexity takes the form of a hierarchy [7][1], whereby a complex system is composed of interrelated subsystems that have in turn their own subsystems, and so on, until some lowest level of elementary components is reached. This hierarchy is the major facilitating factor which allows one to understand and describe complex systems. The architecture is a function of the components as well as the hierarchical relationships among them. The value added by a system comes from the relationship between the parts, not from the parts per se.

2) The choice of what components in a system are primitive is relatively arbitrary and is largely up to the discretion of the observer of the system. What is primitive for one observer may be at a much higher level of abstraction for another.

3) Intra-component linkages are generally stronger than inter-component linkages. This fact has the effect of separating the high-frequency dynamics of the components - involving the internal structure of the components - from the low frequency dynamics - involving interaction among components [8][2]. This provides a clear separation of concerns among the various system parts making it possible to study each part in relative isolation.

4) Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements [11][3]. Ideally, many of the complex components are implemented from common subcomponents providing an economy of expression. Complex systems have common patterns.

---

[1] [7] Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM* vol. 28(6), p. 596

[2] [8] Simon, H. 1982 *The Sciences of the Artificial*, Cambridge, MA: The MIT Press, p.217.

[3] [11] Ibid., p. 221.

5) A complex system that works is invariably found to have evolved from a simpler system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system [13][4]. Objects that were initially considered complex become primitive as systems evolve. These objects cannot be crafted correctly the first time. They must be used within their context first and then improved as one learns more about the real system behaviour.

Booch points out that that there are two orthogonal hierarchies in systems. The first is characterized by collection of parts that make up components and can be describes as being "part of" such as an airplanes propulsion system or flight control system which are referred to as the objects. The second is characterized by grouping objects which have common properties such as a turbofan is a specific kind of jet engine using the phrase "is a" referred to as the classes. Combining these two orthogonal hierarchies and the five attributes of a complex system, virtually all complex systems take on the same form. Booch claims that very rarely are complex system successful, delivered on time, within budget, meet their requirements, unless they are designed with the five attributes and have well engineered classes and objects.

Given the knowledge of how to design complex systems, there still remains the question of why serious problem are often encountered in successfully developing them. Initially, when a complex software system is first analyzed, many parts interacting in a multitude of intricate ways are found. There appears to be little noticeable commonality among either the parts or their interactions, which Booch refers to as disorganized

---

[4] [13] Gall, J. 1986. *Systemantics: How Systems Really Work and How They Fail*, Second Edition. Ann Arbor, MI: The Geberal Systemantics Press, p.65.

complexity. The task is to bring organization through the process of design. To accomplish this one needs to think of many intricate things at once. Unfortunately, it is impossible for a single person to keep track of such detail. Experiments by psychologist Miller [Mil56, Shi94, Bda94], shows that the human mind can only comprehend a maximum of seven, plus or minus two, simultaneous pieces of information. Simon [Sim82], adds a further speed-limiting factor of the mind requiring about five seconds to accept a new piece of information. The technique for addressing this problem has been known since ancient times. It is simply the divide and rule approach which translates into taking a complex system and decomposing it into smaller parts, each of which can be refined independently. Thus, to understand any given level, one needs to comprehend only a few parts. There are two distinct approaches to decomposition: algorithmic and object-oriented. Algorithmic decomposition takes the algorithm and separates it into distinct major steps, which in turn are decomposed into smaller processing steps. The focus is on the process or function. This is the basis of the traditional structured approach. Object-oriented decomposition on the other hand, separates the system according to key abstractions in the problem domain. Rather then decomposing into steps such as "get formatted update" or "add check sum", one identifies objects such as "master file" or "check sum" which are derived from the vocabulary of the problem domain. The world is viewed as a set of autonomous agents that collaborate to perform some higher-level behaviour. The "Add Check Sum" step is replaced with a "Check Sum" object which contains the needed data as well as all of the operations associated with the. Similarly "Get Formatted Update" no longer exists as an independent algorithm

but is contained in the object "File of Updates" which contains all other operations associated with the data they operate on. Calling this operation creates another object "update to card". Each object contains its own unique behaviour and models some object in the real world. An object is asked to perform what it does by sending it a message

The natural question that comes to mind is which approach is better. Martin and Odell [Mar92], argue strongly that the OO approach is more natural for people. They state that the world consists of objects and that the natural development of a person starts with identifying the existence of certain concrete objects in the immediate environment characterized by certain types of behaviour. We tend to categorize objects as we discover their behaviour.

Meyer [Mey88], presents a strong argument, viewing the OO approach as structure based on data contrasted with the traditional approach centred on function, stating that OO provides better extendibility and continuity. He defines continuity as the ability of a design method to yield architectures that will not need to be changed abruptly for small changes in system requirements. He points out that in the evolution of complex systems, the functions tend to be far more volatile than the data that they manipulate, at least if viewed from a sufficient level of abstraction. A convincing example presented is one of a payroll system where the functions it performs can be readily seen to expand and change, yet the foundation sorts of data that it manipulates will always be more or less the same. Martin and Odell [Mar92], further state that the function based approach is one where data can assume any structure and processes can do anything to the data that the

programmer desires. They state that the functional approach is an attempt to design and debug by thinking through the order in which the computer does things leading to systems that are difficult to understand. Booch states that both views are important. The algorithmic approach highlights the ordering of events while the OO view emphasizes the agents that either cause action or are the subjects upon which operations act. He states that the approaches are completely orthogonal and cannot be pursued simultaneously.

One needs to start decomposing using one perspective, which results in a structure, which serves as the framework for expressing the other perspective. Booch states that OO has: the advantage of yielding smaller systems through the reuse of common mechanisms; better resilience to change because the design is based on stable intermediate forms; reduced risk because the design evolves incrementally from smaller systems which already work.

4.8 Addressing Complexity through Abstraction

Abstraction is one of the fundamental ways that OO deals with complexity. Booch [Boo94], combined several different viewpoints to define abstraction. Dahl, *et al.* [Dah72], suggest that abstraction arises from recognition of similarities between certain objects, situations, or processes in the real world. A decision is made to concentrate upon these similarities and to ignore the differences for the time being. Shaw [Sha84], defines an abstraction as a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses

details that are, at least for the moment, immaterial or distracting. Berzins, *et al.* [Ber86], recommend that a concept qualifies as an abstraction only if it can be described, understood, and analyzed independently of the mechanism that will eventually be used to implement it. Booch combines these stating that an abstraction represents the essential characteristics of an object that distinguish it from all other kinds of objects. It provides a crisply defined conceptual boundary relative to the perspective of the viewer. The object's essential behaviour is separated from its implementation with the focus on the outside view. One of the main activities of object-oriented design is deciding on the right set of abstractions for a given domain. It also proves to be one of the most difficult. Figure 4-13 and Figure 4-14 illustrate these points.

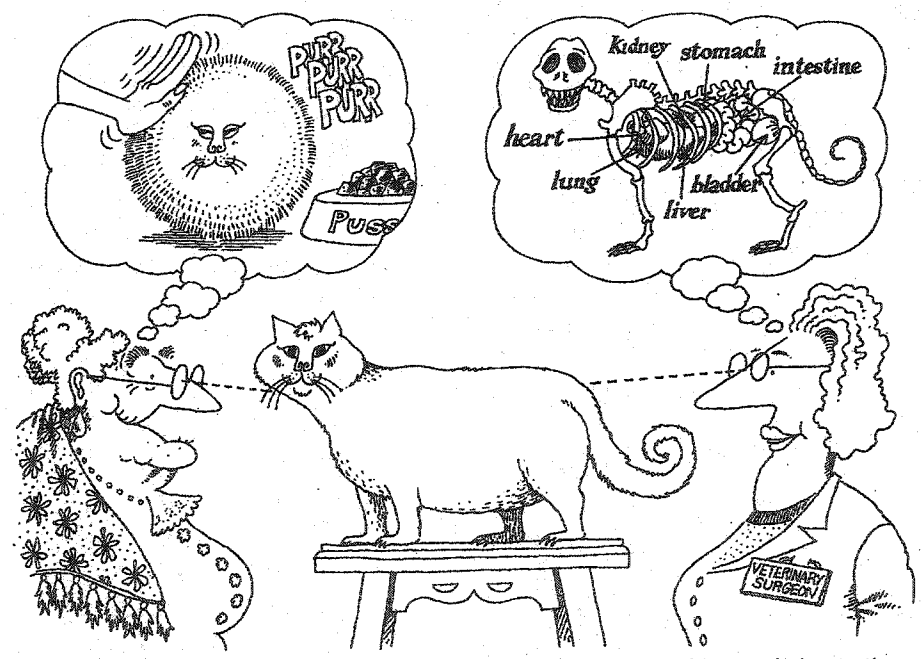
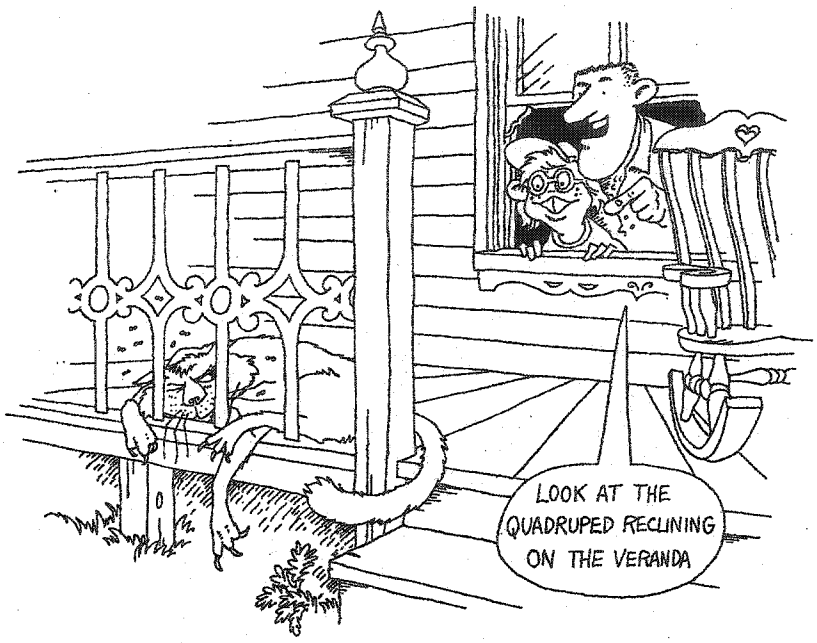


Figure 4-13: Abstraction of Viewer, Taken From [Boo94]



Classes and objects should be at the right level of abstraction: neither too high nor too low.

Figure 4-14: Right Level of Abstraction of Classes and Objects, Taken From [Boo94]

Booch refers to Siedewitz and Stark [Sei86], who suggest that there is a spectrum of abstraction, from objects which closely model problem domain entities to objects which really have no reason for existence. From the most to the least useful, these kinds of abstractions are:

- Entity Abstraction	An object that represents a useful model of a problem-domain or solution domain entity.
- Action Abstraction	An object that provides a generalized set of operations, all of which perform the same kind of function.
- Virtual Machine Abstraction	An object that groups together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations.

- Coincidental Abstraction — An object that packages a set of operations that have no relation to each other.

Entity abstractions are the best ones to use because they exist in the problem domain and are thus readily recognized and understood by the users and domain experts. Martin and Odell [Mar92], state that when we analyze systems, we create models of the application area of interest. The model represents an aspect of reality and is built in such a way that it helps us to understand reality. It is much simpler than reality. We can manipulate the model to help us design the system. Analysts, designers, programmers, and, particularly, end users all use the same conceptual model. They all think of object types, objects and how objects behave. They draw hierarchies of object types or classes in which the subtypes share the properties of their parent. They think about objects being composed of other objects and use generalization and encapsulation. They think about events changing the states of objects and triggering certain operations. The transition from analysis to design is natural. Specifying where analysis ends and design begins is sometimes difficult. Martin and Odell argue that this is superior to the traditional methods where the conceptual models used for analysis differ from those used for design with programming using a third view. When traditional development crosses the walls shown in Figure 4-15, information is often lost and misunderstandings occur. They argue that the OO analysis reflects reality more naturally than the models in traditional systems analysis. Using OO techniques, software more closely models that real world. The analyst creates a model of the area of interest. The model is converted into a design and then code. The model should represent how the end users perceive the area or what the users want the system to do. As far as possible, this model should be in a form that users

can understand and helps them articulate their needs. The implementer uses the model and creates a design from which code can be written or generated. Figure 4-16 illustrates how to build such systems.

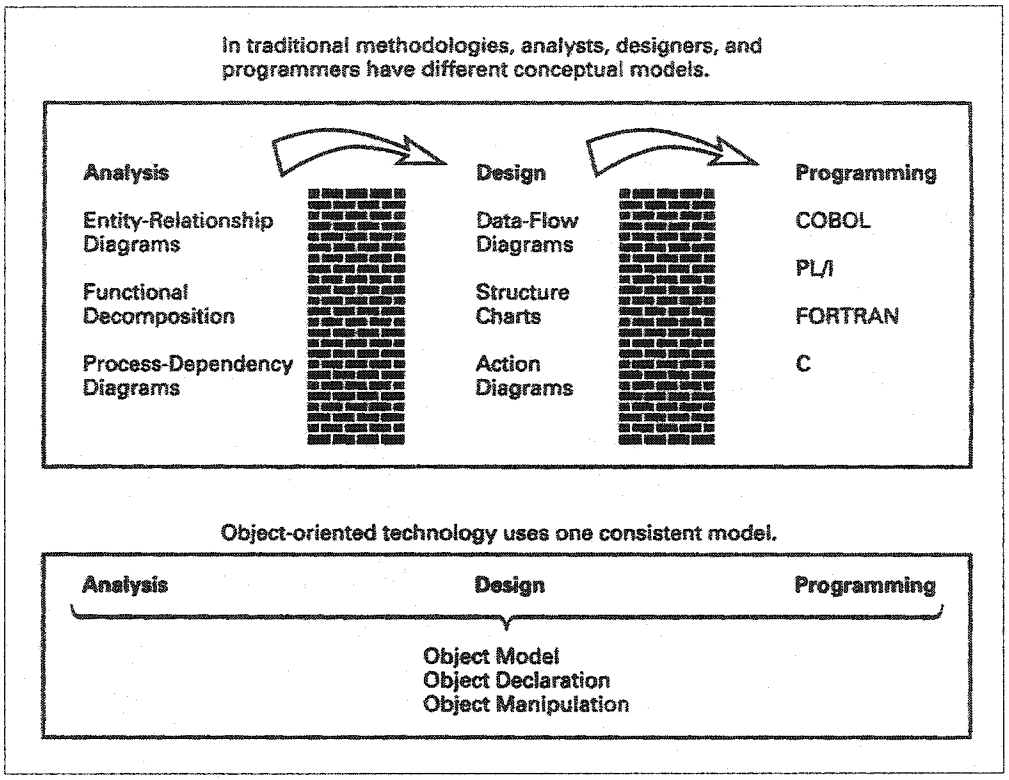


Figure 4-15: Traditional Methodologies vs. Object-Oriented, Taken From Martin and Odell [Mar92]

Central to the abstraction of OO is the concept of a client server relationship. A client is any object that uses the services of another object, referred to as the server, to achieve some goal. Wirfs-Brock, *et al.* [Wir90], describe the object-oriented design method as one, which seeks to model the world in terms of objects collaborating to discharge their responsibilities. The objects have very well defined responsibilities that they are capable of performing based on information they contain within them. The

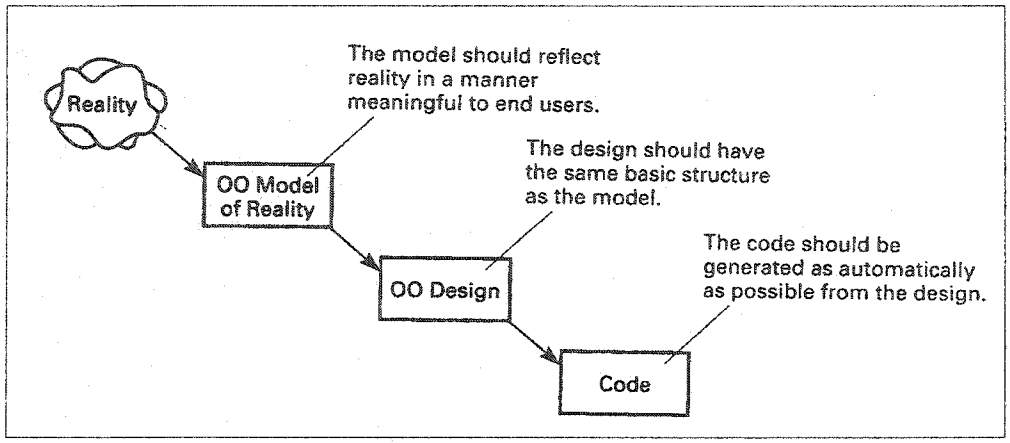


Figure 4-16: Object-Oriented Project Cycle, Taken From Martin and Odell [Mar92]

collaborations can be viewed as one-way interactions where one object requests a specific service of another. The particular ways in which a given client can interact with a given server are described by a contract. A contract is the list of requests that a client can make of a server. Both must fulfill the contract. The client may only request those services that the contract specifies, and the server is responsible to correctly and appropriately respond to those requests. This contract thus establishes all the assumptions a client object may make about the behaviour of the server object. In other words, this contract encompasses the responsibilities of an object, namely the behaviour for which it is held accountable.

4.9 Utilizing Encapsulation to Achieve Information Hiding

Encapsulation is the complimentary counterpart of abstraction. Abstraction focuses on the externally observable behaviour of an object as presented by Booch [Boo94]. Wirfs-Brock, *et al.* [Wir90], state that publicly an object reveals its abilities

stating what services it can do and what information it can provide. However it does not tell how it knows the information or how it does the services. The other objects should not be concerned with that. Encapsulation focuses on the internal or private portion of an object. It deals with the implementation that gives rise to the external behaviour. Encapsulation is most often achieved through information hiding, which is the process of hiding all of the secrets of an object that do not contribute to its essential characteristics. Typically, the structure of an object is hidden, as well as the implementation of its operations. Objects know only what operations they can request other objects to perform. An object requesting an operation from another object or some information acts like a good manager. It specifies the job or asks for the information and then leaves. It doesn't hang around worrying about how the job is done nor how the information is calculated. Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns. Encapsulation through information hiding isolates one part of the system from other parts. The internal representation can thus be changed or replaced with a superior algorithm without changing the object's abstract, public interface. Other objects that count on this objects operation results or certain values are not affected by the change. Code can be modified and extended, bugs can be fixed, without the risk of introducing unnecessary and unintended side effects. Ingalls [Ing78] as quoted by Booch states that no part of a complex system should depend on the internal details of any other part. Gannon, *et al.* [Gan87], quoted by Booch, add that abstraction helps people to think about what they are doing while encapsulation reduces the effort needed to reliably make program changes. Wirfs-Brock, *et al.* [Wir90], explain that

objects are how the principles of abstraction and encapsulation are put into practical use. First the functionality and information that are related, that belong together, are abstracted out. Then they are encapsulated in one object. Then a decision is made concerning what functionality and information other objects will require of that object. The rest is hidden. A public interface is designed that allows other objects to access what they require. The private representation is by default protected from access by other objects.

4.10 Addressing Comprehension through Modularity

Myers [Mye78], as quoted by Booch [Boo94], observes that partitioning a program into individual components can reduce its complexity to some degree. Although partitioning a program is helpful for this reason, a more valuable result is the creation of a number of well-defined, documented boundaries. These boundaries, or interfaces, are invaluable in the comprehension of the program. Liskov [Lis80] uses the definiton of Britton and Parnas [Bri81] where the connections between modules are the assumptions which the modules make about each other. Modules serve as the physical containers in which the classes and objects of the design are declared. In traditional structured design, modularization is primarily concerned with the meaningful grouping of subprograms using the criteria of coupling and cohesion. In OO design, the task is to decide where to physically package the classes and objects based on the design's logical structure. As Britton and Parnas [Bri81], observed, the overall goal of the decomposition into modules is the reduction of software cost by allowing the modules to be designed and revised independently. Each module's structure should be simple enough that it can be

understood fully. It should possible to change the implementation of a module without needing knowledge of the implementation of other modules or affecting their behaviour. It should be easy to making a change in an area of a design where change is likely to be needed. The cost of recompiling the body of a module is relatively small, since only that unit need be recompiled and the application relinked. However, the cost of recompiling the interface of a module is relatively high since all other module that depend on the interface must also be recompiled, and the modules that depend on them, and so on. A balance needs to be reached between the competing desire to encapsulate abstractions, and the need to make certain abstractions visible to other modules. Parnas, *et al.* [Par83], as quoted by Booch, offer the following guidance:

> System details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear between modules are those that are considered unlikely to change. Every data structure is private to one module; it may be directly accessed by one or more program within the module but not by programs outside the module. Any other program that requires information stored in a module's data structure must obtain it by calling module programs

This conforms to the traditional guidelines of building modules that are cohesive and loosely coupled. From this perspective Booch defines modularity as the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. The principles of abstraction, encapsulation, and modularity are stated to act synergistically where an object provides a crisp boundary around a single abstraction. Encapsulation and modularity both provide barriers around this abstraction. Modules also serve as the elementary and indivisible units of software that can be reused across applications. There is a desire to package classes and objects into modules to make their

reuse convenient. Meyer [Mey88] offers five criteria to help designs with respect to modularity: decomposability; composibility; understandibility; continuity; and protection. Modular decomposability is achieved by partitioning a problem into several sub-problems, whose solution may be then pursued separately. Software elements that may be freely combined with each other to produce new systems, possibly in an environment quite different from the one they were initially developed for have modular composability. The aim is to find ways to design pieces of software performing well-defined tasks usable in widely different contexts. Modular understandability aims at enabling a human reader to comprehend a module in isolation from other modules. At worst, the reader should need to look at just a few neighbouring modules. This criterion is key to the maintenance problem. Most maintenance activities involve having to dig into many existing software elements. Ideally, full comprehension of a module can be obtained from only its documentation instead of needing to first read and understand the documentation of several modules. This reduces the chance of introducing new problems and increases the potential of correcting existing ones. Modular continuity implies that small changes in a problem specification result in a change of just one or at most few modules in the system. Such changes should not affect the architecture of the system, that is to say the relations between modules. Modular protection is characterized by architectures in which the effect of an abnormal condition occurring at run-time in a module will remain confined to this module, or at least propagate to only a few neighbouring modules only.

4.11 Summary

This chapter reviewed tools from the disciplines of real-time systems control, computer science, electrical engineering, artificial intelligence and technology insertion that were applied to address issues arising from development of Agile Manufacturing Information systems. Modeling of real-time systems was selected for synchronizing monitoring and control codes. Configuring reusable components into a control plan and the lack of context in the collected data was addressed by applying Artificial Intelligence within an application generator. Object Oriented methodology with finite state machines was chosen as the scientific methodology for control code development. Technology insertion techniques and experiments were employed to transition this modern methodology from the theoretical domain to practice in industry. The next chapter details the development of the abstract model.

Chapter 5

Abstract Model Design and Development

5.1 General Approach

Generalization from a representative specific case was utilized to develop the abstract model. The strategy was to supplement an academic perspective with practical industrial knowledge. A practicing manufacturing-control engineer identified real world constraints, common practices, problems and provided a sense of what would be readily acceptable. The control engineer's knowledge provided guidance on these issues, which is not readily available from academic literature. This knowledge served as a guide to what the industrial community would likely accept and how to present new methods to maximize the probability of successful technology transfer. For the first pass a single minimal complexity station of a typical machine was selected. The primary objective was to develop a model for state based programming and the translation from it to PLC Ladder Logic. The secondary objective was to evaluate the appropriateness of using state machines and object oriented Unified Modeling Language (UML) [Eri98, Dou00, Qua98], which is commonly used by the software industry, for modelling of code targeted at PLC's. The strategy was to initially work with the industrial control engineer on a subset of the new method with a relatively low level of complexity and a high

probability of success. The engineer would need to expend a minimum amount of resources to appraise the value to the approach. Given a positive appraisal, more of the engineer's time would be available for more exhaustive research. Thus, with feedback from the RLM engineer, state based PLC control code was first manually designed and written for the single station. From this specific case, a generalized UML model was developed. Once state based programming was refined to a point acceptable for industry, a more complex complete system representative of products that RLM manufactured, was studied. This specific representative design was then generalized into a more comprehensive UML class model. The class model then served as a "template" or guide for the design and coding of control code for similar machines. It should be noted that the primary reason for introducing the new methodology for writing control code was to address the synchronization issue of monitoring and control systems. A minimal level successful solution would be one where the new methodology for control code generation was of equal quality and was readily acceptable by the control engineers. A more desirable solution would be a methodology with a marked improvement, serving as an impetus for change.

An important technology transfer issue is the amount of change one can introduce at a time. Ideally, one would leave a noticeable net gain after the new technology is comprehended and adopted with minimum disruption to the target organization. It has been noted [Poi98] that productivity will initially drop when a new methodology is introduced into a group to apply to a task that they are proficient at executing. It will remain at a lower level while the methodology is being adapted to the application and

assimilated by the organization. Eventually, the productivity will increase to the original level and ideally surpass it. Thus, minimizing the learning effort was a major criterion when selecting a strategy for transforming the abstract model into code. Leduc and Wonham [Led95b] had used Ladder Logic but they encoded the state using the minimum number of coils. In addition, Karnaugh mapping techniques [Mot72] are applied to reduce the encoding logic. When the number of states becomes too large to handle with Karnaugh mapping, computer aided design programs such as ABLE are employed. While this does result in an economy of logic elements used, it requires skills and additional tools beyond what is required in the current programming paradigm. Ideally, for technology insertion, the users should be presented with the method that they are familiar with and the equivalent new method [Sto03a]. The sequence of steps of a machine will map to states in a state machine or places in a Petri net, which in turn will map into PLC code. It is desirable to minimize the amount of data and PLC code that needs to be studied to determine what the controller is currently doing. The method used in VLSI design, that directly identifies the current state, is called "One Hot" [Gol93] adapted from "one relay per state" logic [Huf54a, Huf54b]. This approach was used encoding each state with a single relay.

One of the advantages of Object Oriented (OO) Development is the use of the same objects throughout the entire development process. Concrete objects that are familiar to the users are identified during the requirements analysis phase. A representative specific case is selected which the user is familiar with. A generalization process is applied to derive an abstract class model that can be applied to other control

applications within the same domain. Here the kinds of objects are identified and taxonomy [Boo94] is created. The abstract general model provides a means to constrain future designs to a limited set of classes. Such a constraint is crucial for the synchronization of monitoring and control systems. Without it, there is nothing to prevent the continual proliferation of different control implementations that achieve the same result that the monitoring system would need to comprehend. When these classes are used as templates or "instantiated" for a specific control application, they are called objects. These objects must be readily recognizable in the control code. When the control code is constrained to the relatively small standard set of classes, one can learn to easily recognize their instantiation in the PLC code. One no longer needs to comprehend the individual style of each control code engineer and its evolution through time.

This need for constraining control code generation was a conclusion reached by the author after many years of developing Operator Performance Support Systems in the Oshawa plant. It was observed that even when a guideline for writing PLC monitoring code was provided [Sto86], engineers continually modified it to suit their preferences. Every engineer had a different adaptation. There were as many different ways as there were engineers. The result was data that needed to be interpreted in different ways before someone other than the engineer who created the data collection logic could use it. Since the performance support system was used by a large number of people from different departments with different needs, a common interpretation was mandatory. The same situation held true for the control code. In the Trim Shop, empirically it was shown that the performance support system [Hod96] was far superior to the Body Shop. In the Trim

Shop, the data were collected from the Automatic Guided Vehicle (AGV) [Dun94] conveyance system. AGV systems in Trim shops, where most of the workstations use people rather than automation, are very complex. The majority of AGV system installations at GM assembly plants failed and were replaced by conventional chain conveyors. The only large-scale AGV systems that remain are in the Oshawa facility. Much of the success in the Oshawa Car plant AGV system is attributed to the standardized method of writing PLC control and monitoring code. No variation is allowed. There are standard blocks of control code for the various types of situations. In today's state of the art performance support systems, monitoring code in the PLC is written separately from the control code. This is the practice in the Car Plant Body Shop. The Monitoring code attempts to capture the pertinent information for the Performance Support analysis programs to use independently from the control code. Unfortunately, the initial implementation is typically not available in time for the start up of new systems and the divergence cycle [Lyp95] is always present. In the Car Plant Trim Shop, however, the Performance Support data are captured directly from the control code and its data. The only code that is specifically used for monitoring consists of time stamping, queuing and sending of events identified by the control systems. These events are needed by control system specifically for control. Without them, the AGV system will not function. Thus, one of the major problems is addressed. Resources are guaranteed to be available to generate the control code since cars cannot be built without them. Monitoring is seen as less important since it affects productivity but does not prevent the building of cars. Resources are considerably more difficult to obtain to implement code

that is used only for monitoring. In the Trim Shop AGV control system, the PLC logic and the associated data structure for each type of situation are always the same. To implement the monitoring data collection, one only needs to identify the control type and configure the data collection logic appropriately. This approach has been proven successful over a period of 10 years. Based on this observation, the selected approach was one of constraining the control code to a small set of predefined configurations that address all of the situations for which monitoring data needs to be collected. Whenever a new situation arises for which a configuration does not exist, the set is extended by the controls expert of the organization and then used by all of the control engineers.

The need to implement control code in PLC Ladder Logic was an important conclusion reached from the experience of inserting AGV technology into the Oshawa Car Plant in which the author was involved. The experience gained is fundamental to guiding the research reported herein. This is the appropriate technology for the maintenance and engineering skills that are typically found in such plants. Other installations deployed computers that required a new support team with skills not typically found in plants. The members of this new team were not part of the existing plant floor operation. The information pertaining to the control algorithm while the system was running on a computer was far inferior to that provided by a PLC. The result was breakdowns typically measured in the tens of minutes with complete rebooting of the control computer occasionally required. The cost of lost production when the main conveyance stops, shutting down the majority of the operations, is about 5 to 10 thousand dollars per minute. The breakdown recovery is much superior in the Trim Shop PLC

based systems. This has been attributed to the fact that the typical AGV electrician can correct the majority of the problems by manipulating the control program's data. The PLC provides a live display of all of the control code and its data as it is executing and provides the mechanism for on line, live modification. This is not available with compiled computer control programs. A development environment would be required with a debugger to display the values of variables and trace functions to determine flow of execution. A cycle consisting of edit, compile link, shut down of executing program followed by a load is needed to make a program modification. The only visibility that is available is from predefined status displays. Changes can only be done to data in predefined ways. The original designers provided such functionality and information only for the situations that they were aware of during the initial design. The support team responsible for the computers in the plant are usually just operators without the analysis and design skill of computer analysts or designers. To make a change to the code requires locating such a person, and then recompilation, shutdown of the whole system for loading the new code followed by a restart. This is a complex procedure since there is typically loss of a large amount of system state information that needs to be manually determined and then re-entered. The computer operators are not familiar with the plant floor. The floor maintenance electricians are in constant contact with the day-to-day operation of the system. Thus their problem resolution ability is far superior to that of a computer operator.

From the AGV technology insertion, it is evident that PLC Ladder Logic needs to be used for the initial programming language. The objective is to move the control

engineers and the maintenance electricians to a higher level of abstraction. As indicated in the Portfolio Project proposal [Poe95], one needs to provide both the current method and the new method side by side. This provides the targeted user with a familiar method to refer to as they become accustomed to the new one. State diagrams were selected to visually represent the new method. They require a modest amount of effort to comprehend. To compose the program from reusable components, an application generator was developed. The Unified Modelling Language (UML) was selected to provide the maximum flexibility for future migration to other languages. An application generator program was written based on the UML class model and state machines, of which a partial example is given in sections 6.3.3, 6.4, Appendix B and Appendix C. The user interacts with the application generator to construct the specific object model based on the abstract class model. The class model serves as a common framework or template that is present in every control program or application generated. Since this framework is common, much of it does not need to be specified by the user. The user first identifies which components are needed. Then only the inputs, outputs and sequence of operation need to be specified by the user before the code for the framework can be generated.

5.2 Object Oriented Abstract Model Strategy

The application generator first constructs an object model as an intermediate step. A translation is then applied to generate the PLC code. This separation provides the ability to select other languages as targets in the future as well as different analysis techniques to construct the object model. As the user becomes familiar and comfortable

with the object model by examining the generated PLC code, a step in the technology insertion process becomes complete. A different programming language, at a higher level of abstraction, closer to working at the model level can now be introduced. Sequential Function Charts based on Petri Nets or a language such as Siemens' "HiGraph", based on finite state machines may be considered. Even before the user is ready for a migration to a more abstract language, the mapping to such may be used to take advantage of analysis techniques. By mapping to a marked timed Petri net, the analysis techniques developed by Park, *et al.* [Par99c, Par99d, Par00], may be applied, and by mapping to hierarchical state machines the methods of Leduc, *et al.* [Led00], and Abdelwahed and Wonham [Abd99] can be used. Eventually, once the user migrates to operating at the model level, distributed computer based approaches [Wan99, Wan00, Sto01, Sha02] may be practical. To replace a PLC, however, will require providing the same level of visibility of the running program and ability to make live, on-line changes. In addition, there has recently been focused research in model-based analysis in the MOBIES [AFR02] project that can be utilized when one starts with a UML model.

Object oriented models have been used to describe software-based systems for quite some time. Libraries of reusable classes have been created but as pointed out by Birla [Bir98], configuring them still requires too much effort. The focus of Birla's work [Bir96] was reconfiguration and reuse of control code in support of reconfigurable machine tools. The focus of this body of research is adapting a methodology and developing a technology insertion process to address the divergence between control and monitoring systems. Specifically, automatically generating control code that will allow

automatic generation of context enhanced data. Though the specific goal is different, the underlying strategy is similar. A library of reusable components is constructed based on a particular domain specific model and these components need to be configured into a "control plan" to address a specific control application. Birla relied on the programmer becoming familiar with the components and how they were to be configured. This thesis takes a different approach. The observation of over a decade of control code generation in the Oshawa plant led to the conclusion that there are as many ways to create components, as there are programmers. To allow reuse by others, one needs to capture the knowledge of the specific expert who created the components and the particular model that they support. Within the model, the expert partitioned functionality in a particular way. This partitioning resulted in specific components implementing a specific set of these functions. A different expert would likely come up with a different partitioning resulting in a different model. The functions that need to be performed have not changed, but the model and the reusable components have. At the start of this thesis, it was hypothesised that to achieve synchronization of monitoring and control, one needs to capture the knowledge of the controls expert in the form of an application generator. This application generator would guide the user in selecting and configuring components to create a control program and the needed monitoring data acquisition code. An abstract model would be created by the generator and used by the monitoring system to comprehend the collected data. The particular state of the various state machines would be collected from the control code data by the data acquisition code. This would automatically provide the context missing from current monitoring systems. After

constructing the class model for a representative system, it became evident that it would handle most if not all of the systems that RLM manufactures. When the classes were designed by generalizing from the specific representative system, all of the functionality that the expert was aware of was implemented. The specific model created with the expert's feedback and the partitioning of the identified functionality into particular components was captured in the application generator. The process was one of mapping functionality to a model. So instead of having users select and configure components into a model, the application generator would have them specify the required functionality and specific behaviour. The user is constrained to specifying functionality in an easily recognizable manner. One can then map it to the model and automatically select and configure the library of standard components. One can use the same transformation that the author developed with feedback from the expert, to generate PLC Ladder Logic for these components, or apply different standard transformations for other target languages.

5.3 Single Station Experiment

5.3.1 Objectives and Approach

The starting point for developing the model was collecting the best practices observed at the Oshawa Car Plant. Since the plant was no longer available for research purposes, the control engineer expert at RLM provided an experimental environment. Both the author and the RLM Engineer had each spent more than a decade working with the control and monitoring systems in the Oshawa plant. The approach used was to start with a simple single station machine, model its operation as a state machine and develop a standard representation in Ladder Logic. The machine selected was a flying cut-off die accelerator shown in Figure 5-1. The objective was to determine how to migrate the RLM engineer from a traditional sensor based approach to a state-based design.

The sensor-based approach consists of writing logic equations that identify what the machine is doing based solely on the state of its sensors, operator selector switches, push buttons and actuators. Based on this identification, appropriate actuators are energized. Since most machines operate in a cyclical manner, there is a sequence of operations or steps that the machine repeats. This is equivalent to a state machine, however, traditionally, the code is not implemented using an explicit state machine. Somewhere, the information identifying the current step in the cycle must be stored. Upon closer examination, it becomes evident that the mechanical position of the machine and the state of the sensors and actuators indirectly imply the state. As the machine complexity increases, it becomes more difficult to determine the correct state solely from

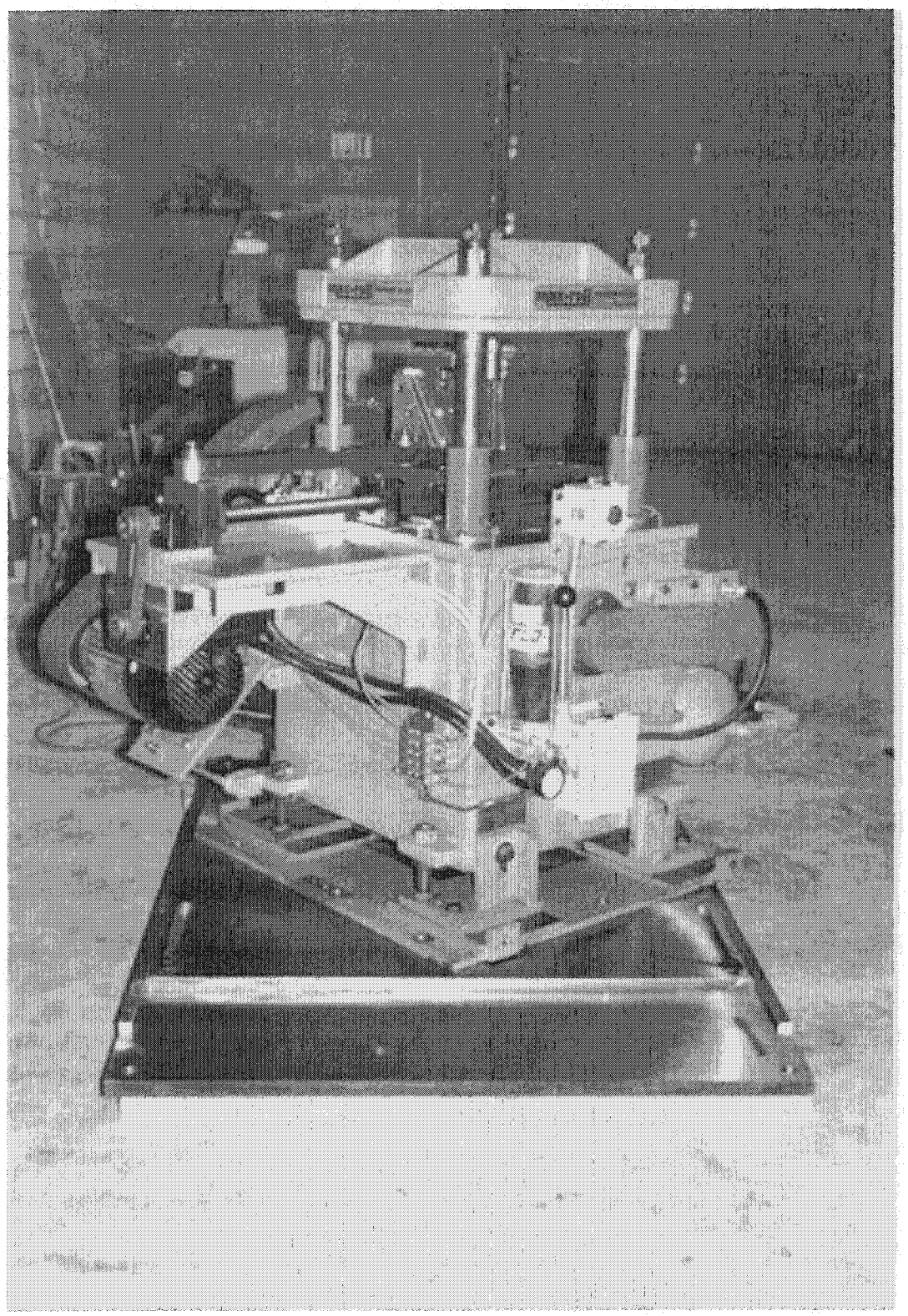


Figure 5-1: Flying Cut-Off Die Accelerator - Courtesy RLM Manufacturing

this information. For example, consider a transfer machine that lifts a part to a safe transfer height, moves across an aisle, descends to a conveyor and deposits the part. It then ascends to the transfer height again and travels back over the aisle. While the actuator that causes the motion is energized, one can determine which direction the transfer is travelling over the aisle. Should the transfer stop over the aisle, however, the actuator is no longer energized and there is insufficient information to determine which direction the transfer was moving. To solve this problem, control engineers will typically latch the direction of travel. Thus some of the state information is stored in the program. There is no particular method to determine where to look to find all of the information needed to identify which actuators the controller should be energizing at a particular time. To further complicate the situation, there are dozens of moving parts, sensors and actuators in a typical automotive plant machine. Many of these are not readily visible. To determine what the control is currently doing, one needs to examine many separate items, scattered throughout the code, have a clear view of the physical inputs and possess a high degree of familiarity with the machine. Since this information is not explicitly identified in the code, it requires a high degree of machine specific knowledge.

A state based approach makes this information explicit. Each step in the sequence of the machine is represented by a unique state. The sequence is represented by states connected with transitions. The transitions identify the switches and sensors that are relevant. The actuators that are to be energized for a specific state can be easily determined by noting the presence of the state in the actuator logic. A consistent

grouping of the Ladder Logic for each of these sections provides an organization for quickly locating all of the information that one needs to fully determine the behaviour of the controller at the time of observation.

In addition, the state based approach provides a simple automatic method for organizing and retrieving documentation pertaining to the controlled machine. Locating the relevant support information is a common problem when troubleshooting a machine fault. The documentation in printed form, including the code, is typically several inches thick. The nested state machines form a framework for organizing this documentation. The monitoring system can readily identify the current sequence step from the control sate machine. The documentation can be electronically stored in the framework based on the sequence step so it can be automatically retrieved. With the video capture technology that is readily available today, one can easily film the motion of the machine corresponding to the current sequence step. One can point out the locations of the sensors and switches that recognize the event and conditions needed to make the transition to the next step. The actuators that cause the current step's motion can also be identified and located in this manner. A difficulty faced in documenting a system is extracting complete information from the design staff and then organizing it in a form useable by the maintenance staff. Typically the documenter requests a written description of "relevant" information. The designers are not particularly well trained in documenting since this is not needed to make the machine work. The documenter does not have a framework for organizing this information for use in machine trouble shooting. The framework created by the control state machines address the organization

problem as well as provide a means to determine when the information is complete. Coupled with video and audio capture, documentation becomes much simpler. The designer is taken to the machine. The machine is stepped through its sequence one step at a time. At each step, the designer points out the actuators that are used by each step and explains how they are coupled to the mechanical system. The normal mechanical motion of the step is video captured and the sensors and switches that are used to indicate the completion of the step are identified and located. It is simpler to have the designer verbally explain and point, prompted and organized by the sequencing of the machine than it is to obtain a written explanation at a desk remote from the machine where the designer relies mainly on memory. Additional information pertaining to wiring, pipe routing, mounting, adjusting, etc. is added to this framework. More general information provided by the manufacturer of the parts such as specifications, assembly drawings, repair procedure, is linked. Thus, during a breakdown, the repair technician is presented with complete relevant information to correct the problem and allow the next sequence step to be reached. This framework can also be used to organize and retrieve a knowledge base of repair methods built throughout the life of the machine. The repair technician can be presented with past repair history pertaining to this sequence step, and they can record the corrective action they used. In this manner, the sequence of steps provides a framework for organizing and automatically retrieving information needed for resolving machine breakdowns and capturing knowledge on the repair.

5.3.2 Control Design

The die accelerator served to introduce RLM to state based programming. This system, consisting of a single station with limited inputs and outputs (I/O) and a distinct sequence, was selected because of its simplicity. Such simplicity aids technology insertion by reducing the amount of ancillary information that is not pertinent to the new material being assimilated. The targeted user can concentrate on the new method without being unduly distracted by the complexity arising from the magnitude of detail associated with more complex systems. To keep the level of complexity to a minimum, control of station sequencing using a state machine was implemented only for the automatic mode. Figure 5-2 shows the final state diagram that was derived after several design iterations.

A major objective of this project was to develop a standard topology for the Ladder Logic where one could readily recognize the various elements of the state machine. A tenant of technology insertion is to provide the user with the familiar method along with the new. The approach was to use the familiar Ladder Logic to implement the state machine. By utilizing a standard topology for the various sections, the user would become familiar with the concepts of state machines through the Ladder Logic code. To implement the states, the previously described "one relay per state" approach was used. This approach was selected since it aids in the recognition of the state machine in the code as well as simplifies debugging and determination of the current state by a monitoring system. It also helps meet the objective of providing a simple method

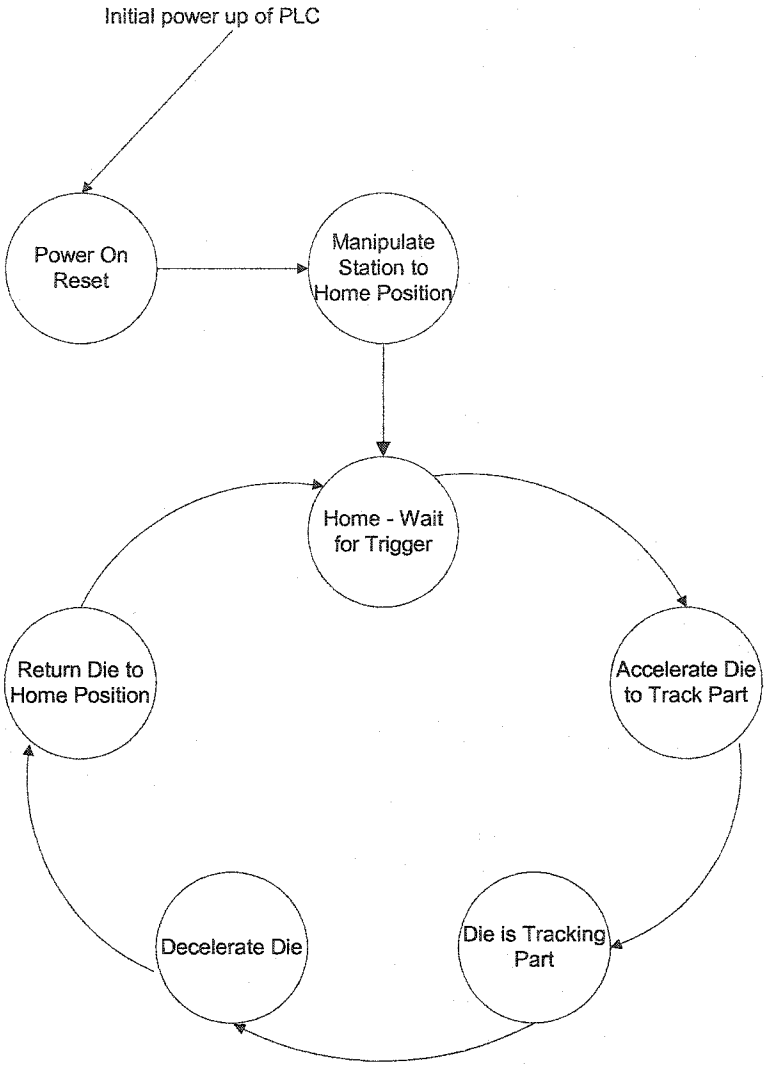


Figure 5-2: Die Accelerator State Diagram

for initially generating the PLC Ladder Logic code manually from a state machine diagram and subsequently modifying it in the field. This was of greater importance then compact code that typically requires encoding states into a binary value using a coil for each encoding bit. If the states were encoded into several coils, the technician would need to examine all of them and translate the code into a specific state. Using a single

coil per state requires finding only the single coil that is currently energized and noting from its name, which state it represents. To create the logic representing a state transition into a specific state, only the rung associated with the state needs to be manipulated rather then each encoded coil rung. To determine which states activate a specific output, single contacts controlled directly by the state coils rather than state decoders are used. The specific implementation in Ladder Logic is detailed in the Appendix B. This particular topology was arrived at through an experimental iterative design followed by refinement process with evaluation and feedback provided by the RLM control engineer.

5.3.3 UML Model

A UML model was developed, based on the design of the Flying Die cut-off machine. The generic machine that it represents is of a single station size containing a single sequence. Such a station is typically part of a larger machine that controls the operating mode. The operating modes consist of automatic, manual and emergency stop. The UML class diagram for a station sequence, referred to as a device state machine, is shown in Figure 5-3. A class is defined for the state machine and is shown at the top of the diagram. Associated with a class are attributes and methods. An attribute is an item of information that describes the specific instance of the class. The state machine has a descriptive name that the user can identify as well as an unique identifier that is to be used by the application generator. The name can be changed at any time while the identifer will retain the value it was assigned when it was created. The unique identifier

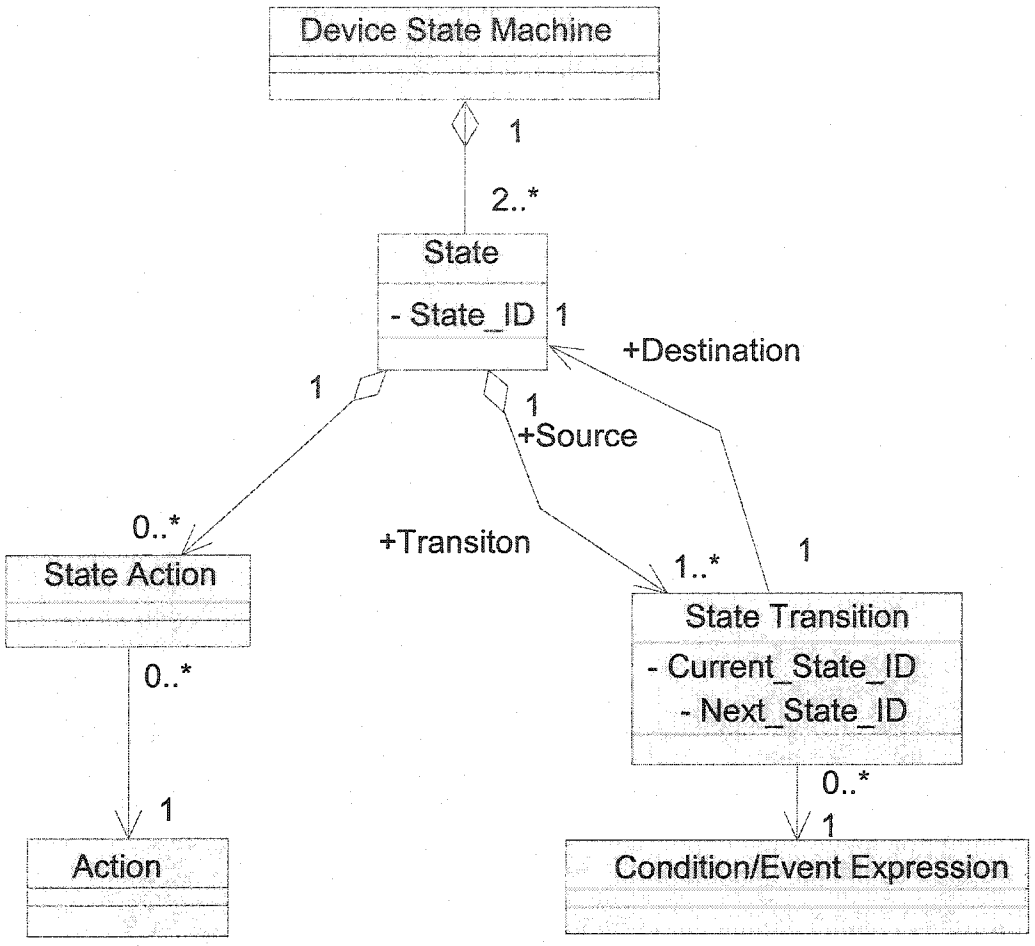


Figure 5-3: Device State Machine Class Diagram

is used when building a model to refer to the specific state machine. Thus when a change to the name occurs, it needs to be updated only in the device state machine object while all the references to it that may exist do not need to be changed. Since the state machine is part of station, a station identifer is included. The station is envisioned to follow the same approach of having a name as well as a unique identifier. Thus only the the identifier needs to be stored in the state machine. Should the station name be required, the identifier will be used to find the station object with the matching identifier.

Having found the station object, one can then access the station name. The relation of states to state machine is shown by the "state" class rectangle and the "association" line joining it to the state machine class.. At the state end the numbers "1 ..." indicate that there is a least 1 and there can be as many states as required, in a state machine. At the state machine end is the number 1 indicating that a state can only be part of one particular state machine. These numbers are referred to as the "cardinality". As with the previous objects, a state has a unique identifier as well as a descriptive name.

Each state has one or more state transitions associated with it. This transition leads to the next state that is to be entered in the sequence of the station. Since a station sequence always repeats, every state must have a transition to another state. The sequence may have more than one path so a state may have more than one transition to different states or even the same state. Thus the cardinality of the state end of the asociation joining the state to the transition is 1 and the transition end is "1 ...". Since a transition must join the source state with a destination state an association is shown from the transition class back to the state class. In the class diagram this is the same box indicating that it is a state. However, the designation of "destination" indicates that this is usually a different state than the source. A transition can lead from a state back to itself. Each transition has a unique identifer and descriptive name. In addition, the source and destination state unique identifiers are also stored in the transition. To find all transitions from a particular state, all transitions with that states identifier in their source state are located. Then to find the state that will be enterd through a particular transition, the transitions destination state identifier is used. A transition occurs only when a

particular event and or condition occurs. To locate this information a condition/event identifier is used.

Associated with each transitions is an "condition/event logical expression". In UML, conditions and events are usually treated separately. A condition is considered to be a "guard" that enables an event to cause a transition. For example, the "station cycling automatically" would be considered a guard condition that must be present before an event such as "trigger" would be recognized and result in the transition from the "home – waiting for trigger" to the "accelerate" state. An event is typically an occurrence at a point in time such as the trigger requesting acceleration of the die. Analysis of the distinction by the RLM control engineer indicated that there was little practical gain in this distinction at the expense of introducing more classes for the user to learn. The user would need to learn the distinction which can often be subtle in machine control where events can be considered to be conditions. A typical example is a sensor that indicates when a machine has reached a particular position but remains active when the machine has moved beyond the position. The sensor being active (turned on to indicate the presence of a condition) actually indicates the condition of the machine being at or past the position. The transition of the sensor from not active to active is the event. In PLC Ladder Logic, the transition is not usualy explicitly decoded. Only the fact that the sensor is active is used in the state transition logic. However, the way the rest of the logic is constructed causes the transition logic to be active (a complete circuit) for only a short period of time (usually only one PLC program scan) capturing the event of the machine arriving at the position. This is achived by including the source state as part of the

transition logic. As soon as the transition logic is active, the destination state becomes active. As soon as the destination state becomes active, its noramlly closed contact in the source statement's latch opens and causes the source state to no longer be active. This in turn opens the normally closed contact which then causes the transition to no longer be active. To further complicate the definition, the machine may have been manually moved to, or past the position. The machine is subsequently placed into the automatic state that is awaiting the "event" of the machine reaching the position. The logic would treat this as an "event" that has just occured even though the machine has been at this position for some time. This difficulty in differentiating between events and conditions led to the decision to treat them the same and name them the "condition/event" class. This particular class was separated from the transition class to provide a form of reuse where the same condition/event can be used with multiple transitions. The condition/event can be independently defined once and later associated with any number of transtions. This is indicated with a 0 cardinality on the association at the transition end.

To keep the transition logic simple and to provide a degree of abstraction, a transition is assciated with only one condition/event. The descriptive name that is an attribute of the condition/event provides the abstract meaning while the logical expresion that may be quite complex is hidden. Should more than one distinct condition/event lead to a transition between two states, separate transitions should be used. One could create a condition/event that contains all of the combinations. However, to make the control algorithm more understandable, different transtions should be used. This differentiation

is desirable since the different transitions occur for different conditions or events during the machine operation. Thus one can gain a partial understanding by studying a particualr case without needing to comprehend all of the possibilities. Studies [Mil56, Shi94, Bad94] have shown that the human mind can comprehend only a limited number of different items at a single time. During experiments with technicians, it was also noted that there appars to be a limit of complexity that when reached results in the person becoming overwhelmed. At that point they loose the willingness to decipher the function of the logic. This observation also served as a motivation for keeping the transitions as simple to understand and map to the function of the machine as possible.

The control of a manufacturing station is typically accomplished through actuators such as motors, pheumatic or hydraulic cylinders, magnets, etc. These are in turn controlled by electronic modules, electronically activated valves, electronic or electromechanical relays. The PLC turns these on and off using its electronic outputs. The current practice is to name these outputs in terms of the intermediate controls that they are connected to. The technician needs to be familiar with the machine to translate what action the machine will take when the particular output is energized. To aid in understanding of the control code by the monitoring system, information hiding was applied. Instead of using outputs as a class, the action that results is used, hiding the implementation detail. The action class contains a description of the action, an identifier for locating it and the actual PLC output that causes the action.

Typical station cycles often have the same action present for several consecutive steps. This led to an approach where the first state in which the action was needed caused the action to be activated. The action remained active until the last state in which it was need de-activated it. This approach caused difficulty when troubleshooting machine breakdowns. The current state was easily identified from the code, but which outputs should be on were not obvious. The state that turned the output on and the state that turned it off were readily apparent, however, the technician needed to know the sequence to determine if the current state happened to be located between the two. To address this situation, each state in between the ones that turned the output on and off were included in the logic as states that "maintained" the output. Upon examining this approach it became apparent that all of the states where the output was on appeared in the logic. There was no need to distinguish the states which turned it on, maintained and then turned it off since this distinction was not needed in the output logic. The coil that represented the output could only be in one of two states: energrized or not energized. The conclusion reached with feedback from the RLM contol engineer was to not differentiate the states and simply include all of the required ones in the output logic (using a logical OR fuction).

An intermediate class "state action" was placed between the state and the actual action in the UML model. Its purpose is to link the state to the action. The state identifier and the action identifier are stored in this class. When generating the code for the action, all of the state action objects that have the specific action identifier are collected. From them, the state identifers are then used to locate all of the states that

energize the action's output. There were two alternatives to the use of the intermediate state action class to link actions to states. One was to store the action identifier directly in each state while the other was to store the state in each action. The alternatives are more efficient when there is only a one to one relationship where only one state controls one action. However any state can control a number of actions and any action may be controlled by a number of states. This would require either a varying length of storage or a fixed maximum of action identifiers in the state or state identifiers in the action along with a mechanizm to determine the exact quantity. Using the intermediate state action places the state and action identifiers in it so one simply creates as many as there are actions controled by a specific state.

The condition/event is used as an information hiding mechanism that allows the Ladder Logic to be more easily understood. A good example is its use in the transition from a "manipulate machine into home position" to the "home position" state. This condition is typically named "machine is in correct position to enter home". It consists of a logical expression containing several individual elements. Each of these elements represents an individual aspect of the machine that needs to be in particular position or state before the machine can be considered to be in the home position from which automatic cycling can commence. Normal coding practice would place all of the logical equations into the state transition equation along with its logic. When looking at the resulting complex Ladder Logic to determine its function, one needs to mentally group it into the separate protions and realize that a large part is simply indicating that the machine is in the correct position to enter home. To avoid this, a single contact is used in

the transition logic. The abstract function of "machine is in correct position to enter home" serves as it name. The coil that drives this contact implements the logical equation. Several representative programs that RLM had previously written were examined to determine what elements would be used to construct the logic of such a condition. The elements are shown in the UML class diagram in Figure 5-4.

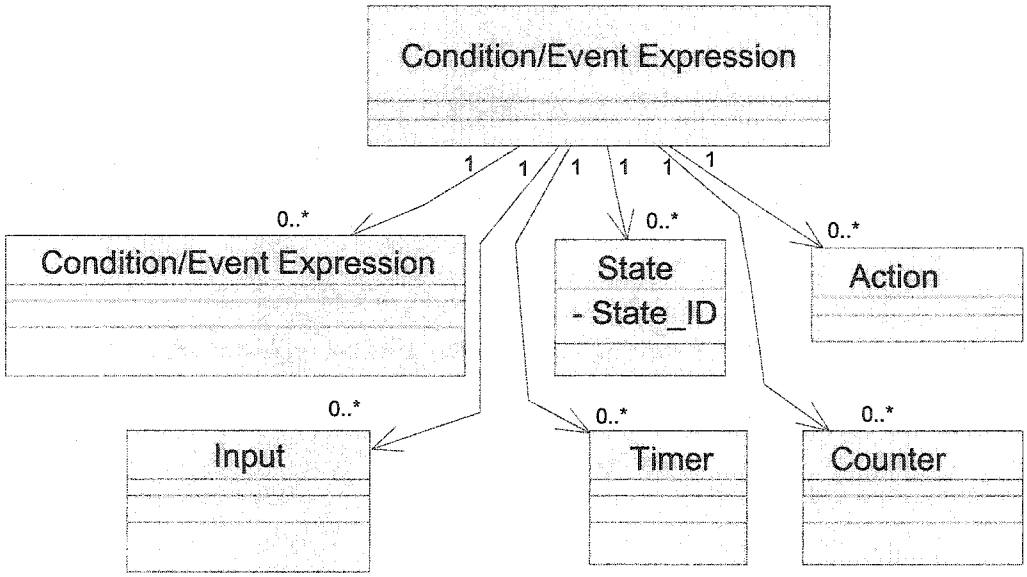


Figure 5-4: Condition/Event Expression Class Diagram

The most common element of a logical expression is an input. Common inputs are switches and sensors used to determine the position of portions of the machine or of the part that the machine is processing. There are times when a condition may be used as

a part of a more complex condition, thus an entire condition may be an element. In this case that condition will have its own definition and it will be represented with its own coil in the Ladder Logic. A contact driven by this coil will appear as an element in the logic for the more complex condition. When examining typical logic for "machine is in correct position to enter home" it became apparent that there were cases where there was no input nor sensor to indicate a particular situation. An output would be energized that caused a condition to occur but there was no direct input indicating this, such as an air cylinder that was retracted or an interface signal sent to another station. To recognize that this condition exists, one needs to examine the action that controls the output. Thus actions can also be elements of a condition. Timers and counters are found within most common PLC's. A timer can often be used to estimate a condition that is not directly measured. For example, when a rotating device is started, it is expected to reach it normal operating speed after certain time. Another example is the need to lubricate a machine. Rather then using sensors to determine when more lubricant is needed, feedforward control can be implemented. A timer is used to track how long the machine has been operating in a manner that consumes lubricant and indicate when lubrication is required. Counters are also used in a similar manner. Rather then sensing when a container is full, knowing how many items it can contain allows a counter to indicate the condition. Rotational position sensors often provide a pulse each time a certain amount of rotation has occured. A counter is used to keep track of the total rotation. This use of timers and counters required their inclusion as elements of a condition expresion. States were added as possible elements to provide a complete set. At the time of implementing

the code for the die accelerator the need to interface with other stations was noted. It was invisioned that the interfaced machine would also use state based logic for all of its stations. In that case, a state could be viewed as a form of information hiding that one normally expects for interfaces between software modules. Common practice for interfacing stations is to directly use various elements belonging to another station to determine when a particular situation exsists. Should both stations be implemented with state machines, one would only need to determine when the state indicating the condition was present. The details of how this state was determined would remain a "secret" [Bri81, Par77] of the station. Thus states were included in the UML class model for a condition to provide the interface mechanism between stations.

5.4 Representative Complete System Experiment

5.4.1 Stamping/Roll-Forming Work-Cell

Having successfully implemented a single station, the RLM control engineer was convinced that the advantages of the new methodology warranted expanding it to address the complexity of a full system. He chose a stamping/roll-forming work-cell system shown in Figures 5-5 through 5-9 as the application to model because it is representative of the more complex found in Roll Form manufacturing.

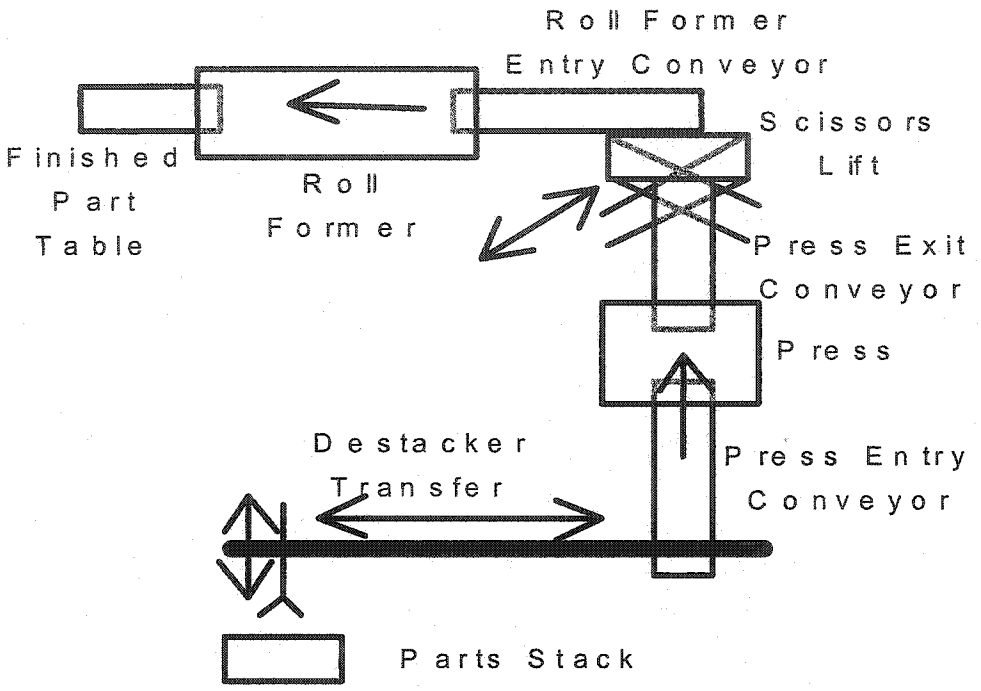


Figure 5-5: Stamping/Roll-Forming Work-Cell

Before an object-oriented model could be developed, a thorough understanding was required of the functionality provided by control code for such a work-cell. RLM has derived a complete set of functionality through years of system development. There was no need to develop new functionality. Since RLM was now familiar with state-based programming, the approach was to write the control code for the work-cell in this manner. The complete set of functions was identified from this specific case. With the knowledge of the RLM engineer, generalization was then applied. Common functions were grouped into standard services and placed into classes. A standard method of writing PLC Ladder Logic was developed for these classes to minimize the amount of unique code. Hierarchy was employed to deal with comprehending the amount of

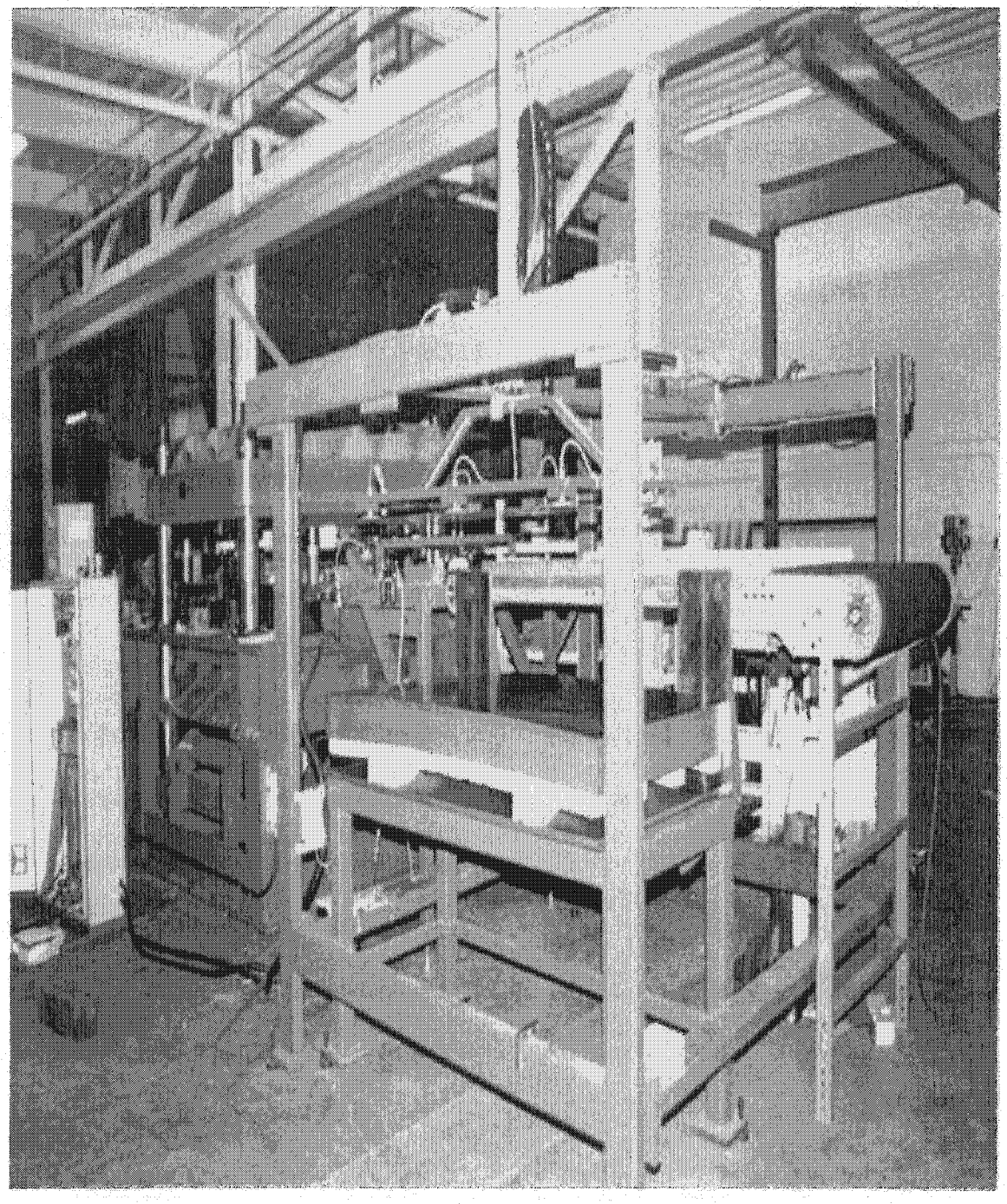


Figure 5-6: Sheet Metal Destacker and Transfer
Courtesy RLM Manufacturing

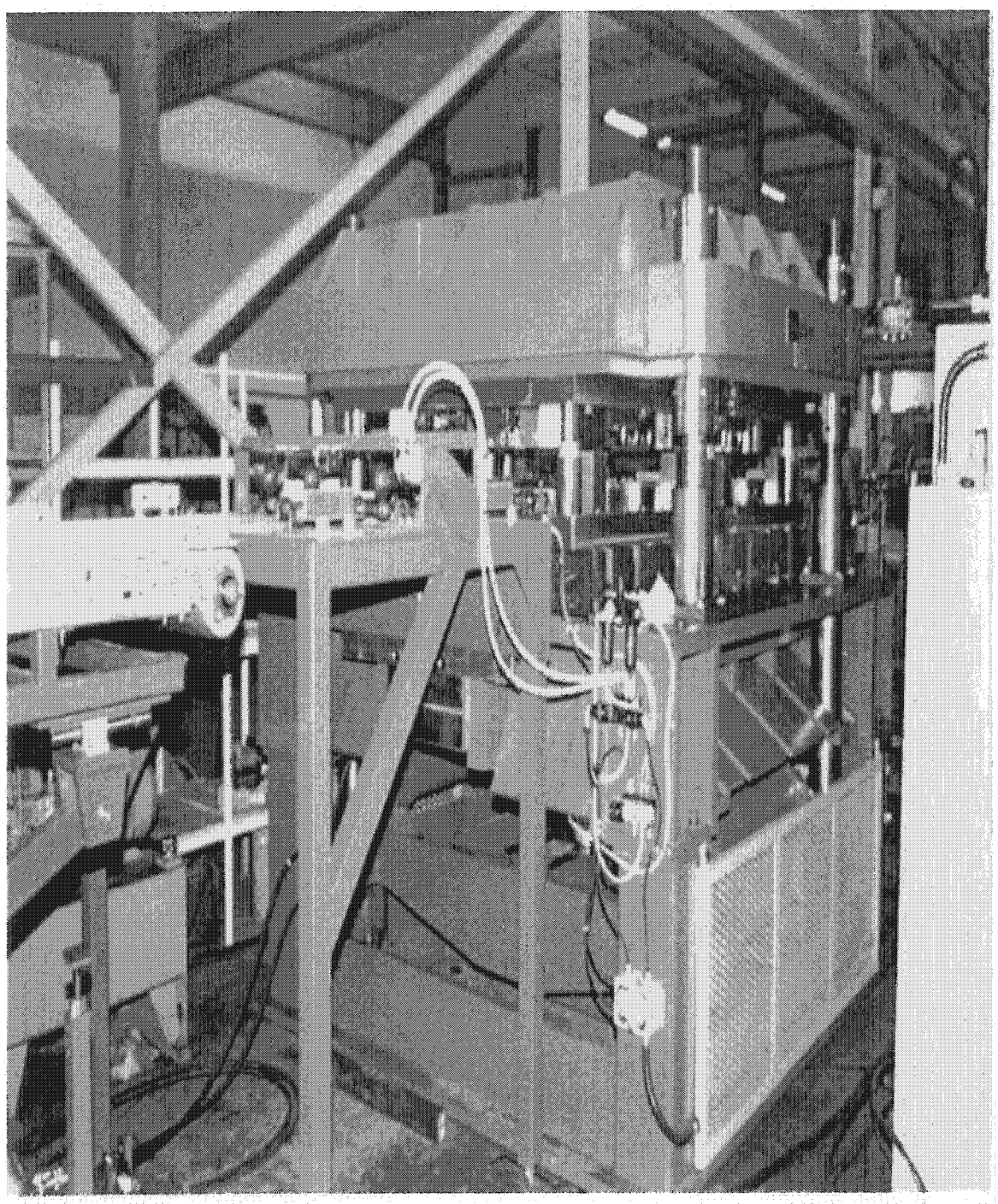


Figure 5-7: Press and Transfer - Courtesy RLM Manufacturing

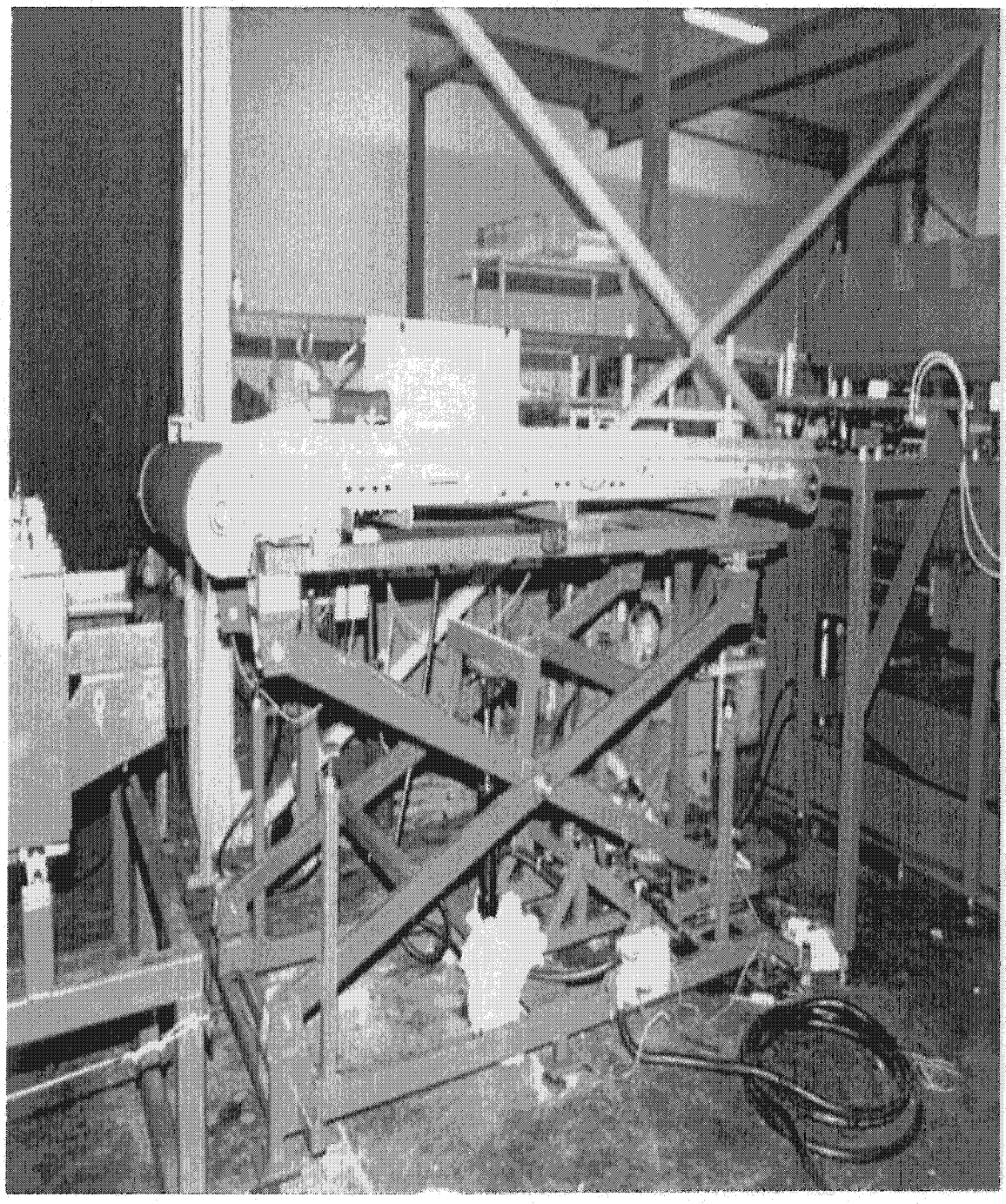


Figure 5-8: Scissors Lift and Roll Former Entry Conveyor
Courtesy RLM Manufacturing

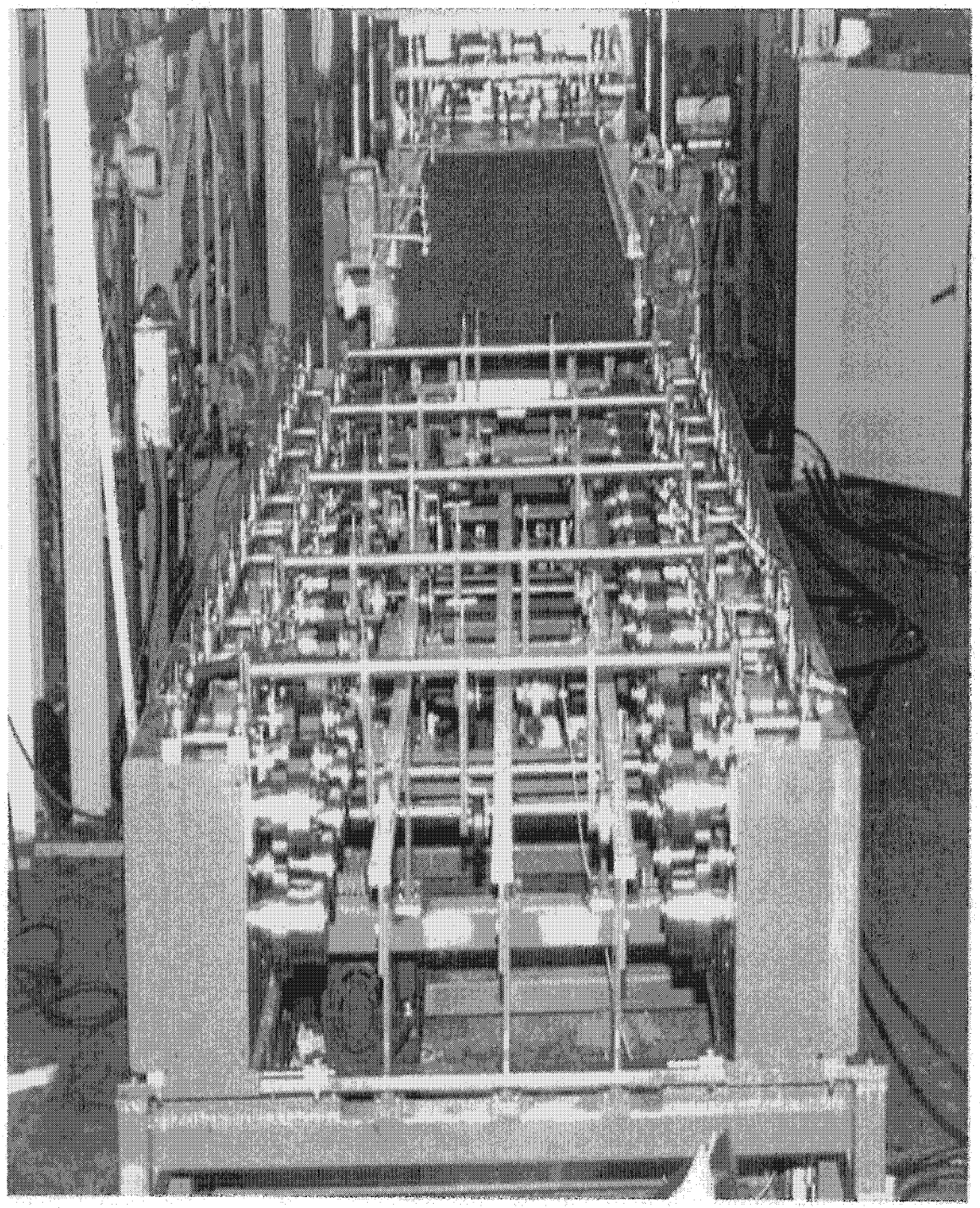


Figure 5-9: Roll Former and Roll Former Entry Conveyor
Courtesy RLM Manufacturing

functionality. This was accomplished by partitioning the functionality into the abstract set of standard classes in a manner that were then arranged in a hierarchy that according to Object Oriented design theory, reflected how a work-cell is naturally operated. There were tangible counterparts to the classes in the machine control such as separate control panels or groupings of operator switches and buttons. Functions that applied to the entire work cell were placed at the highest level. Then the functions that applied to natural groupings such as stations were collected at a lower level. The functions that were specific to the lowest level were placed at the bottom.

5.4.2 Hierarchical Object-Oriented Model

The work-cell object-oriented model was partitioned into a hierarchy of three distinct levels. At the top level, the entire work-cell is treated as an independent system, which can be operated by itself. According to OO theory this level or object should be readily recognizable in the physical domain. Indeed it is. The operators and technicians will recognize the system level as the control panel that contains the mode selection switches, push buttons and indicators that provide the user interface to the system level functions. The behaviour of the overall system can be specified with a system-level FSM. At the next level, the system is divided into distinct subsystems as shown in Figure 5-10, coordinated by the system-level state machine. Within the literature these are often referred to as workstations or modules. The behaviour of each subsystem is specified with a subsystem-level FSM. At the lowest level, each subsystem is then further divided into smaller groupings called components, which are coordinated by their subsystem-level state machines. Analysis revealed only four different components at this level.

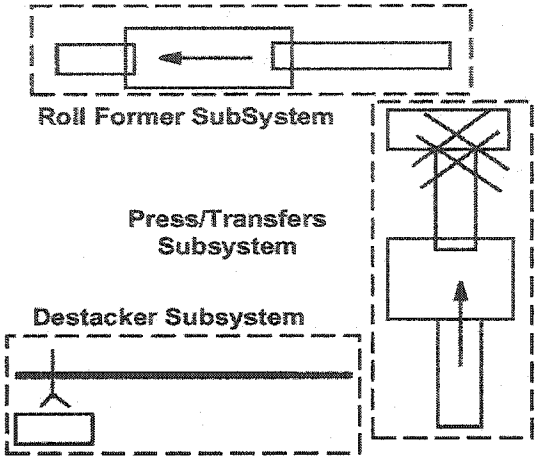


Figure 5-10: Subsystem Partitioning of Work-Cell

The components at each level can be modeled as classes in an object-oriented model, and can be implemented as software components. The abstract classes and their nesting are shown in Figure 5-11. The entire class model consists of only seven different classes. The industrial controls domain expert from RLM has adopted this model and has been using it for more than two years. He has found that it is applicable to most work cells of similar complexity.

The abstract class model that resulted from the analysis is fairly simple. This lower complexity is desirable since it reduces the initial effort required to learn the new methodology, thus increasing the probability of successful technology insertion. Such simplicity is also a key to successfully implementing an application generator and its ability to be used by a domain expert (i.e. mechanical engineer) who is not familiar with control code generation. A simple model also enables the automatic generation of monitoring code since relatively few rules are needed for the limited finite set of different

situations.

The PLC code was initially manually coded based on the object model. Working with feedback from the domain control expert the author developed PLC Ladder Logic code topologies, which were captured in a prototype application generator. Using an iterative process, the Class Model and its mapping to specific PLC code topologies were refined. The end result is that when one has a designed the specific object model for a particular system, a mechanical mapping or transformation is applied to generate the PLC code. The application generator is capable of producing of the PLC code topologies identified with the exception of the Sequential Constraining Condition component. This topology was not implemented because the code generated for a Sequential Control component could be used with minor manual modifications (one normal coil replaced with setting and clearing a latched coil) and did not hinder the experiment with the pipe fitter presented later. The user interface posed a far greater problem, so time was spent there instead. The RLM controls engineer has found this methodology sufficiently superior to the normal practice even when applied manually without the aid of an application generator (through the use of a PLC editor's cut and paste) to adopt it as the standard practice at his company.

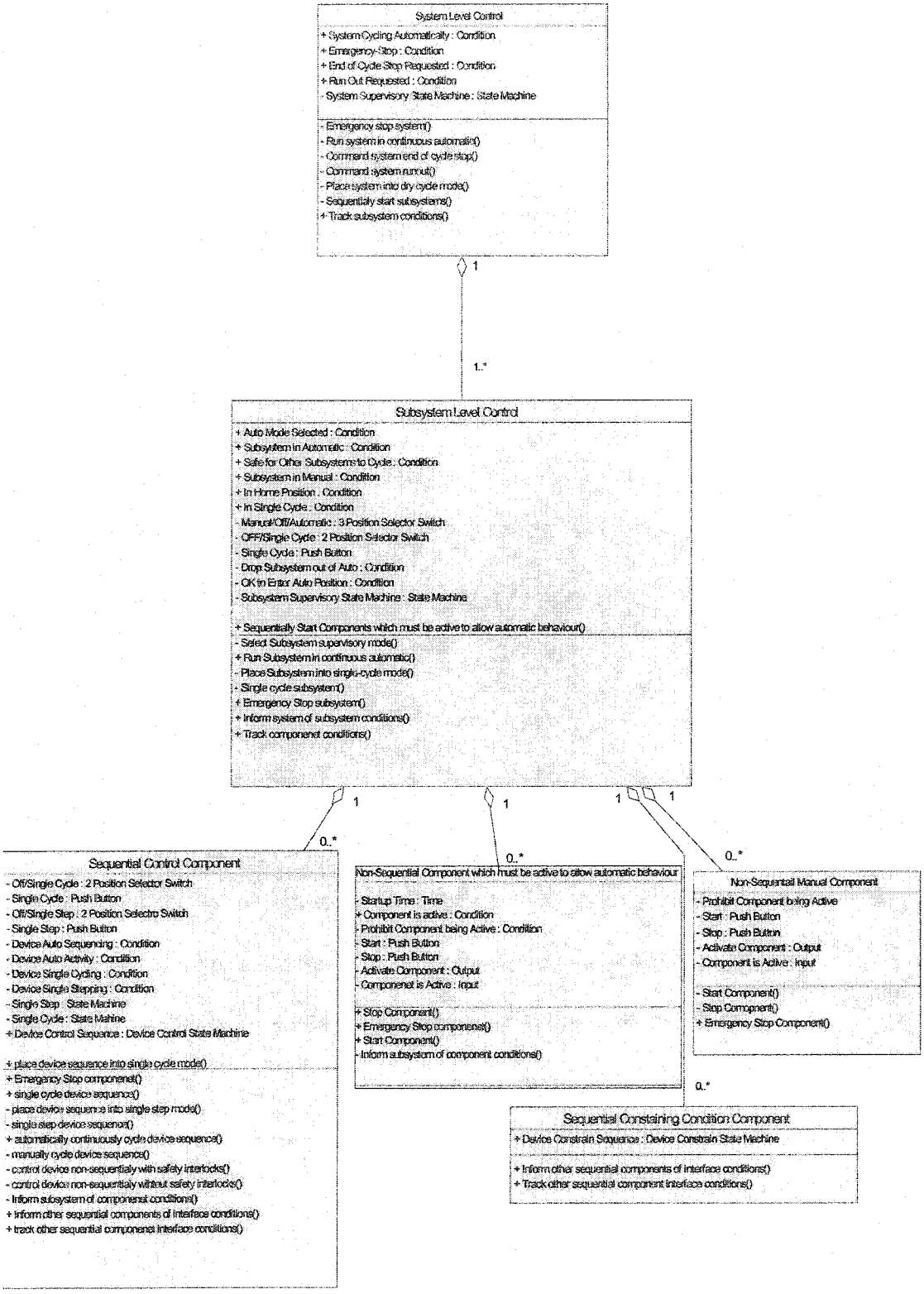


Figure 5-11: Abstract Class Model

5.4.3 System Level

The system-level control class is responsible for coordination of overall work-cell functions. The purpose of such a system is to automatically process or produce some form of part or work-piece. The system level provides the operator with the system-wide setting of modes related to this. These primary modes are: off; automatic; and emergency stopped. There are standard ways to transition between these modes provided by this class. One should note that the manual-operating mode is not included. Analysis showed that manual-operation is typically used on a particular subsystem. It is not generally applied to the entire system at the same time. Thus, this behaviour is provided only at the subsystem level.

The state machine model of the system-level control behaviour is shown in Figure 5-12. At this level, the operator selects and initiates automatic operation of the entire work-cell. Emergency stop conditions are monitored here and communicated down to the lower subsystem level. The state of all of the subsystems is monitored to determine when the system can enter the automatic state and when it no longer can be maintained. Prevention of accidental machine cycling is one of the basic principles of manufacturing controls. This is achieved by requiring two distinct operator actions before entry into automatic to insure the operator truly wants automatic.

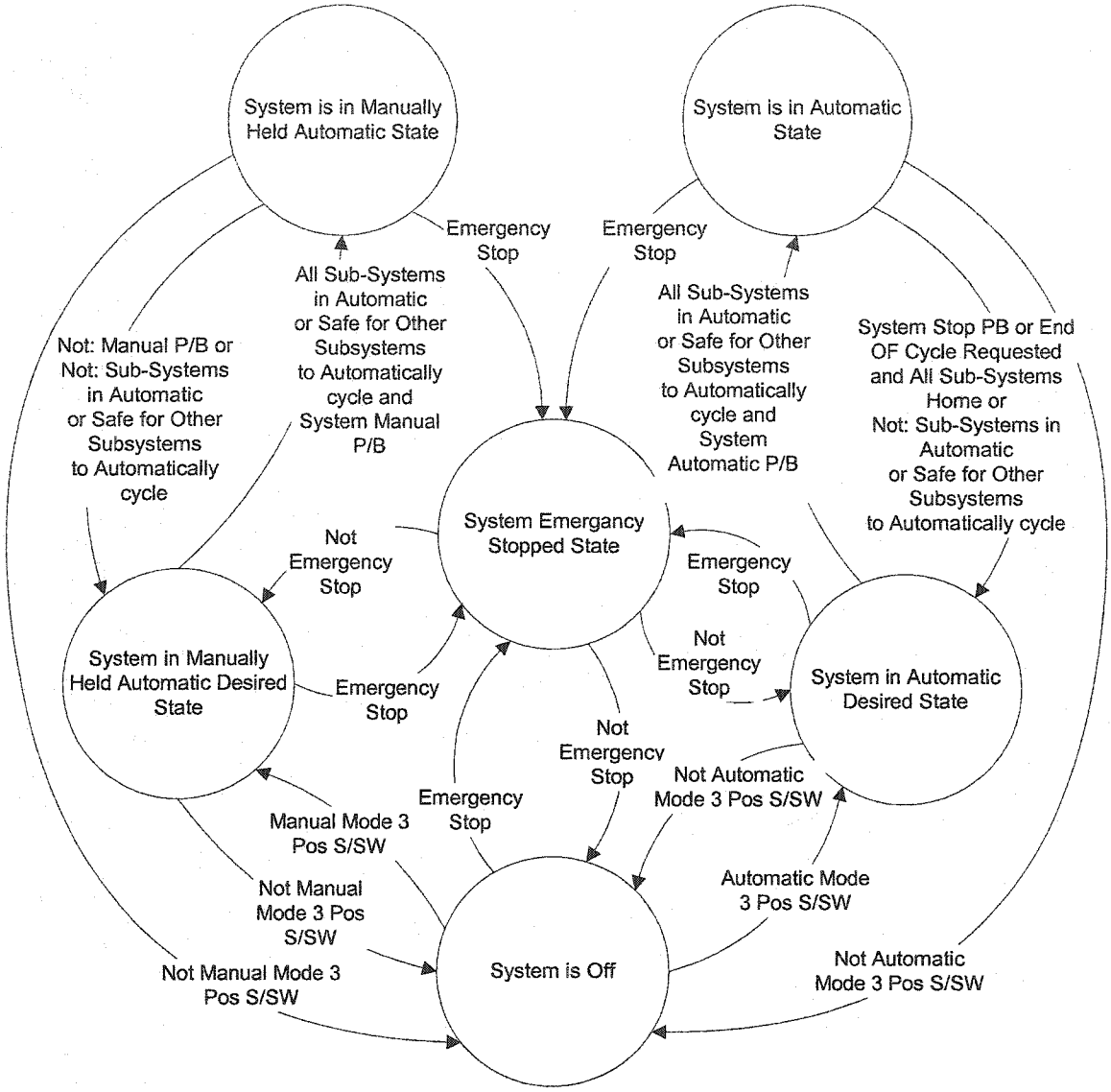


Figure 5-12: System-Level State Machine

To reach the "System is in the Automatic State" from the "System Is Off' state the operator must first place a three-position mode selector switch into the "Automatic Mode" position. This will place the state machine into the "System Automatic Desired" state. To place the system into automatic, the operator must then press the "System Automatic" push button. There are two different forms of maintaining automatic operation of the work-cell. The first as described above, is the normal form where the operator initiates it and the system continues to operate automatically until the operator presses the "system stop" or "end of cycle stop" push button, an emergency stop occurs, a subsystem drops out of automatic or the operator moves the system mode selector switch from automatic to off. The second form of automatic operation is one where the operator must keep a "system manual held in automatic" push button pressed for the system to cycle automatically. The "System Manual Held Automatic Desired" and "System is in Manually Held Automatic" states model this form. This form is used when adjusting the machine or determining where a problem exists.

An "End of Cycle Stop" function stops the entire work-cell when all of the subsystems have finished processing the part they are currently working on. This is modeled as the transition from Automatic to Automatic Desired when the "End of Cycle Stop" function has been selected and all of the subsystems have reached their "End of Cycle Stop" position. The subsystems are informed by the system that the "End of Cycle Stop" function has been requested. They will coordinate the components that they are responsible for and report back to the system when all of the subordinate components

have stopped.

The system level also provides the following utility functions. It coordinates sequential starting of subsystems to prevent overloading the electrical distribution system. If all of the subsystems were started at the same time, the high initial starting current of all of the motors would exceed the capacity of electrical system. The system level also provides the ability to empty out the work cell of all parts. This is referred to as "run out" and is selected by the operator at this level. The "run out" condition is communicated down to the subsystems, which continue to process the parts already in the work-cell but do not feed any new parts into it. Similarly, the system level provides the operator the ability to select the QS9000 "dry cycle" modified automatic operation. This mode cycles the machine through all of its motions with no parts present. It is intended for initial commissioning of the machine by running for an extended period of time with no malfunctions.

5.4.4 Subsystem Level

The next level down in the class hierarchy is the subsystem control. It groups closely related components together to provide coordination and ability of independent subsystem operations. Classical principles of modular software design, such as coupling and cohesion, can be used to determine the boundary and function of a subsystem. As pointed out by Park, *et al.* [Par98], there is a natural partitioning into modules that group common machining operations into common configurations that are reused in various different machines.

The distinguishing attribute of this level is the ability to single cycle all of the bottom level components contained in a specific subsystem together as a group. Single cycle usually refers to the execution of all of the steps in an automatic sequence. Typically, this starts with the loading of a part, processing it, unloading it and preparing for the load of the next part. The step where this sequence stops at is referred to as the "home" step. However, a sequence is not limited to a single home step. If there is more than one step where it is convenient to stop in, then each one is a home step and a single cycle will step the sequence from the current home step until the next home step is reached.

The subsystem-level controls the overall behaviour modes of its contained lower level components. Figure 5-13 shows the state machine that models this. When power is first turned on, the system is checked for an emergency stop condition. The subsystem will stop in the "system emergency stopped" state if one is present. Otherwise, a three-position mode selector switch indicates which state to enter.

The "off" state is used to freeze the current position of the contained lower level component sequences. The simplest form is the removal of power from all actuators. However, in some cases, the result may be undesirable if power is removed. For example, when an electro-magnet or vacuum is used to hold a part while transferring it, removal of power can result in the part being dropped. In the most general case, the off mode implies stopping motion while maintaining the current state.

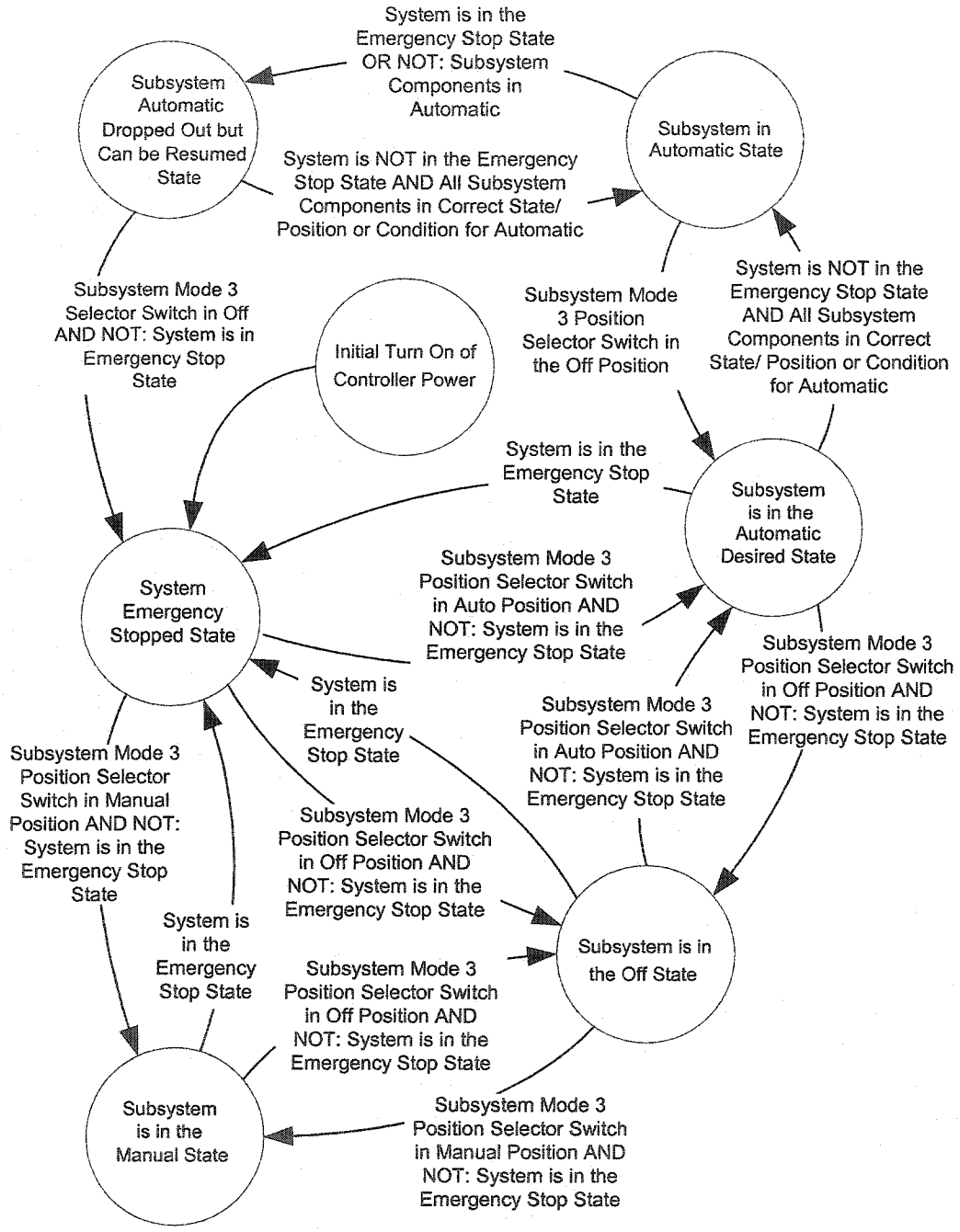


Figure 5-13: Subsystem-Level State Machine

The "manual" state is used to allow operator manual control of the contained lower level components. Analysis followed by discussion with the RLM control expert identified three types of manual control. The most common type (the only one that RLM usually implements) allows the operator sufficient control to manipulate the machine into the initial home position. From there, the operator is limited to manually stepping through the normal automatic sequence of steps and control of only those actions that occur while in automatic operation in each step. In addition, when the machine or part would not be harmed, the reverse action is permitted along with entry into the previous step allowing reverse stepping through the sequence. The actions that result in motion or the application of force are limited to their safe range. The second type of manual allows the operator to control all of the possible actions regardless to the current step in the sequence. Again actions that result in motion or the application of force are limited to prevent damage to the machine or part. The last type of manual control allows the operator to invoke any action without any control system imposed limit. The operator is responsible to limit the actions into a range that does not cause damage or result in an unsafe condition. This type of control is often named "operator emergency override".

The "Subsystem in Automatic Desired" state is entered when the operator places the three-position selector switch into Auto and the system is not emergency stopped. It is used as an intermediate state where the subsystem checks to see if all of its lower level components are in the correct condition or position to allow automatic operation. If they are, then the "Subsystem Automatic" state is entered. Otherwise, the operator is expected to go back to the manual state and place all of the lower level components into the

required condition or position. This appears to be different from the system state machine where an additional action from the operator is required in the pressing of the automatic push button. The reason a second action is not required lies in the fact the contained lower-level components will not commence automatic operation until the operator presses another button. This can be the system automatic, the subsystem single cycle or the component single step push button.

The "system emergency stopped" state is entered from any of the above states whenever an emergency state condition exists. In this state all of the contained components are placed into a safe condition. This condition is the same as "off" where motion is stopped but outputs may remain energized to prevent undesirable situations such as the dropping of held parts. The state that emergency stop was entered from is restored when the emergency stop condition is no longer present.

Automatic operation of the contained lower level components occurs only in the "Subsystem Automatic" state. As mentioned previously, this is only an enabling state. Other states actually initiate automatic operation of the contained lower level components. To remove the need to go through the previous sequence of entering this state again whenever an emergency stop condition occurs or a component needed for automatic becomes inactive, the "Subsystem Automatic Dropped Out But Can Resume" state is used. As long as the operator does not switch the three position selector switch to the off position, the "Subsystem Automatic" state will resume when the emergency stop condition is no longer present and all of the components needed for automatic become

active.

The subsystem also provides the following utility functions. It receives from the system the command to start components that need to be active before auto can be entered. The system sequences through the subsystems to prevent overload of the electrical distribution system. The subsystem in turn, is responsible to sequence starting its own contained components if their collective starting current is excessive. The subsystem monitors all of its components to determine when they are all in the correct position or condition for automatic. The subsystem relieves the system from needing to know the details of the components by informing the system that the subsystem is either in automatic or in a bypass mode where it is safe for the other subsystems to operate automatically.

5.4.5 Subsystem Components Level

At the lowest level of the hierarchical model are the classes that directly control various types of machinery found in a work-cell. The descriptive name selected for these classes is "component" to indicate that these are the lowest granularity represented by the model. The stamping/roll-forming work-cell served as the representative system used to identify all of the kinds of components that needed to be modeled. The knowledge of the RLM control engineer was used to insure that the work-cells they manufactured earlier as well as others he encountered could be fully represented by these components.

Based on the operations of various devices in the work-cell, only four distinct classes are necessary to model such a system. This was far less than the RLM control

engineer expected, based on the large quantity and variety of work-cells he had either designed or encountered. Initially, there were many more components. By applying different grouping of common characteristics, the total number was eventually reduced to four. Analysis of the stamping/roll-forming work-cell components reveled the following sequence of distinguishing features shown in Figure 5-14. The major difference was between components that either controlled a device or served as an interface only without directly controlling anything. The controlling type is then differentiated based on the controlled device having a distinct sequence of steps or not (beyond simply on and off). Then amongst the non-sequential kinds a further division was made based on components that controlled devices that were needed to allow automatic cycling and the ones used only for control only in the manual mode. The following section introduces the four components. A detailed discussion is provided in the application generator section.

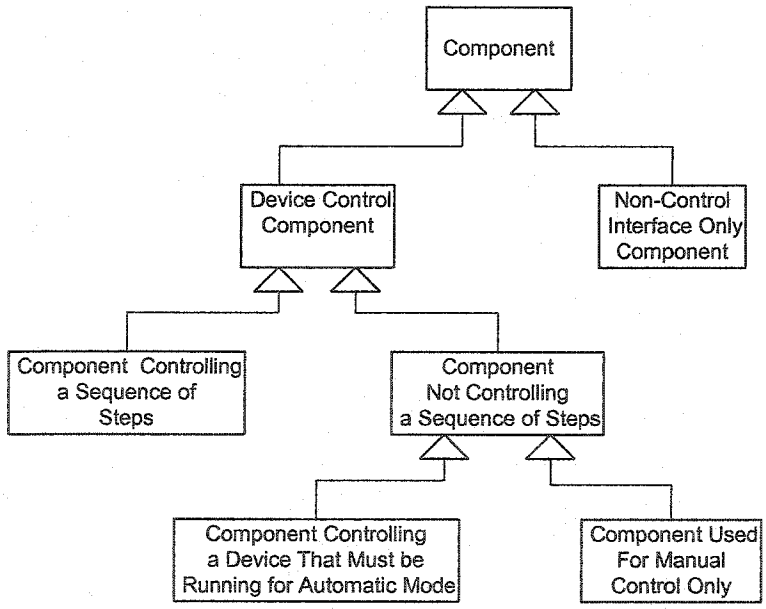


Figure 5-14: Component Classification

The first class is named "sequential control component" derived from the Destacker Subsystem. The distinguishing characteristic of this class is a sequence of steps that it cycles through when the system is operated automatically. The Destacker Subsystem automatic mode sequence was modeled as a state machine shown in Figure 5-15. Thus, this class will represent any component that exhibits sequential behaviour while in the automatic mode. Analysis of the Destacker identified a second simplified sequence that was used to control it in the manual mode. The state machine modeling this sequence is shown in Figure 5-16. This state machine constrained the operator from unloading a part onto the press conveyor until the press finished processing the previous

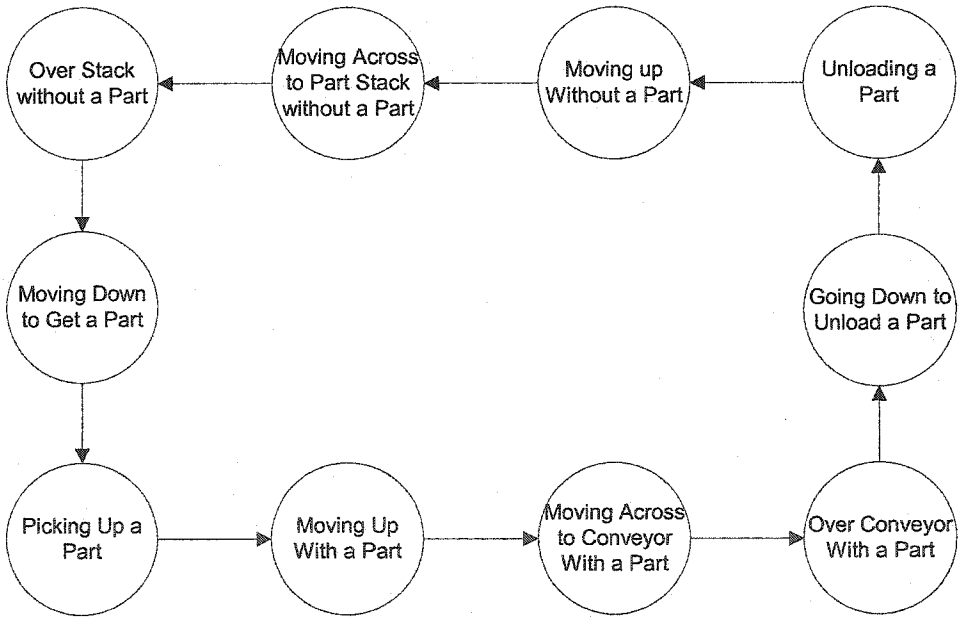


Figure 5-15: Destacker Automatic Mode State Machine

part and was empty. Basically, the operator was allowed to manipulate the Destacker in any safe manner as long as the rule of only one part in the press was not violated. Unsafe actions such as transverse motion while that transfer was not in the top position were

prevented by the logic that controlled the outputs in the manual mode.

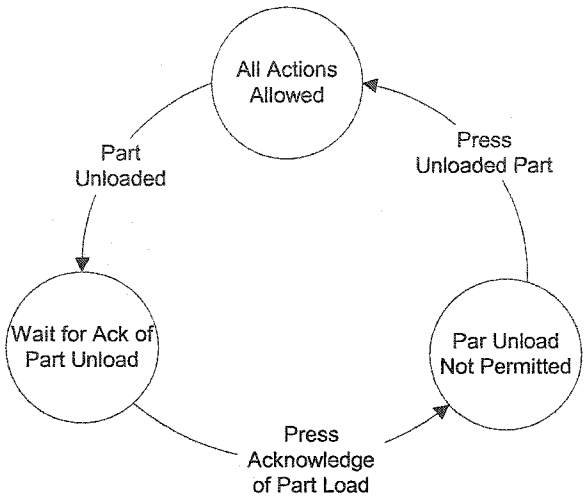


Figure 5-16: Destacker Manual Mode State Machine

A more general pattern for manual operation was identified when the Press/Transfers subsystem was modeled using this type of component. This can be best illustrated by drawing a combined state machine containing both the automatic and manual state machines. By comparing the two sequences shown in Figure 5-17, one will note that the manual mode constrains the operator to follow the same sequence as in automatic. In addition, where it is safe to do so, the manual sequence can be reversed. In situations such as after the part has been stamped, backing up to the previous step is not allowed. This prevents the potential re-stamping of the same part that could damage the die due to the creation of small slivers of metal. The advantage of using a manual sequence that follows the automatic one is that the subsystem may now be switched from manual to automatic without first manipulating it into a home position. From the manual state machine, there is sufficient information to identify which automatic state to enter,

shown by the transitions between manual and automatic states. To allow such transitions, a pair of nested device control state machines models a component of this class. The automatic and manual mode each has their own state machine that defines the sequence in the subsystem-level auto state and manual state, respectively. When the subsystem is in the automatic mode, the automatic sequence state machine is selected. When the subsystem is in the manual mode, the manual sequence state machine is selected. The FSMs for components have only a partial observation of the system external to the controlled device. Typically, only interface states or conditions of components that the device interacts with, such as the Destacker wishing to unload a part onto the press entry conveyor, are observed. Sequential control components can also contain supervisory state machines to single step or single cycle through the sequence. These are detailed in the

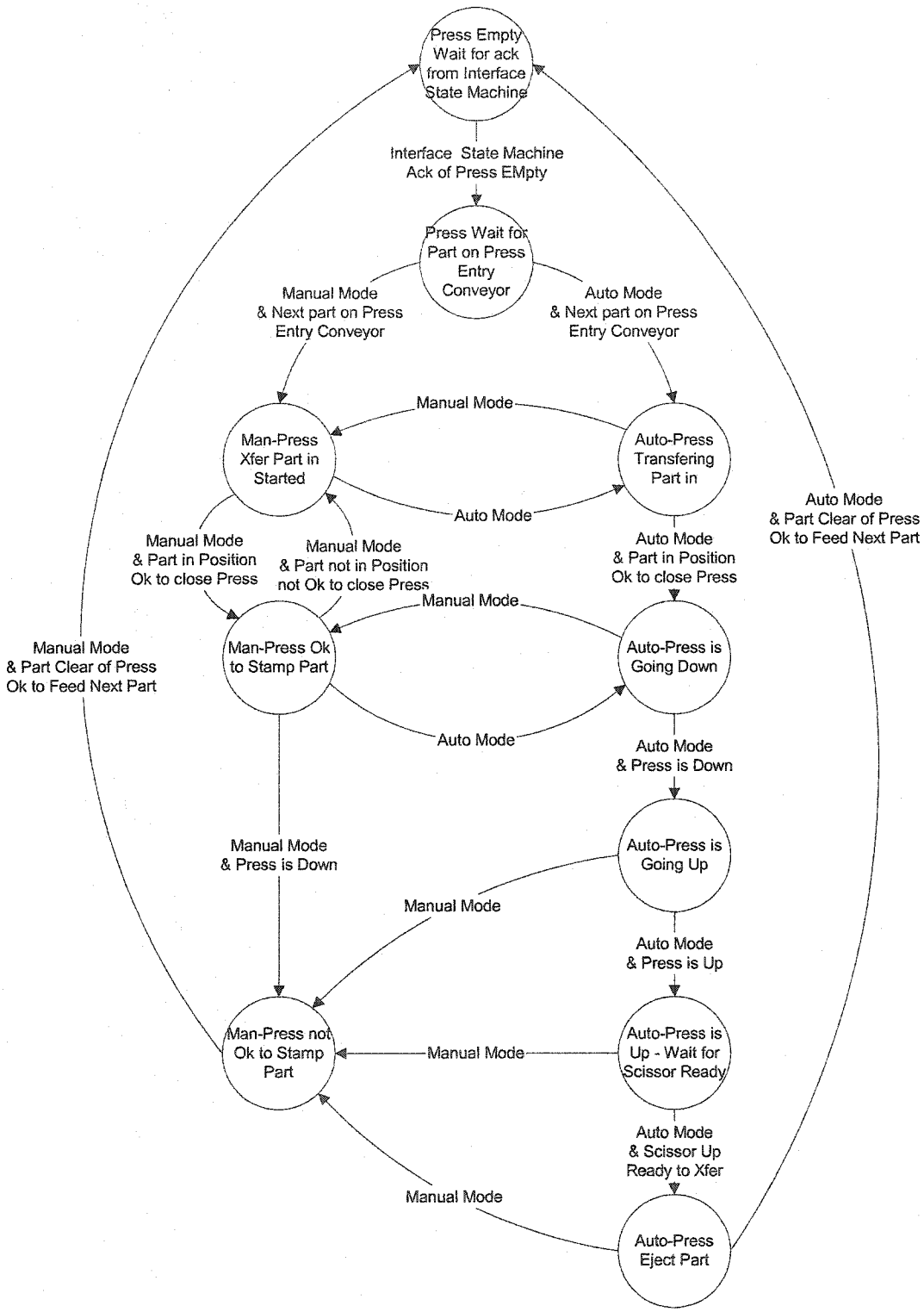


Figure 5-17: Press/Transfers Automatic and Manual State Machines

application generator section. Single step is the sequencing of the component one step at a time performing all of the part processing that would normally happen in continuous operation. Single cycle on the other hand, sequences the component through the automatic steps and stops when it has reached the next "home" step.

The second subsystem component class is named "non-sequential auto". This type was identified when the Press/Transfers were analyzed. A hydraulic pump powered the press. The pump motor needed to be started before the press could be placed into the automatic mode. The automatic mode cannot be maintained should the motor stop due to an external cause such as over-current fault or simply by the operator's stop push button. The class used to model this type of behaviour does not exhibit a sequence of steps. However, the equipment controlled by this type of components must be running to allow the subsystem that contains it to be placed into the automatic state. Typically, these are devices that supply energy such as hydraulic pumps, flywheels, motors connected to devices with clutches, etc. or transfer devices such as conveyors that run continuously. Should such a device stop, the subsystem that is dependent on it must be removed from the automatic operating mode.

The third class is named "non-sequential manual". It is very similar to the second class and it typically controls the same devices, but in a mode which is not allowed while in automatic. This class was identified while modeling the roll former. In the automatic mode, the roll former motor is run in the forward direction and can be modeled by a non-sequential auto component. The roll former can also be run in the reverse direction, but

only for maintenance or setup reasons, never while in automatic. To allow such control and to differentiate it from automatic, the "non-sequential manual" class was developed. In the general case, it is used for the operation of non-sequential devices in a mode not permitted while in automatic such as running the devices in a reverse direction. Physically, this may be the same device controlled by a "non-sequential automatic" component, but from the model's point of view, it is treated as a different class due to its different characteristics. When a technician is analyzing a system, it is usually for a specific mode. Thus using a separate class for each mode limits the information needed to be comprehended to that which pertains only to the particular mode. Otherwise, one would need to first determine which portions were not applicable to the specific situation had behaviour for both modes be modeled as a single class.

The fourth class is named "sequential constraining condition". It is primarily designed to track constraining interface conditions that cannot be determined directly from system sensors. A common constraint is allowing only one part at a time in a machine. In the roll former system, the press entry conveyor may not have more than one part in at a time, but there is no sensor to indicate this. The Destacker cannot determine directly when it can unload a part onto the entry conveyor. During normal operation, it would need to monitor all of the press states where a part is present. This is undesirable from software principles of coupling and information hiding since the Destacker now needs to have detailed knowledge of the internal implementation of the press. Furthermore, after a loss of PLC power, there is no automatic manner to determine if a part is in the press. An operator would need to physically check the press. While the first

three classes do not remember the state they were in when the controller is shut down this class needs to. This requirement is forced by the fact that the constraining condition cannot be directly determined from sensors. A state machine is used to implement this function by keeping track of the constraining condition using persistent (the current state is saved when power is lost) states. The first three classes will directly control the devices through outputs that they are associated with, while this class typically does not drive any outputs. The first three classes exhibit distinctly different automatic and manual behaviour while this class does not. This class is primarily used as an interface coordinating components that are directly connected.

5.5 Applying the Abstract Model

Once a general abstract class model was developed, it was applied to represent the control code for the stamping/roll-forming work-cell. Figure 5-18 shows the implemented hierarchical object model based on the abstract class model. At the system-level, a **wrapper line** object of the system-level control class is implemented to coordinate the entire system allowing automatic cycling, emergency stopping, emptying out of parts, etc. Three **subsystem-level control** objects, which correspond to the partitioning of the system shown in Figure 5-10, are implemented to control subsystems. Each of the three objects contains a supervisory FSM to allow manual or automatic behaviour in each subsystem, as shown in Figure 5-13. Each subsystem-level object controls one or more component-level objects. The Destacker subsystem contains only the **sequence destacker** object of the sequential control class. Two nested state machines

model this object's behaviour. A manual sequence state machine, shown in Figure 5-16, is nested in the manual state of the Destacker subsystem-level FSM to specify the manual operations. An automatic sequence state machine, shown in Figure 5-15, is nested in the automatic state of the Destacker subsystem-level FSM to specify the automatic operations.

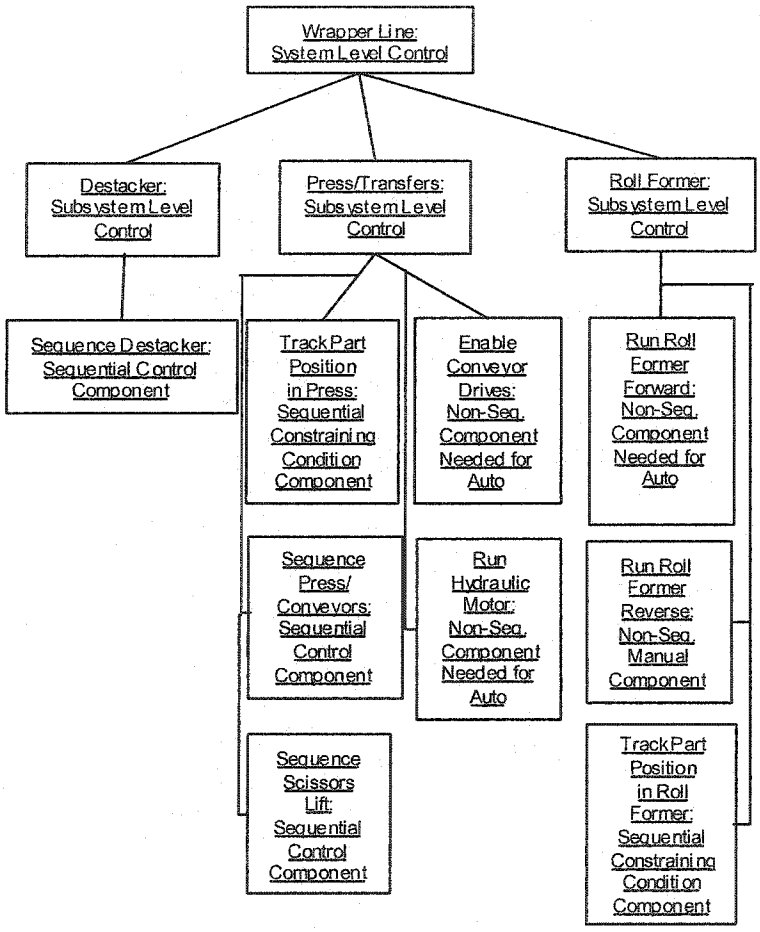


Figure 5-18: Stamping/Roll-Forming Work-Cell Object Model

The press-transfers subsystem contains five component-level objects. The **sequence press/conveyors** object of the sequential control component class specifies the sequencing of the press, its entry and exit of conveyors. An automatic and a manual state

machine controls the behaviour for auto and manual mode, respectively. Similarly, the **sequence scissors lift** object of the sequential control component class specifies the sequencing of the scissors lift with its automatic and manual state machines. The hydraulic press must have its hydraulic motor running before it can be run automatically. The **run hydraulic motor** object of the non-sequential component needed for auto class provides this behaviour. Similarly, the conveyors must be enabled before the subsystem can enter the automatic state. The **enable conveyor drives** object of the non-sequential component needed for auto class serves this purpose. Since only one part is allowed in the press subsystem at a time, and the part may be present but cannot be detected by the sensors, a **track part position in press** object of the sequential constraining component class is used. A state machine is implemented in this object to keep track of the loading of a part in the press and its delivery out of the press subsystem.

The roll former subsystem does not have a sequence (beyond simply running or being stopped), thus it does not contain any objects of the sequential control component class. This subsystem needs to control the roll former and its entry conveyor running in the forward direction to allow automatic behaviour, therefore a **run roll former forward** object of the non-sequential component needed for auto class is used. To allow the roll former to be run in reverse when the subsystem is in manual, a **run roll former reverse** object of the non-sequential manual component class is used. As in the press subsystem, the presence of a part can not be determined by sensors, therefore a **track part position in roll former** object of the sequential constraining condition component class is needed.

5.6 Summary

This chapter presented the general approach used to develop an abstract model for control code. One of the key issues addressed was the technology insertion of a new method for generating control code that automatically provides context enhanced monitoring data. The approach used was experimental iterative design by the author with feedback and evaluation from an industry expert control engineer. Experience previously gained in an effort to introduce the new technology of Automatic Guided Vehicles into an Automotive Assembly Plant led to the selection of Programmable Logic Controllers. Generalization from a specific case was used to derive the model. First control code for a simple single station system was written in a state based manner experimentaly showing its feasibility. The expert industrial control engineer provided the knowledge of the subtle details that needed to be addressed and fedback on how to present the method in a manner that made it acceptable to industry. After the first system was completed, a model for automatic behaviour of automatic stations which have a sequence of steps was developed. A general transformation from the model to PLC code was derived from the experiment. On completion of the first experiment, the expert control engineer judged it of sufficiently greater merit than the conventional practice for writing control code and adopted it as the standard method to be used in his company. Next, using the new methodology, control code for a complete representative system was implemented. This specific case was generalized to expand the model into one that addressed generating control code for typical machines of similar complexity. The modeling method selected was the Object Oriented Unified Modelling Language. The consistent use of the same

objects from the analysis phase through to the coding phase offered by Object Oriented Modelling was selected since this aids in comprehending the methodology. To address the issue of complexity arising from the large amount of detail addressed by a typical control system, hierarchy was applied. The next chapter presents the design and implementation of the application generator.

Chapter 6

Application Generator Design and Implementation

This chapter describes the design and implementation of the application generator. The application generator provides the mechanism for synchronizing the monitoring and control systems. Instead of separate groups writing code for each systems, the controls engineer interacts with the application generator at a higher level of abstraction, specifying the desired behaviour. The application generator then builds the specific model for the system and generates the control code as well as provides the monitoring system with the information needed to obtain context when collecting data. Automatic generation of the model, control code and monitoring information eliminates the multiplicity of implementations that occur when the control code is manually developed. The application generator, which is a knowledge system that behaves as an expert system, addresses the issue of configuring reusable components into a control plan. Capturing a control engineer's knowledge and encoding it in the software where it is used to build the model for the system and then generate the PLC code achieve this. This captured knowledge can be applied by a non-controls knowledgeable person, who is familiar with the operation of a machine, to generate control code for it. A pipe-fitter tradesman as presented in chapter 7 demonstrated this capability.

The application generator design went through several different iterations. The initial approach was an extension of a commercial UML tool. When this proved too

cumbersome, an object oriented language and an abstract set of classes for developing a user interface was pursued. However, the amount of time required to gain the knowledge required for such an approach exceeded the bounds of the thesis. A less flexible but simpler approach based on representing the object model with a relational database supplemented with a rapid prototyping language was finally used to construct the application generator. The user interface went through three iterations. It started with a fairly abstract representation with many user controls for examining the internal data structures of the generator. The complexity of the interface was then simplified leaving only the minimum needed. Finally, the interface style was completely rewritten when it became apparent that the targeted user needed a more concrete representation and a simpler method of navigation. The third iteration was based on Raskin's Zooming Interface Paradigm [Ras00]. The implementation of the "system" object is used to provide a representative detailed description of the different iterations in Appendix D. This chapter concludes with a description of a code generator, that uses the object-oriented model built by the application generator, to produce code used to create PLC Ladder Logic.

6.1 Evolution of Approach

The development of the application generator went through three distinct phases. The first was to expand the abstract model to a degree of detail where code could be automatically generated. A commercial UML tool would be used to build the abstract model with class diagrams. The tool would be extended with custom code to provide "wizards" or "agents" that would guide the user in instantiating the abstract classes into the required objects to implement the specific application. A code generating utility would be written that would use the object model to produce Ladder Logic code targeted at an industrial Programmable Logic Controller (PLC). Difficulties were encountered in attempting to add the desired functionality to the commercial tool. This lead to the second approach of writing a custom application where customization issues would not be present. Since the abstract model was object oriented and a high degree of graphical interaction with the target tool user was envisioned, Microsoft Visual studio version 6.0 was selected. To support the object oriented approach the C++ language was initially selected. The graphical user interface would be implemented using the Microsoft Foundation Classes (MFC). To keep the design and programming time within the bounds of a thesis, the user interface would replace the graphical representation of the UML model as found in the commercial software tool called Rational Rose from Rational Software, with lists and textual description. ActiveX controls would be used to store and present audio-video information captured by a video camera as previously described. The author attended a course on Microsoft MFC and started to write C++ code. The learning effort proved to be extremely difficult. The hope was that MFC would provide a

high degree of abstraction and easy to use interface to the Windows operating system functions. This proved not to be the case. Literature on application development for this platform [Kru98] indicates that becoming proficient in using Windows can take half a year of full time effort. One needed to learn more detail than time would permit. Within Visual Studio, Visual Basic (VB) provides a higher degree of abstraction than MFC. VB does not provide the flexibility, as rich a support for object oriented programming, although techniques [Lho97] are available to overcome this. One strategy suggested [Sol97] incorporating Microsoft Office Products particularly in the user interface. This would minimize the amount of custom code required. Since serialization of objects is not supported in VB 6.0, a method was needed for storing them. A relational database can be used to represent both a class model as well as instantiated objects. This approach was used to implement the first version of the application generator. Upon completion, experiments detailed in the next chapter were conducted with the targeted user. It became apparent that the user interface style was presenting a major obstacle. The experimental subject did not exhibit difficulty in understanding the abstract model and how an actual machine would be represented. However, when faced with implementing the control code using the application generator, he was constantly struggling relating the various screens, drop-down boxes, selection lists and command buttons, to the model. The problem lay in the need for the user to build an image of the model in his mind and then construct it by communicating with the application generator. This problem was not apparent when working with the RLM control engineer. Engineers are trained to think in terms of abstract models and use them in their day-to-day activity. The RLM control

engineer was able to comprehend the model and apply it without even using an application generator. All that he required was an editor that allowed him to cut and paste PLC Ladder Logic code segments representing the objects. He formed the model in his mind and constructed the control code by selecting code sections and configuring them to suit the specific application. For this research, the experimental subject was a pipe-fitter trades-person. The author's experience supervising a crew of trades has led to the observation that they typically work with concrete objects. Abstract thinking is not usually required in the tasks they perform. Unfortunately, the user interface of the application generator had been designed for someone who was adept at mental abstract modelling. To determine if this was truly the problem, a portion of the user interface was rewritten using the Zooming Paradigm (ZIP) [Ras00]. This approach is far more tangible and less abstract than the approach that had been used. Experiments outlined later, showed a marked improvement in the trades-person's ability to use the application generator to create control code.

6.2 UML Tool Extension Approach

The first approach used to develop the application generator was to extend a commercial Unified Modelling Language (UML) tool [Qua98]. The abstract model would be represented by the series of class diagrams developed in the previous chapter. How these are to be applied and instantiated into an object model from which control code could be automatically generated would be provided by Use-Case-Scenarios (an overview diagram showing a systems intended function, its surroundings and the

relationship between them), sequence (a view of how a specific system function is implemented showing the sequence in a time based order of actions/reactions and where in the system they are initiated and serviced) and collaboration diagrams (a big picture view showing how the objects are linked to provide a specific function). A fully developed sample application would further explain the methodology. This section expands the abstract model presented previously to the level of detail needed for automatic code generation. A discussion of the approach outlines the difficulty experienced and the reasons for switching.

The initial approach was to use Rational Rose as the basis for the application generator. The abstract model was expanded in detail and represented by a UML class model in the Rose modeling package. The plan was to apply the extensibility features of the package to augment it so that it could be used to create a control plan. One approach proposed in the MOBIES research effort [Gue00] was to write code to monitor the user's interaction with the package. The classes would be provided as a starting point. The user would then instantiate objects from these classes, interconnect them and provide the details needed for the specific control application. The various diagrams available in the UML, such as collaboration, sequence, state etc. would be used for this. While the user was performing these tasks, the monitoring code would check for incorrect uses and prevent them. The Rose package provides a graphical method to create object models as well as some rudimentary syntax checking at the UML level. However, it does not comprehend the domain specific model. It does not understand how objects need to be connected. How the required functionality is partitioned and what objects need to be

present is the responsibility of the user. This knowledge is not captured by the package. The user is expected to understand the various classes, when to instantiate them into objects and how the objects are to collaborate. The proposed MOBIES approach of extending Rational Rose to constraining the user to correct configurations addressed the construction of a correct specific application object model. This is of great advantage when writing an application for automatically generating code from the object model. One is relieved of need to check for model correctness and handling associated errors. However, the user still needs to acquire thorough detailed knowledge of the abstract model and determine how to express the requirements of the specific application being implemented. The user needs to know how to map the requirements into objects. In addition, the user needs to understand UML and how to apply it. The targeted user for the MOBIES project was an engineer so that type of knowledge might reasonably be expected. The targeted user for this thesis is a trades-person familiar with how machines operate but not expected to know any software modeling theory. To overcome these issues this thesis would provide "wizards" or "agents" instead of directly using the Rose tool with the added software constraining the design. The "wizard" extension code would provide a series of screens soliciting from the user the information needed to create the various UML constructs. This information would be then used by the extension code to automatically create the UML diagrams within the Rose tool.

The primary purpose of the control system is defining the dynamic behaviour of a work cell. As the design of the various wizards for the application generator progressed, a large variety of types of dynamic control system behaviour were identified. This lead

to numerous entry forms that the user needed to navigate through and comprehend to select the particular behaviour he needed. The OO paradigm uses classes with specialization and generalization to address similar complexity of tangible objects. This concept was extended so that the complexity of specifying the system behaviour could be simplified by partitioning it into separate behaviour classes. By assigning these behaviours to specific classes, a standard method for implementing them can be achieved by the application generator. Further, this simplifies the presentation of the concepts to the intended user of the generator allowing selection of the desired behaviours at a higher level of abstraction. Once selected, they are applied as defaults, removing the need to specify them individually. For example, if the user is enumerating the sequence of steps to create the FSM for a sequential control component, having selected a single step capability removes the need to request it for every step.

Applying object-oriented analysis resulted in a class structure of control behaviours shown in Figure 6-1. This is an unusual and believed to be a novel use of class diagrams, which typically represent tangible objects in the application domain. The advantages of modelling class relationships were applied to managing the complexity of comprehending and specifying the entire range of the system dynamic behaviour in a compact manner.

System dynamic behaviour can be specialized into the three classes of automatic, manual and off. The automatic class of behaviour is used to control the automatic manufacturing process the system is designed to perform. Maintenance personnel

manipulate the system in a non-automatic manner such as for set-up, repair, adjustment, etc. by using the manual class. The off class is used to freeze the system in its current position. The simplest form is placing the system into a totally passive, non-energized mode, with the controller powered up and aware of the particular state of all FSMs. As mentioned earlier, this form may produce undesirable results. In the most general case, the off mode implies stopping motion while maintaining the current state.

Emergency stop modifies the automatic and manual behaviour, placing the system into a safe condition in the context of the current automatic or manual state. When the emergency stop condition is no longer present, automatic and manual behaviour may often be resumed from where it was interrupted. Dry cycle modifies automatic and manual behaviour to allow sequencing of the system as if a part was present. It is used in automatic for validation/buy-off by letting the system continuously cycle in the normal manner for a long period of time without actually making parts. In manual it allows setting up and calibrating without the need of a part.

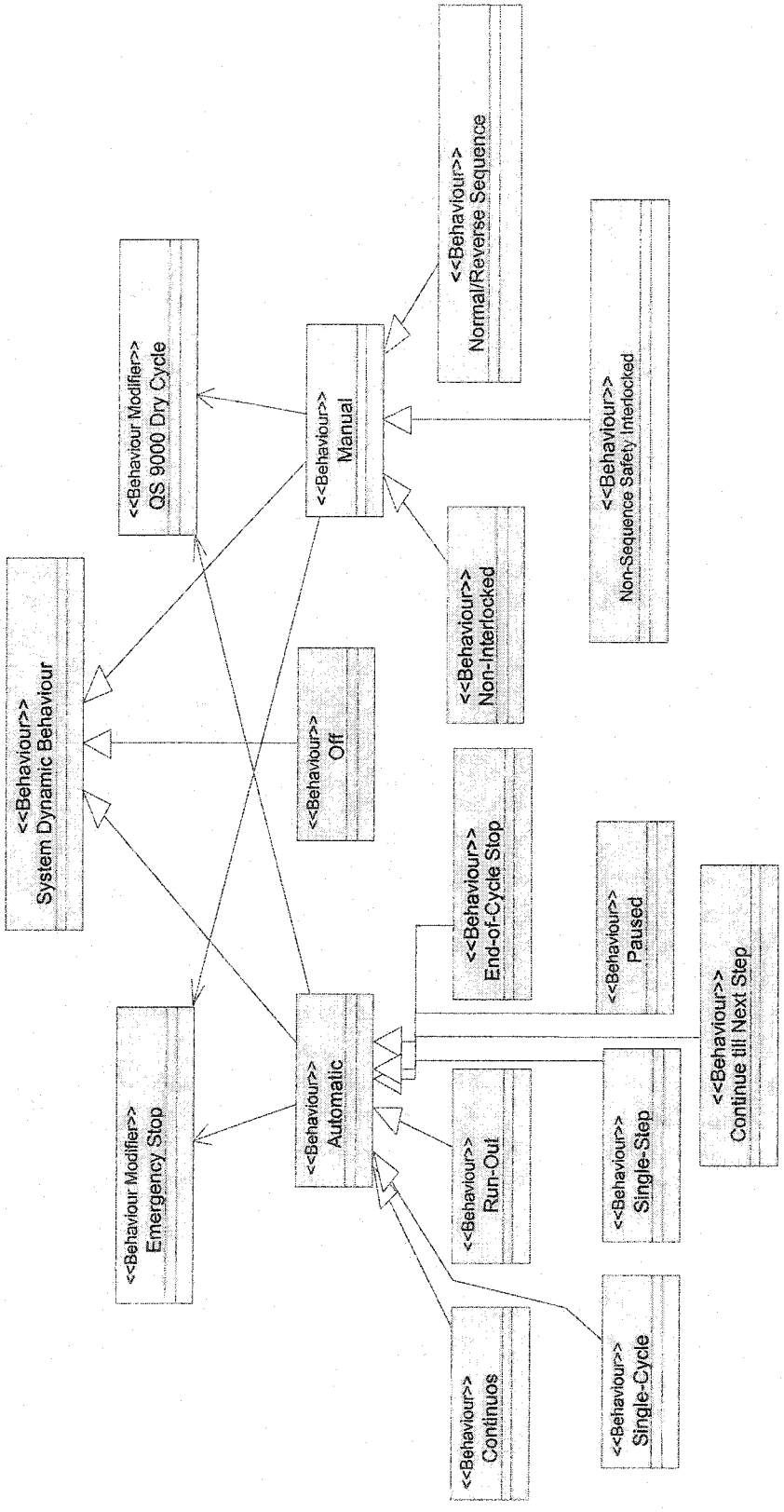


Figure 6-1: Class Structure for Control Behaviours

Manual behaviour has three specializations. Non-interlocked allows a maintenance person direct control of any output without regard for potential damage of equipment or hazardous conditions. It is used only by personnel who have a very deep understanding of the system in situations of severe system failure, which cannot be resolved by any of the other behaviour classes. The RLM control engineers who designed the work-cell do not usually implement this class. Non-sequence safety interlocks restrict control to actions, which will not damage the system nor create hazardous conditions. The normal/reverse sequence allows the maintenance person to step through the normal automatic sequence in either the forward or reverse direction (when possible) having manual control of the actions that are associated with each automatic state.

The "End of Cycle Stop" function stops the entire work cell when all of the subsystem sequential components have finished processing the part they are currently working on. The single cycle specialization stops each sequential component when it has reached the next "home" step. A home step is a convenient stopping location where the machine is out of the way and in a fairly safe condition. Whenever the user selects single cycle behaviour, the application generator automatically creates the supervisory FSM and expands each home state in the device as shown in Figure 6-2.

Single step is the sequencing of the component one step at a time performing all of the part processing that would normally happen in continuous operation. Whenever the

user selects single step behaviour, the application generator automatically creates the supervisory FSM and expands each automatic state in the device as shown in Figure 6-3.

Paused is used to maintain the position of the component while stopping the sequence at the end of the current step. It is used when single cycling or single stepping to pause the sequence and allow subsequent continuation in the continuous, single cycle or single step manner. The continue-till-next-step specialization is used to address the case where the action of a step must be completed even though automatic operation has stopped.

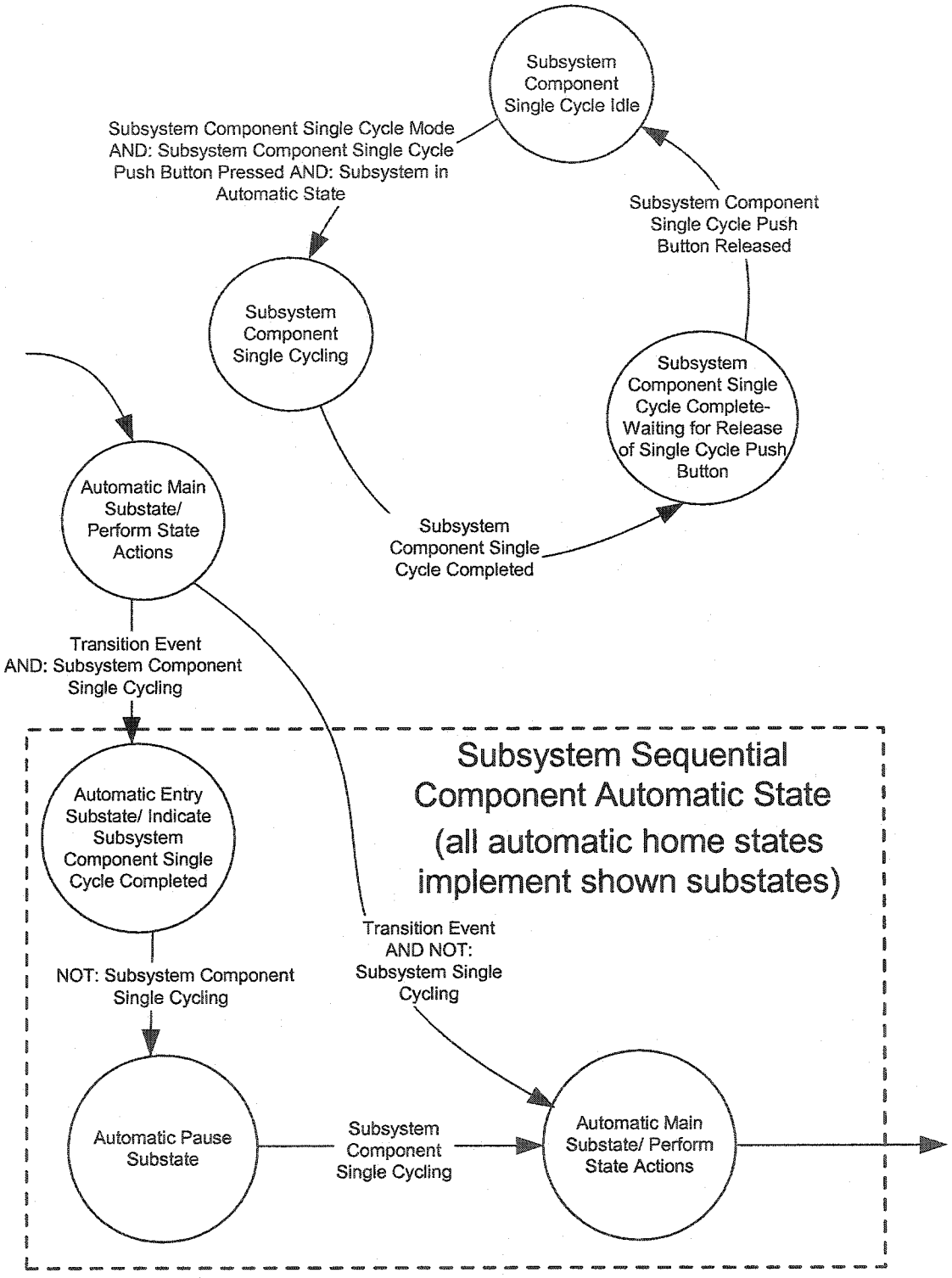


Figure 6-2: Single Cycle Controlling State Machine and Home State Expansion

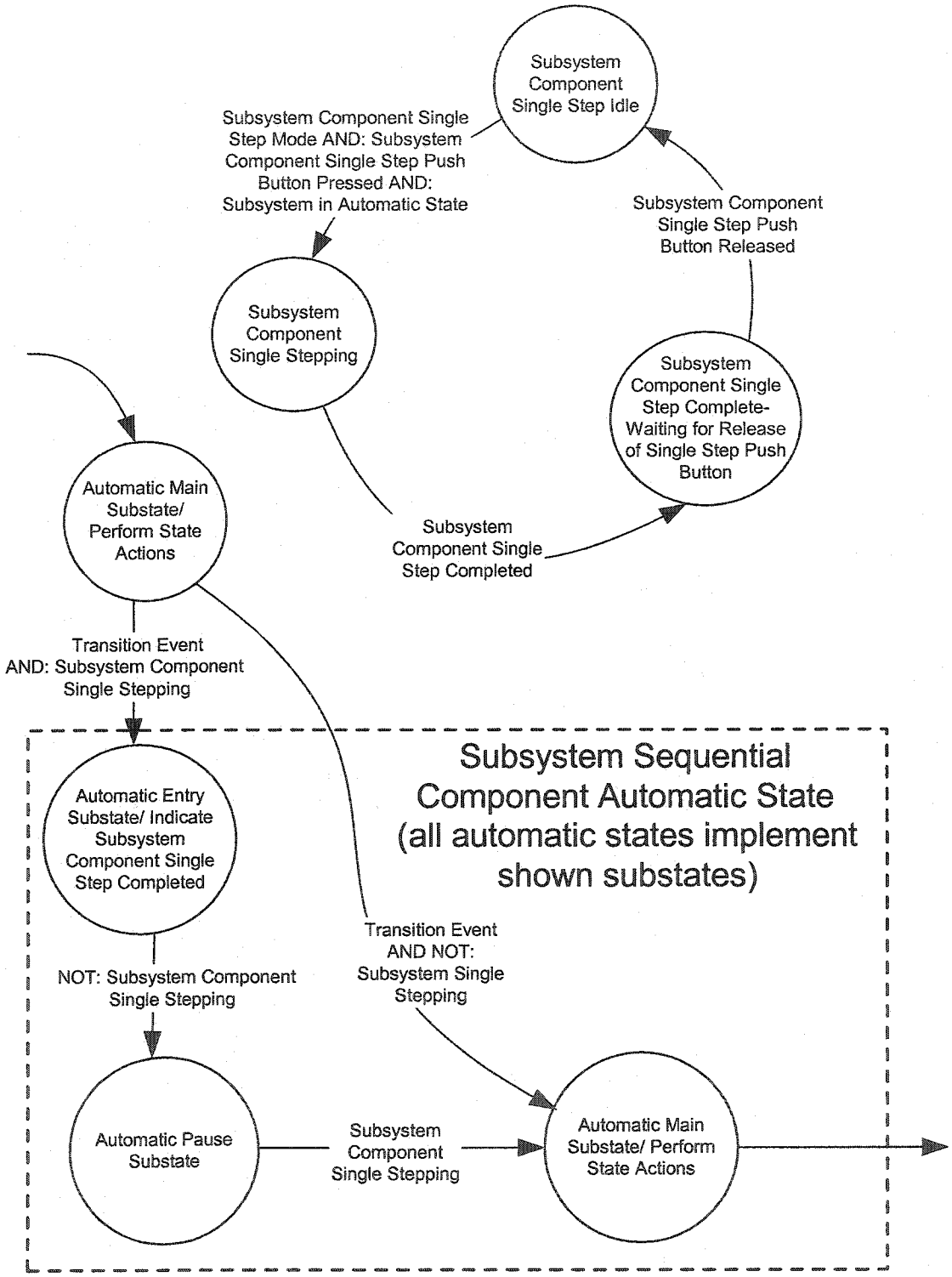


Figure 6-3: Single Step Controlling State Machine and State Expansion

The insight gained from the object-oriented analysis outlined above, led to the expansion of the class structure for the state machine for sequential control components introduced in previous chapter as shown in Figure 6-4. The dry cycle behaviour was implemented by specializing the input class into the part present event qualifier and part present indicator. The indicator class uses the part present input to indicate that the part has arrived. This is typically used as an event, by itself, to trigger a state transition. In dry run, a timer set to the time it normally takes for the part to arrive replaces this input. The part present event qualifier is used in conjunction with another signal that signals the event such as a part carrier arriving. The part present is used to determine if any processing is needed. For this class, the part present input is always forced to its active condition.

Different kinds of states were modeled as shown in Figure 6-5 and different kinds of state transitions as shown in Figure 6-6. The resultant classes encapsulated standard code to implement the specific behaviour. The user simply selects the kind of behaviour they need. The application generator maps the selection to the appropriate specialized classes and then generates the code that implements the behaviour.

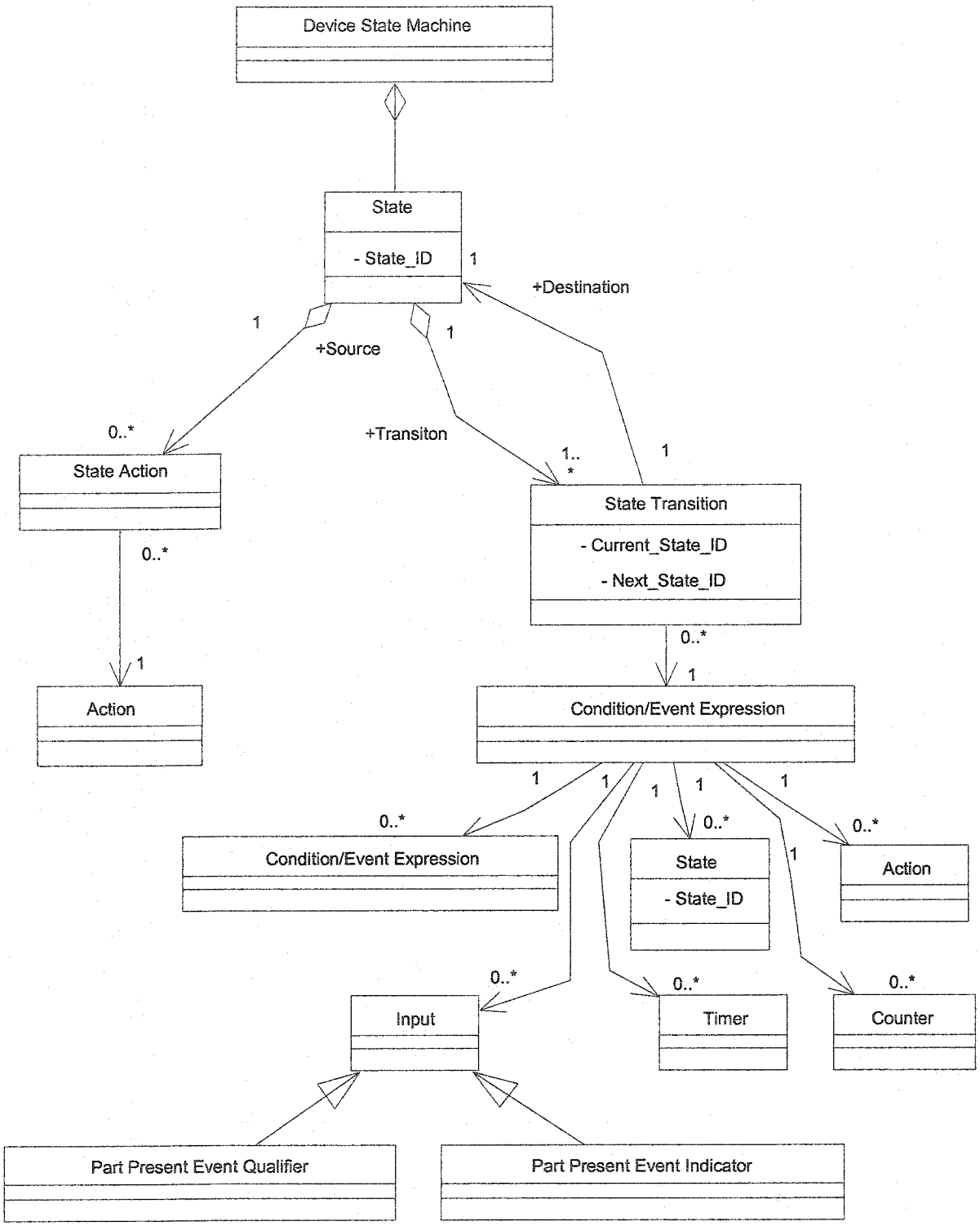


Figure 6-4: Expanded State Machine Class Diagram

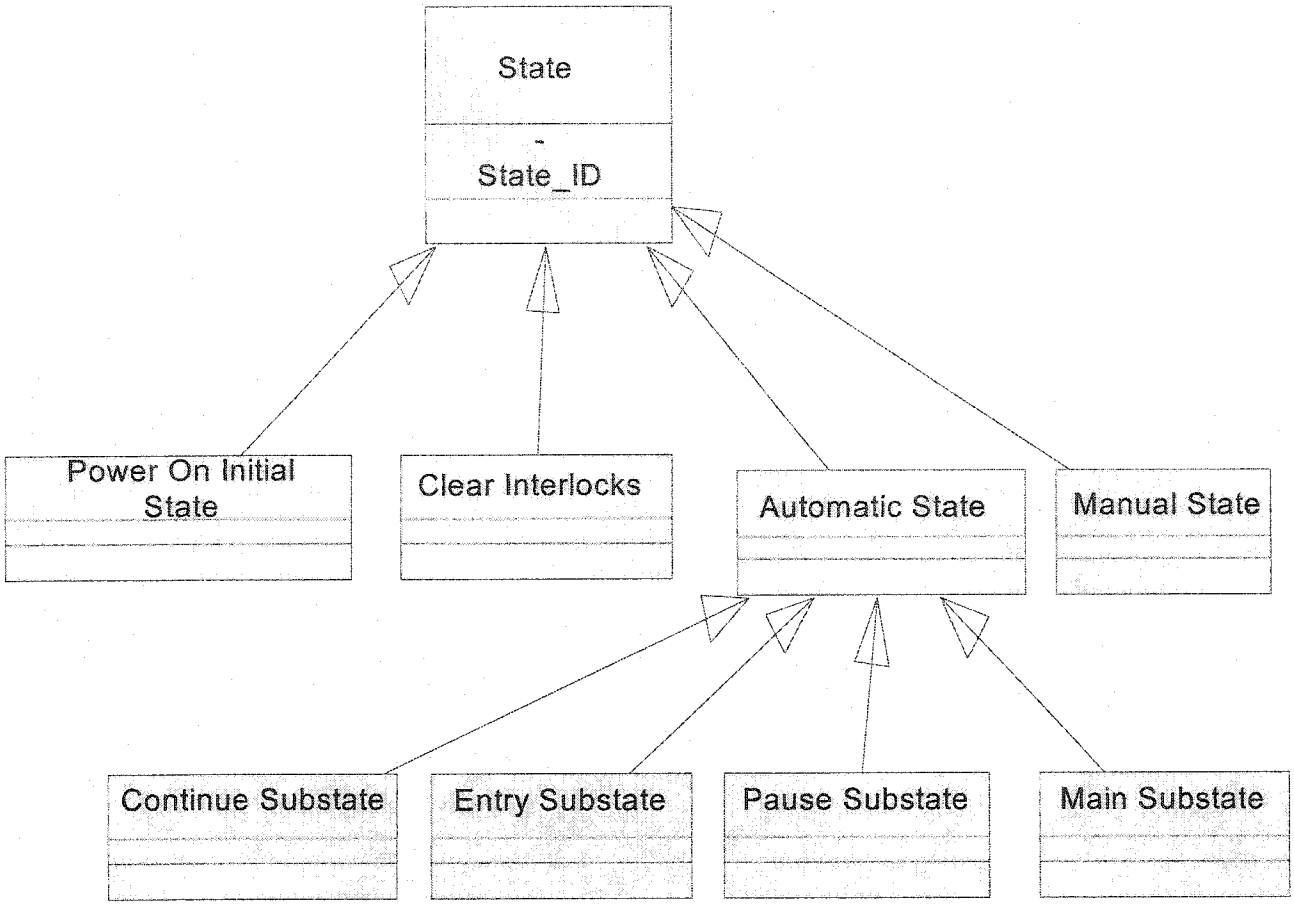


Figure 6-5: State Specialization Class Diagram

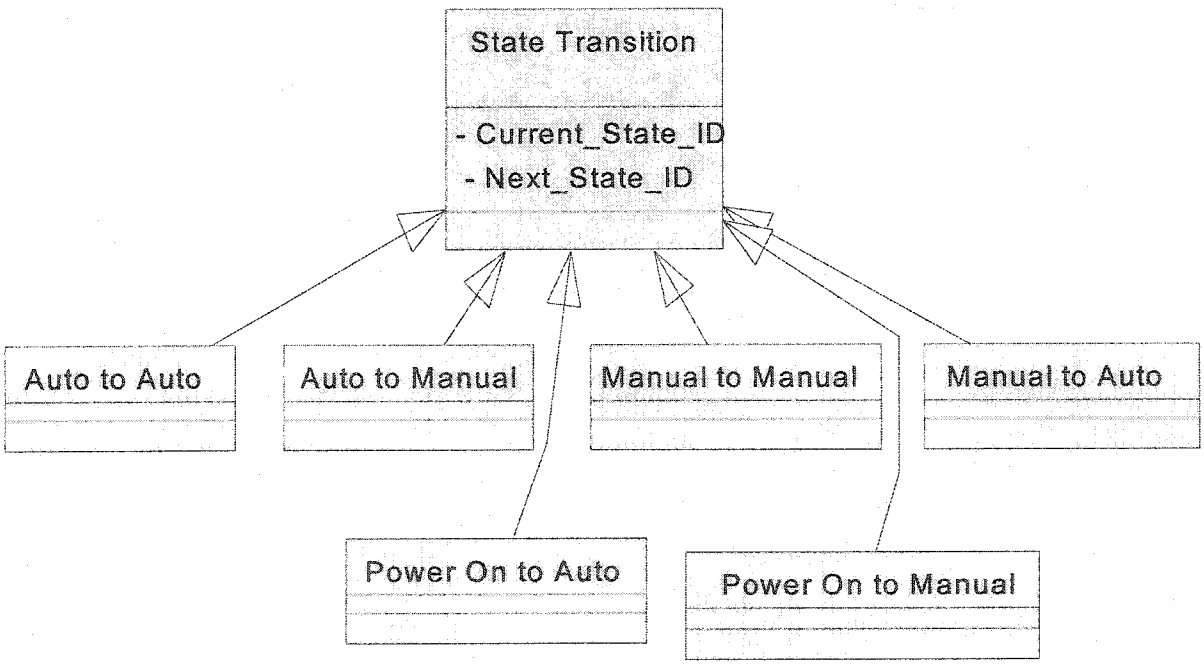


Figure 6-6: State Transition Specialization Class Diagram

6.3 Relational Database Design

6.3.1 General Approach

Once the extension of a commercial UML tool and then the use of C++ to implement the classes and objects required for an application generator was deemed to be too cumbersome, a relational database approach supplemented with Visual Basic was evaluated. In theory, one can represent a class diagram with relational database tables to store the attributes or properties and the index fields and links between the tables to represent the relationships between classes and objects. Visual Basic (VB) code functions and subroutines can be used to represent the class methods. The author chose to group the properties and methods of the major classes inside of VB forms. The form could serve as a container collecting all of the properties and methods of a class as well as implementing the concept of private and public access. Within a form for a specific class, various graphical controls as well as variables would be bound to the relational database table representing the class. The fields in the table were used for the properties of the class. To implement relationships, index fields with links to other tables were used. Since the strength of the OO approach comes from using objects that are readily recognizable in the real world, the author took the general approach of implementing the application generator based on such objects. The overall strategy was to provide the user with forms for all of the major classes. The user would interact with the forms to create or "instantiate" and configure the objects that were required for the particular control application. When the user created an object of the specific class represented by the form, a row would be added to the associated database table. The form would then select

this row providing access to the newly created object. The controls bound to the database table would provide access to the individual fields. The data bound controls allow the user to select values from lists or manually type in the information. Automatically generated values would be provided by code in the form, which would store these values into the record. Since the development was on a Microsoft platform, Microsoft Office Objects such as Word documents were utilized to provide a rich functionality the user was familiar with without the need to write custom code. Other vendor's products could have also been used with different trade-offs between familiarity and licensing costs. A word document object could be stored as a field in a record. The document then served as a container for various forms of user supplied information. The audio-visual knowledge capture previously mentioned using video was implemented in this manner. A video clip would be digitally captured and the resultant file embedded in the Word document. This automatically provided a control for playing the video clip without requiring writing of any additional special code. Objects created by other tools could also be embedded and manipulated in the same document. The user could easily add any additional information to the document utilizing skills he already has. When the model was complete, a method of the application itself would then be invoked to generate the control code in the form of PLC Ladder Logic.

The relational database selected was Microsoft Access for three primary reasons. First, being part of the Office suite, it provided a high degree of integration. It recognized objects such as a Word document saved as a field in a record. One could launch the application that supported the object directly by simply clicking on the field in

the database. Changes could be made, copies taken or new documents substituted without needing a fully functional application generator. Secondly, Access provided a user interface directly into the database. This was particularly useful during development and debug. Large sections of the class model could be implemented and objects created directly through the Access user interface. A design approach could be quickly prototyped and evaluated without needing to make changes to the application generator. Utilities were available to view and analyze the data representing the objects. By using sequential query language (SQL) (a standard language for retrieving, relating and manipulating database objects) procedures available in the Access user interface, relations could be tested without writing special code. Changes could be evaluated directly though the interface. In contrast, the development approach of using C++ to create the classes and objects was much slower. Code was needed for everything and special routines needed to be written to allow viewing and manipulating the objects outside of the application generator. The third reason for selecting Access was its availability and extensive integration with VB. Using the data control, user interface controls such as list and text boxes could be bound directly to the fields of a record. No additional code needed to be written. Using the VB with Access approach for developing the application generator resulted in progress several times faster than with C++. One did not have the fine granularity of control provided by C++ nor the full features of a true object oriented programming language such as inheritance. However, the higher level of abstraction when working with Windows graphical interfaces and the smaller learning curve more than compensated.

6.3.2 Evolution of Design

Once the relational database for the object model with Visual Basic for the user interface approach was selected, the author proceeded to write the application generator. The design evolved through three iterations. This section presents a summary of each iteration, or pass, while Appendix D provides a detailed discussion.

The first pass began by implementing the class model as database tables with links between them for the required associations. Then forms were written for the major classes. These were used to provide the user with the means to create and manipulate the objects. The original plan was to use a single pass through the application generator. It was envisioned that the user would enter all of the information needed for the complete control program and then generate the code. The output would be a text file containing segments of the rung logic and a symbol table as shown in Figures 6-7 and 6-8. The rung-logic text file would be manually entered into a PLC Ladder Logic editor while the symbol file would be read in directly. Should corrections or modification be required, the user would enter them into the application generator and totally regenerate the PLC code. This would result in new assignments of symbolic names to PLC control bits (addresses). The application generator was to be developed iteratively, one section relating to a class at a time, tested and refined by the author. As each class was completed a new section would be added, up to the point where code to control the majority of the functions for the wrapper line described in the previous chapter, could be generated. The author's initial evaluation of the partially implemented generator showed that the amount of manual entry and re-entry of the PLC rung logic into the PLC editor was prohibitive.

The targeted user would have far less familiarity then the author with the wrapper line. Thus the targeted user would be expected to require more re-work with the associated re-entry. It was felt that the additional time simply spent in typing would obscure evaluation of the difficulty to learn the methodology. The single pass approach was deemed unsuitable. The application generator was then redesigned to allow an iterative approach. This approach is often cited in OO literature as highly desirable. It allows the user to design a small portion of a system, implement and test it. This is in keeping with Miller's [Mil56, Bad94, Shi94] research that indicates a limit of the amount of information a human mind can process at one time. The iterative requirement added a fair amount of complexity since the user would be allowed to keep the PLC code previously entered that did not need changes intact and not need to re-enter it.

```
B3/20
[1]---( )--|
I4/0  B3/21
|---] [----] [----[3]
I4/3  B3/22
|---] [----] [----[2]
B3/23
[2]---] [----[1]
B3/8
[3]---] [----[1]
B3/20  B3/24
|---] [----]/[----[1]
B3/25
[1]---( )--|
I4/2  B3/24
|---] [----] [----[2]
B3/23
[2]---] [----[1]
B3/25  B3/22
|---] [----]/[----[1]
B3/21
[1]---( )--|
B3/20  B3/25  B3/24  B3/22
|---]/[----]/[----]/[----]/[----[1]
```

Figure 6-7: Rung Logic Segments Text File

```
B3/0 System E-Stopped
B3/1 System Cycling Automatically
B3/2 System End of Cycle Requested
B3/3 System Run Out Requested
B3/4 All Subsystems in Auto
B3/5 EOCR & All Subsystems Home
B3/6 System Automatic State
B3/7 System Manual State
B3/8 Destacker Auto Mode Selected
B3/9 Destacker in Auto State
B3/10 Destacker in Manual State
B3/11 Destacker Safe for Others to Cycle
B3/12 Destacker In Home Position
B3/13 Destacker in OK enter Auto Position
B3/14 Destacker Drop Out of Auto
B3/15 Destacker Lost Auto Can Resume
B3/16 Destacker Sequence Local Auto Cycling active
B3/17 Destacker Sequence Auto Cycling active
B3/18 Destacker Sequence Auto Activity active
B3/19 Destacker Sequence Auto Idle active
B3/20 At Stack State
B3/21 At Power On State
B3/22 At Moving towards Part Stack State
B3/23 Destacker Sequence Auto Cycling active
B3/24 At Moving towards Transfer Location State
B3/25 At Transfer State
```

Figure 6-8: Symbol Table Text File

Debug capabilities were added to the user interface forms for the first pass of the application generator. This was in the form of additional visual control that allowed navigation and modification of the underlying database tables. This functionality was not strictly required by the application generation function, however the author felt at the time, that it would provide the experimental subject with a greater insight into the use of the tool. Unfortunately this did not prove to be the case. On completion of the first pass a pipe fitter tradesman was chosen as the experimental subject. The methodology of

partitioning the system based on the abstract model was explained. The tradesman had little difficulty in understanding the concept. He stated that the model closely represented the way machines are controlled in the plant. He then proceeded to work through an example training system. When interacting with the application generator he experienced difficulty relating the various forms with their numerous individual controls with the abstract model. The feedback of the tradesman was to simplify the user interface as much as possible. The nomenclature used in the model as well as the application generator was also confusing. Long descriptive names such as "sequential component which must be active to allow automatic behaviour" though accurate, were difficult to relate to in the actual machine. To correct these difficulties, the application generator underwent a redesign referred to as the "second pass".

The second pass of the application generator removed the debug graphical interface components. The remaining ones were reorganized into logical groupings by physically locating closely related controls in clusters. The author utilized the tradesman's critique to replace the long descriptive names with shorter ones that would likely be familiar to other trades-people. Having completed the second iteration, the experimental subject started the training again. The progress seemed much slower and more difficult than the author expected based on the assessment of the tradesman's understanding. The partitioning of the control functions was understood, and what he wanted to specify was correct. Relating this to the application generator seemed to be the problem. The two major complaints that arose involved the navigation through the screens and the amount of abstract thinking required. As the objects for controlling the

specific training machine are specified, they become individual records in various database tables. The user is visually presented with only one record or object of a particular type at a time. Each major different type is on a separate form. The objects are typically selected from lists that are populated based on selections made on the other forms. For example, to work with a specific component's state machine, the user must navigate to the system form, select the appropriate system, navigate to the subsystem for to select the subsystem, and then to the component screen to select the component that contains the state machine. The relationships of the abstract model are stored in the links in the database and are reflected by changing the contents of the selection lists on individual forms. They are not visually represented in the application generator in a compact manner. Thus, to keep track of what has been built at any moment in time, the user needs to form a mental image of the model. Unfortunately, the user interface of the application generator had been designed for someone who was adept at mental abstract modelling and proved unsuitable for the pipe-fitter in his capacity as the experimental subject.

To overcome the need for abstract thinking and simplify the navigation through the model a third iteration of the application generator was implemented. A portion of the user interface was rewritten using the Zooming Paradigm (ZIP) [Ras00] to determine if a more concrete style of presenting the model would address the difficulties experienced by the pipe fitter. This approach is far more tangible and less abstract than the approach that had been used. Instead of needing to view a number of screens and keep track of objects in ones mind, the entire system was displayed on a single form. The ZIP paradigm uses

the analogy of a landscape and flying in an airplane to navigate it. The ability of a human to find their way through complex landscapes as can be observed from the ease in which people find their way from work to home. People have the ability to find landmarks and assimilate them to the degree that they can navigate at a subconscious habit level without being consciously aware of the process. To get an overview of the entire landscape one would fly an airplane higher or "Zoom Up". The desired location would be seen and one would move to it. To get at the detail one needs to simply fly lower or "Zoom Down". Once at the desired level, one would navigate to the desired object and zoom down again to be presented directly with all of the functions that can be performed on the object. This is in contrast to the use of menus with drop down lists that bring up new forms with their own lists. To perform a function in the traditional interface a "maze" of menu and link navigation is required. The ZIP instead provides a visual representation or "landscape" of all of the objects with the ability to move up and down in amount of detail presented. The final iteration of the application generator used this approach. As the user created new portions of the model, they were placed on the "landscape". Panning the landscape to display a portion at a time was used to address limited display screen real estate. To manipulate detailed information of a particular object, such as to create new states for a components state machine, the user would zoom up and to locate the component. The he would zoom in, which would replace the screen with a more detailed screen showing the portions of the located component. The state machine would be located and a button on the screen would be used to create the new state. Then the user

would zoom out which would replace the more detailed screens with ones of less detail but more landscape up to the highest level where the entire model is presented.

6.3.3 Input Information Class Model

One of the difficulties encountered when analyzing current PLC control code arises from the programming practice for symbolic naming of input conditions. Typically the PLC tools allow only one symbolic name for an input. The symbolic name usually given is used to describe the input when it is present or active and is drawn as a normally open contact. The same symbolic name is also used in the PLC code in the negative sense, when the input is not active or not present and drawn as a normally closed contact. To determine what the absence of the signal implies requires the reader of the code to mentally perform the logical negation since the same name is used. In addition, the same input condition may have a different meaning in a different context. For example, in the wrapper line Destacker, a sensor is used to determine if a part is present in the part gripper. When the Destacker is unloading a part onto the press conveyor, this sensor being in the non-active condition is used to indicate that the part has been successfully unloaded. However, when the Destacker is carrying a part and traversing over the aisle to reach the conveyor, the same sensor being non-active is used to indicate an emergency stop condition of a part being dropped and possibly dragged. Unfortunately, only one symbolic name can be used, limiting the ability to accurately describe the condition in both situations. To alleviate this problem, a separate class was used to provide the ability of having multiple symbolic labels applied to the same input when it is used in different context. The table representing this class is named

"Input_Condition_Descriptions" and is used to store a separate description for both the presence and absence of the input that describes its function in the context within which it is used. This table serves as another source of context enhancement to a monitoring system in addition to making the control code simpler to understand.

Analysis of typical manufacturing control panels reveals another relation that inputs participate in. At times several inputs are part of the same input device such as the system mode selector switch. There are also situations where an input will be grouped with an output in the same device. A typical example is a backlit push button, which is a push button switch that has a light inside of it. The light is actually controlled independently of the user pressing the button. The light can be used to bring the user's attention to a button that needs to be pressed to cause a needed action to occur. It may be used to indicate that the pressing of the button has resulted in condition that remains when the button is released. An example is the system end-of-cycle-stop. When the system is running automatically and the user presses this button, its internal light will be turned on indicating that the system is still running but will stop when all of the subsystems have reached their end-of-cycle-stop positions. When using a touch sensitive video monitor such a combination has even more potential to communicate with the user through the use of different colours as well as flashing. One or more inputs can be part of or associated with an input/output device. Each input can be specialized with one or more context specific descriptions for both the case when the input is on and when it is off. The condition/event expression class diagram previously shown in Figure 5-4 now needs to be changed. Instead of inputs being directly contained as elements of a

condition, they are now separated by a context specific description. This provides a higher level of abstraction and a richer degree of freedom in describing the function of the input as used in the context of the specific condition/event expression rather than using a single symbolic name for the input for all uses. The implementation of this class structure in the relational database is shown in Figure 6-9. The "input_output_classes" table represents the top-level "input/output" class. Within it is found the descriptive information for each different kind of input/output object. The "Input_Output_Class_ID" indexed field is used to link instances of the class with the class description. The line joining the "Input_Output_Objects" table "Input_Output_Class_ID" field to the class table shows this relationship. This table is used to store information on the actual devices or in OO terms, it is used to store the objects that instantiate the "input/output" classes. Each device represented by the class is stored in the objects table. The "Input_Output_Object_ID" indexed field uniquely identifies each device in the machine. The "Input_Object_ID" field in the "Input_Info" table links the individual inputs to the device they belong to. This table represents the "Input" class and the "Input_ID" indexed field uniquely identifies the individual input. The "Context Specific Information for Off" as well as the "Context Specific Information for On" classes are both represented by the "Input_Condition_Descriptions" table. This table is linked to the "Input_Info" table through the "Input_ID" field. Since each context can at most have a single on or off, both descriptions can be stored in one table. To locate the input condition description for a specific condition, the "Input_Condition_Description_ID" as well as an indication of either the present or absent referring to the "on" and "off" condition are specified.

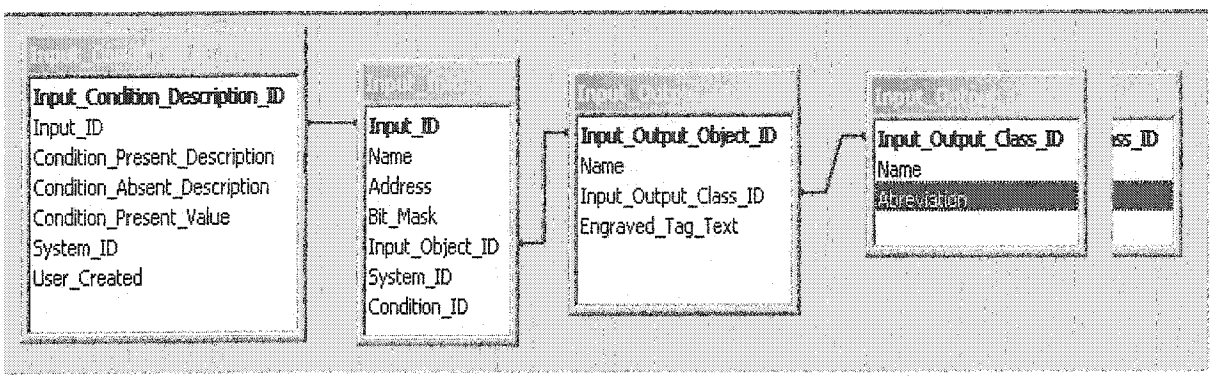


Figure 6-9: Object Diagram for Input Information

6.4 Code Generator

The design of the code generator was based on finding mappings or transformations from the objects the users created in the application generator to PLC Ladder Logic code. The number of different classes of objects that the user could specify was kept to a minimum. This resulted in a tractable solution where the number of transformations was reasonably small. While manually developing Ladder Logic with feedback from the RLM control engineer, patterns or common topologies of the code became evident. The use of OO specialization of a class was used in the development of these topologies. This section will detail the generation of PLC Ladder Logic for the various types of state shown in Figure 6-5 as a representative sample of the code generator design.

The code that transforms the description of a required state into the actual code was knowledge based (encoded how a control engineer would write the code as a series

of decisions). The application generator obtained the information needed to make the decisions from the user and stored it as attributes of the state object. As the series of decisions is evaluated, portions of the Ladder Logic code are generated. All possible topologies are known ahead of time and the user is constrained to state types for which implementations already exist. Thus the code generation is simplified to identifying the distinguishing features and then selecting the appropriate predefined code segments. During the design of the code segments, it became evident that the targeted language did not need to be exclusively PLC Ladder Logic. Initially, the code generator was envisioned outputting Ladder Logic directly. However, if an intermediate form of representing the logic could be used, other languages could be targeted at a later date. Ladder Logic is a specific manner of implementing Boolean logic. If the Boolean equations are generated, any language that supports Boolean arithmetic could be used for the final implementation.

Object Oriented methodology was applied in the design of the intermediate logic representation. The class that contains the logic expression, or Boolean equation, was named a "condition". This name was assigned because it was believed to be one of the simplest and most familiar use cases for a tradesperson. In the abstract model a condition is used to allow a transition from one state to the next to occur. A tradesperson is familiar with the concept of manipulating a machine into its home position. The term condition was used to denote the case of all of the "home position" sensors being in their specific condition to indicating that the machine was "home". This is typically represented by a Boolean equation so the term condition should be easily related to.

Boolean Equations are typically evaluated in a sequence from start to finish. Thus a means was needed to identify the start and finish. When mapping to Ladder Logic, the start is the vertical bar or rail on the left hand side representing one side of a voltage source. A restriction was placed on the definition of the end point to limit the amount of code patterns required. In Ladder Logic the finish end is the rail on the right hand side, representing the other side of the voltage source. The element that is energized by current flowing between the two rails when the logic elements in between provide a complete continuous circuit is typically drawn next to the right hand rail. However, one can place more than one such driven element in parallel and one can place other elements between the driven one and the right hand rail. Typically this driven element is a coil but it can also be a more complex element such as a counter or timer. The RLM control engineer stated that the extra complexity of more than one driven element in a circuit and placement at other than adjacent to the right hand rail was not needed. In his experience such topologies could be expressed with rungs with single driven elements adjacent to the right hand rail. This observation was used to make the simplifying constraint on the definition of the end point of always having the driven element placed directly adjacent to the right hand rail. Thus, the end point is defined as a "node" or connection point of the left hand side of the driven element. This node was assigned the value "1". The node concept was then applied to the opposite rail. Since this rail is considered to be the start of the logical expression the node representing connections to it was assigned the value "0". Node "0" is the left hand node of all elements connected to the left hand rail.

Ladder Logic structure served as the basis for the class model used by the code generator. Examination of a typical "network" will show a topology of connected line segments containing logic elements such as normally open and closed contacts, inputs and outputs. Elements that are connected to each other in series in horizontal line segments representing the logical "AND" function. The logical "OR" function is represented by horizontal line segments drawn in parallel, arrange above one another and connected with vertical lines at each end. Thus the smallest individual section is a horizontal line of logic elements between vertical connecting lines, or when there are none, between nodes 0 and 1. The "Condition Subchain" class represents this smallest section in the OO model. The relation between the subchains and the containing condition is made on a unique identifier in the condition named "Condition_ID". Each condition subchain has the containing condition's ID stored internally. The concept of labelled nodes is used to provide a mechanism for assembling subchains into a logic equation. As mentioned previously, node 0 is the start and node 1 is the end of the equation. Each condition subchain has a left node and right node associated with them. The general approach to build the equation would have the code generator first collect all of the subchains whose left node is 0. The code generator would then connect together all of the right hand nodes of these subchains that are of the same value forming a logical "OR". Each unique right node forms a "branch" of the network. The application generator than takes each branch, one at a time and extends it to the right. This is accomplished by selecting all condition subchains belonging to the condition (matched on "Condition_ID") where the left node value is equal to the right node of the branch

being extended. This process continues until the branch reaches node 1. As each new left node is formed, a check is made to see if it already exists in a branch belonging to this condition that has already been processed. If the node has been previously created, the new branch is connected to that node and it is complete.

Each condition subchain can be made up of more than one logic element. Each element in a subchain is combined in the logic equation segment representing the subchain using the logical "AND" function. The "condition element" class in the OO model are represented by the elements. The types of elements needed were first identified in the condition/event expression class diagram shown in Figure 5-4. Then, further analysis and implementation decisions expanded the input elements into the "input condition description", for both the present and absent case. The classes specializing the condition element class represent these types. The sequence of evaluating the elements within a particular condition subchain does not matter since the logical operation performed is the "AND" function. However, in the case that order may aid in understanding of the function, a sort field is provided to maintain a specific sequence.

The application generator used individual database tables and indexed fields to implement the OO model. The condition has fields for storing the PLC word and bit number to identify the PLC "coil", as well as the symbolic name associated with it. State variable were used to keep track of when these fields were assigned a value as well as whether or not the logic equation represented by the subchains was already generated.

Thus in the application generator, conditions were associated with all objects that required a symbolic name associated with a PLC control bit. This resulted in a large volume of conditions. In the first two passes of the application generator, lists formed the major interface with the user. Conditions needed to be selected in several forms. Since the forms were limited in display area, lists that contained all of the conditions required scrolling to accommodate the length. Unfortunately, the user ended up requiring excessive scrolling and close attention to the descriptions. To overcome this situation the concept of scope or selective visibility was implemented.

When a user is interacting with a particular object, various selection lists are provided. To make the system more comprehensible, one needs to show only the objects that are relevant to the "context" in which the user is working. One method of determining which objects to include is to use the concept of "scope of visibility". This can be applied to states, actions, conditions as well as inputs and outputs. For example, some inputs are of interest only to a specific component, while others are of interest at the subsystem level and some are important at all levels. When the user defines an object, the level of visibility will be defaulted to the most general. The user is provided with a selection list to specify a more specific visibility. When the user defines a new condition, action, input, or output, an option is presented to specify the visibility or "scope" of the condition. The most restricted scope is that of a state, followed by state machine, then component, then subsystem and system. In future enhancements where that system is integrated into a manufacturing facility a wider scope such as "plant wide" will be required. For example a condition that indicates that the entire plant is in a state where

normal production should be running, as opposed to not running, will be needed by most of the systems. Thus the plant will form a higher level than that of system resulting in a wider scope. When dealing with the plant, one should subdivide it further into building, area etc.

The construction of logic to implement a state will be used to illustrate how the code generator used the abstract model. The state type was first used to select the general topology pattern. The "power on initial" state has its own topological pattern shown in Figure B-1. A power on initialization feature of the PLC simplifies the implementation of this state. All normal, that is non-latched coils in a PLC, are set to the de-energized state when power is first applied to the PLC. Thus at power-on, none of the coils representing the states of a state machine will be energized. This will result in normally (when the coil controlling it is at rest or not energized) closed contacts being in their closed position. The logic equation used to detect the power on condition can simply check that the normally closed contacts for all of the states in a state machine are al closed. This was implemented with a single subchain that contains the normally closed contact of the condition of all of the states except the "power on initial" state belonging to a particular state machine. The use of a relational database with SQL capability simplifies locating these contacts. Each state object is represented with a record in the state table. Each record has a field within it that has the ID of the state machine that it belongs to so a simple SQL statement will return all of a state machine's states. Each state record also contains a field that stores the ID of the condition that is used to implement the logic for the state. So to build the "power on" condition subchain, the

code generator loops through the selected states and adds a condition type of condition element indicating the off state with the condition ID retrieved from the state record.

A more complex example is the generation of the logic for an Automatic state. There are three basic types of subchains that make up the logic. The first is used to identify each state that this state is entered from. This information is needed to build the first portion of the logic for the state transition from each of these states. The second takes into account the state of the various supervisory, or higher level state machines that control the behaviour of the component state machine this state belongs to. The third is the subchain that "latches" this state. A description of this function is presented in Appendix B and shown in Figure B-4. It is used to maintain this state once it is entered and also to "drop out" or make this state no longer present when a transition form it to the next state is made. The objects used in the general topology pattern for the automatic state is shown in Figure 6-10. The subchains and the connecting nodes used to implement the topology pattern are shown in Figure 6-11. The right node of the subchain representing it can be determined by checking the type of each state with transitions to this state. Based on the types that were found, the specific supervisory subchains that are needed can be then determined. Additional first step subchains and the required joining supervisory subchains are determined by checking if the state machine this state belongs to implements single step or if this state terminates a single cycle. All of the states that this state has state transitions to are found to build the latch subchain. The rules and sequence of evaluation is shown in flowcharts found in Appendix D.

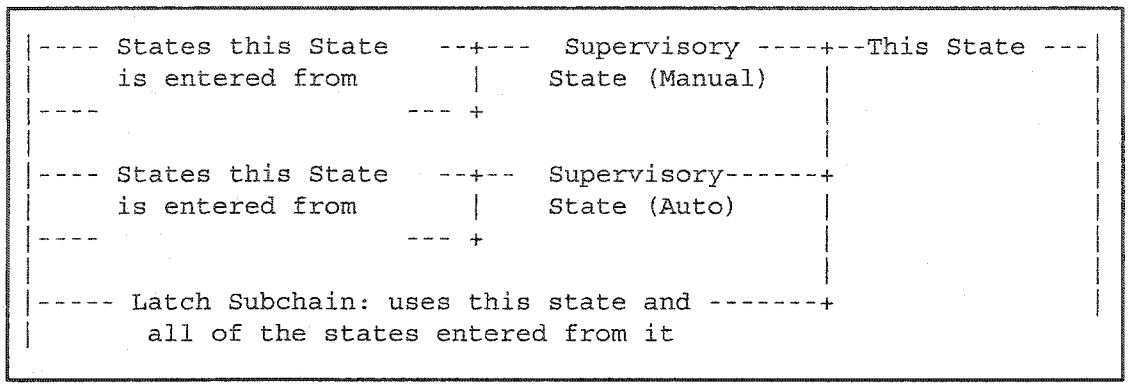


Figure 6-10: General Topology Pattern of Logic

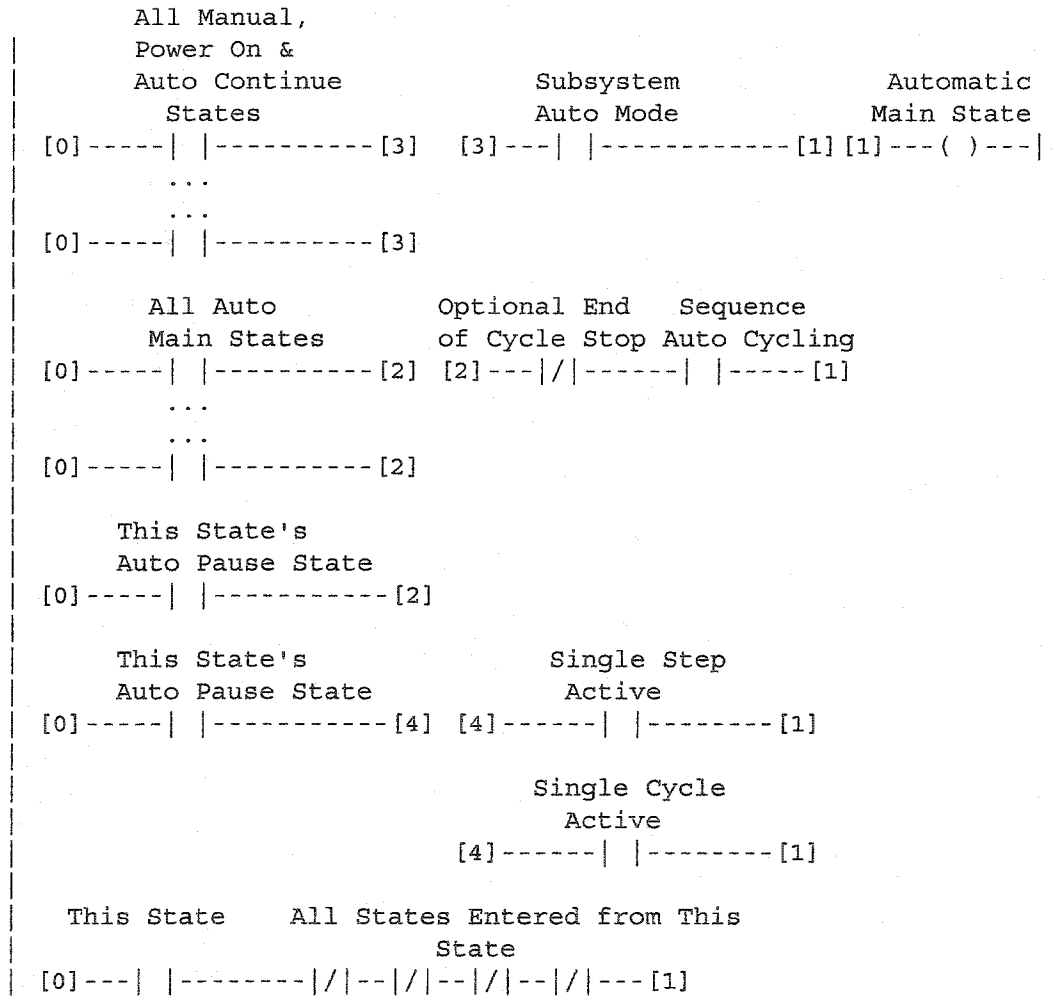


Figure 6-11: Subchain and Node Location for General Topology

The code generator output was first implemented as two text files. The first file contained the symbolic names to PLC word and bit number assignments. The PLC Ladder Logic editor could read it in so the user would not need to manually enter this information. The second file contained each coil followed by the condition subchains on separate lines was previously shown in Figure 6-7. The user was expected to manually enter the subchains connecting them by using the node numbers provided to each other and the coil they controlled. As will be presented in the evaluation chapter, the decision was made to provide a less user intensive method of entry. A general algorithm to form the complete graphical logic network was fairly complex. Searching for an alternative approach, the author found a text-based language named Standard List (STL) [Ber00] that could serve as input to the PLC tool chain. The code is compiled and loaded into the PLC. Then a conversion tool is applied to generate the Ladder Logic. STL is an assembler like language that will take a Boolean statement type of input. The application generator uses a Boolean representation for the logic so minimal coding would be required. The input format does require a specific sequence and the use of parenthesis. A general case creation of STL statements from the condition subchains would still be complex. However, a simplification is possible because all of the different types of logic equations are known. The application generator creates them from the user created model. Users are not allowed to write their own. Thus a simpler type recognition approach could be used. The required pattern was selected based on the type of state, condition, output etc. Knowing the pattern, all of the possible nodes of the subchains and how they are connected are known. With this information, the subchains can be located

in the appropriate sequence to build the STL source file. Then the elements can be located to provide the detailed information such as the type, address and bit number. A segment of a STL source file without symbolic names, with symbolic names and the Ladder Logic resulting from it are shown in Figure 6-12 through 6-14.

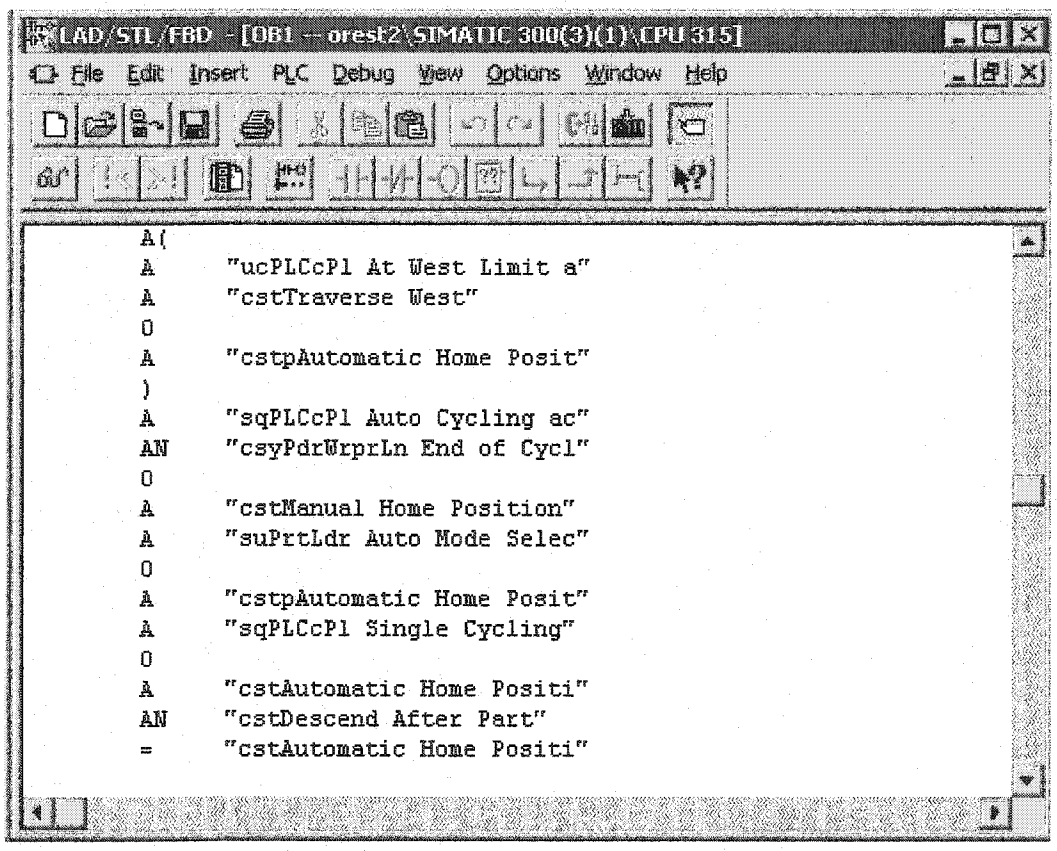


Figure 6-12: STL Source File Without Symbolic Names

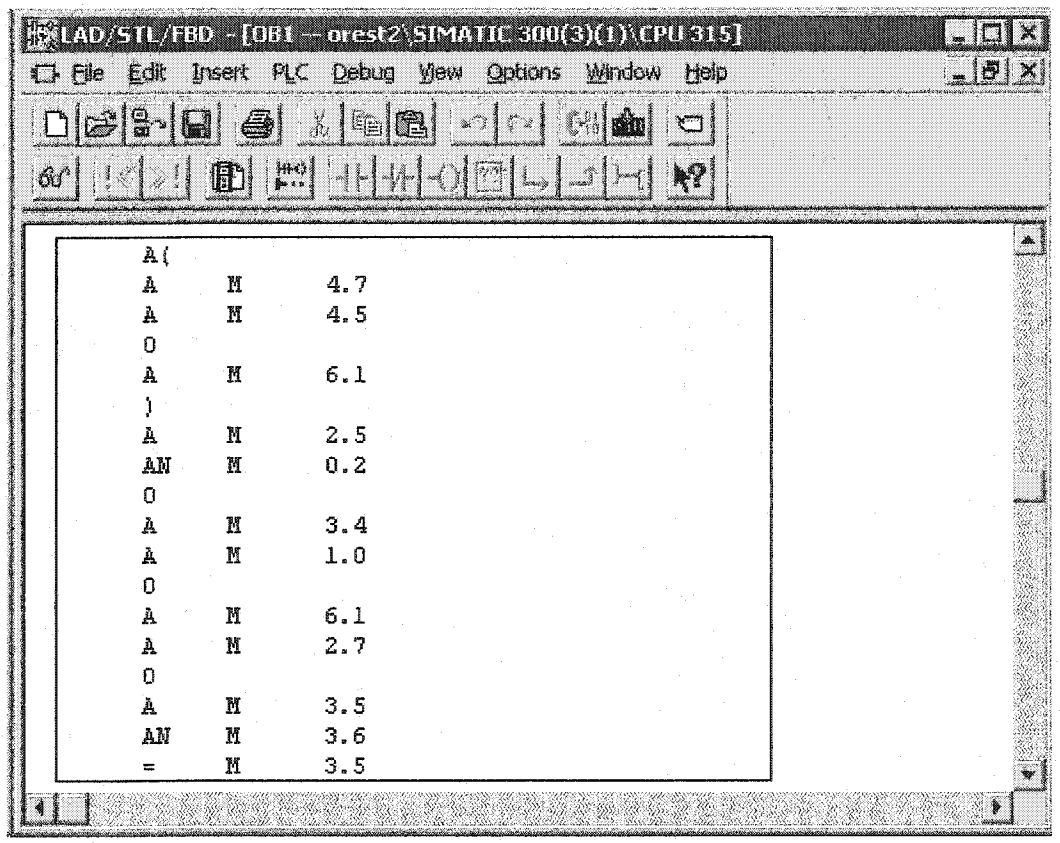


Figure 6-13: STL Source File With Symbolic Names

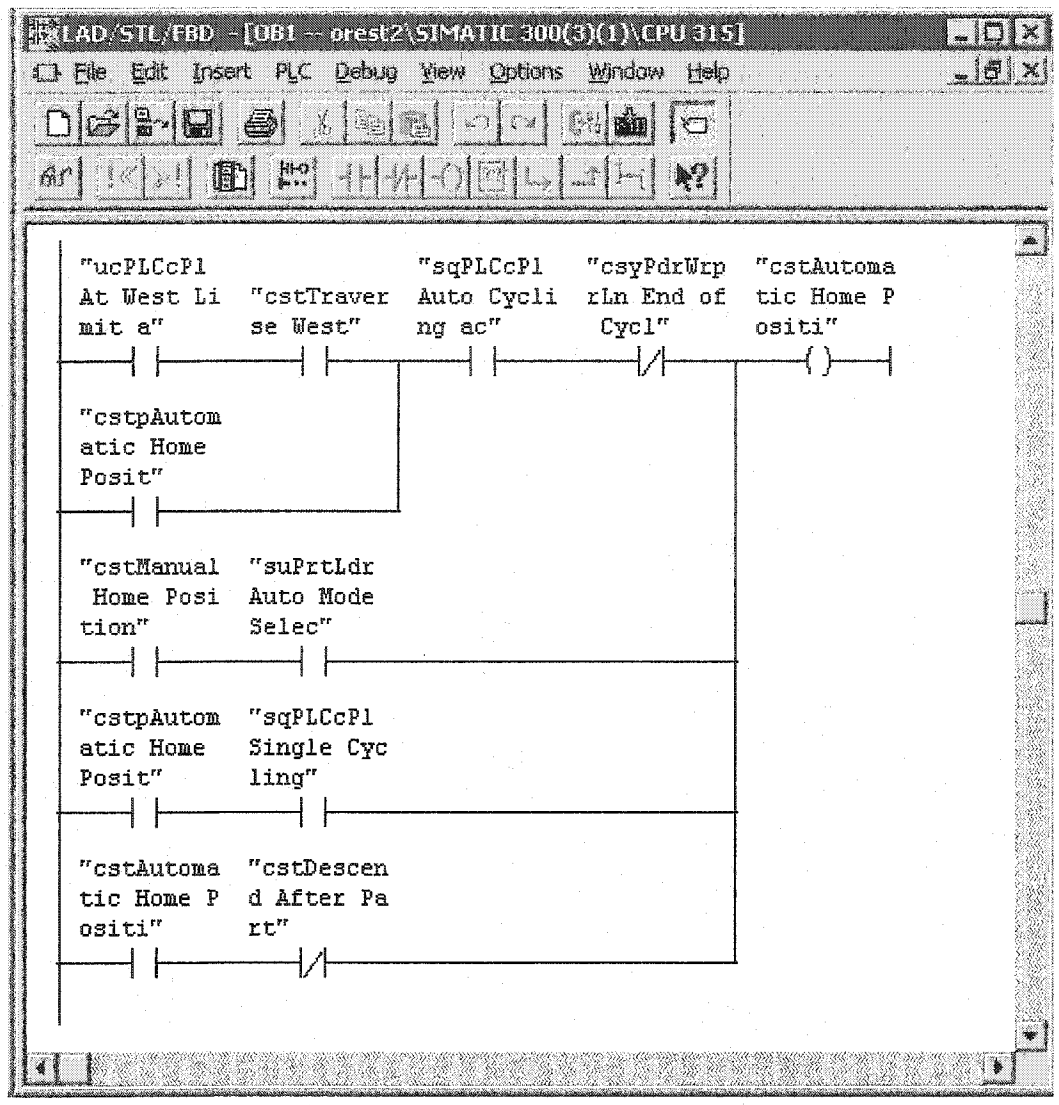


Figure 6-14: Generated Ladder Logic

6.5 Summary

This chapter described the design and implementation of the application generator. The three iterations in approach: extension of a commercial UML tool; object oriented language with an abstract set of classes for the user interface; and representing the object model with a relational database supplemented with a rapid prototyping language were presented. The implementation of the "system" object was used as a representative example to explain the details. The evolution of the user interface from a fairly abstract representation with many user controls to a simpler one with only the minimum needed controls to finally a more concrete representation and a simpler method of navigation based on the Zooming Interface Paradigm was presented. The chapter concluded with a description of the code generator that produces code used to create PLC Ladder Logic. The next chapter describes the replicable experimental environment that was developed to evaluate the proposed methodology.

Chapter 7

Experimental Environment Design and Implementation

This chapter describes the design and implementation of the experimental environment. The obstacles to software experimentation that commonly arise are discussed and a solution proposed. A design based on a commercial product intended for PLC applications that uses Ladder Logic to program a plant simulator is presented. The shortcomings of the simulator and acceptability concerns that led to the second design are discussed. A second design that replaced the Ladder Logic simulator programming language with a visual language is presented. The difficulties experienced with the second design are reported. A final approach based on using a small-scale, functional physical model is presented. The resulting experimental environment using a model of the Wrapper Line as well as a separate, distinctly different model for training is described.

7.1 Software Experimentation Issues

One of the criticisms of software engineering is the lack of experimental evaluation. Much of this lack arises from shortage of real systems on which to conduct the experiments. It is very difficult to find developers of industrial controls who will participate in such experiments. The controls that they need to develop are on very tight

budgets and time schedules. Once the controls are integrated with the machine and debugged, the entire system is shipped to the final customer as soon as possible. The author had the good fortune of having previously worked with a RLM control engineer and establishing an excellent working rapport. This provided a sufficient degree of comfort for RLM to participate in the experimental development of the control code for the two systems previously mentioned. This is a rare situation that is not likely to be available to others. To allow research without having access to a manufacturer of controls equipment, the author designed and implemented an experimental environment that simulated real systems. The objective was to create a replicable and portable experimental environment at a modest cost. By using a tradesperson who lived close by and volunteered to learn and critique the system, the experimental apparatus was installed at the subject's home and left there for the duration. Even with the close proximity to the tradesperson's home, the experiment took much longer to conduct than expected even though that location alleviated the need to synchronize time schedules.

7.2 Ladder Logic Based Software Simulator Approach

The first approach was to use a software simulator. The author was made aware by Bone [Bon99] of successful use of such a simulator by Pettit at Mohawk College. Pettit had evaluated several simulators [Pet99] and selected a product named "Pics" [SST01] produced by SSTECH in Waterloo Ontario. Pics is a program the runs on a PC simulating the equipment controlled by a PLC. The intended use of the simulator was to connect to an actual PLC or to a "Soft PLC" which is a program running on a PC emulating a PLC. When used with an actual PLC, Pics uses an interface card to connect

to the PLC I/O network that communicates with remote I/O modules or "racks". Microsoft's Dynamic Data Exchange (DDE) [Pet92] protocol (an early shared-memory process-to-process communication protocol) is used to connect to PC based PLC emulators. In both cases, the actual I/O racks are removed. The Pics simulator is set to the same addresses and in effect replaces the actual I/O. The PICS simulation software intercepts the PLC output values and uses them to drive a model of the controlled equipment or the "plant". The plant model drives the inputs to provide the feedback to the PLC that the actual sensors and transducers would have. User interface screens are provided to graphically represent the plant and to monitor the various inputs, outputs, variables and graphical screen objects indicating the state of the model. The plant model is actually a Ladder Logic program using the IEC 1131-3 [IEC98] syntax. This was chosen to minimize the technology insertion effort by using a programming language already familiar to the target user. To build a model of a sequential device such as the Wrapper Line Destacker subsystem, one would basically need to implement a state machine. It would have the same states as the control program since the control program models the machine behaviour as a state machine. The difference would be in the state transition logic, inputs and outputs. In the control state machine, a state is entered based on the information obtained from the inputs. For example, in a state that models the horizontal traversing towards the conveyor, the next state representing the part located above the conveyor is entered by a transition driven by an input indicating that the position has been reached. An output would be energized to cause the transverse motion and de-energized to stop it. On the other hand, the simulator would use the same basic

states but with a different treatment of the I/O and the use of nested sub-states. It would monitor the transverse motion towards conveyor output. In the normal sequence this output would become energized and remain so throughout the entire controller horizontal traverse state. In that case, the model would use a timer to simulate the time it normally takes to complete the motion. When the timer accumulates that amount of time, the simulator would then enter the next state and drive the controller's input indicating this condition. However, there are other cases that need to be addressed. While traversing, the controller may de-energize the output. The simulator would need a sub-state to keep track of this and either stop the timer or add time to the final time when the conveyor would be reached. The controller might not only de-energize the output but it might energize the output that causes traverse motion in the opposite direction. This will require an additional simulator sub-state that will either subtract time from the timer or add more time to the arrival time. A graphical representation would also be programmed to provide feedback to the user of the Destacker's current position. A newer version of the simulator allows the use of ActiveX controls. This makes the use of video clips of the actual machine possible to reduce the amount of graphical programming required for constructing the simulator as well as providing a higher degree of realism.

As the development of the simulation program progressed, it became apparent that the Ladder Logic used for the plant could also be used for the controller. As far as the Pics program was concerned, it was simply code that needed to be executed. If this proved feasible, a very compact experimental platform could be developed. The actual PLC or PC emulation of one, along with the I/O network and the simulator's I/O network

interface card could be eliminated. In the configuration intended by the Pics designers, a separate man-machine interface would also be part of the I/O network. After working with the simulator it was evident that the majority of objects that such a man-machine interface required were already available. The product also had the capability to construct custom objects to meet any additional requirements. Thus the man-machine interface could also be replaced. The result would be a single PC program that would execute the PLC control code, simulate the plant and provide the operator man-machine interface. The availability of a DDE interface provided connectivity to a monitoring program. Through this interface, all of the internal variables of the PLC program could be read at a rate fast enough to accurately follow typical automotive plant processes.

The Wrapper Line was chosen to determine the feasibility of using the Pics product for a machine of representative complexity. Sufficient capacity in terms of number of I/O and internal variables as well as execution speed was available. The man-machine interface (MMI) proved to be constrained by the limitation of video screen space. One needed to use either separate screen images or tabbed pages. However, this was no less capable then the intended PC based man-machine interfaces that share the same limitations. Industrial practice for overcoming screen real estate issues is to use multiple physical screens. This option can also be made available with the simulator by using a display driver card capable of supporting multiple displays or by writing custom code to provide communication between multiple PC's. The newer release allows integrating ActiveX controls so one could readily implement such a configuration. Providing a realistic visual representation of the plant model proved more difficult.

There was limited support for simple animation. Creating custom controls proved to be time consuming. Commercial animation packages are available but they also require considerable time and effort to yield realistic results. The limitation of screen space was far more noticeable than in the MMI. In the MMI use case, the operator is focusing on only a section of the machine, so being presented with the visual representation of the portion of the MMI dealing with it is not a problem. The machine on the other hand consists of independent subsystems all running independently as well as interdependently. To observe its operation or verify correct behaviour of the control code one needs to see a large portion of the machine or ideally all of it. In addition, there is a three dimensional aspect to the machine that the MMI does not need to address. When observing the operation of the machine various different views are required since the sensors and the various parts and positions of the machines that they keep track of are located in many different three-dimensional locations. One would either need the capability to rotate and zoom in and out or have separate two-dimensional views of all of the points of interest. The Pics product is primarily aimed at testing the PLC code and training machine operators. A realistic visual representation of the plant is not needed. The simulator is built based on the requirements documents used for generating the PLC control code and is used to check correct normal behaviour and induce faults to verify correct error processing. It is not used as an aid in the initial design of the control code or to help in obtaining an understanding of the machine itself. A further concern was raised in the acceptability of the simulator in experiments aimed at evaluating programming methodologies. The same program executed the code as well as evaluated its correctness.

The same type of Ladder Logic is used for both the control and the simulated plant. Both are based on the same state machine representation. The accuracy of the plant model in simulating the machine behaviour, particularly subtle interactions, is only as good as the programmer's comprehension.

7.3 Lab VIEW Based Approach

The Pics product provided the experimental environment with an acceptable MMI and execution engine for PLC Ladder Logic. To address the potential acceptability issue with the simulation of the plant, a different language was evaluated. The visual representation of the states of the Ladder Logic code as it executes was an important feature to maintain. The National Instruments Lab VIEW [Joh94] language provides the same visual display of program execution. It was designed to be a visual language with graphical display of the structures commonly found in software including state sequences. Its strong support for visual displays such a control panels of virtual instruments led the author to investigate it suitability to model the machine. Researching applications of Lab VIEW, led to work by Erwin, *et al.* [Erw98], aimed at teaching control by using graphical objects to represent various sensors and motors as well as states, counters and timers [Wol98]. The same elements are found in Ladder Logic. These objects were constructed using Lab VIEW elements to provide a higher level of abstraction when dealing with controls. The user interacts with these objects through a graphical editor and is not even aware of the underlying Lab VIEW code. This approach had proven successful in teaching basic discrete event controls even to young students

[Erw98]. This methodology is also used at the Chicago Museum of Technology to teach children how to program model robots.

A representation of the control of the Wrapper Line Destacker is shown in Figure 7-1. The general method of representing a control algorithm is to show a connected graph. The starting point is represented with a traffic light symbol where the green light is illuminated while the same symbol with an illuminated red light is used for the end. One starts at the green light and follows the connecting line to the first object. In this case it is a motor control object that lifts and lowers the Destacker arm. Attached to the motor are attributes. The diamond with the numeral 4 inside is the speed setting. The arrow pointing to the right indicates forward and the black diamond with the letter A identifies the specific output that this motor is connected to. Since there is nothing intervening between the starting point and the motor, it is turned on in the forward direction that corresponds to lifting the arm. By the convention of this control language, any device turned on remains on until specifically turned off. Thus the motor remains on lifting the arm and control is "passed through" it from the left upper side to the right upper side or from the "begin" port to the "end" port. Control is passed on along the connecting line to an object representing a switch. Modifying attributes identify it as the switch connected to input number one that is active when it is pressed in. This particular switch is closed or "pressed" when the Destacker arm reaches the "top" position. The semantics of this configuration are:

1. Wait doing nothing until control reaches this switch at it's "begin" port.
2. Check to see if the switch is pressed in.

3. If the switch is pressed in, pass control through to the "end" port and go back to step 1.
4. If the switch is not pressed in do not pass control through to the end port
5. Go to step 2.

When the switch is finally pressed in, control is passed on through the "end" port to the next object. The next object is a "stop sign". When control reaches its input port it stops, or de-energizes the output associated with it. The output is identified by a letter attribute inside of the stop symbol.

This control language is a powerful abstraction of state machines yet it retains a very concrete representation of the controlled objects, actions and events. It would be well suited to trades people who feel more comfortable with tangible objects and visual representation of a condition that they see in their daily activity such as a switch is pressed in, then they are with building abstract mental models. The traditional concept of states, outputs within a state and state transitions are replaced by a sequence of graphical objects through which a "flow of control" is passed. For example, examine the Destacker state machine sequence for lifting a part from the parts stack. The state transition into this state is represented by the sequence of:

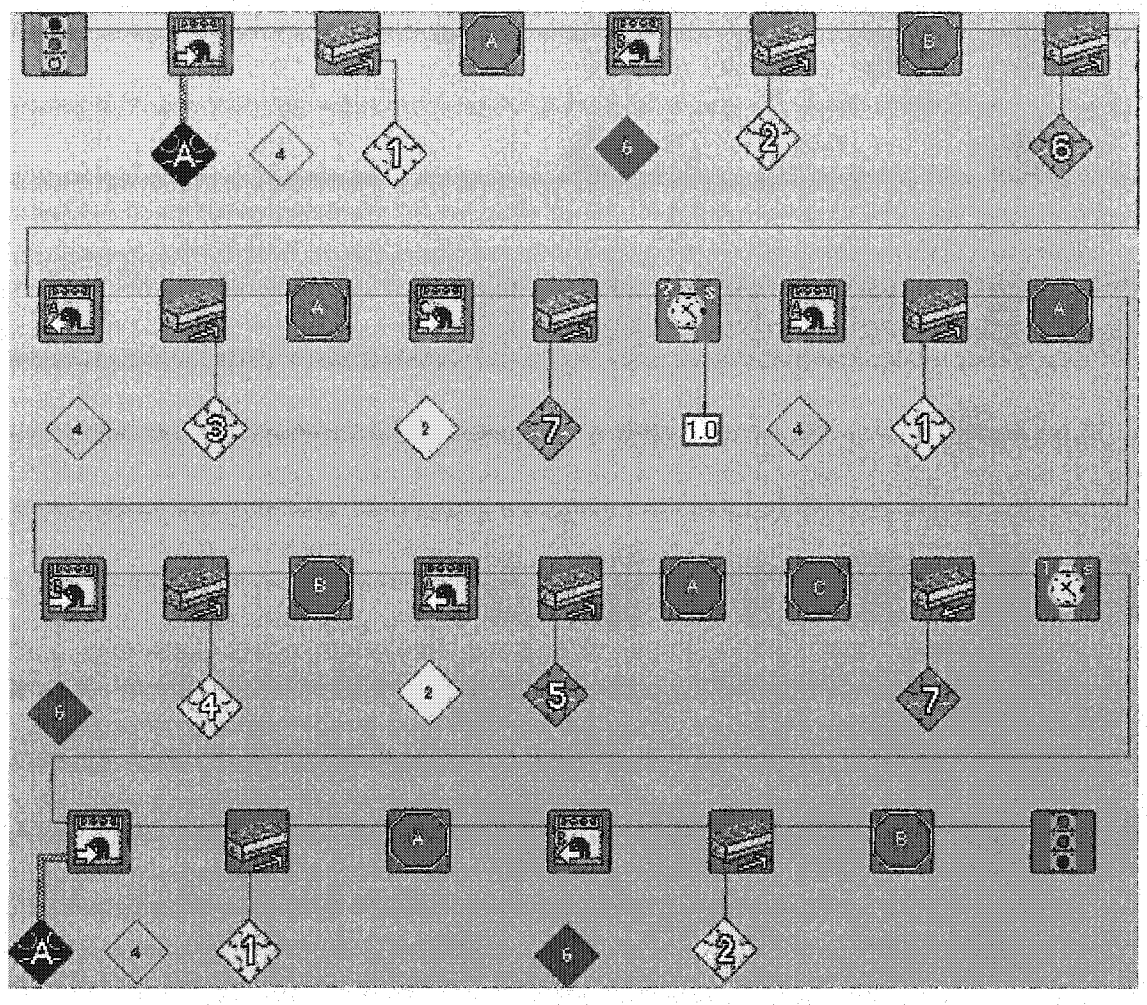


Figure 7-1: Lego Engineer Based Destacker Control Code

1. Control flow reaching the switch indicating the arm has reached the top of the stack being pressed. The control flow is then passed on when this switch is pressed.
2. Control flow then is applied to the switch indicating that the part is in the gripper being pressed in. When the part is present the control flow continues.

3. The "stop downward travel" of the Destacker arm occurs next and control is passed on.
4. Control flow then reaches the next object that now energizes the part gripper magnet.
5. A new state now is introduced to illustrate the use of time delay states. Control flow now reaches a time delay object graphically represented by a watch. An attached attribute indicates how long to wait before passing on the control flow. This introduces a wait state that allows time for the magnet to fully make contact with the part before lifting.
6. After the time delay, control is passed on to the motor object that energizes the Destacker arm motor to lift the part. Control flow is passed on to the "at top" switch.
7. This "state" is maintained until the "at top" switch is pressed.

In addition to the graphical representation, the control can be single stepped and the flow of control can be displayed as the machine cycles. The flow of control is not limited to a single straight line. Parallel paths can be implemented and the traditional software constructs of: case; for loop; while loop; sequence structure; and a formula node for evaluating mathematical statements are graphically represented. The graphical representation of the control algorithm can be further augmented with a "front panel" as shown in Figure 7-2. Various graphical objects are connected to nodes in the control program to provide a real-time feedback display of values or to act as inputs. One can

create custom graphical front panels. These features led the author to attempt to use this approach to build a simulator of the machine.

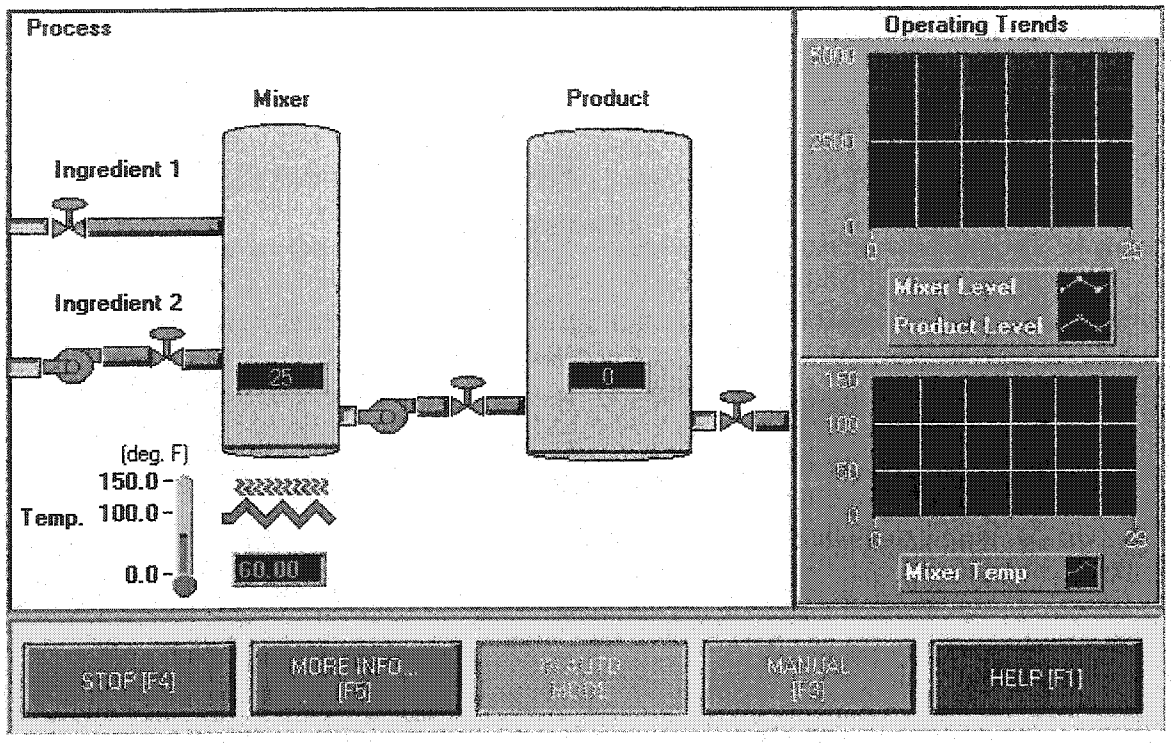


Figure 7-2: Sample Front Panel, Taken From [NIC99]

To represent a simulation of the controlled machine, the semantics of input and output needed to be reversed. When the controller turns on an output such as a motor, this serves as an input to the simulator. Instead of waiting for an event such as the pressing of a switch, the simulator now uses a timer to wait for the normal length of time required for a motion to complete. When the time has expired, control flow is passed on to a switch that actually is an output that informs the controller that the switch has been pressed. A normal sequence consists of the control sequence with time delays inserted between the control programs control flow outputs and inputs. Time delays that are

found in the control program are simply removed. The simulator code with a reversal in semantics for input and output is shown in Figure 7-3.

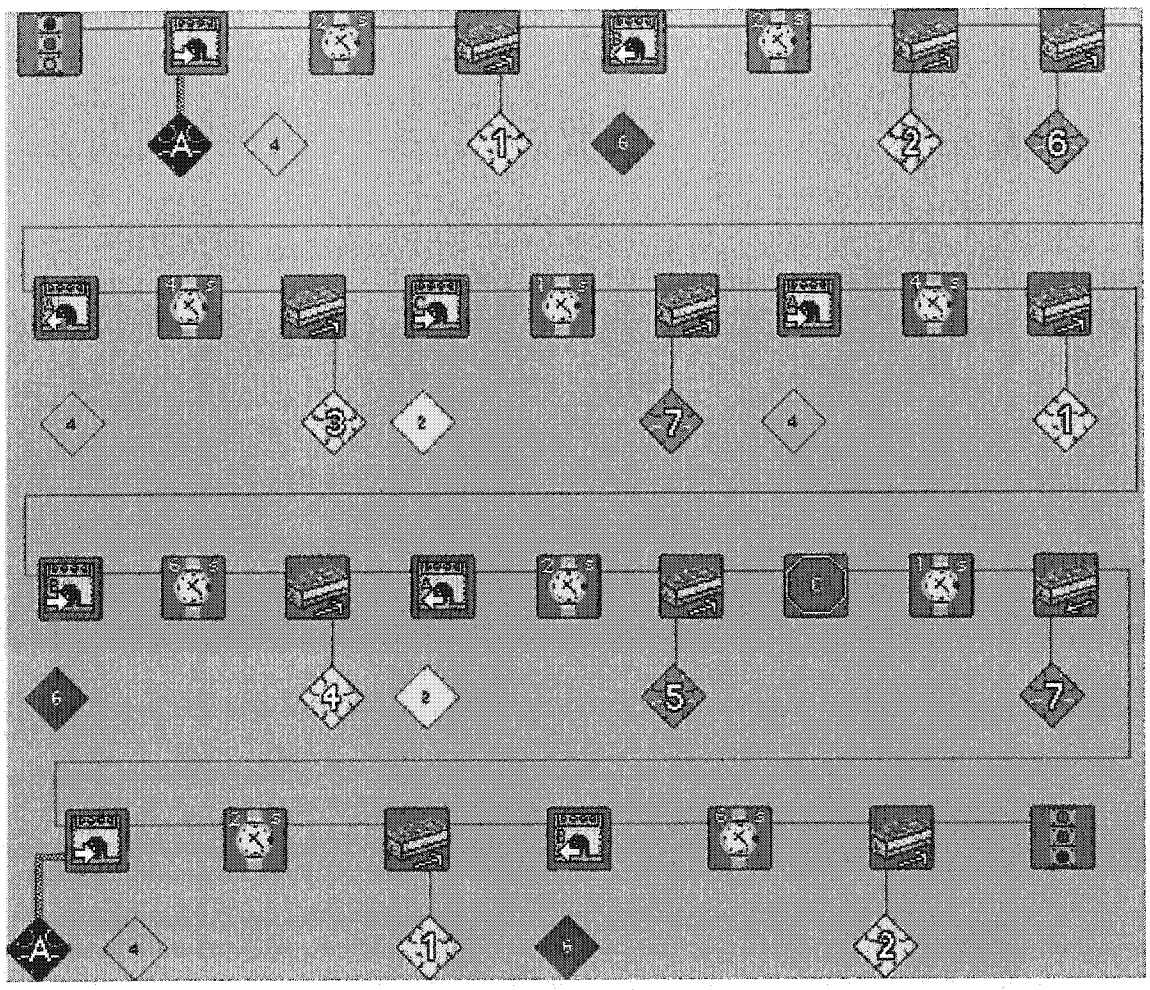


Figure 7-3: Lego Engineer Destacker Simulator

The author undertook learning how to program in LabVIEW. Simple programs did not require much effort to master. However, gaining sufficient understanding of the code used to implement the abstract symbolic Lego Engineer, to allow modifications was very time consuming. In addition, programming the simulator to provide exception behaviour, such as a motor being stopped before completion of an action, or a part being

dropped was far more complex than the normal sequence. Lab VIEW has the advantage over the Pics product of nesting objects of increased detail inside of simpler ones as a visual form of subroutine. This helped hide some of the complexity and addressed the issue of insufficient screen real estate since as in ZIP, Lab VIEW provides a zooming mechanism to navigate through levels of detail. However, the issue of the simulator being only as good as the programmer's understanding of the machine and the problem of using the Lego Engineer's front panel to represent the three-dimensional detail of the machine previously noted still remain unaddressed.

7.4 Physical Implementation of Plant Model Approach

Throughout the period of time this research was attempting to modify Lego Engineer to be used as a simulator and writing code to handle all of the potential exceptions to normal machine sequencing, various references were found of physical models made from Lego. None of these were of a complexity approaching that found in a representative industrial machine such as the Wrapper Line. However, an example of a working model Gantry Mill [Fay01] indicated that such a level of complexity was possible. Bergendahl, *et al.* [Ber95b], demonstrated a high degree of modelling complexity by constructing an operating assembly line from Lego. Figures E1 through E5 in appendix E show samples of these models. Correspondence with Cassandras [Cas00] led to the conclusion that a physical model of the Wrapper line might be built from Lego components. An operating physical model that could be controlled by a PLC would address the problems encountered with simulators. Even if a part could not actually be processed the physical model would still represent the operation of the

machine in terms of transporting of parts and motion of the machine. Exception events such as the dropping of a part do not need to be foreseen and coded ahead of time as in the case of using software based simulators. The problem of visually representing the three-dimensional aspects of the machine with a two-dimensional media of a display is totally eliminated.

The first subsystem to be modeled with Lego was the Destacker. Vertical motion was accomplished by adapting the reverse driven worm gear shown in Figure E-6 [Wil01a]. The horizontal axis was constructed from a rack and gear arrangement shown in Figure E-7 [Wil01b]. Difficulty was encountered in constructing rigid beams of the length desired and mounting them to posts. Gripping the part also posed a problem. Designs for various grippers were found [Knu99, Bau00] but they proved to be bulky and did not grasp the part firmly. One of the major difficulties encountered arises from the friction fit post and socket fastening method used with the Lego parts. To make assembly and subsequent disassembly possible, the force required to pry the parts apart cannot be too great for a young child that the product is targeted for. Unfortunately, this force is not great enough to reliably keep the assembly together while it is operating or being transported. There is an alternative method of assembly that can be used to overcome this issue. As shown in Figure E-8 [LEG00], beams and pins can be used to hold assemblies together. The disadvantage of this approach is the increased physical size. Further research into industrial models uncovered an alternate product made by FischerTechnik. Instead of using the LEGO stud-and-tube coupling system in Figure 7-4, an interlocking grove and mating shaped pin, similar to the sliding "ways" and slots in

tables for mounting clamping fixtures found in milling machines, is employed as shown in Figure 7-5. This product has been used to build models of complex industrial systems shown in Figures E-9 through E-11. King [Kin00] supplies such models to both academic and industrial markets.

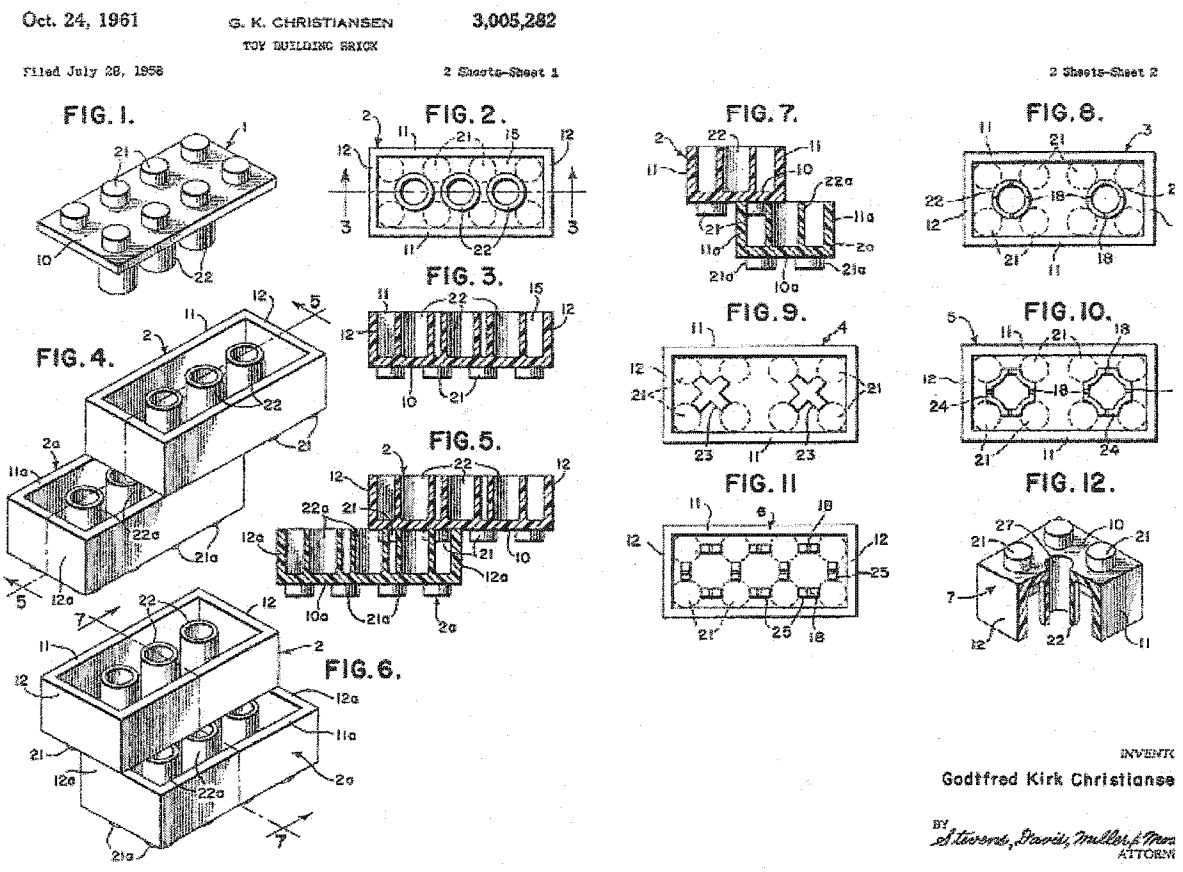


Figure 7-4: LEGO Stud-and-Tube Coupling System, Taken Form [Chr58]

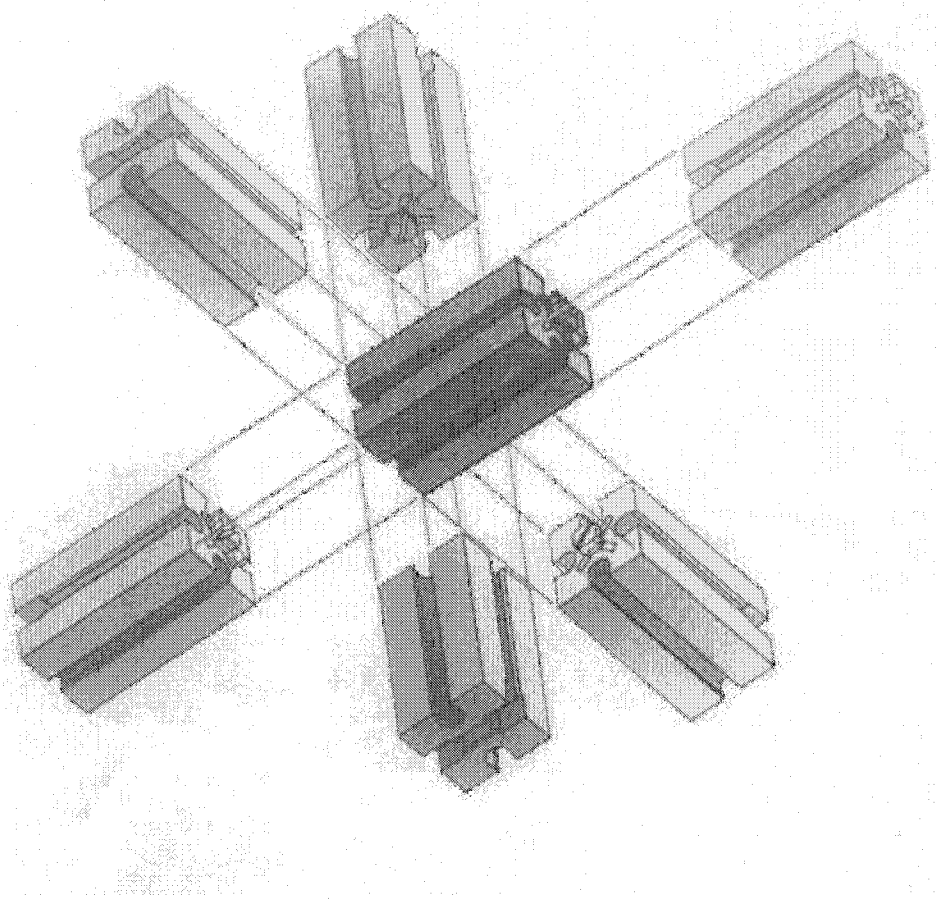


Figure 7-5: FischerTechnik Coupling System - Taken From [Fis94]

The feasibility of constructing a model of the Wrapper Line was discussed with King. King's observation was that his ability to build models that functioned reliably was developed over years of trial and error. He cautioned against this research undertaking designing the Wrapper Line since it consisted of assemblies that he had not previously designed. The author had no previous experience with the FischerTechnik product. King recommended adapting one of his proven designs to the experimental environment. This

may have been a better approach than the one used since it took more then a year of elapsed time to complete the Wrapper Line model as well as a second model to be used for training the experimental subjects in the methodology.

Research into construction of industrial models made with both the Lego and FischerTechnik product led to the observation that each had distinct advantages over the other. The Lego product allowed construction of smaller assemblies that required the use of gears. Mechanical part graspers using a small pneumatic cylinder could be more readily made. The Lego shaft encoder was smaller and did not exhibit backlash as experienced with the FischerTechnik method of using a cam activated switch. The Lego switch required less force and travel to activate and could be activated from more different angles. Lego had beams with bends in them that could be used for clamping parts as well as moving or ejecting them. The FischerTechnik parts produced assemblies that were robust enough to provide reliable motion. The models were rugged enough to withstand the shaking and jarring encountered in transportation. Much larger assemblies could be constructed and a far superior base plate mounting method was available. The use of long extruded aluminium columns provided the mechanism for constructing supports for long axis of motion such as the horizontal axis of the Destacker. The use of steel rods of any length facilitated the construction of slides. The continuous slots that can be made by using the individual blocks or columns, allowed the positioning sensors anywhere along the slot. Motors and switches were easier to mount securely to the model. A greater variety of pneumatic parts were available along with electronically activated air solenoids. To allow PLC control of Lego pneumatics required the coupling

of motors to manual air valves as shown in Figure E-12 and E-13 [Wil01c]. Since both products had advantages, this research devised methods to integrate them together shown in Figures E-14 through E-20 that are believed to be novel.

The model of the Wrapper line constructed of integrated Lego and FischerTechnik parts is shown in Figure 7-6. The part transported through the model was made from wood with a metal disk embedded in it shown in Figure 7-7. The metal disk allowed the use of an electromagnet to grip the part. The actual machine used vacuum to grasp the part. An electromagnet was already available, as a standard part whereas a vacuum based gripper would need to be custom built. The gripper, at top of stack and part present switch assembly is shown in Figure E-21. Proximity switches are used in the real machine to sense the presence of the part and the top of the parts stack. A plunger-actuated switch was used in the model shown mounted on the aluminium bar near the top of Figure E-21. The electro-magnet and part present switch mounted next to it comprise the assembly connected with two sliding bars to the bottom of the aluminium bar. As the bar descends, the assembly makes contact with the top part forcing up the rod to close the upper switch. Metal sensing proximity switches are available from FischerTechnik but they are about five times more expensive. The arm shown in Figure E-22 was constructed from a single aluminium bar providing straight rigid construction. This provided convenient mounting for switches used to sense the various vertical positions of the arm. Vertical motion of the arm was accomplished by using a motor, worm gear

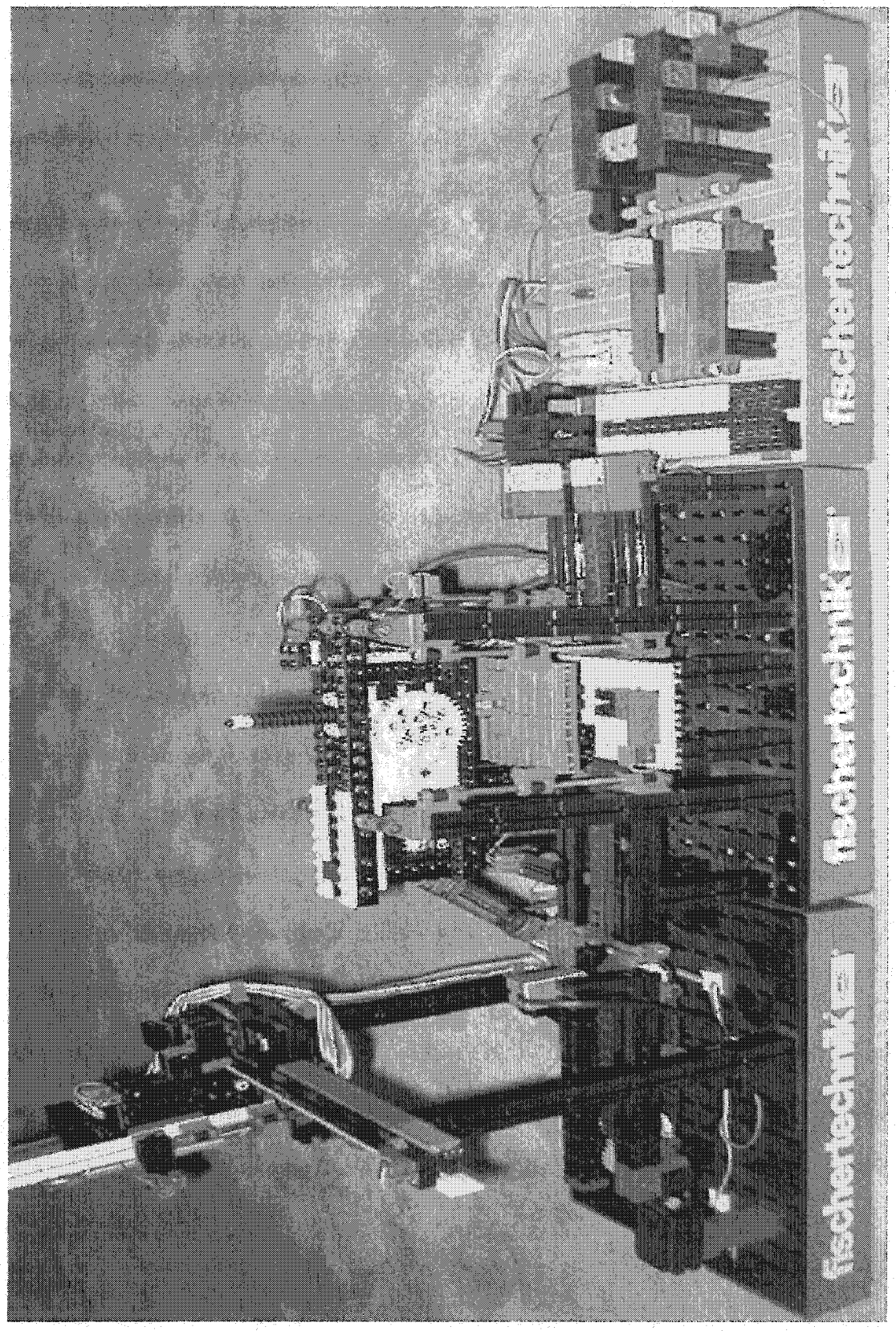


Figure 7-6: Wrapper Line Model

assembly driving a small gear that engaged in a ribbed track shown in Figure E-23. Horizontal motion was accomplished in the same manner detailed in Figure E-24. The press entry conveyor shown in Figure E-25 was an integration of the FischerTechnik motor to Lego gears and belt supported with FischerTechnik blocks. The press vertical drive is constructed from Lego part as shown in Figure E-26 illustrating beam and pin construction and the use of small gears. A pneumatic cylinder is used to move a Lego bent beam through a series of gears to act as a part clamp and ejector as shown in Figure E-27 with a shaft encoder to provide position feedback. A scissors lift is constructed from a combination of Lego and FischerTechnik parts as shown in Figure E-28. The Roll-Former entry conveyor shown in Figure E-29 is an adaptation of the Bergendahl, *et al.* assembly line model [Ber95b].

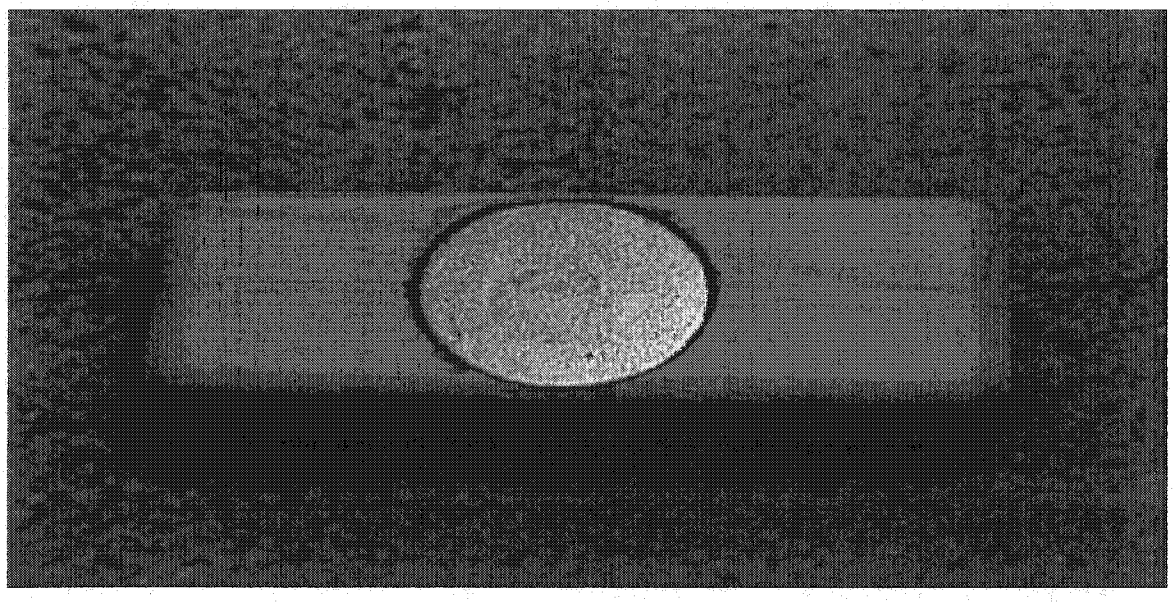


Figure 7-7: Transported Part

After constructing the Wrapper Line model that was to be used in experiments to evaluate a tradesman's ability to create control code, the conclusion was reached that a separate model would be required for training. Initially, it was envisioned that a training document would be sufficient to prepare the experimental subject. However, once the abstract model based methodology was developed and the code generator constructed, it became apparent that training based on constructing control code on an actual model would be superior to one based solely on a written description. If one used the Wrapper Line model for training, it could not be proven that the experimental subject later developed control code for it solely from his understanding of the methodology and applying the knowledge captured in the application generator. It could be argued that he simply copied portions of the training exercise. Thus the second model shown in Figure 7-8 was designed to serve for training.

The mechanical assemblies used were purposely different from the Wrapper line model so that the experimental subject could not just copy the control code from the training exercises. Instead of picking parts from a stack as shown in Figure E-30, a magazine was used as shown in Figure E-31 with a pneumatically powered part feeder. Instead of using the drive mechanism of the Destacker, a pneumatically actuated indexing table shown in Figure E-32 was used. A pick and place robot shown in Figure E-33 was used to load the part into a saw station instead of a conveyor. The clamp used switches to locate its positions instead of a shaft encoder as shown in Figure E-34. Instead of using a part ejector as in the press, a part diverter and ramp was used as shown in Figure E-35.

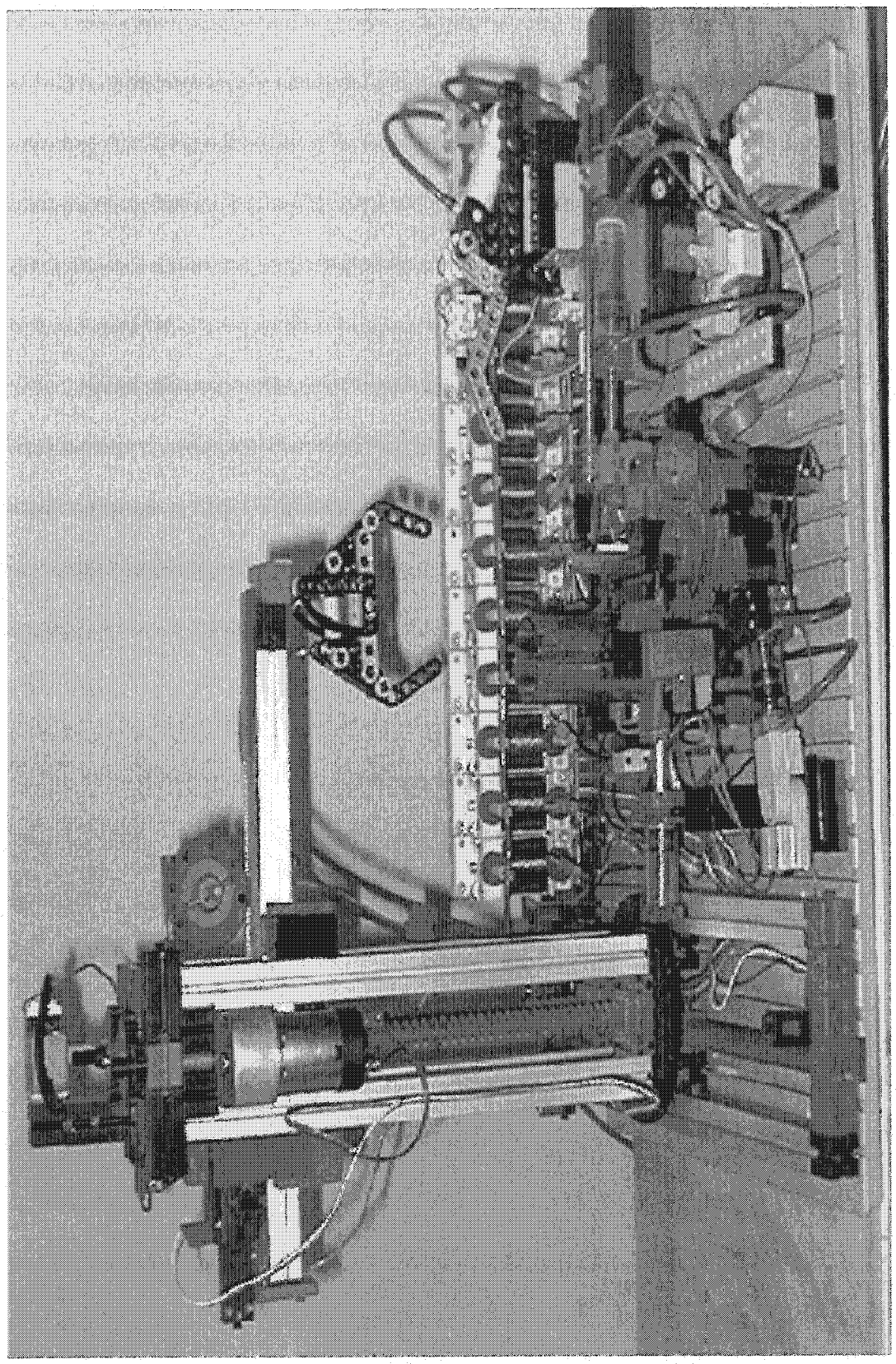


Figure 7-8: Training Model

The scissors lift with its drive and sensors was replaced with a second Pick and Place robot that used a lead screw and threaded block drive as shown in Figure E-36. The position sensors used were a potentiometer shown in Figure E-37 and a shaft encoder shown in Figure E-38. The one common use was photo-sensors for locating the part at the bottom of the ramp and magazine unload shown in Figure E-39. This model provides the experimental environment with the means to train experimental subjects without unduly showing them solutions that can be directly copied to the actual experimental model. A greater appreciation of Mechanical engineering was acquired after completing the training model. The Wrapper Line model was simpler to construct because the mechanical design was copied from the actual machine. In the experimental model, much of it needed to be developed. This required far more effort than anticipated particularly in the subtle modifications and adjustments needed to achieve reliable repetitive cyclical operation.

Three alternatives were evaluated for the interface between the PLC and physical model. King provides an interface that allows connecting the PLC directly to the motors and sensors. This interface consists of relays configured in a full "H bridge" to drive the DC motors in either direction. Single relays provide interface to electro-magnets or DC motors driven in a single direction. Safety circuitry is used to insure that the power supply is not shorted through accidental incorrect signals from the PLC. Level shifters and relays provide the interface from infrared LED's and photo-sensors to PLC inputs. Lego provides two types of interfaces. The first consists of a microprocessor with three inputs and outputs intended for their robotics kits. An infrared optical interface is

provided for communication. Information exists [Knu99, Bau00] on how to directly interact with the microprocessor as well as how to expand the limited I/O capability. The second interface consists of an interface that provides 8 inputs and 8 outputs. The outputs are bi-directional DC that can be set to 8 different levels. These are capable of directly driving DC motors. The inputs are self-contained 10-bit resolution analog to digital converters. Four of the inputs provide excitation current for active sensors such as shaft encoders, potentiometers and photo-sensors. A serial communication protocol [Hud98] using RS232 provides the means for interacting with the I/O. Appendix F lists the functions supported and format used by this protocol. Of the three alternatives, the last one was selected. It provided variable speed control of DC motors, a simple method of connecting to a PC without the need of special interfaces and fairly large number of connections. Multiple interface cards can be used if more I/O is required. The inputs driven by switches were directly connected to the PLC inputs reducing the need for interface card inputs. The interface circuitry used by King for phototransistors was also used further reducing the required interface card input count.

The RLM control engineer recommended using a physical PLC rather than a PC software-based PLC. He stated that this would provide a better chance of acceptance by other control engineers in the industry. Since the new methodology was considerably different than what they currently used, anything that did not need to be changed from what they were familiar with should be left the same. The particular brand chosen was Siemens primarily due to a documented file format that would allow direct input of the code generator's output. The integration of PLC with its I/O cards, Lego I/O interface

modules, physical model into a portable experimental platform is shown in Figure E-40. The various power-supplies, connections and phototransistor interfaces are located in the back of the wooden enclosure. Connectors facilitate exchanging the physical model on the top surface.

A PC based VB program was written that used an interface card to communicate with the PLC and a serial port to communicate with the LEGO interface. Direct reading and writing of the PLC inputs and outputs was available. With this functionality, the PC program functionality was expanded to provide the operator panel MMI removing the need to purchase a commercial MMI. Figure 7-9 and 7-10 show the implemented MMI. The support of objects by VB was sufficient to allow rapid configuration of the interface and MMI applications. Classes were written as shown in Appendix G, to represent: a bi-directional motor drive; momentary push button; retentive push button; three position selector switch; shaft encoder; and others. The visual screen objects could be simply copied with changes made to the captions. When the experimental subject designs the controller, the interface and MMI can be quickly written by simply instantiating object of the appropriate class. The new .NET release of VB now allows creation of not only code objects but also graphical forms and placing graphical controls on them, under program control. A future enhancement to the experimental environment would be the writing of a program that would automatically create the MMI and PLC to LEGO interface program. This would remove the need for a programmer to configure the MMI for the experiment based on the subject's developed control code.

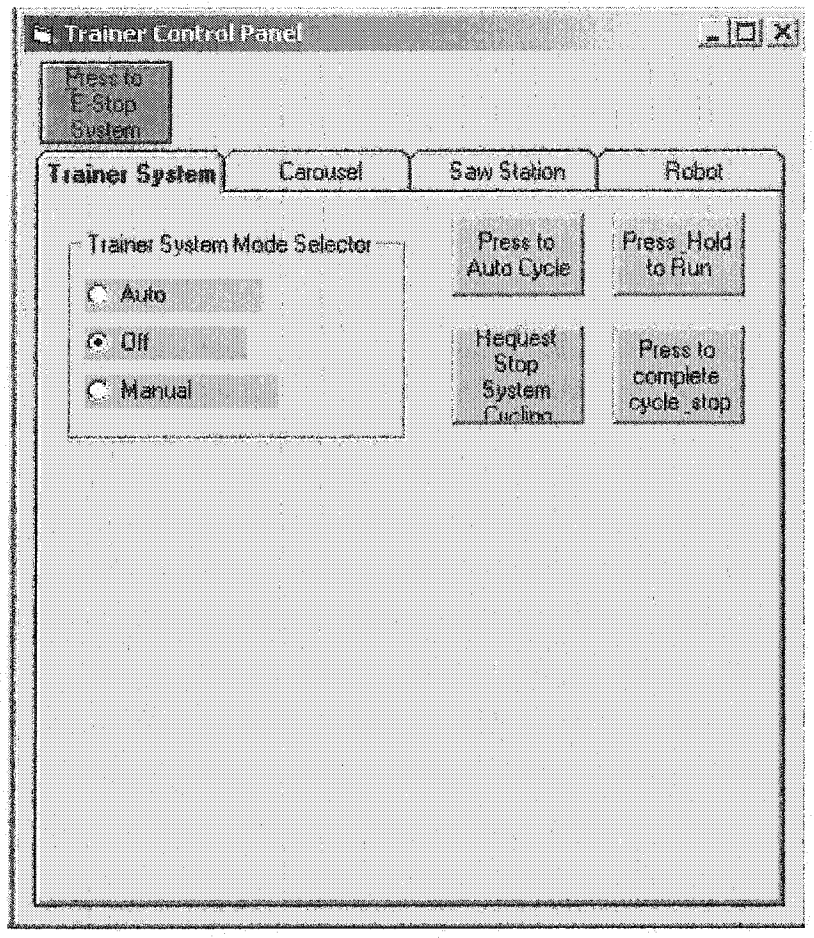


Figure 7-9: System Level Control Panel MMI

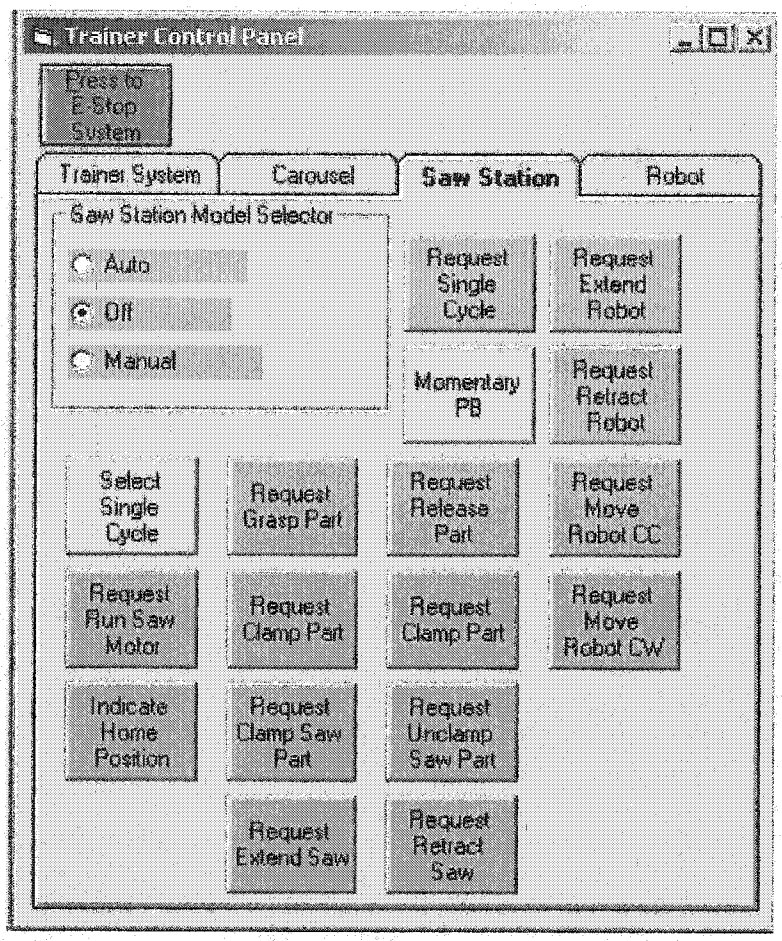


Figure 7-10: Subsystem Level Control Panel MMI

7.5 Summary

This chapter covered the design and implementation of an experimental environment. The design objectives addressed the common obstacles of: shortage of real systems; availability of test subjects; and cost of laboratory systems, encountered when attempting to experimentally evaluate software methodologies. Approaches for the plant simulation based on: Ladder Logic programmed; visual programming language; and a

functional physical scale model were presented. A detailed description of the final system based on a small-scale functional model for conducting experiments as well as a separate distinctly different model intended for training were provided. The next chapter evaluates the results of using the experimental environment.

Chapter 8

Results and Observations

This chapter describes the results of the research. The synchronization of control and monitoring systems was achieved through the use of a consistently applied abstract model. The developed abstract model accurately represents the majority of control systems used in roll forming manufacturing. A mapping from the model to PLC Ladder Logic was developed. Once the model for a specific application is developed, a simple mechanical process is used to manually generate PLC code by applying the mapping. A method of developing a control plan from reusable components was provided through the use of an application generator. The knowledge of a control engineer was captured and represented by the application generator based on the abstract model. As a result a tradesperson who understands machines but is not familiar with writing control code can generate control code using the application generator. Observations of and feedback from the trade person's interaction with user interface led to two redesigns significantly improving its usability. The development of a replicable experimental environment led to the evaluation of three different simulator designs. The one selected met the criteria of implementing representative complexity, low cost, robustness and portability.

8.1 Abstract Model and Methodology

The major motivation of this research was to establish a method of obtaining context enhanced data by a monitoring system and keeping it synchronized with the constant evolutionary change of the control systems. The method proposed to achieve this consisted of moving the control engineer to work at a higher level of abstraction, specifying requirements rather than directly writing code. An abstract model described in chapter 5 provided the higher level. A previous attempt to standardize Ladder Logic code through the use of documented code topologies [Sto86] failed due to different interpretations and modification that engineers used when manually generating the code. To address this problem an application generator described in chapter 6 was developed. The control engineer interacts with the application generator where the abstract model is built and control code automatically generated. The information consisting of context-enhanced data is provided by the control code and the meaning of the information is obtained from the abstract model that the control code is based on.

The result of this research is an abstract model that provides the higher level at which the requirements are specified. Experimentation with the RLM control engineer resulted in a simple mechanical mapping from the abstract model to standard Ladder Logic code topologies. At this point the work began on the application generator that would automatically generate the Ladder Logic code once a model was built for a specific application. At the same time the RLM control engineer began to use the methodology manually.

8.2 RLM Application of Methodology Case Study

Based on the results of the Wrapper Line control code development, the RLM control engineer concluded that the methodology had sufficient merit to use manually before an application generator was developed. He replaced their previous methods for designing the control code with the model based OO methodology supplemented with state based specification of dynamic behaviour. The set of standard Ladder Logic topologies was manually coded with an editor and then using "cut, paste and modify" the set was reconfigured for new applications. This process has now been in use for more than two years. Based on the benefits the company has experienced, the engineers adopted the methodology replacing the methods previously used. The company reported [Has02] software development using the NFSM model showed the following distinct advantages over the method used in current industrial practice when applied to a new design. The assumption made regarding current practice is that it is at CMM level 2. Existing control designs are well documented and processes are in place to repeat earlier successes on projects with similar applications. Adapting and expanding an existing well-documented design is used to develop a new design. Abstraction and hierarchy are used in the form of code patterns modularized in subroutines or function blocks. Higher level, more formal methods based on the Object Oriented approach, Nested Finite State Machines or Perti Nets are not used. The code development tools used are the ones provided by the PLC manufacturer.

The first advantage RLM reported was that the development cycle was shortened and code functionality and quality were improved. The initial system design and code generation required approximately one half of the time compared to the time spent on systems of equivalent complexity using the old method. RLM attributed the improvement to a basic change in the design process. The typical industry state-of-the-art process consists of the control designer obtaining an initial incomplete comprehension of the system requirement from the customer and then starting to generate control code. During the code generation, questions arise which result in a consultation with the customer (person specifying the function of the system). During this consultation the specific questions that the control engineer was aware of are resolved. Frequently, new I/O is required and existing code has to be changed. The cycle repeats as additional code is generated and new questions arise resulting in further consultation. Unfortunately, the customer usually does not comprehend the full system requirement at the beginning and the application code has to be created through a series of meetings. Quite often the control engineer discusses use-case scenarios of which the customer did not think. New decisions are made often invalidating previous assumptions that existed based on requirements that have changed. It is difficult for either the customer or engineer to remember all of the implications of the decisions that have been made along the way. The code ends up being a series of patches often without the removal of obsolete and redundant segments. RLM reports that the NFSM method forces a control engineer to obtain a full understanding of the requirements before any coding can begin since the NFSM must be fully specified. The NFSM provides a compact and complete model of

the system behaviour facilitating communication with the customer, removing ambiguity. Once the NFSMs are drawn, coding is a simple mechanical process. It can be written cleanly in one pass. The resultant quality and functionality is superior to the current practices in industry. Feedback from the Wrapper Line customer as reported by RLM [Has99] indicated that the control system functionality was better than what they normally obtain. The Ladder Logic code topologies developed by this research were optimized for ease of comprehension. Practices for reducing the logic to yield smaller code or optimizing for speed of execution were purposely avoided since the resulting code obscured the original design intent. The expected result was larger code size and longer execution times. However, the industrial experimental partner observed neither noticeable code size, nor execution speed penalties. Their explanation for this result was that the elimination of obsolete and redundant code, plus the single-pass for code development as opposed to a series of patches offset the costs of the new method.

The second major benefit of the method developed by this research observed by RLM in over two years of application was the reduction of maintenance effort. Approximately one third of the total traditional development cycle is consumed by debugging of the control code. RLM reported that the new methodology required about two thirds less time to debug than the code developed under the traditional approach. RLM attributes the improvement to the following. The programs based on the NFSM model provide an easily understood documentation of the system behavior. The FSM drawings provide a concise method of graphically stating all of the possible scenarios the control system has implemented. The paths through the state machine with the events

and conditions that cause a particular path to be traversed and the actions or outputs that occur in each state are readily discerned. The direct mapping from the diagram to standard topology of code implementation provides a simple correlation. When the machine stops in a particular position or fails to behave in a specific manner, one can quickly diagnose the problem. The state, in which the control code places that machine, is readily observable from the single coil per state implementation. The corresponding state in the FSM diagram can be easily determined. The actions that should occur are identified from the diagram. By looking for the standard patterns in code, omissions or errors can be quickly found. If the code is correct, one then observes the machine to determine if there is a malfunction or wiring mistake when the modeled and coded action is missing. The other possibility is that a particular requirement was not comprehended and is missing from the model. In this case, the state diagram is modified and by applying the standard mapping, code is updated. The RLM engineer stated that enablers for this capability:

- Ease of comprehension of the model.
- Straightforward mapping from model to code.
- The corresponding ability to find the section of the model that corresponds with a particular segment in code
- A relatively simple modeling method that allows manual updates without the need for special design tools.

For the same reasons, modification of a component sequence by someone other than the software developer also requires one third of the normal time. RLM has found that their technicians, having become familiar with the aspects of the new methodology, now will expect to be provided with the state machine diagrams before they leave for a service call. An additional advantage was noted during the diagnosis of system failure, which was had not been accounted for within the implemented diagnostics. The hierarchical nesting of finite state machines provided a natural sequence of diagnostic steps. A technician starts by checking the state machine from the top down through examination of the current state, expected next state and conditions needed to achieve it. The location of an error where the missing condition exists, can be quickly determined, and mapped to the physical malfunction. The transition from one state to the next typically requires examining only a few lines of code. The state machine method effectively eliminates the need to examine and understand large sections of code by automatically indicating the current state. Using a standard topology for implementing the code allows a technician to quickly understand the functions and logic, and correlate it with the sequencing of the physical machine. Normally this is far more difficult in traditional control logic where the current state of the system cannot be readily determined. There are no distinct state indicators. The state needs to be deduced by studying the conditions of the machine, the current value of the logical equations represented by the control logic and correlating the two. The logical equations represented by the code have usually undergone simplification making it more difficult to determine their functions.

The third advantage reported by RLM was ease of integrating MMI and diagnostic code. A common method for providing diagnostics is to monitor how much time machine motions consume compared to the time they normally require. When the normal time is exceeded an alarm is raised. To implement such a scheme requires isolation of individual sequence steps. This leads to the same difficulty that monitoring systems experience associated with the difficulty in recognizing the steps within the code. The standard code topology simplifies integration of such diagnostic code. The use of states and implementing each state with a single contact provides the signal needed to operate the diagnostic timing logic. The diagnostic logic also needs to determine when it should be enabled. It should report alarms only when the machine is cycling in the normal automatic mode. The hierarchical levels with distinct interface conditions provide this information directly without the need for custom diagnostic code. The diagnostics discussed so far were for normal automatic operation. A different use case scenario exists for diagnostics that assist with the starting of the machine. As previously described, most machines must be manually placed into a specific position, commonly known as the home position, before the automatic mode can be entered. The new methodology explicitly models this use case and then employs a standard Ladder Logic code topology for implementation. To implement the diagnostic, the programmer locates the condition that indicates when automatic cycling can begin. Then the individual condition subchain elements provide the detailed information of what is missing to allow entry into automatic. These are conveniently collected into one section of code reducing the effort of determining what conditions need to be examined and then locating them.

RLM reports a reduction in effort of approximately one third when using the new methodology. A standard function of a MMI is to indicate which position in the sequence of operation the control is currently in. The single coil per state implementation provides this signal directly. The indicator of current sequence step can be driven directly by the state coils. When designing the MMI one no longer needs to analyze the control code to identify the steps in the sequence. These are directly available from either the state machine diagram of from the standard code patterns. In the traditional control code, these were not explicitly implemented. One needed to have good understanding of both the machine operation and control code.

This body of research demonstrates that state machine based modeling and code development methods are superior to the unstructured methods used today for programming sequential logic such as that found in the component level of the abstract model. Based on this observation, the author designed a translation from the model to Ladder Logic for the system and subsystem level state machines shown in Figure 5-12 and 5-13 based on the standard state logic code topology shown in Figure B-2. The resulting code is shown for the system state machine Automatic and Automatic Desired States in Figure H-1. However, a detailed analysis and review with the RLM control engineer, led to a redesigning of the topology. The RLM engineer indicated that a more compact topology could be achieved using the traditional method of using indirect indicators of state such as mode selector switch positions. Further analysis indicated that the function was not likely to change and that only the interface states were needed for determining context by the monitoring system. A different Ladder Logic topology was

then developed using the traditional approach with the addition of encoding the context states as single coils. The RLM technicians are familiar with the traditional method so little training was required for an operational understanding. One of the major advantages of the state based pattern is the ease for a person to comprehend it and make manual changes. Since change to the function was unlikely and training fairly simple, these advantages were not an issue. The topology would remain unchanged and easily recognizable. Encoding the context information as single coils and generating the topology automatically, provides the monitoring system with consistency and facilitates locating this information. The particular Ladder Logic code pattern developed behaved in the manner modeled by the state machines. This is one of the desired features of OO design. Well-designed classes should be generic and representative of the application domain to the degree where they can withstand the need for change for most implementations. This could be achieved with the traditional method so little would be gained in replacing it with the new state-based pattern. Thus the decision was made to use the traditional method. The code for the system Automatic and Automatic Desired States based on state-based pattern shown in Figure H1 is replaced with the traditional code shown in Figure H-2. Examination of the logic reveals that the abstract model states are represented. The difference is that the mode selector switch positions are used to indirectly implement some of them. The code generator was written to output these Ladder Logic patterns for the system and subsystem level dynamic behaviour. The following general conclusion was reached from this analysis. Within an organization, there will likely exist a considerable investment in development of specific solutions or

patterns used to address common design issues. If these patterns are easy to comprehend, have not changed in a long time and are not likely to change in the future, they should be incorporated into the abstract model and into the code generator. Unless there is a rapid turnover of employees resulting in a potential savings in training to be realized from the easier to comprehend state based Ladder Logic patterns, the organization will realize a savings by incorporating such patterns in the application generator. As with the system and subsystem dynamic behaviour, these patterns should still be modeled with state machines to minimize initial comprehension and communication of the behaviour with others.

Generalizing from the RLM experience, other organizations are likely to have developed different specific solutions to the same design problems that fall into the category of inclusion into the application generator in their existing form. When addressing the issue of reuse through the use of models this research has determined the following. At the highest level of abstraction a general methodology or principle exists in a pure non-domain specific form. For the underlying principles of this investigation this is taxonomy (classification) and directed arc graphs. To make these principles usable, specific domains such as software and electrical engineering adapt them to their particular needs. In this case the adaptations are object-oriented analysis and design and nested finite state machines. However, on their own, these adaptations still do not provide a high level of reuse of investment in solutions. Parnas [Par93] points out that methods typically provide detail on the form but lack direction on the content. As indicated by Birla [Bir97], even with an extensive library of reusable components it is

still too difficult to decipher how to configure then into a control plan for a specific application. As observed by the author through many years of experience with control engineers, there are as many different ways to partition a solution to a problem into "reusable" components, as there are engineers. Each one brings their own experience and collection of approaches that they have accumulated through their career. Each is sufficiently different from the others that often, it takes more effort for someone not familiar with the specific partitioning of functionality to learn how to reuse the components than it would to develop them again. Two practices should be followed to enable an organization to effectively reuse a library of OO design solutions. First, a clear documentation of the partitioning and assignment of functions to a class must be developed. This research shows that a model represented with the OO paradigm supplemented with Nested Finite State machines can be used for this purpose. The second practice that must be followed is for personnel in the organization to understand and adhere to the partitioning of function represented by the model or else reuse will not occur. When new design problems occur that are not covered by the model, the organization must extend the model. The new solutions must build on the old as extensions protecting the organization's investment. Only when a new approach can show sufficient improvement to offset the retraining of everyone involved, should it replace an existing one.

8.3 Technician's Effectiveness Case Study

The metric of effectiveness applied to the object-oriented model supplemented with state machines for specifying dynamic behaviour can be observed in the following case study. A technician of typical capability was used in the capacity of troubleshooting systems manually designed and coded by RLM using the new methodology. The methodology was explained to him and system documentation in the form of graphically represented state machines was provided. The transformation of the state machine to Ladder Logic code was also explained. The technology insertion used the principle of presenting both the old method along with the new. Using standard patterns in PLC Ladder Logic code to represent the elements of the state machine provided the familiar method while the state machine diagrams represented the new method. The technician could see and relate to the pattern in the generated code and gain an understanding of the state machine approach. The first indication of the acceptance of the new method was the technician's insistence of being provided with the state machine diagrams before going on a service call. This showed that the model was indeed being used to help troubleshoot the machines. The technician had considerable previous experience with the unstructured approach to control logic where one used only the code listing when diagnosing problems. Thus he replaced his previous practice of mentally relating the code to machine operation through observing the two and relying on previous history with the use of the state machine representation and mapping to code. By debugging several systems, he learned the state-based technique through a process of familiarization. The RLM control engineer reported that during this time the technician required less support

both in gaining understanding of a particular system and resolving complex problems than was previously needed for traditional systems. The engineer attributed this to the organized structure provided by the methodology. Hence information provided by the structure reduced the time needed to learn. In addition, the technician could handle a greater level of complexity. After three months of debugging state-based systems, the technician was able to make changes such as adding new states. Specifically the technician actually made the changes to the state-machine diagram before he modified the code illustrating the value of the methodology. Typically, documentation methods are used after the code is written. The documentation is regarded as more for the use of others rather than as an aid to designing the solution. After a year of troubleshooting and making modifications to the state machines, the technician acquired a sufficient level of skill to design the controls for entire new systems. He was promoted from the position of a troubleshooting technician to the responsibility of a control designer. The RLM engineer stated that such a transition is not common for technicians of typical capability, citing the new methodology as the reason for the exception to the norm. He stated that compared to the unstructured method used in the industry, the state-based approach developed from this work was easier to learn and accommodated greater levels of system complexity.

8.4 Applicability of Methodology to a Different Domain

Having completed the abstract model based on the Wrapper Line as a representative control system for roll-forming systems, a different domain application was sought to determine adaptability of the new methodology. RLM is considering retrofitting the control of some of their older computer numerically controlled (CNC) machines (computer controlled lathes, milling machines and machining centres). The mechanical construction of these older CNC machines is superior to that of currently manufactured machines. However, the controls of the older machines are no longer reliable and their repair is very costly. The hierarchical levels of the abstract model were found to be appropriate to the CNC machines. An overall coordination of the various sections of the machine was still required. Partitioning into subsystems could also be used with a smaller scope. A separate commercially available continuous motion controller that controls the axis of the CNC machine was treated as a subsystem, although PLCs were not required. Analysis of the remaining functions of a representative CNC machine such as a lathe showed that in addition to the axis, they could be partitioned into several other distinct subsystems. These subsystems included: tool-changer; lubricator; control of tailstock position along with pressured applied to hold the work-piece; and headstock speed gear changing. The control of such devices required a sequence similar to that found in the Wrapper Line. Thus the same PLC control code development methodology could be used. Two approaches were available for implementing the system level control. The axis controller kept track of a similar set of modes. These could be read by the PLC and then used to generate the same interface conditions as

before, instead of implementing the system state machine. The other approach would use the same PLC implementation of the system level with the interface conditions being read by the axis controller and used to set its modes. The RLM engineer who sets up the CNC machines was asked to evaluate the two approaches. He observed that there was more flexibility in implementing the system level control in the PLC. The PLC was designed as a general-purpose controller while the axis controller was specialized to numerical axis control. It used the "G code" language that was well adapted to axis control but lacked the flexibility of expressing logic functions that is available in the IEC 1131 Ladder Logic language. His conclusion was that a less complex and simpler to understand overall system design would be achieved with a PLC implementation of the system level control functions.

The RLM technician who repairs and maintains their CNC machines was interviewed to gain insight to the requirements for the subsystems. The first subsystem analyzed was the tool-changer. There were a wide variety of tool-changers with differently implemented control strategies. It became quickly apparent that a simple application of the state based methodology by the technician would result in different solutions to each case influenced by his familiarity of the existing hard-wired mechanical relay controls. The RLM engineer felt that a general design pattern could be found that would apply to all of the cases.

The author utilized knowledge and feedback from the technician to develop such a general pattern for the tool-changer. The approach most likely to be taken by a

technician would be directly mapping each tool-changer position to a separate state. Observing the normal automatic operation one will find that the tool changer can be commanded to stop at any tool position. When moving from the current tool to the selected one, the tool-changer released a clamp and advanced at a high speed. When reaching a tool position close to the selected one, the motion is slowed down to prevent overshooting the target location. When the desired location is reached, motion is stopped and a clamp applied. This sequence is based on tool locations so the simple mapping would be to use a state for each tool position. Unfortunately, different machines have different numbers of tool positions resulting in different numbers of states required. Applying OO analysis one can group the tool positions into a small number of classes. These classes then become states. The first class to consider is the one where no motion is required. This occurs when the tool-changer is at the desired position. The next class is the one where rapid motion can occur. This includes all of the tool positions that are far from the desired position. The last class is the one where only slow motion is allowed. This class includes the tool positions that are close to the desired one. The resulting automatic mode states shown in Figure 8-1 become: at desired position; advancing rapidly; and advancing slowly. The events or conditions needed for the state transitions are: "not at desired position"; and "near desired position". The occurrence of the "not at desired position" while in the "at desired position" state, event causes a transition into the "advancing rapidly" state. The "near desired position" event occurs as the tool changer approaches the desired position resulting in entry into the "advancing slowly" state. In this state, the deactivation of the "not at desired position" condition

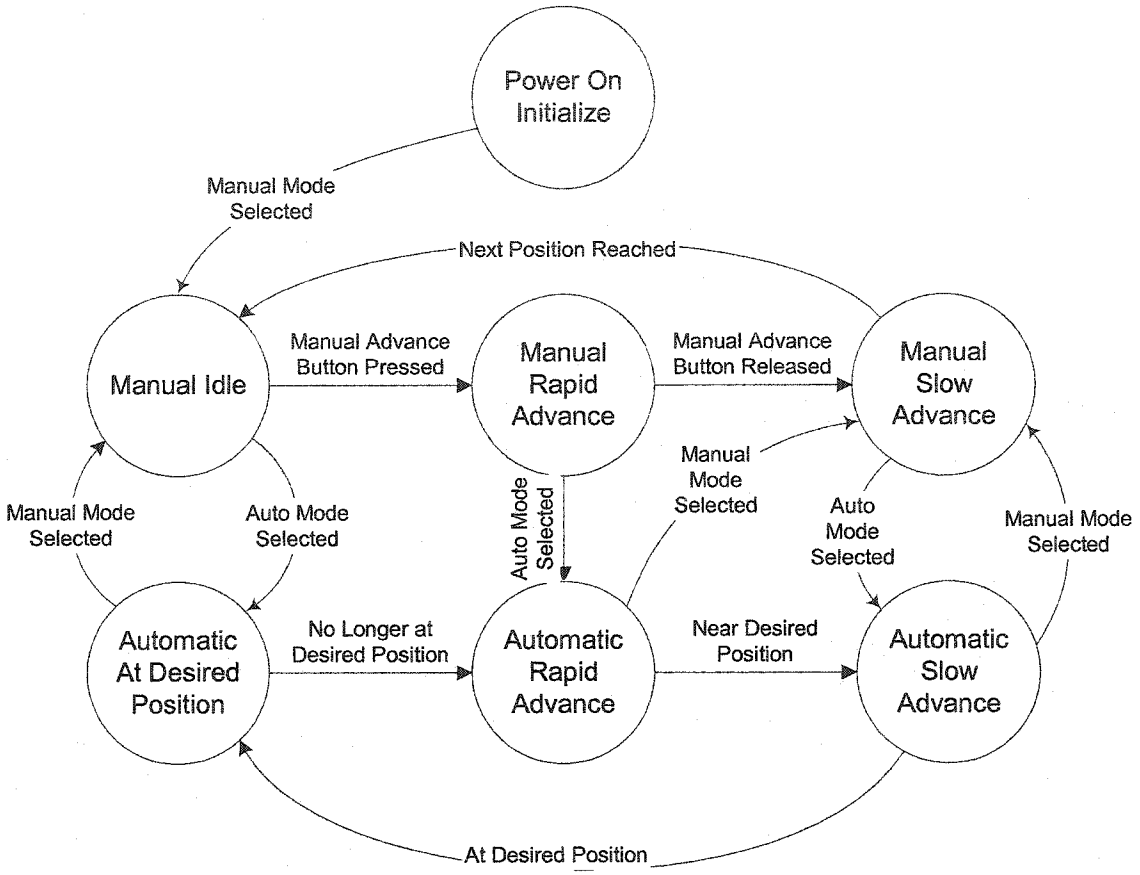


Figure 8-1: Tool Changer State Machine

signals the arrival at the correct position and entry into the "at desired position" state where motion is stopped and the clamp applied. A simplification can be applied to reduce the number of positions that need to be decoded for these conditions. The tool-changer is constructed so that it always moves through the same sequence of positions. Thus when selecting the positions that are near a desired position, one only needs to specify the first position. All the intervening positions between it and the destination do not need to be decoded since a transition into the "advancing slowly" state has been

made. In the case of a single direction tool changer only one position, the first of the "near" ones needs to be identified for each possible destination position. A mechanism is needed to select the appropriate "near" position. This can be accomplished by qualifying the "near" position with the code for the currently selected destination position.

Using the states and conditions described provides a general solution for a tool-changer of any number of positions. The user first needs to supply the number of positions that there are. Then he needs to provide the number of inputs from the tool-changer used to encode the positions and the number of inputs from the CNC controller that request these positions. From there, the specific code patterns for these inputs need to be specified. Then the user indicates which position is the first "near" one for each destination position. The only remaining information needed is identifying the outputs that cause the needed actions. From this information, code for the specific tool-changer can be generated. The application generator was expanded to first collect the number of positions, position encoder inputs and CNC controller selection inputs. This is accomplished with the screen shown in Figure 8-2. Having obtained this information, the application generator then modifies the screens shown in Figure 8-3 to collect the selection and position encoding information and the identification of the first "near" position for each possible position. When this information is complete the state machine and its various contained objects such as states, transitions, events, and state actions are automatically created. From that point PLC Ladder Logic can be generated in the same manner as for state machines defined by the user. The application generator does not treat the tool-changer state machine differently.

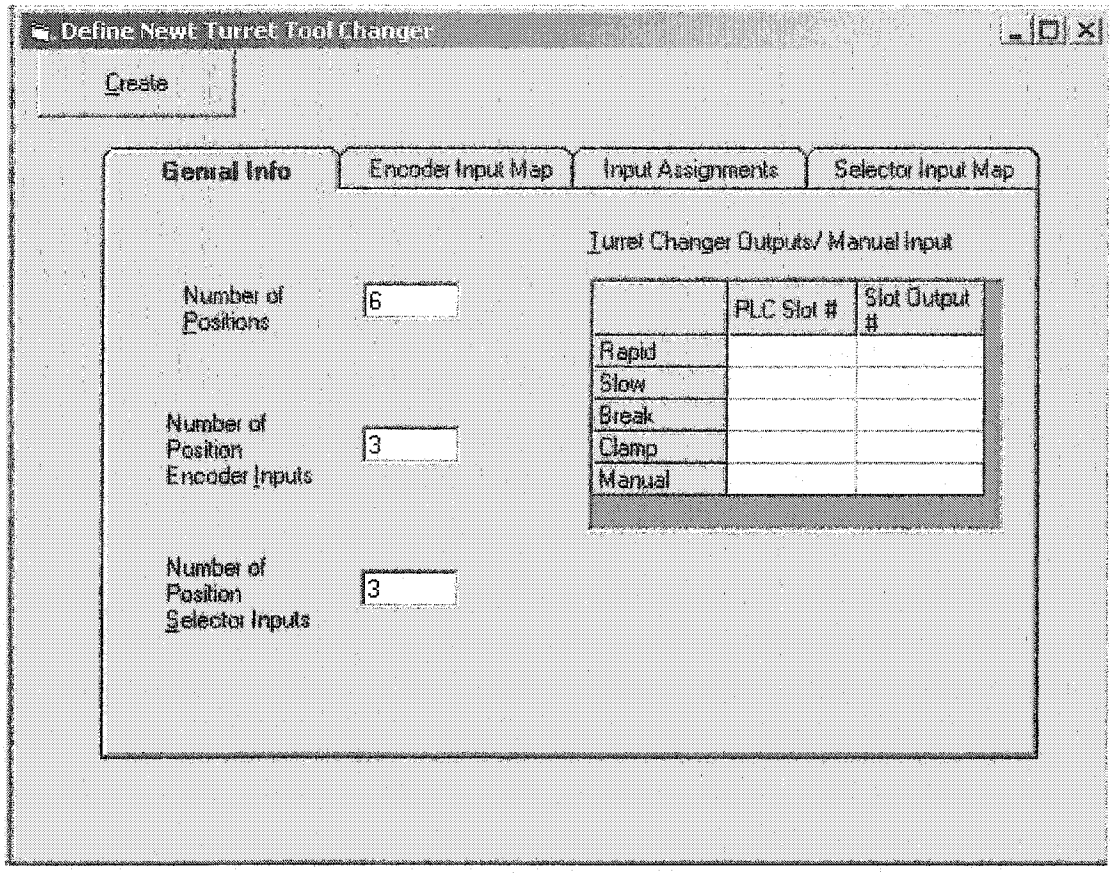


Figure 8-2: Tool Changer Configuration Screen

Contrasting the state machine of the tool-changer with the Wrapper Line Destacker component shows a different type of design. The steps of the Destacker sequence are reflected directly in states of its state machine. The physical conditions and events that lead from one step to the next have exact counterparts in the model. Anyone familiar with the Destacker operation can readily specify the model. On the other hand, the tool-changer model is an abstraction of its actual sequential operation. Instead of using states for each tool position, abstract states representing groups of positions under different conditions are used. The selection of a specific first "near" position using the current

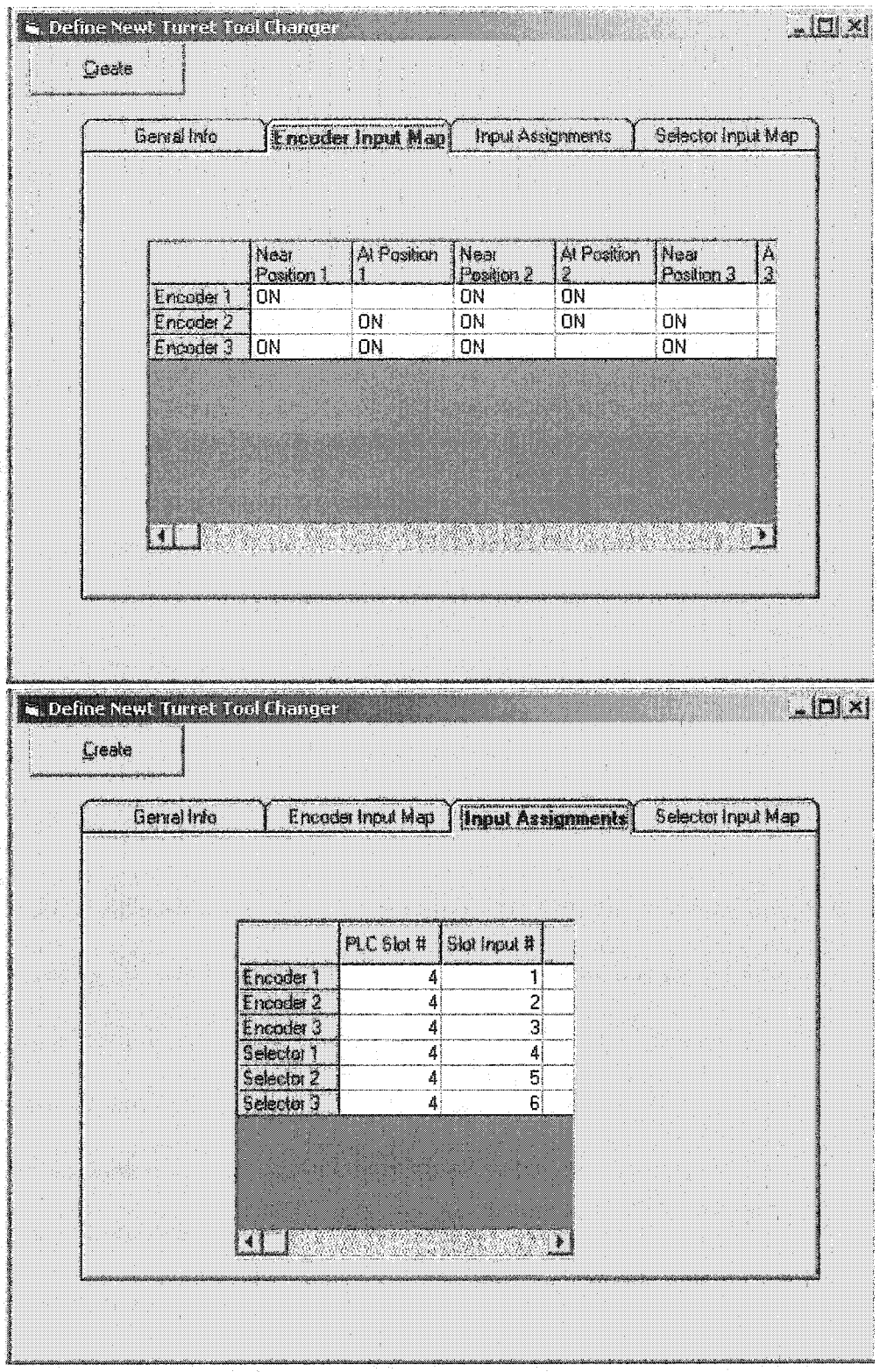


Figure 8-3: Position, Encoder and Near Position Information Input

destination tool as the qualifier allows the use of a general condition for all destination positions. These abstractions make it more difficult for a technician to understand the implemented code. However, using the abstraction provides the application generator with a configurable solution that is independent of the number of actual tool positions. In addition, the skill level to derive such abstractions is higher than would be expected from a technician.

Having completed the tool-changer design, the lubrication subsystem was addressed next. This subsystem was different from the others modeled so far in that it needed to keep track of conditions for which there was no direct input and it was to provide a mechanism to identify faults and clear them. This required an approach that was different from the standard mapping of sequence steps to state machine states. The state machine model is shown in Figure 8-4. When power is first applied to the system the "power-on-initialize" state is entered. When the automatic mode is selected the "cause lubricant to flow" state is entered and lubrication of the system occurs. This insures that the machine is lubricated when it is first turned on. An accumulator of time (implemented as a retentive timer) elapsed since lube was applied is set to zero. An input such as a pressure switch located at the exit of the lubricant line indicates that the lubrication was successful. At that point, the "lubricant being consumed - wait for next lubrication" state is entered. This state is used to keep track of elapsed time and is maintained as long as the machine is performing an operation that consumes lubricant. Inputs indicating such actions are used to create a condition indicating the presence of machine motion that needs lubricant. Should this type of motion stop, the

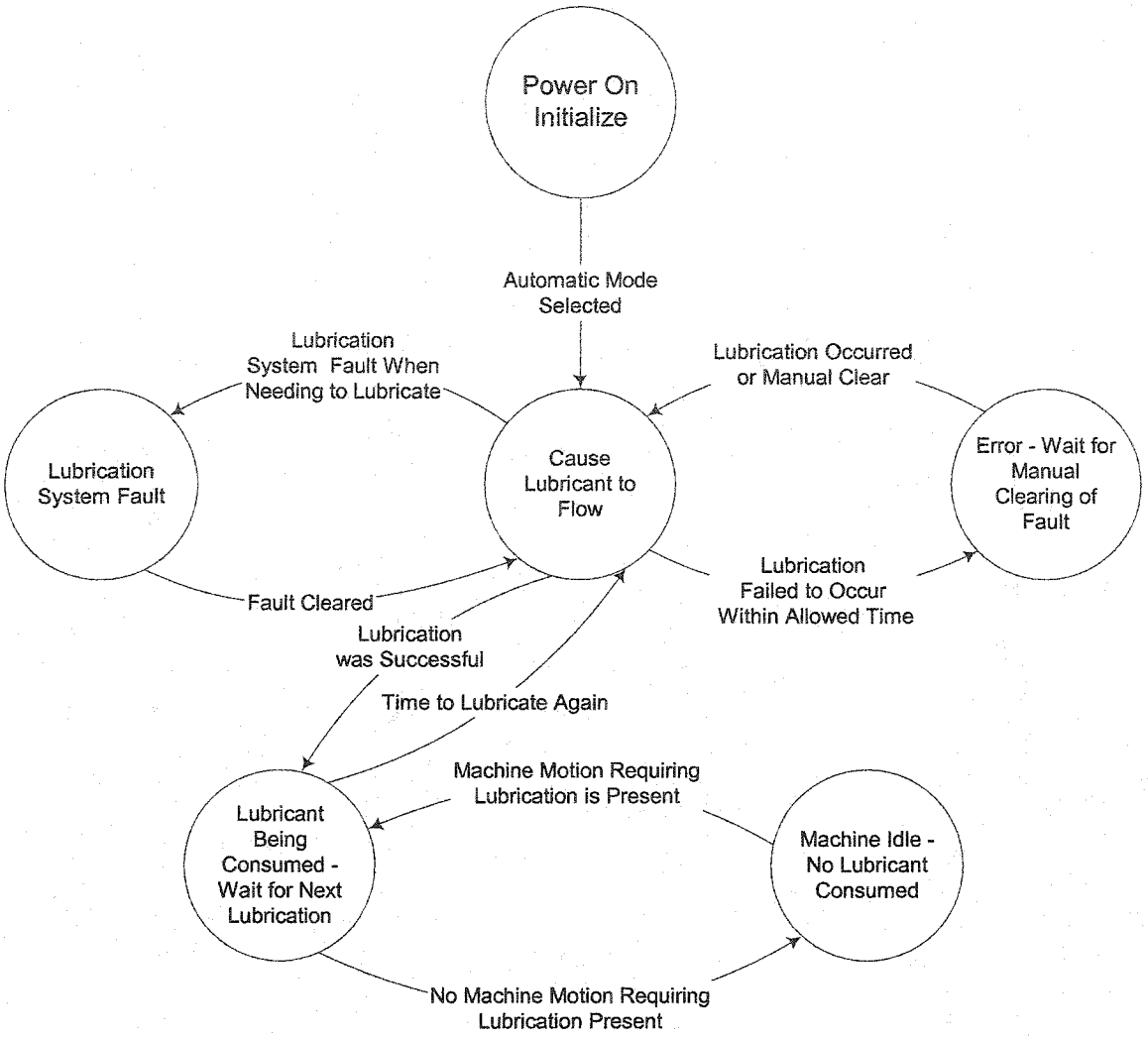


Figure 8-4: Lubrication Subsystem State Machine

"machine idle – no lubricant consumed" state is entered. In this state the time accumulated while machine motion that consumed lubricant is held at its value when the state was entered. Basically a retentive timer is stopped from accumulating more time. When lubricant consuming machine motion commences again, the "lubricant being consumed - wait for next lubrication" state is entered again. In this state, the time accumulation continues from the value at which it had been interrupted. When the timer accumulates the amount of time that indicates sufficient lubricant has been consumed, the "cause lubricant to flow" state is entered. In this state the accumulated time is set back to

zero and lubricant is applied to the machine. If the lubrication was successfully applied the sequence repeats.

Two failure use case scenarios need to be addressed. The first is a fault that can be directly recognized from inputs. This includes such conditions as low lubricant in the supply storage or in the case of air activated lubricant pumps, low air pressure. The CNC machine service technician indicated that a fault should not occur in such conditions until the machine was actually being lubricated. Separate indicators of these conditions exist to alert maintenance people of these situations. The fault is used to stop the CNC machine. It is desirable to run the CNC machine as long as possible so there is no need to stop it until the conditions prevent a currently needed lubrication as opposed to one in the future. In most cases the maintenance organization will correct such conditions before they will cause a fault. In the event that the conditions persist to the point in time that lubrication is to be attempted, the "lubrication system fault" state will be entered. Lubrication is not attempted since some of these conditions, such as lack of lubricant can cause air to be introduced into the lines that will subsequently need to be removed. This state is maintained, informing the CNC machine of the fault and preventing further attempts to lubricate. The inputs that monitor the fault conditions are used to indicate when the fault condition is corrected. At that time, the "cause lubricant to flow" state is entered and lubrication commences.

The second type of fault is one where lubricant is being pumped but does not reach the end of the lubricant line (used as an indirect method of determining that lubrication

occurred) within the expected time. The cause of this fault is not provided by inputs. The "error – wait for manual clearing of fault" state is entered and the CNC machine is informed of the fault. This state is maintained while a maintenance person diagnoses and corrects the problem. When the problem is corrected and lubrication does occur, the "cause lubricant to flow" state is entered. An alternate means to enter this state is the pressing of a "manual clear" button. The use case is one where the maintenance person has cleared up the problem and then needs to run the lubrication pump.

The lubricator and tool-changer applications demonstrated the applicability of the methodology developed for roll-forming systems to be applied to the different domain of CNC machines. The use of a state machine for the lubricator allowed the author to effectively communicate with the technician when gathering requirements. The design went through several development iterations where new requirements were identified or existing ones changed. Use case scenarios were used to describe the requirements but on their own did not provide a compact manner of showing all possible behaviour, as did the state machine. When reviewing the functionality with the technician the state machine was consulted to see what would occur in any given situation.

The CNC applications indicated a need to expand the application generator's representation of a control engineer's knowledge. In the roll-forming systems, a technician could create a sequential component base on his knowledge of machine operation. He simply mapped the sequence into states and assigned inputs to conditions used by state transitions. To cause a particular machine motion required assignment of

an action to a state. There was a one to one correspondence between the machine sequence and I/O and the state machine model. The framework of the other levels of control was automatically provided. In the CNC case such a direct correlation to the model developed was undesirable due to the multiplicity of different tool changer configurations. To enable a technician to generate one standard code pattern required an extension of the application generator. The author needed to form a specific configurable solution, which then needed to be coded into the application generator. In effect, each specific solution can be modeled as a specialization of the sequential component class. The control engineer's knowledge is captured in these classes and the ability to be applied by others is supported by the application generators screens that allow configuring the general pattern to the specific application.

Experience developing the tool changer and lubricator components led to the conclusion that such general solutions or patterns needed to be included in the abstract model and supported by the application generator. The author was trained in applying the process of abstracting a general solution from a specific case. The technician was not and thus could not be expected to develop a general design pattern. He could develop custom solutions to each type of tool-changer or lubricator. The RLM control engineer judged that this was inferior due to the proliferation of different code, with the additional effort expended in developing and later maintaining unique solutions. A common approach that was configured to the specific need by the technician avoided these costs. The general design patterns developed by the author were hard coded into the application generator. The RLM control engineer's experience indicates that such design patterns

will continue to emerge through time as an organization develops new systems with new requirements. The analysis of the specification and implementation of such patterns with the goal of obtaining a general OO abstract model of these design patterns is a topic for future research. The goal of such a model would be its implementation in the application generator in the same manner as the control model. The intent would be to enable an organization's control engineer to extend the application generator with such new patterns without the need for custom reprogramming of the generator. The rest of the organization could then configure these new patterns to address the new controls requirements.

8.5 Application Generator Feasibility

The capability to creating an abstract model and generate PLC Ladder Logic code for the Wrapper Line by the application generator was demonstrated. The same partitioning with the generator as was manually used for the actual machine. The majority of objects shown in Figure 5-18 were instantiated by the application generator. Only the "sequential constraining condition" components had not been implemented due to time constraints. They require the same kind of logic as the sequential components. The difference is the omission of the mode subchains since the constraining condition is present in all of the modes and the need to use logic to unlatch latched coils. One can generated their code by creating a normal state machine, automatically generating the code, then manually removing the mode subchains, replacing the normal coil for the latched relay with a latched coils and adding an unlatch rung. After creating the object

model, Ladder Logic code was generated and compared to the actual code that had been manually created earlier. The generated code was targeted for a Siemens PLC while the actual code had been written for an Allen Bradley PLC. With the exception of formatting differences, the same functionality was present in the application generator code for the implemented objects. The next step was to test the code. For this purpose the functional physical model shown in Figure 7-6 was used since the actual machine had been shipped to the customer by the time the application generator was completed. The design was modified to account for the differences between the model sensors and actuators and the actual machine. Code was automatically generated for all but the "sequential constraining condition" components. The missing code was added manually and the PLC loaded. The system was functionally tested and operated in the same manner as the actual machine. To prove that the application generator was capable of creating systems other than the one on which it was based, the training system shown in Figure 7-8 was used. The application generator was successfully used to create the PLC control code and its correct functioning on the training model was verified. The successful completion of these two experiments demonstrated the feasibility of implementing an application generator based on an abstract object oriented model using state machines to specify dynamic behaviour. The resulting object model and standard control code topology provide context-enhanced data for monitoring systems. Constraining control code modifications made only via the application generator keep the monitoring system synchronized with the evolution of the control system.

8.6 Application Generator Usability

The usability of the application generator was assessed through observation of the tradesman's ability to interact with it during the experiment described in chapter 7. The metric applied was how often the user needed assistance when interacting with the application generator. Ideally, the user would not need to refer to guides or need any assistance. In addition, the length of time taken to accomplish a task was measured.

At the start of the experiment the abstract model was verbally presented and notes were provided. Examples of typical plant machine control functions were used to explain the functions in the model and their placement in the different classes. The tradesman grasped the concepts easily being able to relate them to the machines with which he was familiar. The next step consisted of working through a training manual that showed in detail how to construct an entire control application using the training model. The first difficulty that became apparent as the tradesman started to interact with the user interface was related to using the mouse. He is right handed but due to the type of work he has been performing for many years, he had developed carpel tunnel syndrome and needed to use his left hand. His fine-motor hand control was far less accurate than the author anticipated when designing the user interface. The characteristic of smaller interface objects requiring more time to be selected predicted by Fitts' Law [Ras00] was greatly magnified by the journeyman's condition. Figure 8-5 is the interface screen for defining state that the user interacts with the most. There are many small sized objects that the user must click on with the mouse. The targeted user of the application generator is a

tradesperson, and this type of injury is common amongst them. Thus the problem would affect a large segment of the intended users.

The next difficulty observed was caused by the abstract nature of the application generator. The foundation of design was based on the abstract model. Each major class was represented with its own user interface screen. The user was expected to find the appropriate screens in the correct order, and create the objects based on the classes. The hierarchical layering was used for navigation. To access the form representing a particular object one would start at the top-level class and work down through the classes contained within it or associated with it. Thus to access a particular state one would start at the system form, select the system the state was in, then select the subsystem, then the component and then the state machine. The selections were often from lists where the individual items were narrow rows making it difficult for the tradesman to manipulate the mouse. To effectively use the application generator one needs to be intimately familiar with the abstract model. This proved to be a disadvantage. Knowledge of the abstract model relations between the classes was needed to know which forms to navigate through and in which sequence. Overall, it requires far more user activity to perform a specific input than a tradesperson typically has patience for. The method one would use to quantify this activity in terms of typical time is the GOMS model [Ras00]. This yields a typical time that needs to be adjusted to include degradation of fine-motor motion. Based on the tradesman's questions it was clear that he understood the abstract model and how to partition the experimental training system into system, subsystem and the various components. The difficulty was in building this model using the application generator.

The objects that were created were displayed on separate screens in lists. The relationships between them and the overall structure were not visually displayed. The user was expected to first form the image of the objects needed in their mind and then enter it. The author had no difficulty with this approach but he had been working with the model for several years and was thus very familiar with it. The tradesman felt the problem could be improved if the application generator explicitly showed him the parts of the model that he had already built and how they fitted together. In addition, he had difficulty finding where to input the information that was needed. For example, he knew that there is always a selector switch to set the system into automatic. But it was not clear to him where one would input the information about it. He did not automatically relate the switch with the two inputs labelled "Auto S/Sw" and "Manual S/Sw".

The labels and captions used were an additional area that needed improvement. The names of objects were selected to be descriptive and precise resulting in long phrases. Unfortunately the tradesperson found them confusing. As stated by Raskin, a "natural" or "intuitive" interface really translates into one that is easy to use due to familiarity. Names of objects and functions familiar to a tradesman were easier for him to understand. Thus the experiment was halted and with critique from the tradesman, new names were selected. For example "request manually held automatic" was replaced with "press & hold auto" for the function of a button that needed to be pressed and held to allow the system to cycle automatically. In addition, with feedback from the tradesman, the interface was simplified, removing any graphical objects that he did not absolutely

need. The improvement in speed to use the interface when removing the need to choose amongst alternative actions can be determined by applying by Hicks Law [Ras00].

After these changes were implemented an unacceptable degree of difficulty was still experienced by the tradesman. The conclusion was reached that the application generator user interface needed considerable improvement. It may have been suitable for engineers who are use to abstract reasoning with minimal visual aids, however, trades-people work at a more concrete level in their day-to-day activity. The application generator user interface underwent a third pass redesign to address this issue. The ZIP paradigm was selected to provide a more tangible, less abstract form of interface. The steps screen shown in Figure 8-5 was replaced with the screens shown in Figure 8-6 and 8-7. Selection lists and option buttons were replaced with large individual buttons. Instead of selecting from a list and setting options to further characterize an object, all variations were explicitly represented with large buttons. The large buttons now had space to better explain their function. Relationships and detailed information was explicitly shown. This approach was used instead of selecting the main object from a list such as the state and then changing the other lists to reflect the objects such as actions and state transitions and deducing the state machine sequence from the transitions. One is graphically presented with all of the states and their sequence. One simply selects a particular state and "zooms in" to navigate to its detailed information. The lists and

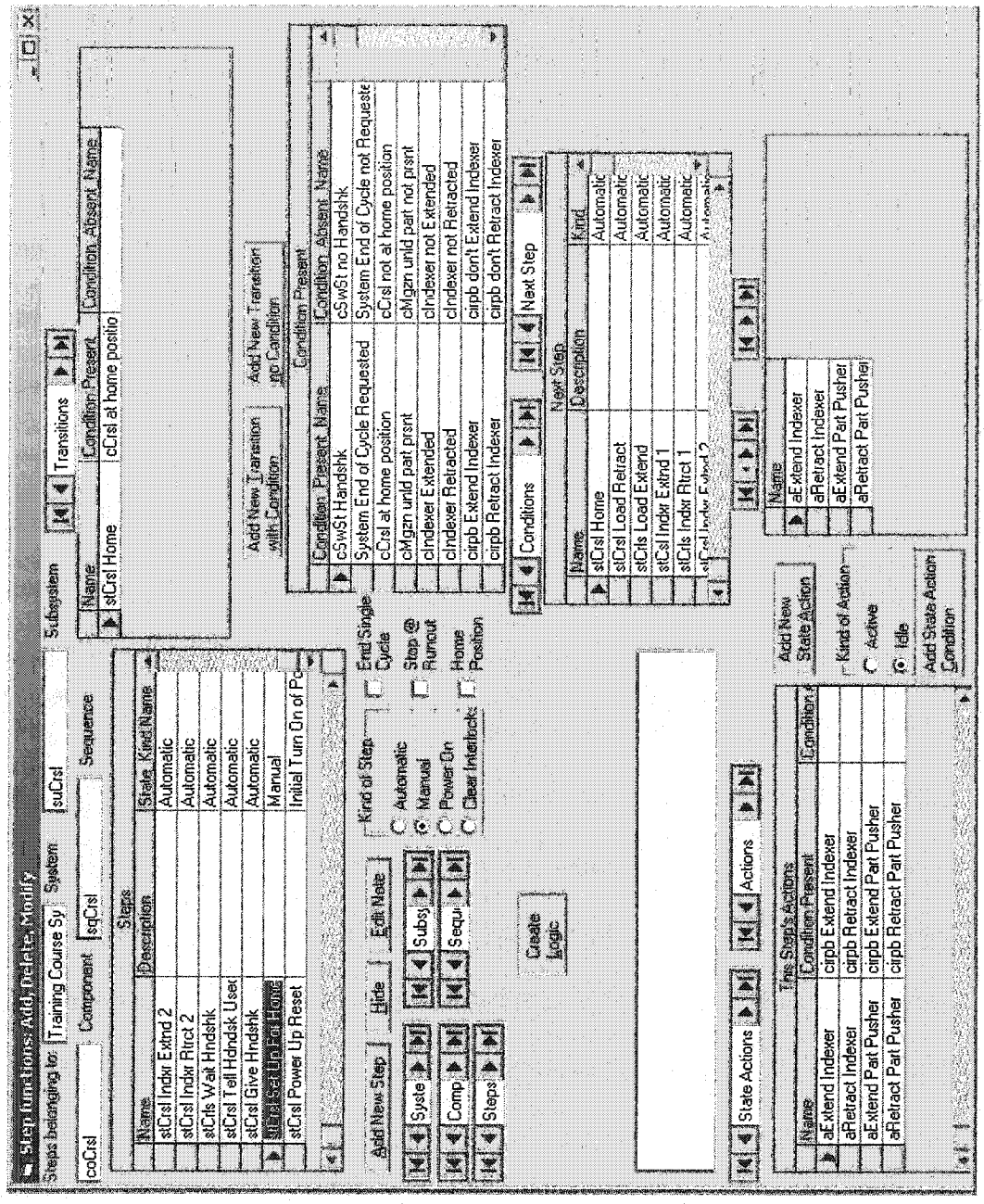


Figure 8-5: Windows Format Steps Screen

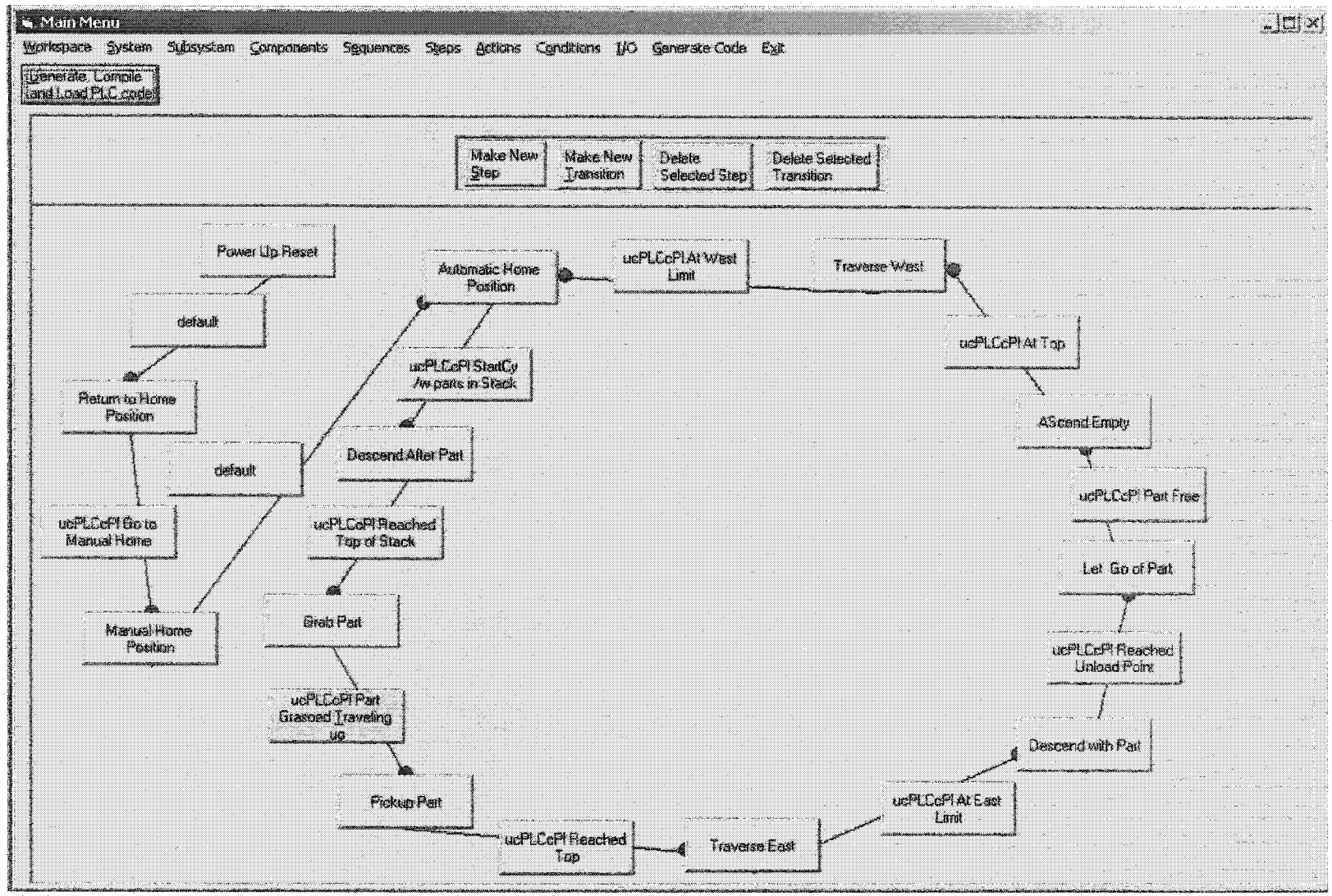


Figure 8-6: ZIP Format State Machine Interface Screen

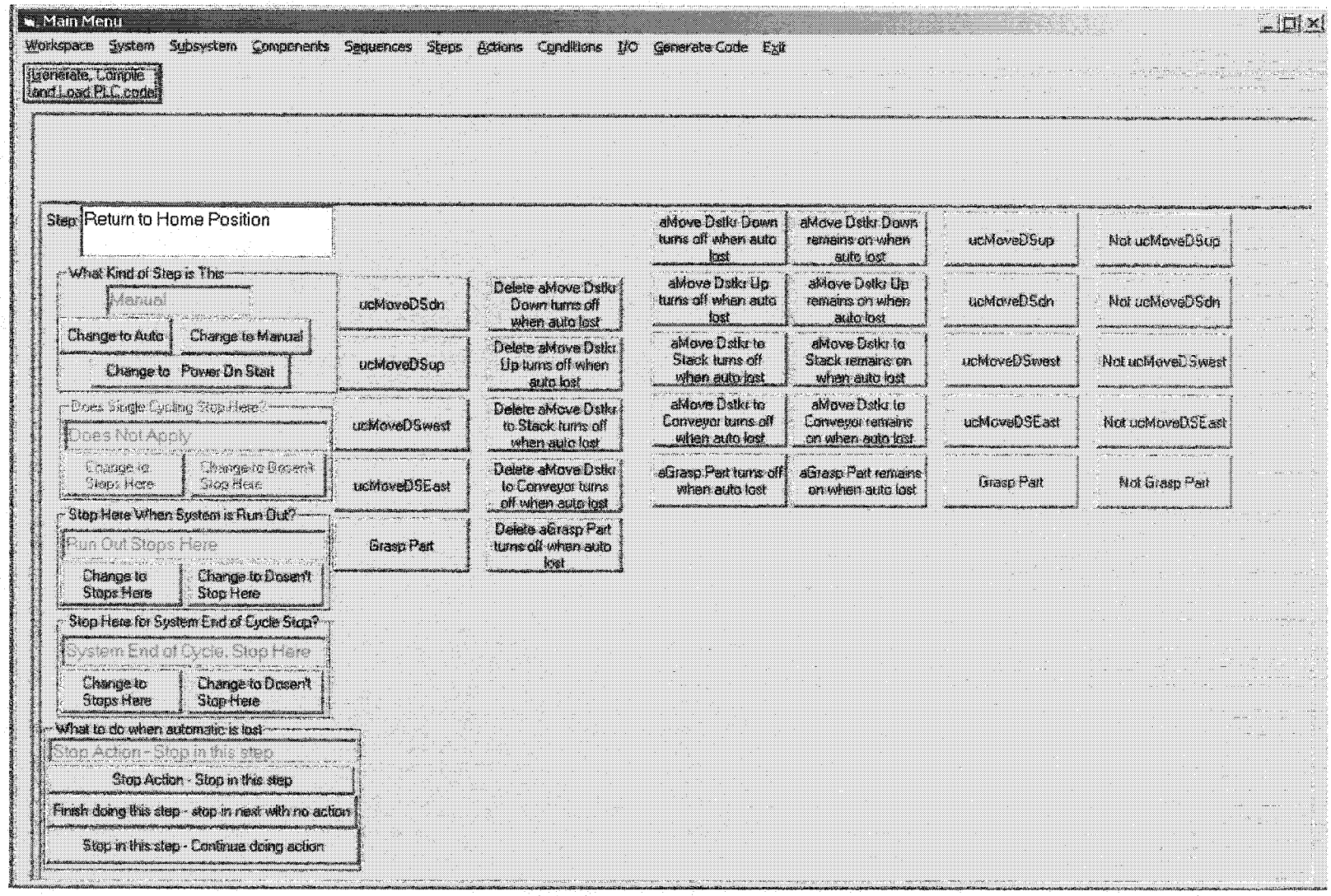


Figure 8-7: ZIP Format State Information Screen

options on the detailed view are also replaced with large buttons that explicitly perform functions. For example, there are separate buttons for selecting the output behaviour such as "finish doing this step-stop in next with no action" instead of small radio selection buttons.

The tradesman recommended further improvements. The application generator should provide a utility to automatically create any pattern that was common. For example, interface between state machines implemented as "partial observation" is implemented in two ways. The first is the use of the "sequential constraining condition" component. This was renamed to "mailbox handoff" to provide a familiar analogy that should reduce the amount of explanation needed [Ras00]. This type of interface keeps an internal state machine allowing asynchronous interaction. Basically one component indicates to the interface that it had reached a point in its sequence where it was ready for the next part and continues on. The other component eventually has a part available, checks the interface, provides the part informing the interface and moves on. The user needed to create each individual state and condition to create the interface state machine. To do this he needed to either remember the pattern or refer to the training information. The recommendation was to provide a screen where the unique information would be entered. From that information the interface state machine would be automatically built as well as the states in the two interfacing state machines that interact with the interface. This is in keeping with Raskin's principle of a "humane" interface where the computer is of service to the user instead of the user performing needless entry for the benefit of the computer. The second type of interface was synchronous where one state machine

needed to wait on the other. This was implemented directly in the two interacting state machines and was named "baton handoff" to describe the behaviour. Again, a very specific configuration of states and conditions were manually entered into both interfacing state machines. The RLM engineer and the author have both observed that in the traditional controls approach interfacing between sequential components causes the most difficulty both in design and in debug. Many different variations are found. The conclusion reached is that this is a more abstract concept than stepping through the sequence of machine operation. Thus the generator should automatically create these configurations instead of relying on the user to manually enter them with the need to remember how they are constructed. This particular enhancement is left for future work.

During the training session, the tradesman provided his experience with control systems to create specializations of the "non-sequential component which must be active to allow automatic behaviour" class. The name was changed to "needed for auto" since it was descriptive enough to recognize a type of control that was common to most systems. It is common for systems in an automotive plant to have "needed for auto" type of devices controlled from a central location. Thus the local control code generated for this type of component is not needed. However, the condition that indicates that the device is active is still required since the system cannot operate without it. To accommodate this situation this class was specialized to create an "externally controlled" class. The control code generated is simplified to an input that indicates that the externally controlled component is active driving the interface condition that informs the rest of the system. Two other specializations were identified. They provided the capability to keep a locally

controlled device active in the case of E-stop or when the subsystem mode selector switch was set to the off position.

The portions of the application generator that were used the most for the experiment were rewritten using the more concrete ZIP and incorporated the tradesman's feedback. A marked improvement in the tradesman's performance was measured. Using the new interface he was able to create the wrapper line system, part loader subsystem, sequence and all of the states in 33 minutes. He spent three, 2-3 hour-long sessions to accomplish the same amount using the previous interface. Using the old interface, he needed 20 minutes to create 4 conditions. Using the new interface he create 14 transitions with 12 conditions in only 24 minutes. Using the new interface he assigned 15 actions in 22 minutes. He never reached this point with the previous interface. These measured improvements as well as the feedback from the tradesman show that the more concrete ZIP type of interface, where all created objects are graphically displayed, navigation through moving between more and less detailed views and the use of large graphical objects with adequate descriptions of their function, was more suitable than the traditional Windows type for applications when targeted at trades people.

To support the experiment, an extensive training course was developed. A detailed training manual was written that was to be used with the training functional physical model. The general philosophy was one of information hiding where the user was presented with the minimal information needed at a higher abstract level. The focus was on the steps needed to identify the needed information about a system and then enter it

into the application generator. The details of what these steps produced was purposely omitted on the premise that a tradesperson would be more interested in getting the job completed than understanding how it was done. The sequence was presented with pictures of the screens and examples of how to input the information. PLC Ladder Logic code and a symbol table were generated as text files that needed to be manually entered into the PLC editor. Then a PLC compiler was applied to generated code that then needed to be downloaded into the PLC and executed. As the tradesman began using the training course it became quickly apparent that the basic premise was incorrect. He was uncomfortable approaching development of the control code, as executing a "ritual" that he did not understand. The author's hypothesis for this discomfort is the environment in which the tradesman works. Down time caused by a malfunction is very costly in an automotive plant. A tradesman needs to have a wide range of knowledge to quickly diagnose and correct such problems. Their comfort level is proportional to their understanding of the equipment that they are responsible for. In the author's experience of supervising maintenance trades people, he noted that they were adamant at being involved during the commissioning and debug of new systems. They stated that this was one of their best opportunities to become familiar with the details since they would be exposed to many problems in a short period of time. The tradesman was not previously involved with the control code in his capacity as a pipe fitter. However, as soon as he started to use the application generator, he felt the need for detailed understanding what he was creating so that he could service it effectively in the future.

A further problem with the training was the required amount of manual manipulation of different tools. The only place in the tool chain where the user enters information needed to create the control code is in the application generator. The remaining steps are only of a clerical nature but they still require knowledge of how to perform them. The tradesman explained that if he used the entire tool-chain on a daily basis he would become familiar enough with it. However, the experiment was conducted in over 20 two to three hour sessions with several days in between them. This was not sufficient for the tradesperson to be able to use the tool-chain without constantly needing to refer to the training notes. This additional time and effort negatively impacted his learning of the methodology. To address this issue modifications were made to the application generator. First, the PLC code output format was changed so that it could be input directly into the Siemens tool chain. This removed the need for time consuming, error prone user entry. Then the manipulation of the Siemens tool chain was automated and controlled by the application generator removing the user's need to know its details.

The training approach was reworked to accommodate the need to understand what the application generator was creating. First the tradesman was taught how to read PLC Ladder Logic. Then the manual was expanded to describe the PLC code that was created by each step of the process. The tradesman did not need to learn how to manually create the PLC Ladder Logic code patterns, but he did need to know how to recognize then and understand what they did. He needed to understand the consequences and the results of his inputs into the application generator.

The training material had been revised and the main portions of the application generator rewritten. The tradesman then continued with the experiment. Time constraints prevented his completion of the entire training program. He started on the wrapper line model once he felt he had sufficient familiarity to construct a subsystem. He successfully used the application generator to create the system, subsystem, component and state machine for the Destacker subsystem. He used the automated Siemens tool chain to read in the generated code into its editor, compile it, load it into the PLC, display it in Ladder Logic format and run the program. During this experiment it became apparent that the effort spent in teaching him to understand Ladder Logic was needed to support the iterative design process he used. This knowledge provided the ability to observe the Ladder Logic execution, force control bits on and off and relate this back to the model. The tradesman stated that this was integral to his development and diagnosis of the control code.

8.7 Divergence of Monitoring and Control Systems and Context Enhancement

The automatic generation of control code, combined with the information captured by the model (built in the application generator), provides the desired synchronization and context enhancement. By providing the required context (the states and conditions of the control system needed by the monitoring system) as single, specific bits in the control code, the need for custom monitoring PLC code is eliminated. The location and identification of these bits are provided by the information in the application generator. The monitoring system only needs to read the bits at a rate that provides the desired

accuracy to time the changes from one state to the next. Experience with monitoring systems in the Oshawa Car plant indicates that 10 samples per second are sufficient. This rate is easily achieved due to the relatively small amount of data that needs to be read and high bandwidth of PLC communication channels. Initial interpretation of the function of control code prior to implementing the monitoring system is not required since all of the information needed already exists in the model from which the control code was generated. The use of the application generator to modify the control code provides an up to date model for the monitoring system addressing the synchronization issue. By reading in the entire PLC control data and time stamping it, context in terms of the state of the various portions of the control system is available. The collected data can be later sorted based on the various states. For example, cycle times for normal automatic operation can be obtained from the time stamps of the state machine state sequencing filtered based on the system state machine state.

The manual use of the methodology can also reduce much of the synchronization effort. As was observed with the technician, designing and making changes at the model level first, followed by a mechanical transformation to Ladder Logic is feasible. If this sequence is followed, the information needed by the monitoring system will be available. It will not be in a machine-readable form such as the application generator's database, but at least it will be complete and current.

8.8 Experimental Model and Support Environment

An objective of this research was to provide an environment where other research could replicate the experiment that was conducted as well as study other methodologies. The issues that arose while adapting software based simulators of representative industrial machines resulted in changing to a physical functional scale model. Modeling examples based on Lego component of an operating gantry mill [Gan01] and an assembly system [Cas99] were analyzed. The methods they used were applied to build a model of the Wrapper Line. The results were unsatisfactory. The construction was insufficiently rigid to implement larger structures and the assemblies tended to come apart, particularly while being transported. Additional research into modeling techniques led to industrial models constructed of FischerTechnik components. These overcame the problems but did not allow construction of small gear based assemblies. A method was developed to integrate both types of components. The result was a modeling capability that could be used to construct a functioning representation of the Wrapper Line.

A Wrapper Line model and a separate, distinctly different training model were constructed. Together, these models allow researchers to experiment with systems that are representative of typical Roll Forming industry complexity. The difficulty of obtaining access to real systems is addressed in this manner. When an actual system is available, there is a very short time during which experiments can be conducted. The experimental environment allows the duration of available time to be extended. In a real system that needs to be delivered to a customer one needs to conduct the experiment as

quickly as possible. Delays to the delivery result in delays of payment to the control supplier. The physical model resolves this issue. The experimental subject can take the model to their residence and conduct the experiment at the rate appropriate to his other responsibilities. The entire experimental environment consisting of the physical models, PLC, I/O, interfaces, PC, etc. was designed to be small and to use common household power sources. This allows transportation to and set-up at an experimental subject's residence. The result of this capability is a greater number of available experimental subjects. They no longer need to live within a close proximity to the researchers and they do not need to schedule a block of consecutive time within which to perform the experiment.

A Windows based program was developed to serve as the MMI and the interface between the PLC and Lego interface card. The use of OO methodology resulted in a program that can be easily and quickly configured to match the experiments requirements as they are developed. When the experimental subject creates a new subsystem, all of the required classes exist and the control and MMI objects needed, can be simply instantiated. On occasion the Windows operating system would cause noticeable delays. During these delays communication would be interrupted between the PLC and Lego interface. If the delays were long enough, over-travel would result. The Lego interface provides an emergency stop button that can be pressed in such situations to prevent collision between parts of the model. A potential solution to this problem is to raise the priority of the program. Should this prove to be insufficient, a dedicated microprocessor based interface such as used by MIT [Mar99] for interfacing with Lego components, or a

DSP evaluation card [Mot01] that provide PWM control of DC motors as well as A to D converters and pulse counters should be investigated.

8.9 Summary

This chapter presented the results and observations of this research. The abstract model and OO methodology with finite state machines for representing dynamic behaviour developed was capable of representing typical systems. A mapping from the model to PLC Ladder Logic was developed. An industrial control manufacturer adopted the methodology and has successfully been using it for more than two years. Through the actual use on other systems they found that the method had advantages over the current state of the art. The application generator developed based on the model and methodology was capable of creating control code for typical systems and thus providing synchronization of monitoring and control systems along with context enhanced data. A tradesman, who did not know how to write control code to create a control application, applied control engineering knowledge captured in the application generator to successfully create control code. An experimental environment was developed that is capable of replicating the experiment by other researchers, replacing the need for an industrial system. The next chapter provides a summary of the work presented in this research, draws conclusions about the methodology, application generator and experimental environment, and outlines some areas for further study.

Chapter 9

Conclusion

This research has provided solutions to challenges that arise when implementing Agile Manufacturing Information Systems. The synchronization of manufacturing discrete event supervisory control systems with plant monitoring systems was the primary accomplishment. It also addressed a fundamental shortcoming of current monitoring systems by automatically providing the data collected with context from the control system. The issue of experimentation in software engineering was investigated. Scientific methodology for developing code was transferred from the software community's theoretical domain to practice in industry. The possibility of capturing the knowledge of a control engineer within an application generator was demonstrated. A method of constructing control plans from reusable objects was derived.

9.1 Summary

Chapter 1 introduced the Agile Manufacturing paradigm, which addresses many aspects of the frequent and unpredictable change that characterizes today's global economy. To achieve agility requires an increased dependency on human judgment. This results in the need to provide workers with accurate information in a timely manner resulting in the need for an Agile Manufacturing Information System (AMIS). The synchronization of the manufacturing plant floor monitoring and control software code

problem of such an AMIS is presented. The lack of systems context provided by current monitoring systems; development of the code lacking application of scientific methodology; and insufficient experimentation in the software community to transition technology from the theoretical domain to practice in industry are identified as additional problems faced by an AMIS that this research would explore.

Chapter 2 presented a literature search of the current state of the art pertaining to the problems identified in the previous chapter. The life cycles of monitoring and control systems are presented to explain the cause of their divergence and their ad hoc development practice as the cause of poor quality data without system context awareness. The need to transfer principles and methods of software system development that address these issues from the academic to industrial community is identified with the lack of experimental support for the process attributed for the difficulty. The capture of a control engineer's knowledge in a reusable form is identified as a technology transfer enabler. Current research on discrete event supervisory control is presented and a proposal for a gradual transition to industry is recommended.

Chapter 3 was included to address the difficulty in presenting a multidisciplinary research subject in a concise manner that can be readily understood by readers who do not have expert knowledge in all of the disciplines involved. This chapter briefly presented an overview of the motivation consisting empowering employees to deal with rapid change by providing accurate information to support decisions from an Agile Manufacturing Information System. The issues of non-disruptive technology insertion

were discussed. The problems associated with keeping monitoring and control systems synchronized and the lack of system context in collected data were described. The novel contributions to the fields involved were stated. The methodology developed to address issues arising from development of Agile Manufacturing Information Systems (AMIS) was outlined.

Chapter 4 described the tools available to address the transfer of technology needed to address the issues faced by an AMIS. The methodology is based on an object-oriented composition model augmented with the formal method of finite state machines. For a gradual technology insertion, PLC Ladder Logic code was used to provide the same information in a familiar form. The development of an abstract model and the capture of the knowledge of how to apply it within an application generator were selected as the strategy to: overcome the problem of composing reusable components into a control plan; automatically provide context aware data; and synchronize monitoring and control systems.

Chapter 5 developed an abstract model for the control code. Technology insertion achieved thorough experimentation with evaluation and feedback from an industry expert control engineer was described. The process of generalizing from a specific case illustrated the derivation of the model represented with the Object Oriented Unified Modelling Language. Based on this model the methodology for generating control code for typical machines was presented.

Chapter 6 outlined the investigation of different approaches for the design of the application generator. The artificial intelligence paradigm of knowledge-based systems and the Object Oriented concepts of hierarchy, compositional modelling, abstraction and information hiding were applied to capture and represent the knowledge of a control engineer in a form useable by a trades person. Aspects of human-computer interaction and human-computer interface along with the iterative rapid prototyping software development approach were employed to develop, solicit feedback from an experimental subject and improve the usability.

Chapter 7 described the design and implementation of the experimental model and support environment that addressed common obstacles to software experimentation. Alternative solutions for implementing a machine simulator were evaluated and the final approach based on using a small-scale, functional physical model was presented.

Chapter 8 concluded with the results of the research and an evaluation of its performance. Industrial case studies evaluated the capability and ease-of-use of the abstract model and code generating methodology contrasted to the current state of the art. A tradesman experimentally demonstrated the feasibility of using an application generator to capture the knowledge of a control engineer in a reusable form. Feedback from the experiment was used to iteratively refine and then redesign the computer-human interface. A measure of general applicability of the methodology was obtained by using it in the different domain of CNC machine control.

9.2 Results Assessment

This research set out to resolve the following issues faced when developing an AMIS: synchronization of monitoring and control systems; lack of systems context in the collected data; development of the code lacking application of scientific methodology; and insufficient experimentation in the software community to transition technology needed to solve these problems from the theoretical domain to practice in industry. These goals were accomplished through the novel application of software and electrical engineering principles to develop an original solution. This solution was successfully transferred from the academic environment to industry through: the use of experimentation on representative manufacturing systems; evaluation and feedback from an industrial controls expert; and a gradual transition path providing PLC Ladder Logic as the familiar method along with the new model based method. The process of generalizing from a specific case was used to develop an abstract model that partitioned control code into a standard framework where specific functions were always assigned to standard components. Hierarchy was shown to be an effective mechanism for dealing with complexity. Information hiding was used to reduce the amount of information a user needed to assimilate before he could start applying the solution. The development of a standard transformation of the abstract model to Ladder Logic code patterns enabled an industrial control engineer to manually apply the methodology without needing to wait for an application generator. The utility of experiments with industrial oversight for the purpose of refining and adapting academic theory so that it can be successfully

transferred to industry was shown. After experimentally developing and refining the methodology during two progressively more complex experimental system developments, the industrial partner could ascertain its merit. Subsequently, the industrial control engineer demonstrated the suitability of the methodology for generating control code by using it for more than two years. During that time he observed a reduction in time for initial control code development along with an improvement in debug and later modification. A case study of a technician showed the value of using finite state machines for both debugging and designing sequential control code. The technology insertion principle of providing both the old method he was familiar with in the form of PLC Ladder logic and the new method of state machine models was successfully applied. His efficiency was noticeably increased. The utility of the method was attested to by his choice to modify control code at the higher level of abstraction of the state machine diagram and then applying the transformation rather than working directly at the code level as he had previously. A further indicator of the superiority of the methodology was the increase in the technician's ability over the period of a year of use to the point that he could design complete control systems resulting in his promotion to the level of control designer. The ability to recognize the abstract model in the control code through the use of the standard code topologies removed the need for separate monitoring code. The use of a single coil to represent a state or a condition, and the documentation of the meaning of these in the model reduced the monitoring system to simply reading the control system data and directly using it. The explicit availability of

the state of all of the objects of the model as single bits in the control code provided the context that is typically missing from current monitoring system collected data.

This research sought to solve the difficulty of configuring a library of reusable components into a control plan. To resolve the multiplicity of different partitioning and assignment of function to component, a domain as well as organization specific model was developed. As reported, the industrial control engineer successfully applied this model manually, through a program editor's cut, paste and modify capability, to reuse the components developed. To reduce the effort for others to reuse these components and to prevent possible proliferation of implementation variation, an application generator based on the model was implemented capturing the knowledge of a control engineer in the form of an expert system. The user interacts with the generator at a functional level describing the machine control guided by the expert system, which builds the model for the specific application. Selection and configuration of the reusable components that were previously developed by experimentation, evaluation and incorporation of feedback from a controls expert implements the model. The application generator then creates Ladder Logic code based on the topologies or patterns that were developed. An experiment using a tradesman to generate control code demonstrated the feasibility of constructing such an application generator. The information captured by the application generator can be used by a monitoring system to collect context enhanced data and remain synchronized with the control system provided that control code modification are made through the application generator.

Developing an environment that allowed other researchers to readily replicate as well as extend the experiments presented was a goal of this research. The common obstacles of inaccessibility of representative real systems, cost of experimental systems, distance to test subjects and their ability to schedule a sufficient block of time were addressed through the development of a functional model. This model is self-contained, sufficiently compact and robust enough to be transported to and used at the subject's residence. It is constructed from readily available commercial components at a modest cost within the reach of most researchers.

9.3 Contributions

This research constitutes a body of original work in that it has successfully integrated aspects of real-time systems control, computer science, electrical engineering, knowledge-based systems and technology insertion to address the following issues faced when developing Agile Manufacturing Information systems: synchronization of monitoring and control systems; lack of context in the collected data; development of the code lacking application of scientific methodology; and insufficient experimentation in the software community to transition technology needed to solve these problems from the theoretical domain to practice in industry. A contribution of this work is the extension of the current UM [Bir97] research domain model based on the object-oriented paradigm with an extended finite state model to address the AMIS divergence and context issues. It provides a method for integrating components into a control plan by capturing and representing a control engineer's knowledge in an application generator based on a

generalized abstract model applicable to the domain of roll-forming systems. The general nature of the methodology was then tested in the different domain of CNC controls. There it was extended with standard configurable design patterns for common subsystems to provide an optimized solution and contain proliferation of different implementations. A novel mapping from the model to PLC Ladder Logic code that the target user is familiar with was developed to assist with the technology insertion. The possibility of automatic PLC Ladder Logic code generation from the object model was demonstrated. Through an experimental effort with industry, the methodology was successfully transferred from the theoretical academic domain to practice in the industrial community. The improvement of the new methodology over the state-of-the-art industry practice was experimentally demonstrated. The possibility of constructing an application generator based on an abstract model, knowledge capture of an expert and the removal of the need for a specialized control engineer was experimentally shown. The improvement in graphical user interface efficiency based on a concrete approach offered by the Zooming Interface Paradigm was shown for applications targeted at trades' people. An experimental environment was developed to allow other researchers to replicate the experiments and to extend the research. Through the replicable experimental process, further transferring of software technologies from the theoretical domain into industrial applications can occur.

9.4 Conclusions

The results attained through this research leads to several conclusions. Foremost, new theoretical methodologies can be successfully transferred from the academic domain to industrial practice through applications of technology transfer supported by experimentation. Synchronization of monitoring and control systems and context-enhanced data can be obtained through the use of model-based methods. Discrete event control code for industrial applications can be developed at a higher level of abstraction than the direct manual writing of code. The new methodology results in less time required for the initial design, debug and later modification. It requires less skill to apply than the traditional method. Application generators can be built to capture a control engineer's knowledge to be reused by domain experts such as machine designers who are not familiar with the design of controls to generate control code. Modestly priced, portable experimental environments that simulate the complexity of real machines can be constructed and used for research in industrial applications.

9.5 Further Study

The work presented in this research suggests various topics for further study particularly in the application of model-based methods, artificial intelligence and performance support systems. An abstract model and code generating methodology was developed for the domain of discrete event supervisory control of roll forming machines. This domain could be addressed with a class model of modest complexity. To what other

domains can it be applied? Of particular interest is embedded control of automotive Powertrain applications such as engine and transmission control where the expected number of classes is much larger. Can continuous control be integrated into the nested state based hierarchical scheme? The state based approach provides a natural means for organizing information about a system. How effective would an artificial intelligence system based on this organization be for capturing initial design knowledge and subsequent debug information and using it to assist with rapid repair of system breakdowns? Could capture and organization of knowledge about a system be automated based on the use of audio-visual technology and the structure provided by the model? Can more complicated control engineering knowledge be implemented with an application generator? This research is the first step in transitioning of industrial control development from the *ad hoc* unstructured stage towards working at a higher level of abstraction. It introduced the OO oriented state based supplemented approach with PLC Ladder Logic as the targeted code. What should the next step be and how abstract can the final stage become? Can enough utilities be built into a model to allow control directly from it instead of using a control language for implementation? The application generator provides a machine-readable model of the controller. Can this model be used to automatically generate a discrete event simulation model, which could be coupled with the data collection of the monitoring system to provide the needed statistical characteristics of cycle time, mean failure time, repair time etc? Can the information in the application generator be applied to automatically create monitoring and diagnostic applications? Can the control system MMI be automatically generated from this

information? The application generator demonstrated the possibility of replacing the need for a control engineer's specialized knowledge. A comprehensive training manual was required to introduce the tradesman to its use. How can this material be integrated into the application generator and how can the graphical user interface be improved? The application in the CNC machine domain showed the need for capturing an organization's design solutions to common problems as configurable patterns. These were then hard coded into the application generator. Can a generalized method of specifying the patterns be found and coded into the application generator so that the user can create new patterns without the need for modification of the generator? Using the replicable experimental environment, quantitative studies of various control methods such as controlled automata, Petri Nets, nested finite state machines, when applied in industry could be conducted. This research indicated the possibility of integrating readily available components into functional models representative of complexity found in real machines. A Windows based PC and Lego interface was used. Are other forms of interface better? Can the MMI and PLC-to-model interface code be automatically generated? The design of the models required much time and effort. Can a collection of designs for assemblies that can be integrated into common functions as well as common construction techniques be provided to allow researchers to easily construct other models? What would be an effective manner of organizing the techniques?

All of these questions are worthy of further research and would serve to transfer technology from the theoretical academic domain into industry practice.

This research explored integrating aspects of real-time systems control, computer science, electrical engineering, artificial intelligence and technology insertion to address the specific issues of implementing Agile Manufacturing Information Systems. It has sought to explore and identify the many facets associated with transferring the benefits of these disciplines to industry. In doing so, it is anticipated that similar technology insertion efforts will meet with success in the future.

References

[Abd99] S. Abdelwahed and W.M.Wonham, "Interacting Discrete Event Systems", *Proc. Thirty-Seventh Annual Allerton Conference on Communication, Control and Computing*, Allerton, IL, September 1999, pp.85-92.

[Abd00] S. Abdelwahed, Interacting Discrete-Event Systems: Modeling, Verification, and Supervisory Control, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Toronto (Toronto, Ontario, March, 2002) 180p.

[ABP92] Allen Bradley Publication, PLC-5 Programming Software, 6200-6.4.7, Milwaukee,WI: Allen Bradley Publication, April, 1992.

[ACM96] Theme Issue: Computer Science in Manufacturing *Comm. ACM*, **39#2** February, 1996.

[ACM96b] Theme issue: "Learner-centered Design", *Comm. ACM*, **39#3** (March,1996)

[AFR02] Air Force Research Laboratory, Model-Based Integration of Embedded Software (MoBIES) Web Page (Available January, 2003 at http://spock.deepthought.rl.af.mil/tech/programs/MoBIES/)

[Alb91] J. S. Albus, "Outline for a Theory of Intelligence", *IEEE Transactions on Systems, Man and Cybernetics*, **21#3** (May/June 1991) pp.473-509.

[ARC02] ARC Advisory Board, General Motors Leverages Common Architecture to Strategic Advantage, ARC Strategies, October 2002. (Available to Arc Advisory Service members, February 2003 at www.ARCweb.com)

[Bad94] A. Baddely, "The Magical Number Seven: Still Magic After All These Years?", *Psychological Review*, **101#2** (1994) pp.353-356.

[Bau00] D. Baum, Definitive Guide to Lego MINDSTORMS, Emeryville, CA: Apress, 2000, 385p.

[Ben91] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems ", *Proceedings of the IEEE,* **79#9** (1991) pp.1270-1282.

[Ber95] G. Berio, A. Di Leva, P. Giolitto, and F. Vernadat, "The M*-OBJECT Methodology for Information System Design in CIM Environments," *IEEE Transactions on Systems, Man and Cybernetics*, **25#1** (January 1995) pp.68-85.

[Ber95b] J. Bergendahl, D. Esterman, M. Sullivan and C. G. Cassandras Computer-Controlled LEGO Factory, Technical Report, Amherst, MA: University of Massachusetts, June 1995, 63p.

[Ber91] G. Berry, "A Hardware Implementation of Pure Esterel", Research Report 15, Paris: Digital Equipment Corp., Paris Research Laboratory, 1991.

[Ber86] V. Berzins, M. Gray and D. Naumann, "Abstraction-Based Software Development", *Communications of the ACM* **29#5** (May 1986), p.403.

[Ber00] H. Berger, Automating with Step7 in STL and SCL, Erlangen, Germany: Publicis MCD Verlag, 2000, 436p.

[Bir97] S. K. Birla, Software Modeling for Reconfigurable Machine Tool Controllers, Ph.D. Dissertation, The University of Michigan, (Ann Arbor, Michigan, May 1997) 171p.

[Bir98] S. K. Birla and K. Shin, "Reconfiguration Requirements for Software to Control Automotive Manufacturing Machine Tools," *Proceedings of the 1998 Japan-U.S.A. Symposium on Flexible Automation*, Detroit, MI, July 1998.

[Bir03] Private Communication: S. Birla, Principal Research Engineer, Research & Development and Planning General Motors, Warren MI, February 2003.

[Boa84] B. Boar, Application Prototying, Reading, MA: Addison-Wesley, 1984.

[Boe85] B. Boehm, "A Spiral Model of Software Development and Enhancment", *Proceedings of an International Workshop on the Software Process and Software Environmnets,* Coto de Caza, Trabuco Cancyon CA, 1985.

[Bol87] T. Bolognesi and E. Brinnksma, "Introduction to the ISO Specification Language LOTOS", *Computer Networks and ISDN Systems,* **14#1** (1991) pp.25-59.

[Bon99] Private Communication: G. Bone, McMaster University, Hamilton Ontario, April, 1999.

[Boo94] G. Booch, Object-Oriented Analysis and Design With Applications, Second Edition, Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc., 1994, 203p.

[Bou91] F. Boussinot and R. de Simone, "The Esterel Language, Another Look at Real Time Programming", *Proceedings of IEEE,* **79#9** (1991) pp.1293-1304.

[Bow98] H. Bowman, "LOTOS Based Tutorial on Formal Methods for Object-Oriented Distributed Systems", *New Generation Computing,* **16#4** (1998) pp.343-372.

[Bra91] B.A. Brandin, W.M. Wonham and B. Benhabib. "Discrete-event systems supervisory control applied to the management of manufacturing workcells," *Proc. 7th International Conference on Computer Aided Manufacturing Engineering*, Cookeville, Elsevier , pp. 527-536, 1991.

[Bra92] B. A. Brandin and W. M. Wonham, "The supervisory control of timed discrete-event systems", *Proc. 1992 Control and Decision Conf.*, Tucson, AZ, December, 1992, pp.3357-3362.

[Bra93] B. A. Brandin Real-Time Supervisory Control of Automated Manufacturing Systems, Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Toronto (Toronto, Ontario, March, 1993).

[Bra94] B. A. Brandin and W. M. Wonham, "Supervisory Control of Timed Discrete-Event Systems," *IEEE Transactions on Automatic Control*, **39#2** (February 1994) pp.329-342.

[Bra96] B. Brandin, "The Real-time Supervisory Control of an Experimental Manufacturing Cell," *IEEE Transactions on Robotics and Automation*, **12#1** (February 1996) pp.1-14.

[Bra93b Y. Brave and M. Heymann, "Control of Discrete Event Systems Modeled as Hierarchical State Machines," *IEEE Transactions on Automatic Control*, **38#12** (December, 1993) pp.1803-1819.

[Bri81] Britton, K. and Parnas, D. *A-7E Software Module Guide.* Washington, D.C.: Naval Research Laboratory, Report 4702, December 8, 1981, p.2.

[Buz95] J. Buzacott, "A Perspective on New Paradigms in Manufacturing", *Journal of Manufacturing Systems*, **14#2** (1995) pp.118-125.

[Cas97] P. Caspi et al., "Lustre: a Declarative Language for Programming Synchronous Systems", *Proceedings of 14th Symposium on Principles of Programming Languages,* (1997) pp.178-188.

[Cas00] Private Communication: C. G. Cassandras, Boston University, Boston, MA, August, 2000.

[Cas02] I. Castillo and J. S. Smith, "Formal Modeling Methodologies for Control of Manufacturing Cells: Survey and Comparison", *Journal of Manufacturing Systems,* **21#1** (2002) pp.40-57.

[Chr58] G. K. Christiansen, Toy Building Brick, United States Patent 3,005,282, October, 1961, (Available January 2003 at: http://w3.one.net/~hughesj/technica/history/patents/3005282.php).

[Cie88] R. Cieslak, C. Desclaux, A .S. Fawaz and P. Varaiya, "Supervisory Control of Discrete Event Processes with Partial Observation," *IEEE Transactions on Automatic Control*, **33#3** (March, 1988) pp.249-260.

[CIM03] University of Toronto Computer Integrated Manufacturing Laboratory, (Available January, 2003 at: http://www.mie.utoronto.ca/labs/ciml/).

[Cor02] M. A. E. Coronado, M. Sarhadi and C. Millar, " Defining a framework for information systems requirements for agile manufacturing", *International Journal of Production Economics*, **75#1-2** (October 1, 2002) pp.57-68.

[Coy90] R.D. Coyne, et. al., Knowledge-Based Design Systems, Reading, MA :Addison-Wesley Publishing Co., 1990.

[Dah72] O. Dahl, E. Dijkstra and C. A. R. Hoare, Structured Programming, London, England: Academic Press, 1972, p.83.

[Dav94] R. David and H. Alla, "Petri Nets for Modeling of Dynamic Systems – A Survey," *Automatica,* **30#2** (1994) pp.175-202.

[Dav95] R. David, "Grafcet: A Powerful Tool for Specification of Logic Controllers", *IEEE Transactions Control Systems Technology,* **3#3** (September, 1995) pp.253-268.

[DeG90] P. DeGrace and L. H. Stahl. Wicked Problems, Righteous Soutions: A Catalogue of Modern Software Engineering Paradifms, Englewood Cliffs, NJ: Yourdon Press/Prentice Hall, 1990.

[Dic93] F. Dicesare, G. Harhalakis, J.M. Proth, M.Silva and F.B. Vernadat, Practice of Petri nets in Manufacturing First Edition, New York, NY: Chapman & Hall, 1993.

[Dou00] B. P. Douglass, Real-Time UML Second Edition Developing Efficient Objects for Embedded Systems, Reading, MA: Addison Wesley, 2000, 328p.

[Dub02] P.F. Dubois, "Designing Scientific Components", *Computing in Science and Engineering,* (September/October, 2002) pp.84-90.

[Dun94] A. Dunkin, "Automated Guided Vehicles: An Introduction", *Ind.Engg.*, **26#8** (1994) p.47-53.

[End00] E. W. Endsley, M. R. Lucas, D. M. Tilbury, D.M., "Software Tools for Verification of Modular FSM Based Logic Control for use in Reconfigurable Machining Systems," *Proceedings of the 2000 Japan-USA Symposium on Flexible Automation*, Ann Arbor, MI 2000

[End01] E. W. Endsley, M. R. Lucas, and D. M. Tilbury, "A Logic Control Programming Framework Using Modular Finite State Machines," *Proceedings of the American Control Conference*, 2002, submitted September 2001, (Available January 2003 at: http://erc.engin.edu/Publications/PubFiles/TA2/ProjM3/Acc02Paper.pdf).

[ERC98] Engineering Research Center for Reconfigurable Machining Systems, OSACA Workshop and Open Archetcture Control Round Table, Ann Arbour, MI: University of Michigan, April, 1998.

[ERC03] Engineering Research Center for Reconfigurable Machining Systems at the University of Michigan,. (Available January 2003 at: http://erc.engin.umich.edu/).

[Eri98] H. E. Erikson and M. Penker, UML Toolkit, New York, NY: Wiley, 1998, 397p.

[Erw98] B. Erwin, *et al.* Middle School Engineering with LEGO and LabVIEW, presented at the MIT LabVIEW in Education days and National Instruments Week, Austin, Texas, August 1998 (available January, 2003 at http://ldaps.arc.nasa.gov/Publications/Presentations/mit.ppt,)

[Fay01] B. Fay, The Gantry Mill, (Available January 2003 at: http://www.ozbricks.com/bobfay/anatomy.htm).

[Fen27] C. Fencott, Formal Methods for Concurrency, International Thompson Computer Press, 1996.

[Fer92] L. Ferrarini, "An Incremental Approach to Logic Controller Design with Petri Nets", *IEEE Transactions on Systems, Man, and Cybernetics,* **22#3** (May, 1992) pp.461-473

[Fis94] Fischerwerke, The fischertecknik system General Catalog, Waldachtal Germany: Arthur Fischer GmbH, 1994, p.2.

[Fow93] P. Fowler, and L Levine, A Conceptual Framework for Software Technology Transition, CMU/SEI-93-TR-31 ESC-TR-93-317, Pit PA: Software Engineering Institute, December 1993.

[Fre91] P. Freedman, "Time, Petri Nets, and Robotics", *IEEE Transactions on Robotics and Automation*, 7#4 (August, 1991) pp.417-433.

[Gan87] J. Gannon, R. Hamlet and H. Mills, "Theory of Modules", *IEEE Transactions on Soltware Engineering,* 13#7 (July, 1987) p.820.

[Gia89] J. Giarratano and G. Riley, Expert Systems Principles and Programming, Boston, MA: PWS-Kent Publishing Co., 1989.

[Gol87] C. H. Golaszewski and P. J. Ramadge, "Control of discrete event processes with forced events", *Proceedings of the 26th Conference of Decision and Control*, December, 1987, pp. 24-251.

[Gol93] S. Golson, "One-hot state machine design for FPGAs", 3rd PLD Design Conference, Santa Clara CA, March 30, 1993

[Gra94] M. Graube and O. Storoshchuk, "Fiber-Optic Cable System Implementaion", chapter 19 of Mini-Map '93, Rochester, NY: Open I.T. Corporation, 1994, pp.19-1 – 19-20.

[Gue00] Private Communication: Sam Gue University of Michigan, Ann Arbour Michigan, September 2000.

[Gue91] P. Le Guernic et al., "Programming Real-Time Applications with Signal: Another Look at Real-Time Programming", *Proceedings of the IEEE,* 79#9 (1991) pp.1321-1336.

[Hal92] N. Halbwachs, F. Lagnire and C. Ratel, "Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language Lustre", *IEEE Transactions on Software Engineering,* 18#9 (1992) pp.785-794.

[Hal93] N. Halbwachs, Synchronous Programming of Reactive Systems, Boston, MA: Kluwer Academic Publishers, 1993.

[Har87] D. Harel *et al.*, On the Formal semantics of state-charts, in *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, New York: IEEE Press, 1987, pp 54-64.

[Har87b] D. Harel, "Statecharts: a Visual Formalism for Complex Systems", *Science of Computer Programming,* 8#3 (1987) pp.231-274.

[Har88] D. Harel, "On Visual Formalisms", *Communications of the ACM,* **31#5** (1988) pp.514-530.

[Har90] D. Harel et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering*, **16#4** (April, 1990) pp.403-414.

[Has98] Private Communication: B. R. Hastings, P. Eng., Engineering Manager, RLM Manufacturing Bowmanville Ontario, Canada, October 1998.

[Has99] Private Communication: B. R. Hastings, P. Eng., Engineering Manager, RLM Manufacturing Bowmanville Ontario, Canada, September 1999.

[Has02] Private Communication: B. R. Hastings, P. Eng., Engineering Manager, RLM Manufacturing Bowmanville Ontario, Canada, October 2002.

[Hoa85] C. A. R. Hoare, Communicating Sequential Processes, Englewood Cliffs, NJ: Prentice Hall, 1985.

[Hod91] P. Hodaie, B. Szabados, B. Chan, O. Storoshchuk "Robot and PLC Support System Based on Fiber Optic Map Network", chapter 14 of "Microprocessors in Robotics and Manufacturing Systems", Kluwer Academic Publishers, 1991.

[Hod92] P. Hodaie, O. Storoshchuk, B. Szabados, "Manufacturing Automation Protocol From Application Developer's Point of View" Proceedings of ENE'92 Conference, March 16-19, 1992 Washington, D.C., Application Development.

[Hod93] P. Hodaie, O. Storoshchuk, B. Szabados, M.A. Maslied, L. Van Der Jagt, G. Martin, "Propagation Measurements at a G.M. Manufacturing Plant for Wireless LAN Communication", Proceedings of Wireless 93', Calgary, July 12-14, 1993, Conference Proceedings pp.10.6.1-4.

[Hod96] P. Hodaie and B Szabados. "Intelligent Monitoring System for Asynchronous Production Systems", *CSME* Hamilton, Ontario, Canada, May 7-9, 1996. (Available January, 2003 at: http://power.eng.mcmaster.ca/szabados/papers/paper.html)

[Hol90] L. E. Holloway and B. H. Krogh, "Synthesis of Feedback Control Logic for a Class of Controlled Petri Nets", *IEEE Transactions on Automatic Control*, **35#5** (May, 1990) pp.514-523.

[Hol97] L. E. Holloway and A. Hall, "Principles of lean manufacturing," *Industry and Higher Education*, (August, 1997).

[Hol91] E. Hollnagel, The Phenotype of Erroneous Actions: Implications for HCI Design, *Human-Computer Interaction and Complex Systems,* G. R. S. Wier and J. L. Atly, eds., San Diego, California: Academic Press Inc., 1991, pp.73-121.

[Hor84] E.Horowitz, and J. B. Munson, "An Expansive View of Reusable Software", *IEEE Transactions on Software Engineering*, **10#5** (September, 1984) pp.477-487.

[Hud98] C. H. Huddleston, D. A. Troy, ActiveX Control for LEGO DACTA Control Lab, (Available January, 2003 at: http://troyda.eas.muohio.edu/paper2.html, corrections at: http://troyda.eas.muohio.edu/LegoCorrections.html.

[Huf54a] D. A. Huffman, "The Synthesis of Sequential Switching Circuits," *J. Franklin Institute*, **257#3** (March, 1954) pp.161-190.

[Huf54b] D. A. Huffman, "The Synthesis of Sequential Switching Circuits," *J. Franklin Institute*, **257#4** (April, 1954) pp.275-303.

[IEC98] INTERNATIONAL ELECTROTECHNICAL COMMISSION TECHNICAL COMMITTEE No. 65: INDUSTRIAL-PROCESS MEASUREMENT AND CONTROL SUB-COMMITTEE 65B: DEVICES WORKING GROUP 7: PROGRAMMABLE CONTROLLERS COMMITTEE DRAFT - IEC 61131-3, 2nd Ed. PROGRAMMABLE CONTROLLERS - PROGRAMMING LANGUAGES

[Ing78] D. Ingalls, The Smalltalk-76 Programming System Design and Implementation. *Proceedings of the Fifth Annual ACM Syposium on Principles of Programming Languages.* NY NY:ACM, 1978, p.9.

[Ing81] D. Igalls, "Design Principles Behind Smalltalk", *Byte*, **6#8** (August, 1981) pp.286-298.

[Ise95] H. Isenberg, "The Impact of the Information Explosion", *Industrial Engineering*, (March, 1995) p.15.

[Joh94] G. W. Johnson LabVIEW Graphical Programming – Practical Applications in Instrumentation and Control, New York, NY: McGraw-Hill, 1994, 522p.

[Kin00] Private communications: T. King, Tim King Electronics Allen Park, MI, June 2000.

[Kit97] B. Kitchenham, P. Brereton, D. Budgen, S. Linkman, V.L. Almstrum, and S. L. Pfleeger, "Evaluation and Assessment in Software Engineering," *Information and Software Technology*, **39#11** (November, 1997) pp.731-734.

[Knu99] J. B. Knudsen, The Unofficial Guide to Lego MINDSTORMS Robots, Sebastopol, CA: O'Reilly & Associates, 1999, 247p.

[Kru98] D. J. Kruglinski, G. Shepard and S. Wingo, Programming Microsoft Visual C++ Fifth Edition, Redmont, WA: Microsoft Press, 1998, p.22.

[Kuu91] I. Kuuluvainen, M. Vanttinen, and P. Koskinen, "The Action-State Diagram: a Compact Finite State Machine Representation for User Interfaces and Small Embedded Reactive Systems", *IEEE Transactions on Consumer Electronics*, **37#3** (August, 1991) pp.651-657.

[Lau97] S. C. Lauzon, "An implementation methodology for the supervisory control of flexible manufacturing workcells", *Journal-of-Manufacturing-Systems,* **16#2** (1997) pp.91-101.

[Lay94] J. Layden, "Real-Time Factory Floor Scheduling Enhances Responsiveness", *Industrial Engineering*, (November, 1994) pp.20-27

[Led95a] R.J. Leduc and W.M. Wonham, Discrete event systems modeling and control of a manufacturing testbed, *Proc. of Canadian Conference on Electrical and Computer Engineering*, Volume 2, September 5-8, 1995, pp.793-796.

[Led95b] R.J. Leduc and W.M. Wonham, PLC implementation of a des supervisor for a manufacturing testbed, *Proc. of Thirty-third Annual Allerton Conference on Communication, Control, and Computing*, October 4-6, 1995, pp.519-528.

[Led00] R. J. Leduc, B. A. Brandin, and W. M. Wonham, Hierarchical Interface-Based Non-Blocking Verification, *Canadian Conference on Electrical and Computer Engineering (CCECE 2000)*, Halifax, , May 7-10, 2000, pp. 1-6.

[Led02] J. R. Leduc, Hierarchical Interface-based Supervisory Control, Ph.D. Dissertation, Department of Electrical and Computer. Engineering, (University of Toronto, April, 2002).

[LEG00] LEGO, *Motor Mount,* (Available July 2000 at http://www.lego.com/dacta/robolab/tipsandtricks.htm).

[Lew95] R. W. Lewis, Programming industrial control systems using IEC 1131-*3*, London, United Kingdom: The Institute of Electrical Engneers, 1995.

[Lho97] R. Lholtka, Professional Visual Basic 5.0 Business Objects, Birmingham, UK:Wrox Press, 1997, 636p.

[Lin97b] P.C. Lind, Application of Technology Insertion to Particle Accelerator Modernization and Operations Support, Ph.D. Dissertation, McMaster University, (Hamilton, Ontario, October, 1997).

[Lin94] R. Lingus, "Enterprise Agility: Jazz in the Factory", *Industrial Engineering*, (November, 1994) pp.18-19.

[Lin97] S. Linkman, and H. D. Rombach, "Experimentation as a vehicle for software technology transfer-A family of software reading techniques," *Information and Software Technology*, **39#11** (November, 1997) pp. 777-780.

[Lis80] B. Liskov, A Design Methodology for Reliable Software Systems, *Tutorial on Solfware Design Techniques.* Third Edition. New York, NY: IEEE Computer Society, 1980. p. 66.

[Lip95] R. Lipa, Technical Assessment: The Impact of Agents in Manufacturing, Detroit MI: BRL & Co., May, 1995 prepared for National Center for Manufacturing Sciences, Inc., pp.51-57.

[Lip03] Private Communication: R. Lipa, Ann Arbor MI, February 2003.

[Mar01] Private communications: Aldo Marcuzi, Siemens Industrial Controls, Detroit Michigan, May, 2001.

[Mar99] F. Martin, The Handy Board Technical Reference, February, 1999 (Available January, 2003 at: http://lcs.www.media.mit.edu/groups/el/Projects/handy-board/techdocs/hbmanual.pdf)

[Mar92] J. Martin and J. Odell, Object-Oreinted analysis and Desig, Englewood Cliffs, NJ: Prentice Hall, 1992.

[Mat00] J. Mattos de Assumpção Jr., Pegasus 2000 - User Interface (Available January 2003 at: http://www.merlintec.com/pegasus2000/e_gui.html).

[Met94] P. Mett, D. Crowe and P. Strain-Clark, Specification and Design of Concurrent Systems, London: McGraw-Hill, 1994

[Mey88] B. Meyer, Object-oriented Software Construction, Cambridge, U.K.: Prentice Hall International (UK) Ltd.., 1988, p.41.

[Mil56] G. Miller, "The magic Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *The Psychological Review* **63#2** (1956) p.86.

[Mil89] R. Milner, Communication and Concurrency, Englewood Clifs, NJ: Prentice Hall, 1989.

[Mot72] J. Motil, Digital Systems Fundamentals, New York: McGraw-Hill Book Co., 1972, pp.84-91.

[Mot01] Motorola Semiconductor Products Sector, DSP56F803EVMUM DSP56F803 Evaluation Module Hardware User's Manual, February, 2001 (Available January 2003 at: http://e-www.motorola.com/brdata/PDFDB/docs/DSP56F803EVMUM.pdf).

[MRL97] New York University Media Research Lab, Zooming Pad Demo, (Available January 2003 at: http://www.mrl.nyu.edu/~perlin/zoom/TestButton.html).

[MRL98] New York University Media Research Lab, Pad Web Navigation Demo, (Available January 2003 at: http://www.mrl.nyu.edu/~perlin/zoom/SiteTour.html).

[Mur98] P. Muro-Medrano, J. Banares, and J. Villarroel, "Knowledge Representation-Oriented Nets for Discrete Event System Applications", *IEEE Transactions on Systems, Man and Cybernetics-Part A: Systems and Humans*, **28#2** (March, 1998) pp.183-198.

[Mye78] G. Myers,. Composite/Structured Design, New York, NY: Van Nostrand Reinhold, 1978, p.21.

[NCM96] National Center for Manufacturing Sciences, "NCMS Collaborative Manufacturing Agenda", NCMS Document 0040RE96, May 1996.

[NIC99] National Instruments Corporation, "Control Mixer Process.vi", *LabVIEW 5.1*, 1999, Search Examples|Demonstration|Control Process Mixer.

[Nic86] R.S. Nickerson, Using Computers: The Human Factors of Information Systems, Cambridge, Massachusetts: MIT Press, 1986.

[OCo94] L. O'Connor, "Agile Manufacturing in a Responsive Factory", *Mechanical Engineering*, **116#7** (July, 1994) pp.54-57.

[OMA99] OMAC Working Group, OMAC API Documentation v0.23, Framingham, MA: OMG, April, 1999.

[OMG95] Object Management Group, The Object Management Architecture Guide, Object Management Group, Framingham, MA, June 1995.

[OSC96] OSACA Consortium, Open System Architecture for Controls within Automation Systems,Work Package 4 Deliverable D 2422 Access to MMC, MC, LC, ESPRIT III Project 9155, February, 1996

[Par98] E. Park,D. M. Tilbury and P. P. Khargonekar, "A Formal Implementation of Logic Controllers for Machining Systems Using Petri Nets and Sequential Function Charts", *Proceedings Japan-USA Symposium on Flexible Automation*, Otsu, Japan,1998, pp.683-690.

[Par99a] E. Park, Modular Logic Controllera of Machinig Systems: A Modelling and Analysis Methodology, Ph.D. Dissertation, University of Michigan, (Ann Arbour, MI, 1999).

[Par99b] E. Park, D. M. Tilbury and P. P. Khargonekar, "Modular Logic Controller for Machining Systems: Formal Representation and Performance Analysis using Petri Nets", *IEEE Transactions on Robotics and Automation*, **15#6** (1999) pp.1046-1061.

[Par99c] E. Park, D. M. Tilbury and P. P. Khargonekar, "A Modeling and Analysis Methodology for Modular Logic Controllers of Machining Systems using Petri Net Formalism", submitted to *IEEE Transactions on Systems, Man, and Cybernetics,* (Available January 2003 at: http://erc.engin.edu/Publications/PubFiles/TA2/Proj22e1/PetriNet.pdf)

[Par99d] E. Park,D. M. Tilbury and P. P. Khargonekar, "Performance Analysis of Machining Systems with Modular Logic Controllers", *Proceedings of the IEEE International Conference on Robotics and Automation*, Detroit, 1999, pp.137-144.

[Par00] E. Park,D. M. Tilbury and P. P. Khargonekar, "A Modeling and Analysis Methodology for Modular Logic Controllers of Machining Systems with Auto, Hand, and Manual Modes", *Proceedings of the American Control Conference*, Chicago, 2000, pp. 3158-3164

[Par77] D. Parnas. *Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems.* Washington, D.C: Naval Research Laboratory, Report 8047, June 3, 1977, p. 4.

[Par83] D. Parnas, P. Clements and D. Weiss, "Enhancing Reusability with Information Hiding", *Proceedings of the Workshop on Reusability in Programming,* Stratford, CT: ITT Programming, 1983. p.241.

[Par93] Private Communications: D. Parnas, McMaster University, Hamilton Ontario, April, 1993.

[Par96] K. Parker, "Technologies in Symmetry", *Manufacturing Systems*, (1996)

[Pau93] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, Capability Maturity Mode for Software, Version 1.1, CMU/SEI-93-TR-024, ESC-TR-93-177, Pit PA: Software Engineering Institute, February 1993.

[Per91] K. Perlin and J. Schwartz, Fractal computer user centerface with zooming capability, United States Patent 5,341,466, August, 1994, (Available January 2003 at: http://164.195.100.11/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=/netahtml/srchnum.htm&r=1&f=G&l=50&s1='5341466'.WKU.&OS=PN/5341466&RS=PN/5341466).

[Pet99] Private Communication: G. Petit Mohawk College, Hamilton Ontario, April, 1999

[Pet92] C. Petzold, Programming Windows 3.1, 3rd Ed., Redmont Washingotn: Microsoft Press, May 1992, 1088p.

[Poi98] Private Communication: D. Poirer, Engineering Group Manager, General Motors Powertrain Process Support Milford Michigan, May, 1988.

[Poe93] W. F. S. Poehlman, W. J. Garland, A. A. Bokhari, R. J. Wilson, and C. W. Baetsen, "Performance Support Systems and Artificial Intelligent Considerations", Proceedings of International Nuclear Congress -- INC93, vol.3, on October 3-7, 1993, Toronto, Ontario, Canada, session C21, 8p.

[Poe94] W.F.S.Poehlman, "The OPUS Approach to Domain / Software / Hardware Mapping", Proceedings of Workshop on Performance Support Systems, June 15-17, 1994, McMaster University, Hamilton, Ontario, Canada.

[Poe95] W.F.S. Poehlman and G.R. Chapman, A Proposal for the Development of A System Control Prototype for Agile Manufacturing, Prepared for GM Corp., Detroit MI, January 1995.

[Poe02] Private Communications: W. F. S. Poehlman, McMaster University, Ontario, June, 2002.

[Qua98] T. Quatrani, Visual Modeling with Rational Rose and UML, Reading, MA: Addison Wesley, 1998, 222p.

[Ram82] P. J. Ramadge, "Supervision of discrete event processes," *Proceedings of the 21st IEEE Conference on Decision & Control,* IEEE, vol. 3, 1982, pp.1228-9

[Ram86] P. J. Ramadge, Modular supervisory control of discrete event systems, Analysis and Optimization of Systems, *Proceedings of the Seventh*

International Conference, Berlin, West Germany: Springer-Verlag, 1986, pp. 202-14.

[Ram87] P. J. Ramadge, and W. M. Wonham, "Supervisory Control of a Class of Discrete-Event Processes", *SIAM J. Control and Optimization*, **25#1** (January, 1987) pp.206-230.

[Ram89] P. J. Ramadge, and W. M. Wonham, "The Control of Discrete Event Systems", *Proceedings of the IEEE*, **77#1** (January, 1989) pp.81-98.

[Ras00] J. Raskin, The Humane Interface: New Directions for Designing Interactive Systems, Reading, MA: Addison Wesley, 2000.

[Ras86] J. Rasmussen, Information Processing and Human-Machine Interaction: An Approach to Cognitive Engineering, New York, NY: North-Holland Publishing Co., 1986.

[Rey02] M. Reynolds, et. al., .NET Enterprise Development in VB.NET: From Design to Deployment, Birmingham, UK: Wrox PressLtd., 2002

[Rol92] Y. Roll, R. Karni and Y. Arzi, "Measurement of Processing Flexibility in Flexible Manufacturing Cells", *Journal of Manufacturing Systems*, **11#4** (1992) p.258-268.

[Run93] X. Runkle (Vice-President Engineering Support, General Motors Corporation), as quoted by J. Sheridan in "Agile Manufactuing: Beyond Lean Production", *Industrial Week*, (April 19, 1993) pp.31-46.

[Sch94] T. Schmoyer (Executive Director of the Agile Manufacturing Enterprise Forum,) as quoted by L. O'Connor in "Agile Manufacturing in a Responsive Factory", *Mechanical Engineering*, **116#7** (July, 1994) pp.54-57.

[Sha84] M. Shaw, "Abstraction Techniques in Modern Programming Languages", *IEEE Software,* **1#4** (October, 1984) p.10.

[Sha02] S. Shah, E. W. Endsley, M. R. Lucas and D. M. Tilbury, "Reconfigurable Logic Control Using Modular FSMs: Design, Verification, Implementation and integrated error handling", *Proceedings of the American Control Confrerence,* **5**, Anchorage, AK, May 2002, pp.4153-4158.

[She96] W. Shen, P. Williams, *Computer-Controlled Lego Factory*, April 1996, (Available January 2001 at http://www.ecs.umass.edu/ece/ug/lego/home.html).

[Shi94] R. M. Shiffrin and R. M. Nosofsky, "Seven Plus or Minus Two: A Commentary On Capacity Limitations", *Psychological Review*, **101#2** (1994) pp.357-361.

[Sei86] E. Seidewitz and M. Stark, "Towards a General Object-Oriented Software Development Methodology", *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station.* TX: NASA Lyndon B. Johnson Space Center, 1986, p. D.4.6.4.

[She93] J. Sheridan "Agile Manufactuing: Beyond Lean Production", *Industrial Week*, (April 19, 1993) pp.31-46.

[Shi94] K. G. Shin, and P. Ramanthan, "Real-Time Computing: A New Discipline of Computer Science and Engineering", *Proceedings of the IEEE*, **82#1** (January, 1994) pp.6-23.

[Shi98] C. Shiu, et al., "Specifying reconfigurable control flow for open architecture controllers", *Proc. 1998 Japan-USA Symposium on Flexible Automation*, vol.2, Otsu, Japan, July 1998, pp.659-666.

[Shn91] B. Shneiderman, A Taxonomy and Rule Base for the Selection of Interaction Rules, *Human Factors for Informatics Usability*, B. Shackel, S. Richardson, eds., Cambridge, England: Cambridge University Press, 1991, pp.352-342

[Sim82] H. Simon, The Sciences of the *Artificial*, Cambridge, MA: The MIT Press, 1982, p.217.

[Sol97] C. Solomon, Microsoft Office 97 Developers Handbook, Redmont Washingotn: Microsoft Press, 1997, 578p.

[Spe97] W. Sperling and P. Lutz, "Designing Applications for an OSACA Control", *Proceedings of the International Mechanical Engineering Congress and Exposition*, Dalles, TX, November, 1997.

[Spi89] J. Spivey, The Z notation: A reference Manual, New York, NY: Prentice Hall, 1989, 158p.

[SST01] SSTech, PICS PRO Simulation Software, Waterloo Ontario, Canada: SSTech, September 2001, (Available Feburary 2003 at http://www.mysst.com/pub/drivers/picsdata/epicspro.pdf)

[Ste94] M. P. Stephens and H. Kraebber, "Meeting the challenge of lean production and synchronous manufacturing systems", *Journal of industrial technology*, **10#3** (1994) pp.15-16.

[Sto83] O. Storoshchuk and B. Szabados, "Industrial Local Area Network", 1983 International Electrical, Electronics Conference Proceedings, New York, NY: IEEE, 1983, pp.454-456 vol.2.

[Sto86] O. L. Storoshchuk, D. Marlin and P. West, *C.B.A.P. Facilities Monitoring and Control System Functional Specification.* Internal specification, Oshawa Ontario, Canada: General Motors, October, 1986.

[Sto90] O. Storoshchuk, B. Chan and D. Daly, "The MAP Solution in the General Motors Oshawa, Ontario Car Body Assembly Plant", *Proceedings of the International Society for Optical Engineering,* **1197** (1990) pp.595-601.

[Sto01] O. L. Storoshchuk, S. Wang and K.G. Shin, " Modeling Manufacturing Control Software", *Proceedings of 2001 IEEE International Conference on Robotics and Automation* Seol, Korea, May 2001.

[Sto02] O. L. Storoshchuk, W. F. S. Poehlman, G. R. Chapman, S. Wang and K. G. Shin, "Modeling Manufacturing Control Software", submitted to *IEEE Transactions on Robotics and Automation*, March, 2002.

[Sto03a] O. L. Storoshchuk, W.F.S. Poehlman and G.R. Chapman, "Accommodating Change In A Manufacturing Environment", in preparation for publication (January, 2003).

[Sto03b] O. L. Storoshchuk, W.F.S. Poehlman and G.R. Chapman, "Synchronizing Manufacturing Monitoring and Control Systems", in preparation for publication (July, 2003).

[Sur95] R. Suri, R. Veeramani and J. Church, "Quick Response Manufacturing", *I.E. Solutions*, (November, 1995) pp.27-30.

[Sza86] B. Szabados and O. Storoshchuk, "Microprocessor Adaptive Control of Oil Temperature for Robot Arm Valves: A Practical Implementation", *IEEE Transactions on Industrial Electronics*, **IE-33#1** (1986) pp.21-27.

[Tan81] A. S. Tanenbaum, Computer Networks, Englewood Cliffs, NJ: Prentice Hall, 1981, pp.177-183.

[Tic98] W. F. Tichy, "Should Computer Scientists Experiment More?" *Computer*, **31#5** (May, 1998) pp.32-40.

[Til99] D. M. Tilbury, S. Kota, " Integrated Machine and Control Design for Reconfigurable Machine Tools", *IEEE/ASME Conference on Advanced Intelligent Mechatronics*", 1999.

[Tra94b] M. Tracy, J. Murphy, R. Denner, B. Prince, R. Joseph, A. Pilz and M. Thompson, "Achieving Agile Manufacturing: Part II", *Automotive Engineering*, (December, 1994) pp.13-17.

[Tra94a] M. Tracy, J. Murphy, R. Denner, B. Prince, R. Joseph, A. Pilz and M. Thompson, "Achieving Agile Manufacturing: Part I", *Automotive Engineering*, (November, 1994) pp.19-24.

[Vaz96] N. Vaz, "Agile Manufacturing Information System" in GM NAO Technology Portfolio project 43111. Internal report 1996.

[Vil97] T. Villa, et al., Synthesis of Finite State Machines: Logic Optimization, Kluwer Academic Publishers, 1997.

[Wan99] S. Wang, C.V. Ravishanka and K.G. Shin, "Open architecture controller software for integration of machine tool monitoring", *Proc. of 1999 IEEE International Conference on Robotics and Automation (ICRA'99),* Detroit, MI., May 1999, pp.1812-1820.

[Wan00] S. Wang and K.G. Shin, "Generic programming paradigm for open architecture controllers", *Proceedings of WAC 2000*, Muai, Ha, June 2000.

[Wan00b] Private Communications: Shige Wang, University of Michigan, Ann Arbour Michigan, April 2000.

[War94] C. Ward, "What is Agility", *Industrial Engineering*, (November, 1994) pp.15-16.

[Wil01a] B. Williamson, *The Scanning Tunnelling Lego Probe*, 2001, (Available September 2003 at http://ozbricks.com/benw/lego/probe/zdrive1.jpg).

[Wil01b] B. Williamson, *The Lego Plotter*, 2001, (Available September 2003 at http://ozbricks.com/benw/lego/plotter/xrack.jpg).

[Wil01c] B. Williamson, *The Lego Pneumatic Valve*, 2001, (Available September 2003 at http://ozbricks.com/benw/lego/valve/index.html).

[Wir90] R. Wirfs-Brock, B. Wilkerson and L. Wiener, Designing Object-Oriented Software, Englewood Cliffs, NJ: Prentice Hall, 1990.

[Wol98] J. K. Wolf, D.Glas, B. Erwin, Getting Started: A Teacher's Guide to LEGO Engineer, (Availabel January 2003 at: http://ldaps.arc.nasa.gov/LEGOEngineer/manual.html).

[You93] E. Youdon, Decline & Fall of the American Programmer, Englewood Cliffs, NJ: Prentice Hall, 1993.

[Yun99] M. Yunus, Banker to the poor: micro lending and the battle against world poverty, New York, NY: PublicAffairs, 1999, p.34.

[Zay96] J. Zaytoon, "Specification and Design of Logic Controllers for Automatic Manufacturing Systems," *Robotics & Computer-Integrated Manufacturing*, **12#4** (1996) pp.353-366.

[Zel97] M. V. Zelkowitz, and D. Wallace, "Experimental validation in software engineering," *Information and Software Technology*, **39#11** (November, 1997) pp.735-743.

[Zie91] J. Ziegler and H.J. Bullinger, Formal Models and Techniques in Human-Computer Interaction, *Human Factors for Informatics Usability*, B. Shackel, S. Richardson, eds., Cambridge, England: Cambridge University Press, 1991, pp.183-206

[Zho92] M. Zhou, F. Dicesare and A.A. Desrochers, "A Hybrid Methodology for Synthesis of Petri Net Models for Manufacturing Systems", *IEEE Transactions on Robotics and Automation*, **8#3** (June, 1992) pp. 350-361.

Appendix A

Methodology Overview

A stepwise overview of the methodology follows:

- Identify system by determining boundary of work-cell
- Partition work-cell into subsystems
- Partition subsystems into components
- Create a system object for the work-cell
    - Provide work-cell name
    - Provide all system level input condition descriptions and input information for:
        - E-Stop
        - Auto selector switch
        - Manual selector switch
        - Auto push button
        - Manual push button
        - Run-Out selector switch
        - Start subsystems push button
        - Stop system push button
        - End-of-Cycle-Stop push button
    - Create internal (private) conditions which provides system level state machine and external (public) interface conditions (used by subsystems)
        - Create the "Automatic" condition
        - Create the "E-Stop" condition
        - Create the "System End of Cycle Requested" condition
        - Create the "System Run Out Requested" condition
        - Create the "All Subsystems in Auto" condition
        - Create the "End of Cycle Request & All Subsystems in Home Position" condition
        - Create "System in Automatic State" condition
        - Create "System in Manual State" condition
- Create subsystem object for each subsystem belonging to the work-cell
    - Provide subsystem name
    - Provide all subsystem level input condition descriptions and input information for:
        - Auto selector switch
        - Manual selector switch

        - Single cycle push button
        - Single cycle selector switch
        - Auto cycle push button
    - Create internal (private) conditions which provides subsystem level state machine and external (public) interface conditions (used by system and contained components)
        - Create the "Automatic Mode Selected" condition
        - Create the "Subsystem Automatic State" condition
        - Create the "Safe for Other Subsystems Cycle Condition" condition
        - Create the "Subsystem Manual State" condition
        - Create the "In Home Position" condition
        - Create the "In Single Cycle State" condition
        - Create the "OK to Enter Auto Position" condition & logic
        - Create the "Drop Subsystem Out of Auto" condition & logic
        - Create the "Subsystem Lost Auto can Resume" condition & logic
- For each subsystem, create an object for each contained component
    - If component has a sequence of steps create a sequential control component
        - Provide component name
        - Provide all sequential component level input condition descriptions and input information for:
            - Single cycle selector switch
            - Single cycle push button
            - Single step selector switch
            - Single step push button
        - Create internal (component level) and interface conditions:
            - If single step required, create single step active condition
            - If single cycling required, create single cycling condition
            - Create sequence local auto cycling condition
            - Create auto cycling condition
            - Create auto activity condition and logic
            - Create auto idle condition and logic
            - If single step required create single step state machine conditions:
                - Single step idle
                - Single step wait for push button release condition
                - Single step complete
            - If single cycle required create single cycle state machine conditions:
                - Single cycle idle
                - Single cycling
                - Single cycle complete - wait for push button release condition
        - Create power-up initial state

        - Create "manipulate component into home position" state
        - Create transition connecting above two states
        - Create a manual home state
        - Create an automatic home state
        - Create an automatic state for each automatic step of the component sequence of operation
        - Create a corresponding manual state where applicable
        - Create all input condition descriptions and input information associated with this component sequence of operation
        - Create all actions associated with this component
        - Create all condition/event expressions needed to move from each step to all successive steps
        - Create all condition/event expressions needed to allow manual control of actions (push buttons that must be pressed when in a manual state)
        - Assign actions to all states where they are needed to create the state actions and qualify them with condition/event expressions for manual control
        - Create transitions and assign the appropriate condition/event expressions connecting each state to its successive states
    - If component does not have a sequence but must be running for the machine to be in automatic
        - Provide component name
        - Provide all input condition descriptions and input information for:
            - Start push button
            - Stop push button
            - Component is active
        - Create output that turns on the component in the manner appropriate for automatic operation
        - Create interface condition that indicate the component is fully operational (running and up to speed)
    - If component does not have a sequence but must provide manual control when the machine to be in manual
        - Provide component name
        - Provide all input condition descriptions and input information for:
            - Start push button
            - Stop push button
            - Additional inputs as required (forward, reverse, jog, etc)
        - Create outputs that turns on the component in the manner appropriate for the manual operation selected by the inputs
        - Create output action control conditions by creating condition/event expressions consisting of the inputs
        - Assign appropriate control condition to associated output

    - For each occurrence of a constraining condition (interface situation) that can not be directly determined by inputs/sensors that must be remembered when the machine is turned off, create a sequential constraining component
        - Provide component name
        - Create state machine that keeps track of constraint
        - Create interface conditions
            - Constraint is present
            - Constraint is not present
- For each sequential constraining component assign the appropriate sequential component state that indicates the occurrence of the constraint and the state that indicates clearing of the constraint
- Generate the Boolean logic equations that needed the complete model (were not previously generated) based on the model to logic transformation patterns determined by capturing the controls expert's knowledge
- Assign PLC control bits and generate PLC ladder logic from the logic equations

Appendix B

Ladder Logic Implementation

When PLC's were first introduced, they were designed to operate in the same manner as electro-mechanical relays to improve acceptance. The elements that one will find in most PLC's are: coils; normally open and closed contacts controlled by a coil; inputs, outputs, timers and counters. Normal coils are analogous to standard electromechanical relays. When electrical current flows through them, they will pull in a spring loaded metal bar that will close or "make" the normally open contacts of the relay. Electrical current can now flow through these contacts. When no current flows, the spring will push back the bar and the normally open contacts will break their contact. When they are not energized the contacts that they control are placed into the normal condition; that is the condition in which they are drawn in. A normally open contact will be open when its controlling coil is not energized and will be closed when the coil is energized. The opposite is true for normally closed contacts. The same convention applies to switches and sensors. They are drawn in the condition that they are in when they are not activated (switch is not pressed, no objects present for sensor to sense). To understand how to read Ladder Logic one considers the two vertical lines on either edge of the diagram to be connected to the opposite terminals of a voltage source. Typically, a coil is directly connected to the right vertical line. To energize the coil, a path must be established to it from the left vertical line. One simply starts at the left and follows the

paths through contacts, switches and sensors that are currently closed. Should a complete path from the left vertical line to the coil be present, the coil will be energized or "turned on". One of the benefits of PLC's is a real-time display of the status of these contacts and of the coils. In addition, the ability to manually force a contact into a closed or open condition is often provided. With a descriptive naming convention and a standard topology, one can readily determine the function of the Ladder Logic circuit and see what is currently missing to allow the coil to be energized.

One of the electro-mechanical relay characteristics usually implemented is the de-energizing of all non-latched coils when power is first applied to the PLC. The application of power to the PLC is indicated in the state machine with the "Initial Power On" transition that has no source state. This transition can be simply implemented by taking advantage of the one-to-one relationship between states and coils. The initial power on transition takes advantage of each state being represented by an individual coil, and that all of these coils will be de-energized when power is first applied. When a coil representing a state is de-energized, a normally closed contact controlled by this coil will be closed. Thus the initial power on transition is the Boolean "And" of the series of normally closed contacts where there is a closed contact for each state in the state machine excluding the power-on-reset state as shown in Figure B-1. A simple manual method for constructing this logic is to first create the coil representing the power on reset state. Then a coil is added for each state in the state machine and a normally closed contact controlled by the new coil is added in front of power-on-reset state coil. Should a

state be removed, the normally closed contact that it controlled is also removed from the initial power on transition.

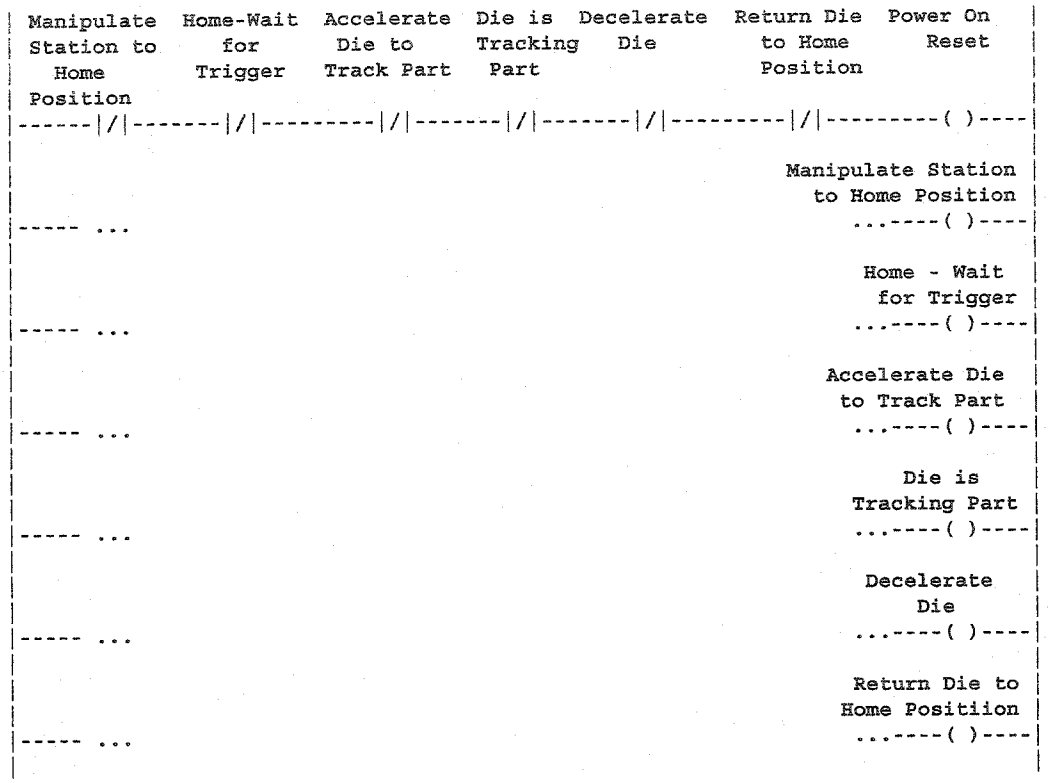


Figure B-1: Power On Reset Logic.

As previously mentioned the aim of the topology was to allow a technician to readily recognize the elements of the state machine in the Ladder Logic. Figure B-2 shows the generic topology where a single coil is used to represent each state. The top segments shown in Figure B-3 represent all of the state transitions leading to this state. At the far right is a contact that acts as a "guard" condition to limit the segments to the left of it to have effect on the state coil only when the station is cycling automatically. This

contact is labelled "Station Cycling Automatically". To the left of this contact, each straight segment going to the left rail is used for each state that this state can be entered from while the station is cycling automatically. These segments start with the contact that is energized when the state machine is in the particular state that the transition originates from. This contact is followed by a contact that is energized when the condition or event is present indicating that the particular transition into this state can occur. Should that condition or event consist of a logic equation, it will be implemented separately with a coil representing it. This event or condition coil will control the single contact in the transition segment. This, in effect, is a form of "information hiding". The technician is presented with only a single contact hiding the details of how the transition logic is actually implemented. When diagnosing a fault, the technician can focus on the higher level of information pertaining to why a state is not entered without needing to first sort through detail that might not be applicable to the current situation.

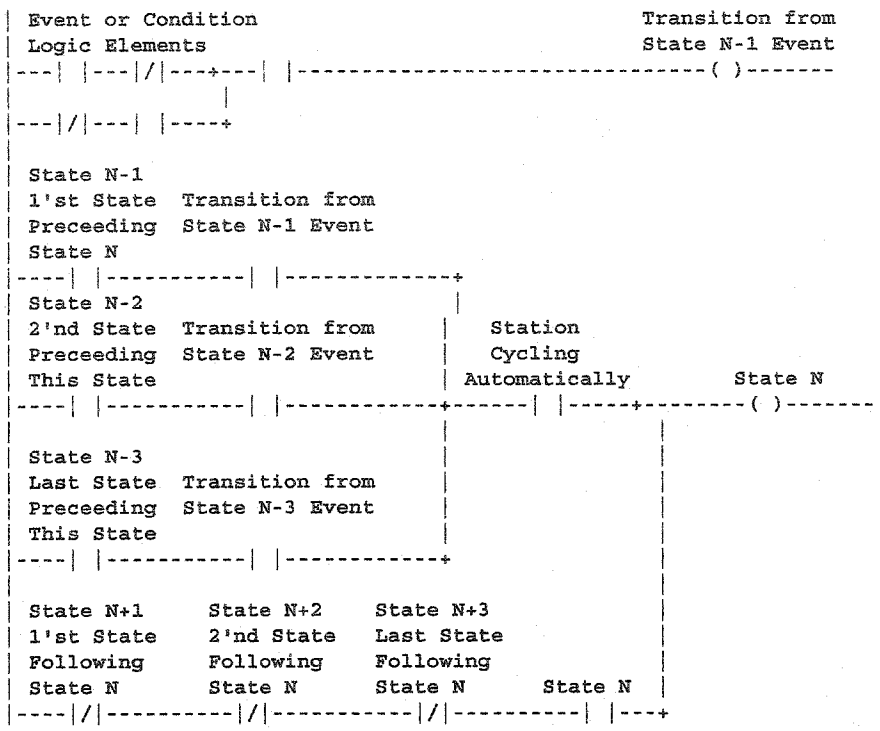


Figure B-2: General Topology For a State.

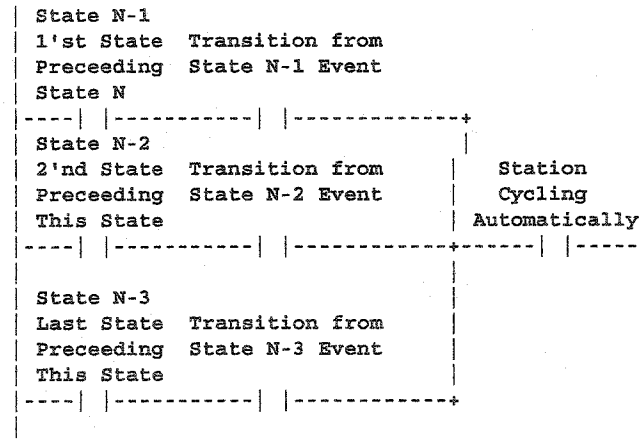


Figure B-3: State Transition Logic.

The bottom section of the state logic shown in Figure B-4 represents the "Latch". The function of the latch is to maintain the state coil energized after a transition section has energized it and to "drop out" or break the energy path to the state coil when a state

that follows is entered. Examining the logic from right to left shows that the circuit will be complete as long the state N coil is energized and the coils for the states that are entered from state N (states N+1 through N+3) are not energized. This occurs upon entry into state N. This entry into state N causes the State N coil to become energized. This in turn completes the latch circuit. The latch circuit now maintains state N coil energized. This is required since the transition logic that initially energized will become open. When the transition into one of the states that follow state N occurs, such as the into state N+1, its coil "State N+1" will become energized causing the normally closed contact "State N+1" to open. The latch logic circuit is now "open" so it will no longer maintain state N coil energized.

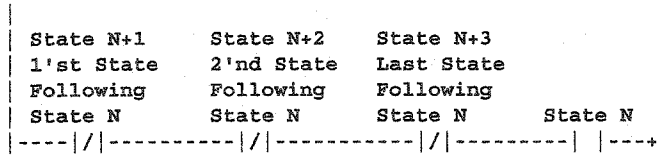


Figure B-4: State Latch Logic.

The sequence of events is as follows for the circuit shown in Figure B-2:

- The station is cycling automatically so the "Station Cycling Automatically" contact is closed
- The station in state N-1 that has a transition to the state that we are observing (state N) so the source state N-1 coil is currently energized, closing the source state N-1 normally open contact.

- Since we are in state N-1, all of the other source state contacts are open preventing current flow through their segments.
- The transition from state N-1 specific event or condition has not occurred so that contact is open preventing electrical current to flow through that segment
- Since we are not in state N, the state N coil is not energized.
- In the latch section, the state N contact is open since state N coil is not energized
- Since there is no complete path, state N coil is not energized
- Transition from sate n-1 specific event occurs closing that contact establishing a complete path through "Source State N-1", "Transition from State N-1 Specific event or condition" and "Station Cycling Automatically"
- State N coil is energized indicating that the machine is now in state N
- The "state N" contact in the latch segment is now closed while the remaining normally closed contacts for the destination states remain closed since we are not in any of those states. This provides a second complete path to keep the "state N" coil energized
- The latch segment of state N-1 coil has a normally closed state N coil in it that will now open removing energy from state N-1 coil
- The source state N-1 contact in the state N rung will now open breaking that path. However, the latch path is still complete since it relies on only state N being energized and none of the states that are entered from state N being energized.
- Eventually, the station sequence causes the conditions for the transition to state N+1 following state N to complete a state transition path in logic for state N+1.

- State N+1 coil is energized opening the normally closed state N+1 contact in the latch segment of stat n coil logic. Since this was the only complete path of state N, that coil will now open, de-energizing the state-n coil.

The Ladder Logic topology used for implementing states lends itself to manual modification. Figure B-5 shows the modification needed to add a new state between two existing states. The state sequentially before the new state requires only a single change in the "latch" rung. The normally closed contact of the state coil that originally followed is replaced with the normally closed contact of the new state. The logic for the last state in the sequence replaces the segment containing the original source state normally open contact with the new state and the transition specific contact is replaced with the new transition.

The logic for activating the outputs was also designed for ease of understanding and manual modification. A typical rung is shown in Figure B-6. At the far left connected in parallel are all of the state coils for which this particular output is to be energized. If the code were optimized for small size, these outputs would be reduced by standard Boolean algebra simplification. However, this would make it more difficult for a technician looking at the code to determine which states controlled the output. In addition, if a later modification were required, the simplification techniques would need to be applied again. It is simpler, at the expense of additional contacts, to simply place all of the states controlling the output in series. The technician can tell by the presence of the state contacts which states are used. Addition or deletion of states controlling this

output is requires only the insertion or deletion of a single contact rather than deriving and changing a Boolean equation.

A common configuration found is pairs of outputs that cause opposing motion of a mechanism or motor. For example a pneumatic clamp will have an output to clamp and one to unclamp. It is a common practice with such a pair of outputs to insure that both cannot be energized at the same time. This is accomplished by placing a normally closed contact of the opposing output before the coil energizing the output as shown.

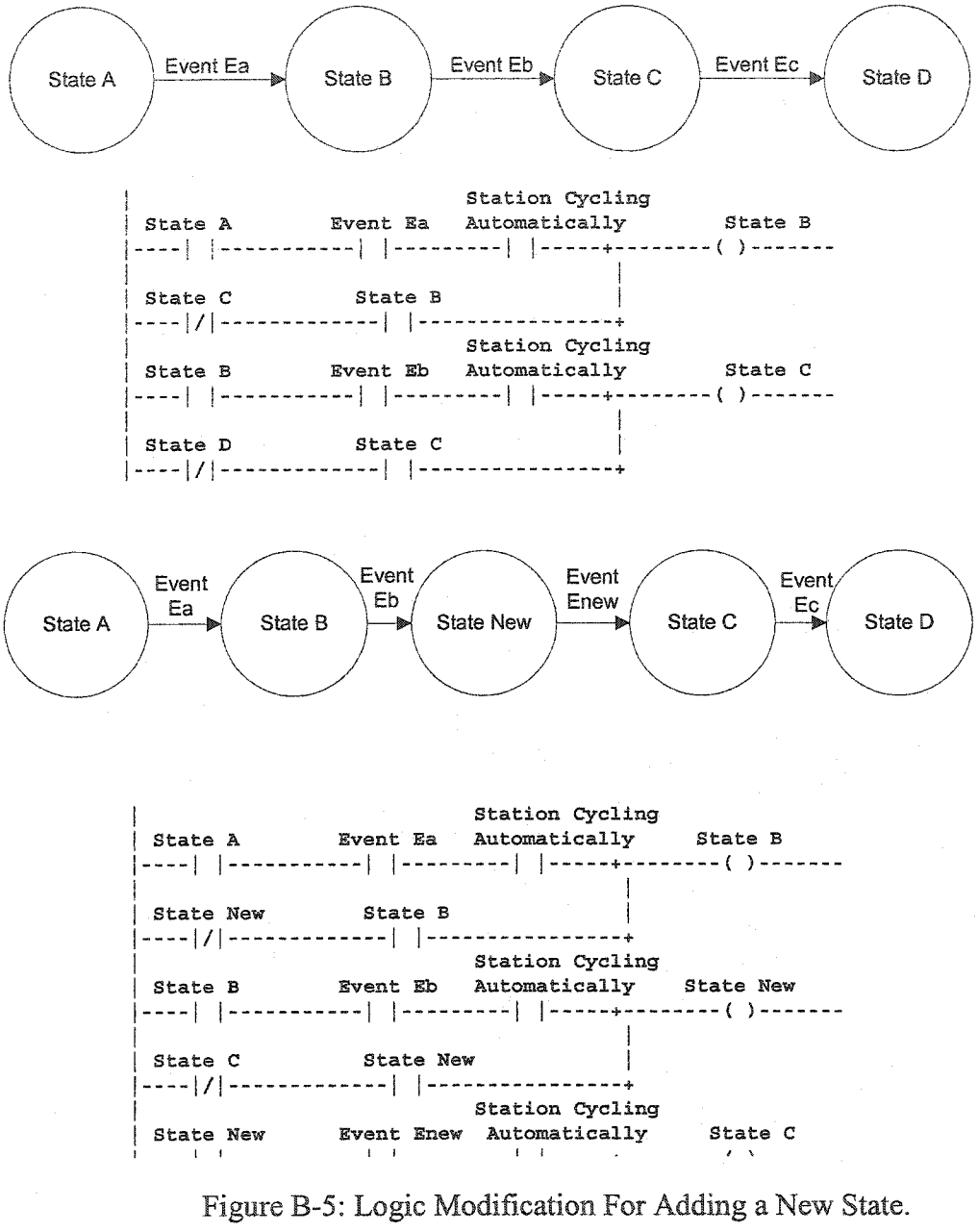


Figure B-5: Logic Modification For Adding a New State.

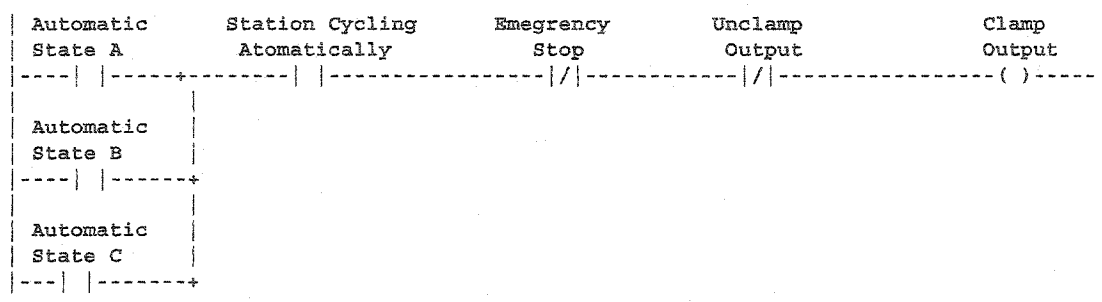


Figure B-6: Output Logic.

The state machine logic will keep the current automatic state coil energized should an emergency stop condition occur or if the station is no longer cycling automatically, but not placed into a manual mode. The majority of outputs controlled by the state machine cause a motion to occur. When an emergency stop condition occurs, the machine is to be placed into a safe condition with no motion. Thus outputs that cause motion need to be de-energized. To accomplish this a normally closed emergency-stop contact is placed directly to the left of the opposing output contact. A normally closed contact used to indicate the emergency stop state is not what one would expect when normally open contacts are used for normal states. The reason for this difference is a "fail safe" approach used for emergency stop that would result in an emergency stop should the wiring break or be left disconnected. When mechanical relays were used, the emergency stop relay coil was connected through a series circuit consisting of normally closed emergency stop buttons and switches. When an emergency stop condition occurred a button or switch opened the circuit de-energizing the coil, causing normally closed contacts to open. Thus, should the wire be disconnected or broken, the coil would become de-energized resulting in an emergency stop. As with emergency stop, when the

station is no longer cycling automatically, but not placed into manual, outputs causing motion need to be de-energized. Station auto-cycling is not considered to be a safety critical condition so a normally open contact is placed to the left of the e-stop normally closed contact. The station must be cycling automatically for this contact to be closed, which allows the normally open state contacts to the left of it to control the outputs.

Appendix C

Detailed Evolution of Application Generator Design

The user interface went through three iterations. The first pass was a fairly abstract representation with many user controls for examining the internal data structures of the generator. The complexity of the interface was then simplified leaving only the minimum needed for the second pass. Finally, the interface style was completely rewritten when it became apparent that the targeted user needed a more concrete representation and a simpler method of navigation. The third iteration was based on Raskin's Zooming Interface Paradigm [Ras00]. The implementation of the "system" object is used to provide a representative detailed description of the different iterations in this Appendix.

C.1 First Pass – System Object

The table representing the system class is shown in Figure C-1. The primary key or main index is the "System_ID". This is a unique value that identifies a particular system. The application generator is envisioned to store several systems and thus needs the ability to locate a specific one. This "System_ID" value is used to implement an aggregation association between systems and subsystems (subsystems belong to, are a part of, are contained by). The abstract model class diagram shown in Figure 5-11 of chapter 5 requires this association. In addition there are other association that serve the needs of the application generator only. The output of the generator is used as input to a PLC tool chain. The PLC tool chain typically requires a table for the inputs, outputs,

timers and counters. This table contains the symbolic name and addressing information such as the I/O card identifier and the particular screw terminal, the identity of the timer or counter and further information such as the time scale and type (count up/down, retentive timer etc.). Since there can be more than one system in the application generators database a method of distinguishing which system they belong to is required. The "System_ ID" field in the database tables provides the necessary linkage as shown in Figure C-2. By using an index one can then use the database SQL utility to readily locate these objects using a minimum amount of custom code. A sample code segment is shown in Figure C-3 and the resulting table in Figure C-4. The previous approach of creating the classes with an object-oriented language such as C++ would have required writing custom code to implement such a function. In the sample code one can see the utility of the "System_ID" index. The value corresponding to the system that is desired is passed in as a parameter. This parameter is then used in building up the text string "Query_Text" to filter the selection to only records representing objects that belong to this system. The database is interfaced through "Data Access Objects" (DAO) where this string is used as an SQL statement that returns the desired objects sorted in numerical order, in a collection named a "record set". In this code segment the record set is named "Conditions_dyn". One then can simply loop through this record set and process each object in turn. This code segment illustrates how using tables in a relational database as classes to store objects as individual records with SQL processing through the DAO application interface (API) to implement common functions such as selecting and stepping through a collection, greatly simplifies the code of an application generator.

Microsoft Access - [System_Info : Table]

File Edit View Insert Tools Window Help

Field Name	Data Type	Description
System_ID	AutoNumber	
Name	Text	Short mnemonic name used for creating PLC symbolic names
Longer_Name	Text	Longer name displayed in user interface
Description	Memo	Text description of the system
Design_State_ID	Number	
Notes	OLE Object	MS Word document used as container for dosumentation & video
Visibility_Kind_ID	Number	
Auto_Condition_ID	Number	indicates to rest of system that automatic cycling should occur
E_Stop_Condition_ID	Number	indicates to the rest of that an emergancy stop condition is present
E_Stop_Input_Info_ID	Number	Used to locate input infofor emergrency stop circuit
Auto_ss_Input_Info_ID	Number	Used to locate auto contact input info of 3 position mode selector switch
Manual_ss_Input_Info_ID	Number	Used to locate manual contact input info of 3 position mode selector switch
Auto_pb_Input_Info_ID	Number	Used to locate contact input info of auto push button switch
Manual_pb_Input_Info_ID	Number	Used to locate contact input info of manual push button switch
Run_Out_Input_Info_ID	Number	Used to locate contact input info of run system out selector switch
Start_SubSystems_Input_Info_ID	Number	Used to locate contact input info of start subsystems push button switch
Stop_System_Input_Info_ID	Number	Used to locate contact input info of stop system auto run push button switch
System_EOC_Stop_Input_Info_ID	Number	Used to locate contact input info of stop system at end of cycle push button switch
Supervisor_Subsystem_ID	Number	use to find next level up grouping in plant - future use
Supervisor_Component_ID	Number	id used by next level up to locate this system - future use
End_of_Cycle_Requested_Condition_ID	Number	indicates to rest of system to stop at end of cycle
Run_Out_Requested_Condition_ID	Number	indicates to rest of system to empty it of parts
All_Subsystems_in_Auto_Condition_ID	Number	private used to determine when all subsystem in auto
EOCR_All_Subsystems_Home_Condition_ID	Number	private used to determine when all subsystems in auto and end of cycle stop is needed
Auto_State_Condition_ID	Number	private used for auto state
Manual_State_Condition_ID	Number	private used for manually held atuo state
Next_Available_PLC_Bit	Number	Next PLC control bit to assign-allows updates without needing to regenerate bit assignments
Input_Conditions_and_Info_Created	Yes/No	Indicates if system input conditions and info have been created
Interface_Conditions_Created	Yes/No	Indicates if system interace conditions have been created

Field Properties

Figure C-1: System Table Representing System Class

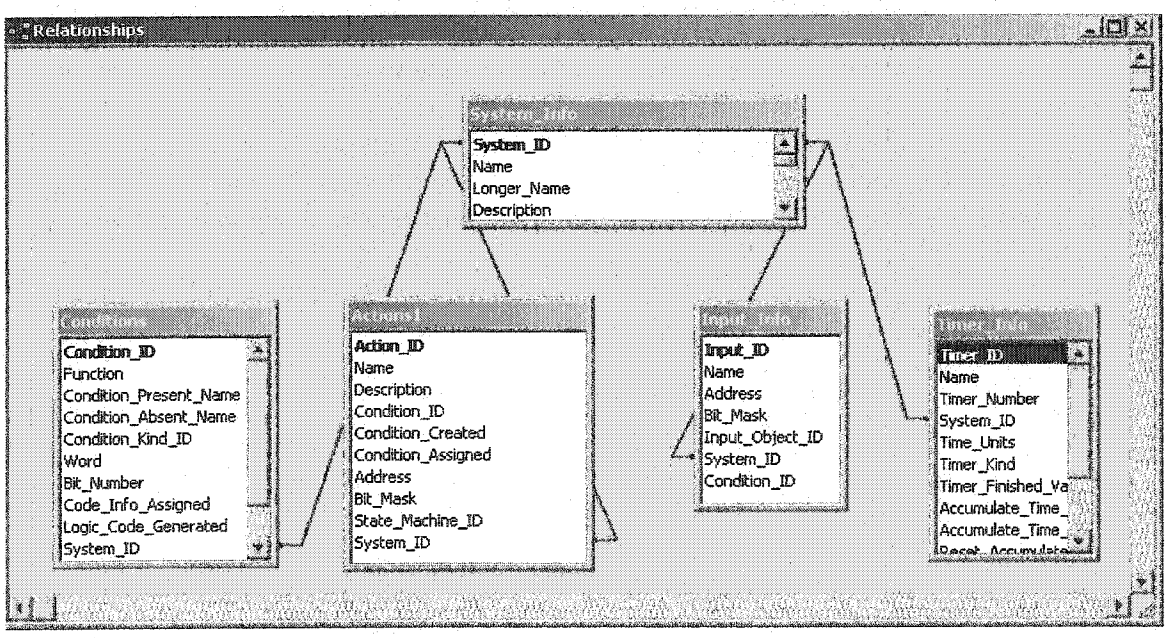


Figure C-2: Use of System ID to Link Tables

```
Private Sub Print_Control_Bit_Symbols(System_ID, Symbol_File)
  Dim Query_Text As String
  Dim Conditions_dyn As Recordset
  Dim str_24_Char_Max_Symbol As String
  Dim str_Siemens_Bit_Number As String
  Dim str_Siemens_Byte_Number As String
  Query_Text = "SELECT Bit_Number, Condition_Present_Name From Conditions " & _
    "Where Currently_Used = Yes And System_ID = " & _
    System_ID & " ORDER BY Bit_Number"
    Set Conditions_dyn = Controls_db.OpenRecordset(Query_Text, dbOpenDynaset)
    If Conditions_dyn.BOF = True And Conditions_dyn.EOF Then
    Else
      While Conditions_dyn.EOF = False
        If optb_Allen_Bradley.Value = True Then
          Symbol_File.WriteLine ("B3/" & Conditions_dyn("Bit_Number") & _
          " " & Conditions_dyn("Condition_Present_Name"))
        Else
           str_24_Char_Max_Symbol = _
                 VBA.Left(Conditions_dyn("Condition_Present_Name"), 24)
          str_Siemens_Bit_Number = (Conditions_dyn("Bit_Number") Mod 8)
          str_Siemens_Byte_Number = (Conditions_dyn("Bit_Number") - _
                                     str_Siemens_Bit_Number) / 8
          Symbol_File.WriteLine (""" & str_24_Char_Max_Symbol & """, ""M " & _
           str_Siemens_Byte_Number & "." & str_Siemens_Bit_Number & _
          """, ""BOOL"" , """ & Conditions_dyn("Condition_Present_Name") & """")
        End If
        Conditions_dyn.MoveNext
      Wend
    End If

End Sub
```

Figure C-3: Using SQL to Retrieve Objects

```
"System E-Stopped",          "M 0.0", "BOOL" , "System E-Stopped"
"System Cycling Automatic", "M 0.1", "BOOL" , "System Cycling Automatically "
"System End of Cycle Requ", "M 0.2", "BOOL" , "System End of Cycle Requested"
"System Run Out Requested", "M 0.3", "BOOL" , "System Run Out Requested"
"All Subsystems in Auto",   "M 0.4", "BOOL" , "All Subsystems in Auto"
"EOCR & All Subsystems Ho", "M 0.5", "BOOL" , "EOCR & All Subsystems Home"
"System Automatic State",   "M 0.6", "BOOL" , "System Automatic State"
"System Manual State",      "M 0.7", "BOOL" , "System Manual State"
"E-Stop Wrapper Line On",   "I 0.0", "BOOL" , "E-Stop Wrapper Line On"
"Auto S/Sw Wrapper Line O", "I 0.1", "BOOL" , "Auto S/Sw Wrapper Line On"
"Manual S/Sw Wrapper Line", "I 0.2", "BOOL" , "Manual S/Sw Wrapper Line On"
"Auto P/B Wrapper Line On", "I 0.3", "BOOL" , "Auto P/B Wrapper Line On"
"Manual P/B Wrapper Line ", "I 0.4", "BOOL" , "Manual P/B Wrapper Line On"
"Run-Out Wrapper Line On",  "I 0.5", "BOOL" , "Run-Out Wrapper Line On"
"Start Sub-Systems Wrappe", "I 0.6", "BOOL" , "Start Sub-Systems Wrapper Line On"
"Stop System Wrapper Line", "I 0.7", "BOOL" , "Stop System Wrapper Line On"
"End of Cycle Stop Wrappe", "I 1.0", "BOOL" , "End of Cycle Stop Wrapper Line On"
"Move Dstkr Down",          "Q 0.0", "BOOL" , "Move Dstkr Down"
"Move Dstkr Up",            "Q 0.1", "BOOL" , "Move Dstkr Up"
"Move Dstkr to Stack",      "Q 0.2", "BOOL" , "Move Dstkr to Stack"
"Move Dstkr to Conveyor",   "Q 0.3", "BOOL" , "Move Dstkr to Conveyor"
"Grasp Part",               "Q 0.4", "BOOL" , "Grasp Part"
```

Figure C-4: Symbolic Name and Addressing Information Table

The system state machine that models the behaviour of the system class shown Figure 5-12 requires specific inputs. A three-position selector switch is used to select the major system level modes of automatic, off and manual. This type of selector switch actually has only two contacts. One is for the automatic position and the other is for the manual position. The automatic contact is closed when the selector switch is in the automatic position and the manual contact is open. The reverse occurs in the manual position. In the off position, both the automatic and manual contacts are open. The information about the inputs that these contacts are connected to is located in a table named "Input_Info". An indexed field named " Input_ID " is used to locate the information pertaining to a specific input in this table. Thus to locate the information for the automatic and manual contact inputs, the " Auto_ss_Input_Info_ID" and "Manual_ss_Input_Info_ID" fields are used to store the index values. There is no input for the off position since there is no contact specifically available for this function. Logic is required to note that both the manual and automatic contacts are open to determine when the selector switch is in the off position. The state machine also requires a System automatic, a System Manual and a System Stop push button switch. These push-button switches have a single contact that is closed when the switch button is pressed. Thus each push-button switch requires an input. To locate the information for these inputs in the input "Input_Info" table the "Auto_pb_Input_Info_ID", "Manual_pb_Input_Info_ID" and "Stop_System_Input_Info_ID" fields in the "System_Info" table are used. The last input needed by the state machine is Emergency Stop. The index value stores in the "E_Stop_Input_Info_ID" field locate it. To support the generation of control code a

method was needed to specify the logic equations of the various states in the state machine. This would be implemented as a coil with its rungs in Ladder logic or as a variable set equal to the specified equation. In effect one could consider this to be a condition, thus a condition class was created and represented with a database table named "Conditions". To locate a specific condition object one needs to find the row in the table that represents the object. This is achieved in the same manner as in the table for systems by using an index field named "Condition_ID". Three conditions are needed to represent the state machine resulting in three condition ID fields in the system table. These are the "E_Stop_Condition_ID", "Auto_State_Condition_ID" and the "Manual_State_Condition_ID" corresponding to the "System is Emergency Stopped", "System is in Manually Held Automatic State" and "System is in the Automatic State". The system level object is responsible for determining the various system level conditions described in the abstract model chapter. The mechanism chosen for interfacing with other objects are the condition objects represented by the conditions table. Emergency stop is already available from the state machine. The system level objects needs to indicate to the rest of the objects when automatic behaviour should occur. This occurs in both the "Manually held Automatic" and the "Automatic" states. The conditions representing these states could have been used as the interface. However, this exposes an unnecessary level of implementation detail internal to the system object. To keep with the principle of information hiding, a separate single condition "Auto_Condition_ID" at a higher level of abstraction is used for this purpose. Had the emergency stopped condition been a combination of several conditions a separate condition would have also been used.

Since it was only a single state, the condition associated with it was sufficient. The remainder of the conditions in the system are used for private internal logic and states, as are the remaining variables.

To provide the user with an interface to manage the system object a form was initially provided shown in Figure C-5. From this form the user could directly manipulate fields in the system table that contained the information directly such as the "Longer_Name". It is displayed in the "Systems Information" grid in the upper left corner. The user could edit the name of the currently selected systems by typing in the grid. For information that is obtained from linked files, such as the inputs shown in the grid on the right side of the form, an indirect method was employed. First, the names of the fields in the system table that contain the linking information were stored in an array. When a specific system was accesses, its record would become available. Using the filed name array, each filed would be accessed by name. From that field the value used to locate the input description information in the "Input Conditions Descriptions" table. This table contains the meaning of the different conditions of the input in the context of this specific use. From this table, the linking information is then obtained to locate the record for the specific PLC input card and the screw terminal used for the input in the "Input_Info" table. Using this information, the grid on the system form is populated with the input information. A code segment shown in Figure C-6 illustrates this indirect access method. The reason for this indirection lies in the DAO API providing access to only one record at a time. Ideally, one would prefer to select all of the input information records and manipulate then directly. Whenever the user changes the information for a

specific input, such as the screw terminal, using the DAO API, requires the indirect process again. The system table field is used to locate the value for locating the linking information in the "input condition descriptions" which in turn is used to access the record for the input in the "input_Info" table. The code segment shown in figure C-7 illustrates this locate and update sequence. Recently Microsoft released a new API for accessing data named ADO.Net. This API does allow one to create an image of a table in memory in the form of an indexed collection. One now has direct access to the memory image of all record through the use of the index. Updates to the values are done directly. The database table values are updated from the memory image at one time when the system form is closed. The use of this API would allow direct manipulation of all of the system inputs, eliminating the need for the code previously shown. It is now appropriate to explain the reason for storing the input information in more than one table or class

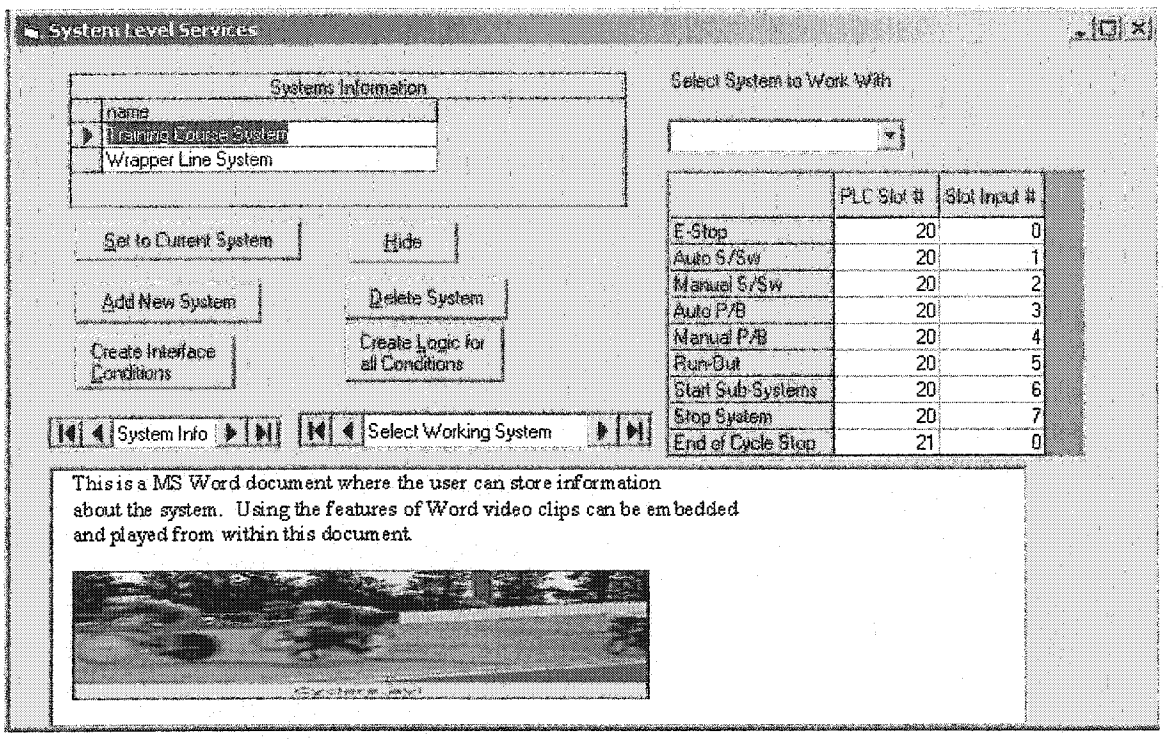


Figure C-5: First Pass Form for System Level Control

```
For Field_Name_Index = 1 To NUMBER_OF_INPUTS
  Input_Assignmnet_mfg.Row = Field_Name_Index
  Input_Assignmnet_mfg.Col = 1

  Input_Condition_Description_ID_lgi = _
    System_Info_dbc.Recordset(Field_Names_str(Field_Name_Index))
  Input_Condition_Descriptions_tbl.Seek "=", _
       Input_Condition_Description_ID_lgi
  If Input_Condition_Descriptions_tbl.NoMatch = False Then
    Input_ID_lgi = Input_Condition_Descriptions_tbl("Input_ID")
    Input_Info_tbl.Seek "=", Input_ID_lgi
    If Input_Info_tbl.NoMatch = False Then
      Input_Assignmnet_mfg.Col = 1
      Input_Assignmnet_mfg.Text = Input_Info_tbl("Address")
      Input_Assignmnet_mfg.Col = 2
      Input_Assignmnet_mfg.Text = Input_Info_tbl("Bit_Mask")
    End If
  End If
Next Field_Name_Index
```

Figure C-6: Indirect Accessing of System Input Information Code Segment

```
Input_Condition_Descriptions_tbl.Index = _
       "Input_Condition_Description_ID"
Input_Info_tbl.Index = "Input_ID"
'System_Info_dbc.Recordset(Field_Names_str(i)) = _
       Input_Condition_Description_ID_lgi
Field_Name_Index = Input_Assignmnet_mfg.Row
Input_Condition_Description_ID_lgi = _
       System_Info_dbc.Recordset(Field_Names_str(Field_Name_Index))
Input_Condition_Descriptions_tbl.Seek "=", _
      Input_Condition_Description_ID_lgi
If Input_Condition_Descriptions_tbl.NoMatch = False Then
  Input_ID_lgi = Input_Condition_Descriptions_tbl("Input_ID")
  Input_Info_tbl.Seek "=", Input_ID_lgi
  If Input_Info_tbl.NoMatch = False Then
    If Input_Assignmnet_mfg.Col = 1 Then
      Input_Info_tbl.Edit
      Input_Info_tbl("Address") = Val(Input_Assignmnet_mfg.Text)
      Input_Info_tbl.Update
    ElseIf Input_Assignmnet_mfg.Col = 2 Then
      Input_Info_tbl.Edit
      Input_Info_tbl("Bit_Mask") = Val(Input_Assignmnet_mfg.Text)
      Input_Info_tbl.Update
    End If
  End If
End If
```

Figure C-7: Locate and Update Sequence of System Input Information Code Segment

C.2 Second Pass – System Object

Once the first pass was completed it was integrated into an experimental environment. A pipe fitter was chosen to represent the targeed tradesperson user. His first task was to complete a training program using the application generator. The high degree of difficulty he experienced using it was first attributed to the large quantity of options on each screen and the lengthy names. Thus for the second pass, all user interface controls on the forms that were not essential were removed. The resulting second pass system form is shown in figure C-8. The user interface objects known as visual controls were reduced to a minimum. To accommodate trades-people who do not

use computers as regularly as engineers, explicit buttons are used for every function. For example, a "Done" button is provided to exit the form instead of relying on the users familiarity with Windows forms to use the built in controls. Attention to labelling is also required to avoid confusion over standard labels such as "close" where someone familiar with computer understands that this is equivalent to done and does not imply "'cancel". A specific button is provided to open the word document. The user is no longer expected to know that a right click on the control is needed followed by selecting "open" from the resulting drop down list. Instead of using controls to move through the database table records to select the system to work on, the users simply clicks on the desired system in the grid that displays all the currently created ones. Underlying code then manipulates the database navigation controls on the users behalf. Theses controls are now hidden reducing the amount of information the user needs to comprehend.

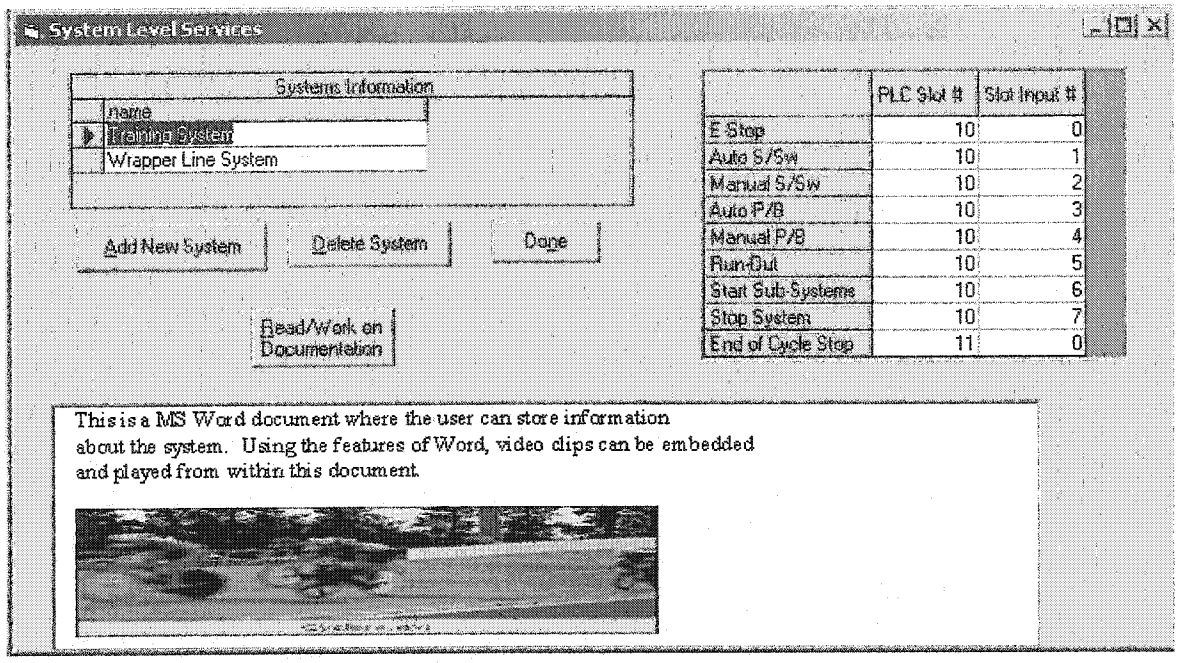


Figure C-8: Second Iteration Of System Form

The pipe fitter found the revised system easier to use but still experienced difficulty. He seemed to understand the abstract model and how to partition the experimental training system into system, subsystem and the various components. The difficulty was in building this model using the application generator. He felt the problem could be improved if application generator explicitly showed him the parts of the model that he had already built and how they fitted together. In addition, he had difficulty finding where to input the information that was needed. For example, he knew that there is always a selector switch to set the system into automatic. But it was not clear to him where you input the information about it. He did not automatically relate the switch with the two inputs labelled "Auto S/Sw" and "Manual S/Sw".

The availability of his time for the experiment was limited to a two to three hour period of use on weekends and on the occasional weekday. Thus not only was there a difficulty in constructing the mental image of the model as it was built during a session, there was further difficulty in needing to reconstruct it after the breaks in time between the sessions. One may argue that this is a special case caused by the circumstance of the experiment and thus not worth addressing. In the typical use case, one would use the application generator to create the initial control code in a single block of time. However, observation of the life cycle of control code in the Oshawa Plant identifies a second use case. Modifications are often made with considerable time between individual changes by different people. Thus the use case of a tradesperson being absent from the model for a period of time and then needing to interact with it again does need to be addressed.

The initial approach required more abstract thinking that was appropriate for use by trades-people. It may have been suitable for engineers who are use to abstract reasoning with minimal visual aids, however, trades-people work at a more concrete level in their day-to-day activity. The application generator user interface underwent a third pass redesign to address this issue. The ZIP paradigm was selected to provide a more tangible, less abstract form of interface.

C.3 Third Pass – System Object

The third pass design was very different from the first two. The first two approaches used the abstract model as the foundation for the design. Forms and underlying tables provided the user with the means to create and manipulate objects of the class represented by the form. These objects were presented in separate lists for selection. The author expected the mapping from the application generator objects to parts of the machine to be easily made in the users mind. Object Oriented theory relies on using tangible objects that are readily identified in the real application domain. Thus the users familiarity with such objects was expected to make the application generator based on these "familiar" objects easy to use. The weakness of the first approaches was in the presentation of these objects and the navigation through the user interface to locate them. For the third approach, representing the portions of the machine in a manner that was familiar to a trades-person replaced the focus of the model. In the first approach one typically navigated by selecting a containing object in one screen to provide a list on a second screen from which a selection of the desired object was made. Then a different screen needed to be accessed where the detailed information about the object was presented. Functions to manipulate the object were then located by navigating through a menu or by further selection of options and clicking on a button. For example one might go to the form representing the subsystem class. There a selection of the particular subsystem object is made. From there one moves to the form for the component class and selects from a list of component objects that are contained in the previously selected

subsystem object. Having selected the desired component, one would then move to the sequences form and finally from there, on to the "steps" form. On the steps form, an individual state would be selected. Upon making the selection, the form would then display the information pertaining to the state. The user knows that they want to work with a specific state machine state to perhaps add an action to it. Then they want to make a similar change to a different state machine. This requires the knowledge of the abstract model relations between the classes to know which forms to navigate through and in which sequence. Then it requires far more user activity to perform the input than a tradesperson typically has patience for. The ZIP approach used for the third pass relied on representing the physical machine as the basis for the design. A graphical representation was used to make navigation simpler and to remove the need for forming a mental image of what part of the model was currently in the application generator. At the highest level from the perspective of flying in an airplane, a screen presents all of the systems in the application generator database. On this screen all of the subsystems contained in each system and all of the components contained by the subsystems are also displayed. In the first approach, multiple forms or screens were used to address the issue of insufficient screen real estate to display all of the information. For the third pass the ZIP principle of using the available screen to present a window or port view of a portion of the entire landscape was used. The user can "pan" or move the window to different sections of the landscape. Unfortunately this type of interface is not used by typical Windows application, so support of the needed features is not directly provided by Visual Basic controls. A method was needed to implement a large graphical surface that

logically exceeded the boundaries of the form and was displayed through a window. This could be accomplished by first placing a "picture box" control on the form and adjusting its size to the desired window or port view. Then a second picture box was placed within it. The size of the second picture box was then increased by the program, to extend beyond the boundaries of the first picture box. Since the second picture box is contained within the first one, only the area currently framed by the first picture box is displayed. This provided the "unlimited" space needed to display all of the information contained in the database for a particular level.

The standard Widows keyboard does not provide a key specifically labelled for panning such a display. The author reviewed other implementations of ZIP [Mat00, MRL97, MRL98, Per91] and found the key mappings they used displayed in Figure C-9. These implementations typically stayed away from using the mouse. The author found that using the mouse provided a faster response than relying on keys remaining pressed and utilizing the auto-repeat keyboard feature to maintain a particular motion. One could use Quasi-Modes [Ras00] through pressing multiple keys at the same time to adjust the speed. However, this would require the user to remember the particular key, labelling the keys or displaying the information on the screen. Neither of these approaches were deemed to be as simple as using the mouse. The author is in agreement with Raskin that the user interface should be of as much service to the user as possible without requiring mental activities to which the human mind is not well equipped. The mouse was used instead of a "gun sight" cursor where the user would pan the landscape to place the area of interest under the cursor and then zoom down. Using the mouse the user moved the

cursor over the desired object. The appearance of the object changed as the cursor entered the area of a particular object to indicate that it was selected.

PC XT key	Normal Function	Shifted Function
up arrow	scroll world up	scroll heaven up
down arrow	scroll world down	scroll heaven down
left arrow	scroll world left	scroll heaven left
right arrow	scroll world right	scroll heaven right
page up	zoom in world	zoom in heaven
page down	zoom out world	zoom out heaven
right control	find	Find
left control	do	Do
right alt	grab	Drop
left alt	clone	Stamp
Esc	undo	Undo

Figure C-9: Key Mappings for ZIP Navigation, Taken form [Mat00]

The current implementation used colour because this was readily provided. This should be replaced in the future with a change in appearance such as using an internal fill pattern to overcome the issue of colour perception deficiency that is common. When the desired object is selected, the page down key is used to zoom to the next level of detail. The user can still position the cursor with the mouse to the centre of the screen (or anywhere else) and pan the display under it. The objects are selected as they pass under the cursor. They remain selected until the next object passes under the cursor. Currently,

the movement of the cursor is restricted to the displayed area. A future enhancement would pan the display in the appropriate direction as the cursor is moved beyond the edges.

The mouse is also used for selecting and moving objects on the display area. This provides the user the capability to arrange objects in a manner meaningful to them. During the training session, the pipe fitter had requested this function when dealing with the list of states for a state machine displayed in a grid. He stated that he could relate to and locate states better if they were arranged in the natural sequence followed by the machine. Otherwise, he needed to read the description of each state and make the mental connection to locate the desired one. By using the ZIP approach, the user can find the desired state through spatial recognition and need to read only as much of the description as needed to verify the correctness. In addition to using the mouse, the user can use the arrow keys with a Quasi-Mode selected by the shift key to achieve the same movement. Both methods were provided to allow experimental evaluation of the two approaches as future work. Currently, an object can be moved beyond the edges of the display. The standard graphical objects provided by the operating system do not provide a simple method of detecting this event so additional code would need to be written that keeps track of the position of the edge of the moved object relative to the current edge of the display. As a future enhancement, the display should pan to keep the object visible as it is moved beyond the edges.

The ZIP paradigm proposes to present the user with a series of such display areas each with a different level of detail shown. Zooming up or down provides accessed in sequential level of increased or decreased detail. The ZIP principle would imply that all information available for each level should be accessible by panning at the particular level. To implement such an approach would require a very large expenditure of Windows resources (objects, lists, etc. and associated memory) as well as processing time to fully populate the levels. The author chose a compromise that provided a similar navigational functionality at a greatly reduced cost in terms of resources and processing time. Navigation over larger distances on a detailed landscape through panning was noted as inefficient by other implementers of ZIP [Ras00]. They indicated that the user would zoom up to a high enough level to locate the desired object directly on the displayed area, move to it and then progressively zoom down to the level of detail required. Thus the author chose to restrict the more detailed levels to display only the information related to the object selected at the previous less detailed level.

Figure C-10 shows the third pass implementation of the main form representing the lowest level of detail. On this screen are displayed all of the higher level objects comprising the abstract model. In the previous passes, the user needed to access four different forms to obtain the same information. Vertical positioning is used to represent the hierarchy of containment. At the top is each system. Then below is each subsystem they contain. Below and to the right are all of the components belonging to each system. As a future enhancement, lines shall be used to connect the objects to better illustrate their relationships. The ZIP paradigm promotes the use of individual command buttons

to perform specific task as opposed to having the user select options and then press a single button that changes its behaviour based on the selections. This approach minimizes the potential of user error resulting from forgetting or not checking which options were selected when pressing a general "execute" type of button. In keeping with this philosophy, separate buttons are provided for creating the different types of components. The long descriptive abstract class model names have also been replaced. Instead of selecting "Sequential Constraining Condition Component" from a list, the shorter more familiar "Create Mailbox Handoff" label is used for a button that creates such a component. To provide a more concrete representation closer in nature to the physical machine, all of the objects that the user directly specifies are shown. They do not need to be navigated to or selected from lists. For example, the component level between system and sequence is removed. It serves a function in OO modelling, but the user sees little value in needing to navigate through this level and then selecting from a list when he wants to work with the states of a specific sequence. Thus, direct access to the sequence is provided from the main screen. Similarly, direct access to the details of the systems is also available.

To provide a more concrete interface, the individual inputs are no longer represented as simply contacts in a list as shown in the system form in Figure C-8. To access this information the user now selects the system of interest on the highest level and then zooms down to a more detailed form shown in Figure C-11. Working with the pipe fitter indicated that in his mind, the control of a specific level was related with the control panel, or at least the collection of switches and lights assigned to it. To emulate

this, where possible, input contacts are grouped together and ideally represented graphically by the device that they are part of. The contact for a mode selector switch is no longer shown in a list. Instead, the mode selector switch itself is shown. Within the selector switch individual positions are then shown. Then instead of referring to abstract concepts such as address and bit position, the physical PLC I/O card position that the wires connect to named "PLC Slot Number" and the exact terminal identified as "Slot Screw Number" are used. A proposed future enhancement would provide an accurate graphical representation of each input and the layout on the screen should match the layout of the control panel. This will increase the use of the "familiar" that the user has from his daily activity of using such control panels. The use of such panels becomes a "habit" and enters a subconscious level. As mentioned by Raskin [Ras00], operating from this level is far more efficient and less prone to error then having to engage the conscious mind to read the labels on the screen and then determine what they represent. By using screens that can be panned and zoomed to increase the amount of display area readily available to the user, pictures could now be considered for identifying all of the sensors on a machine as well as indicating how they are used. Outputs could also be represented by the action that they cause. A graphical representation of control at just the sensors and actuator level has been developed by Wolf, *et al.* [Wol98]. A study of how to effectively use graphical representation for this purpose is for future study.

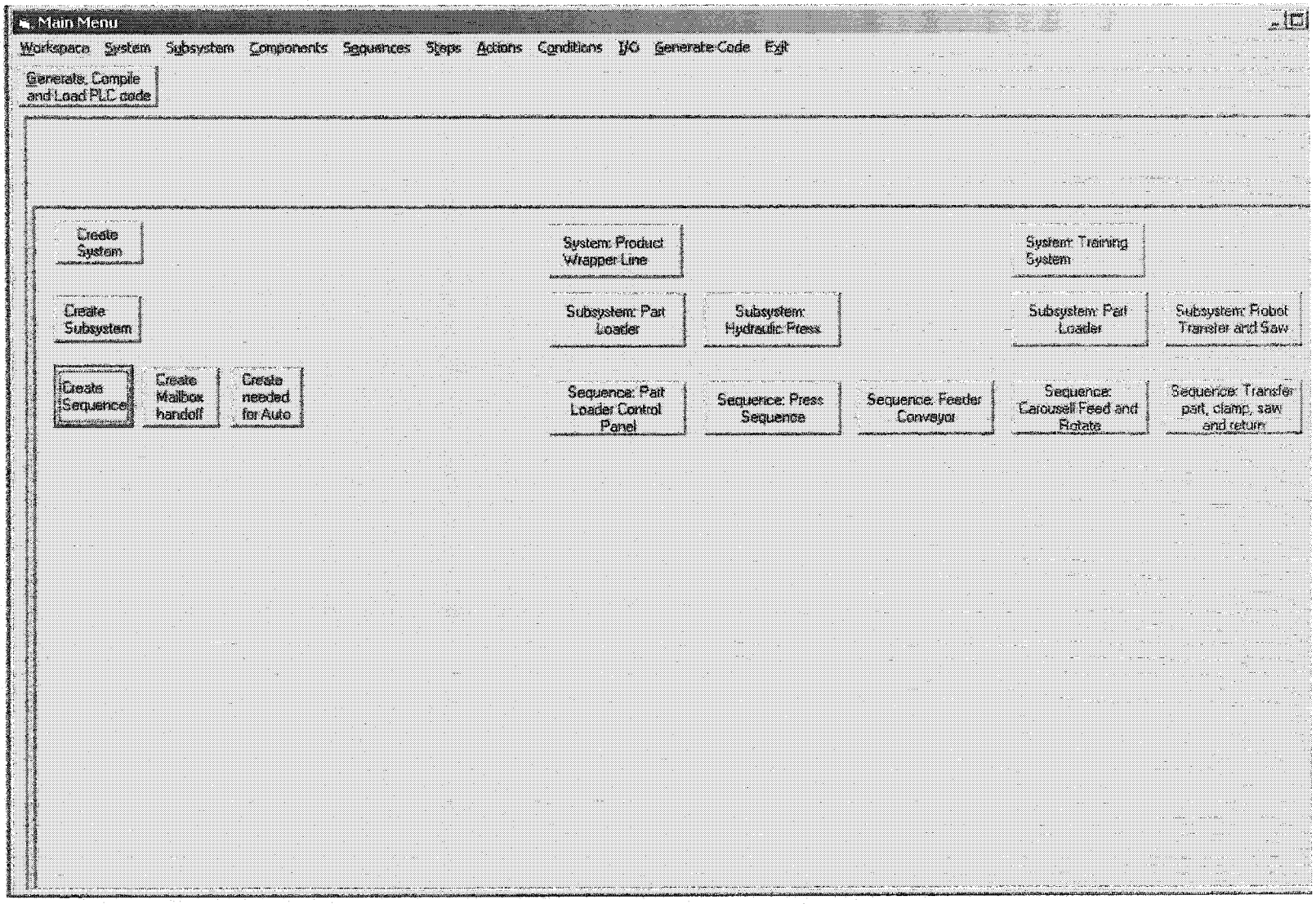


Figure C-10: Zooming Interface Paradifm Main Form

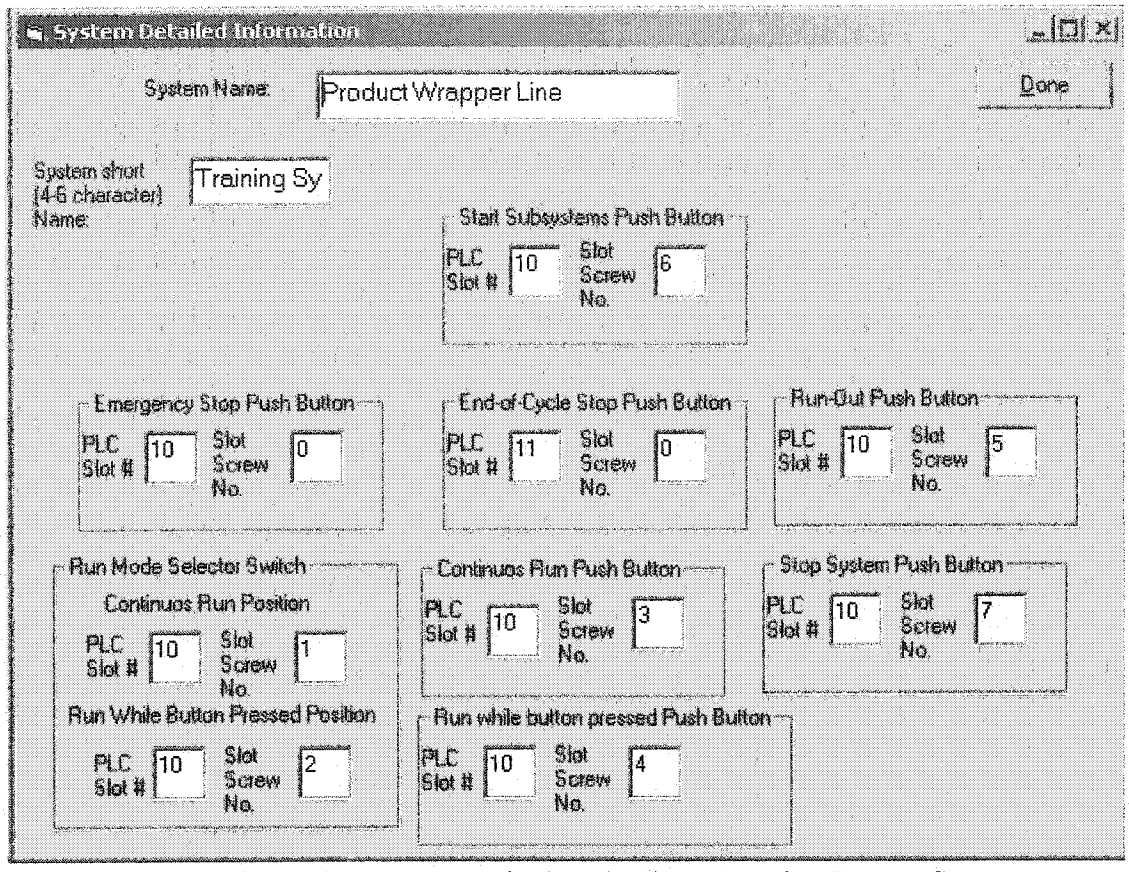


Figure C-11: Form Displayed When Zooming In on a System

Appendix D

Ladder Logic Generation for States Flowcharts

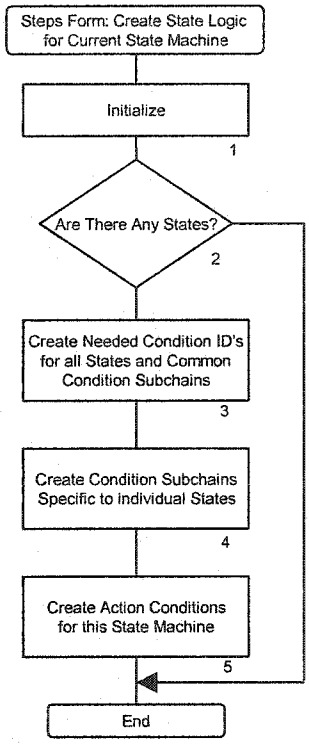


Figure D-1: Create State Logic for Current State Machine

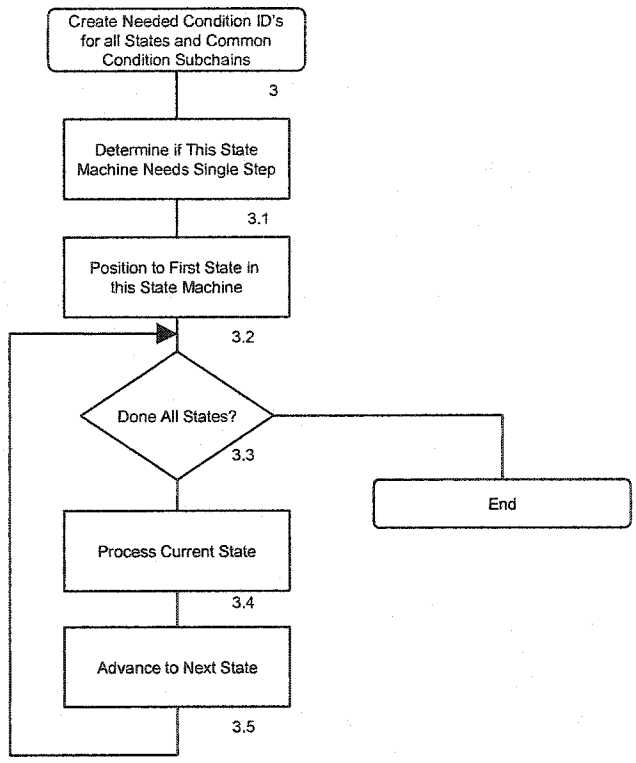


Figure D-2: Create Needed Condition ID's for all States & Common Condition Subchains

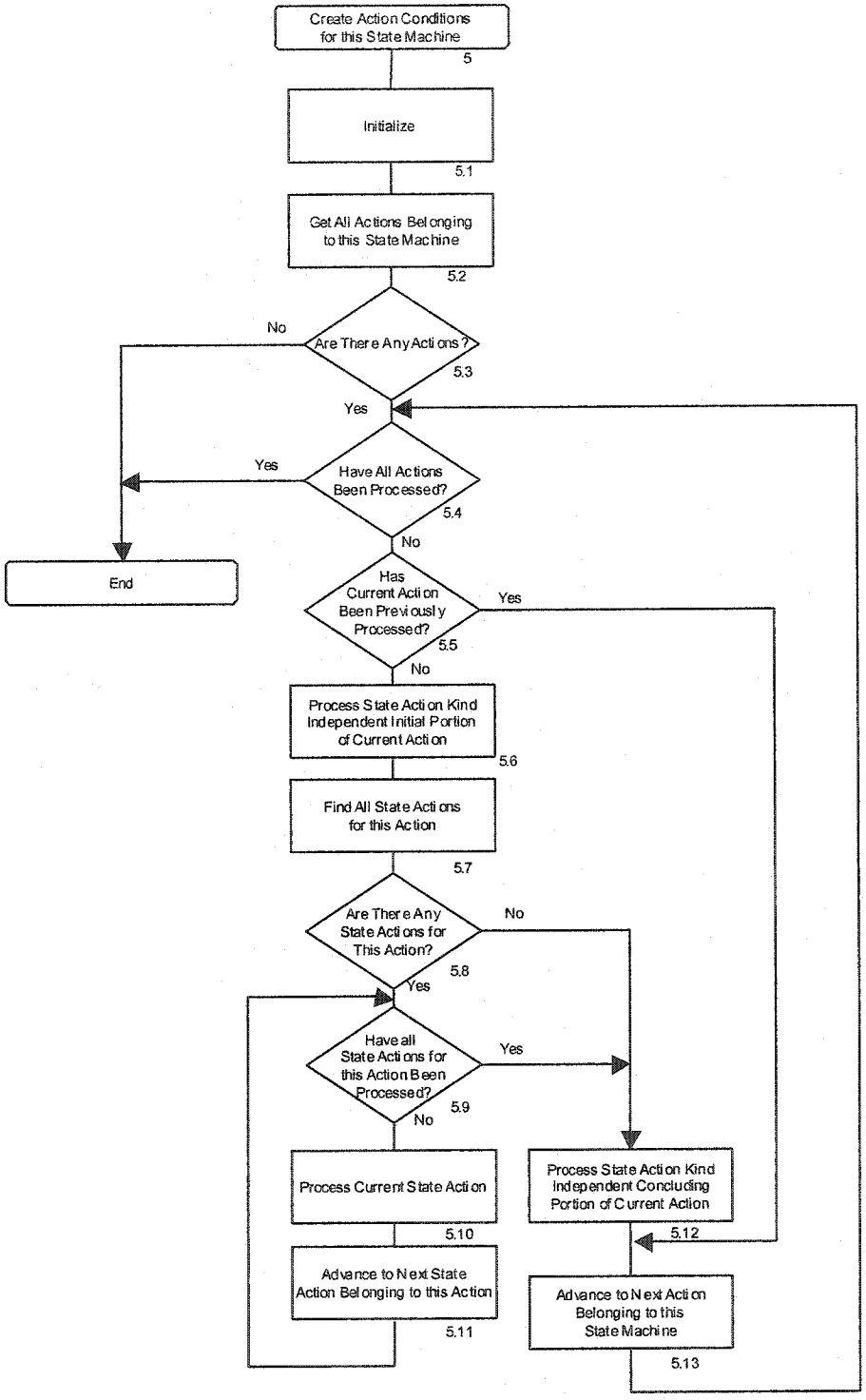


Figure D-3: Create Action Conditions For All State Machines

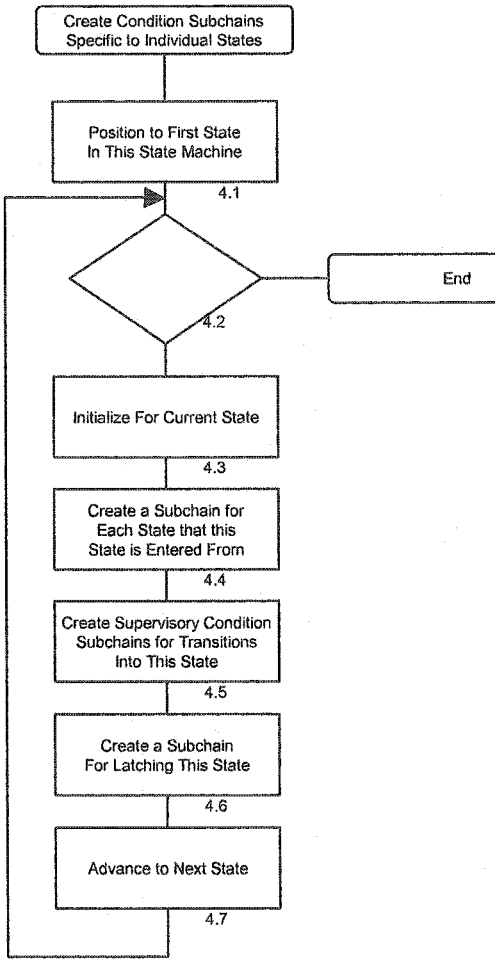


Figure D-4: Create Condition Subchains Specific to Individual States

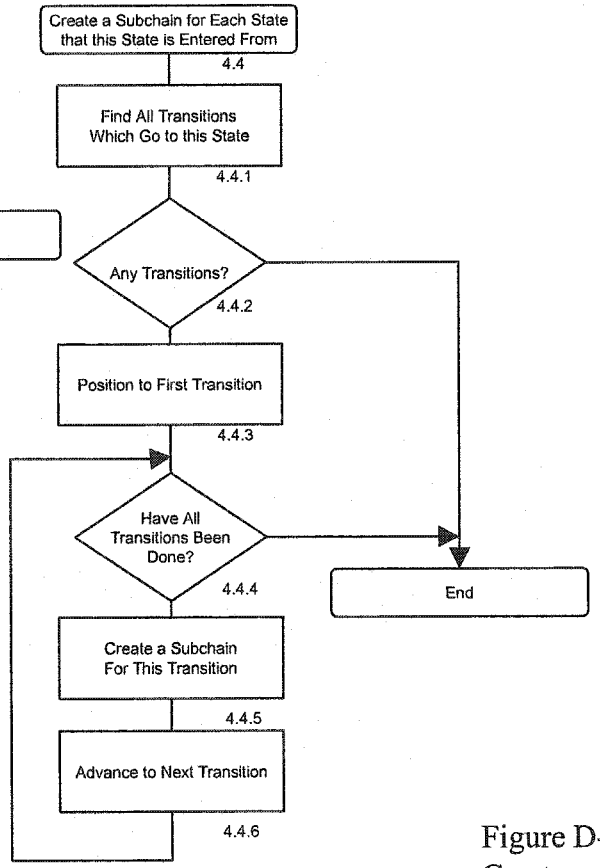


Figure D-5: Create Subchains for Each State that this State is Entered From

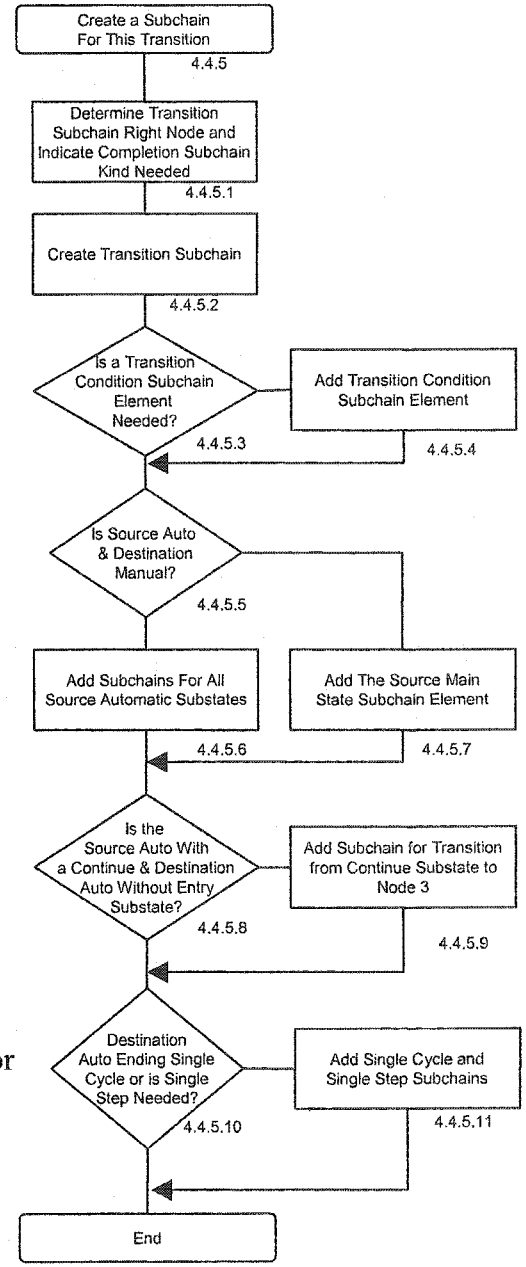


Figure D-6: Create a Subchain for this Transition

Appendix E

Physical Model Illustrations

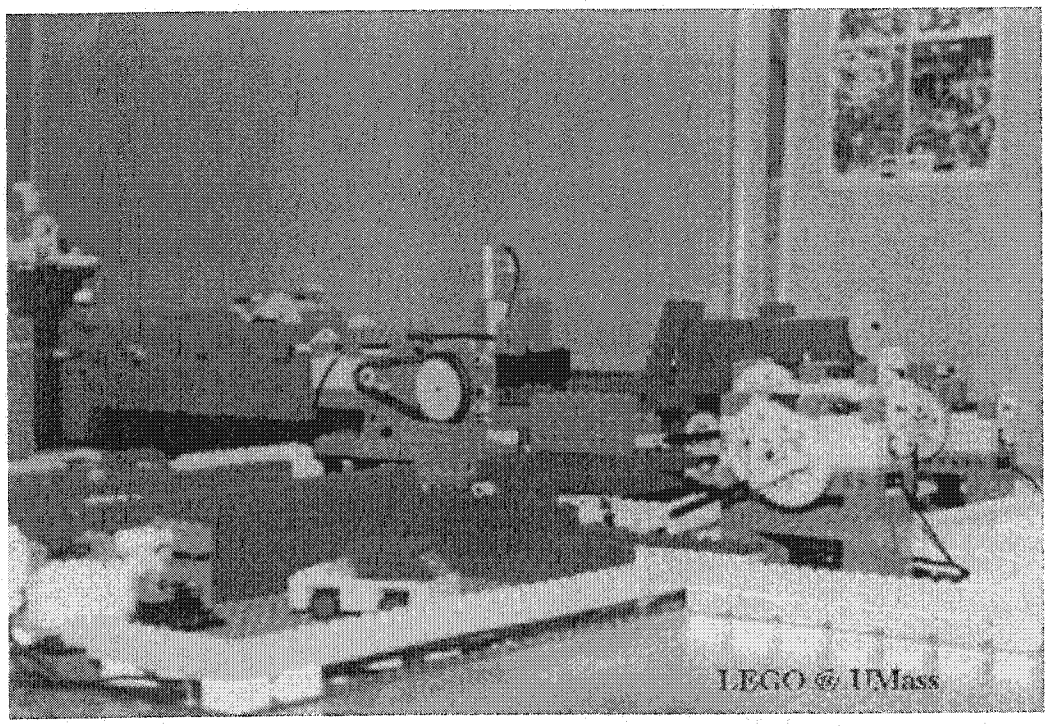


Figure E-1: UMass. Factory Model, Taken From [She96]

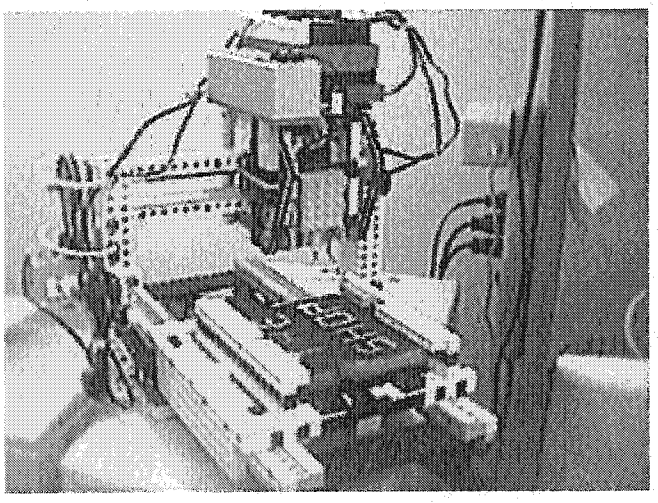


Figure E-2: Functional Gantry Mill, Taken From [Fay01]

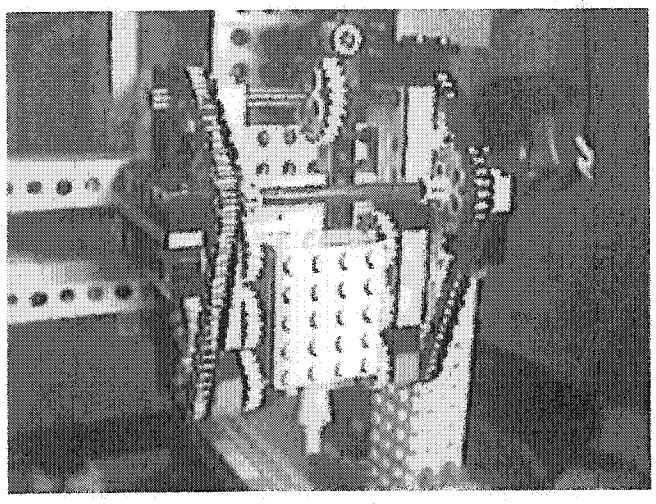


Figure E-3: "Z" Axis and Spindle Motor, Taken From [Fay01]

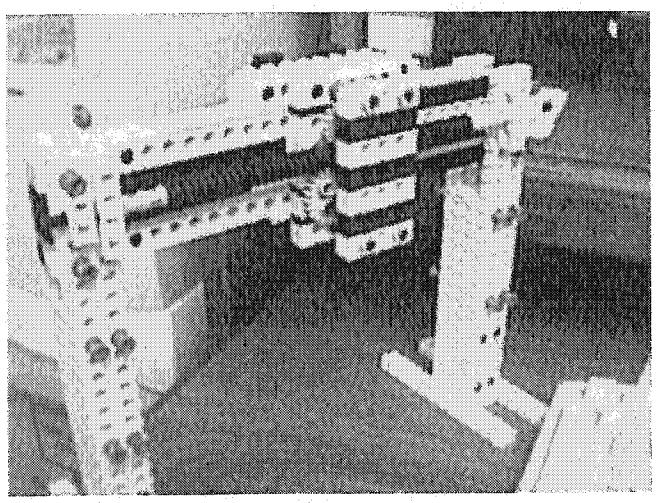


Figure E-4: "Y" Axis Gantry Beam, Taken From [Fay01]

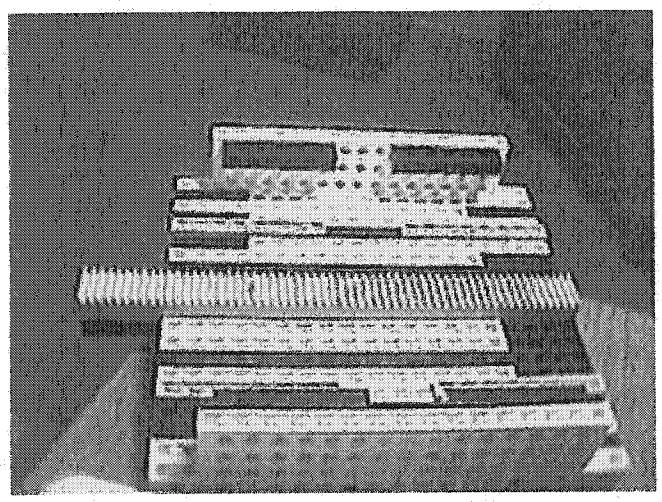


Figure E-5: X Axis Table Bottom View, Taken From [Fay01]

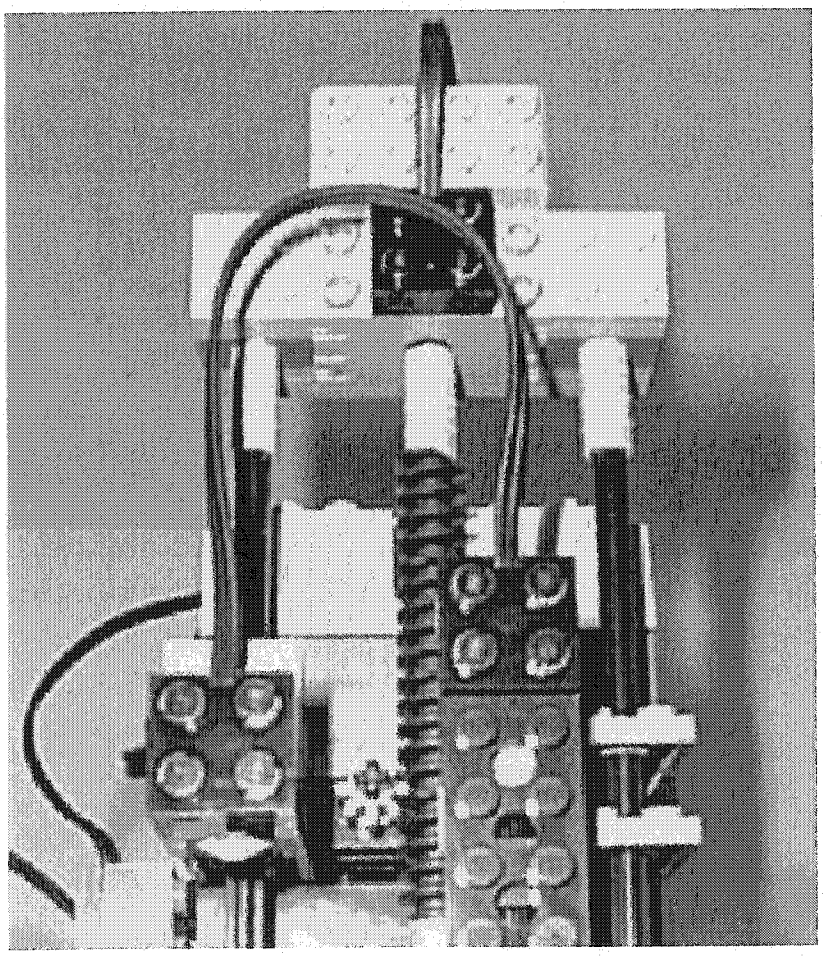


Figure E-6: Reverse Driven Worm Gear, Taken From [Wil01a]

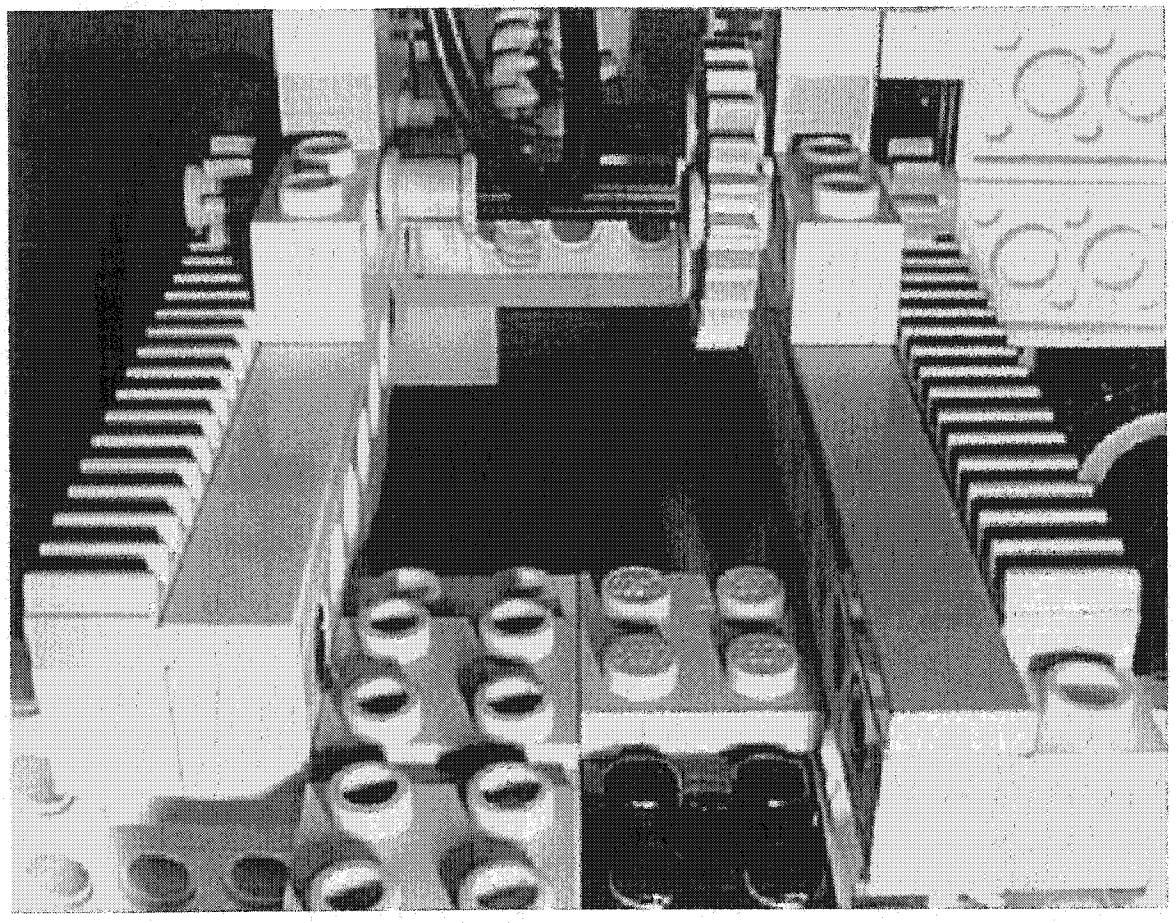


Figure E-7: Rack and Gear Horizontal Axis, Taken From [Wil01b]

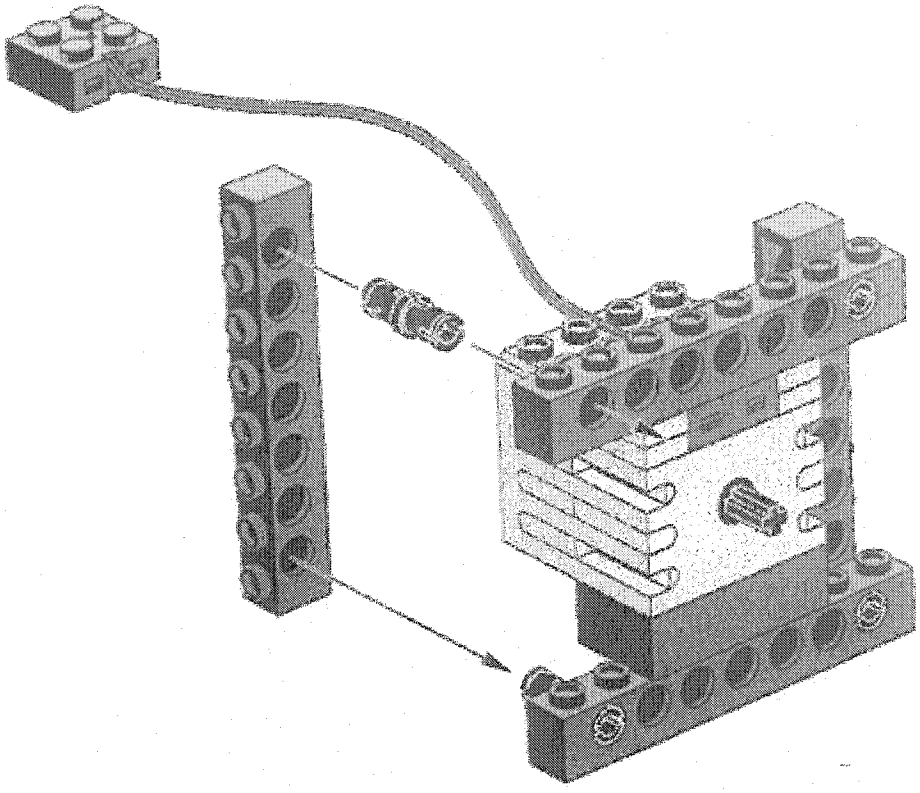


Figure E-8: Pins and Beams Construction, Taken From [LEG00]

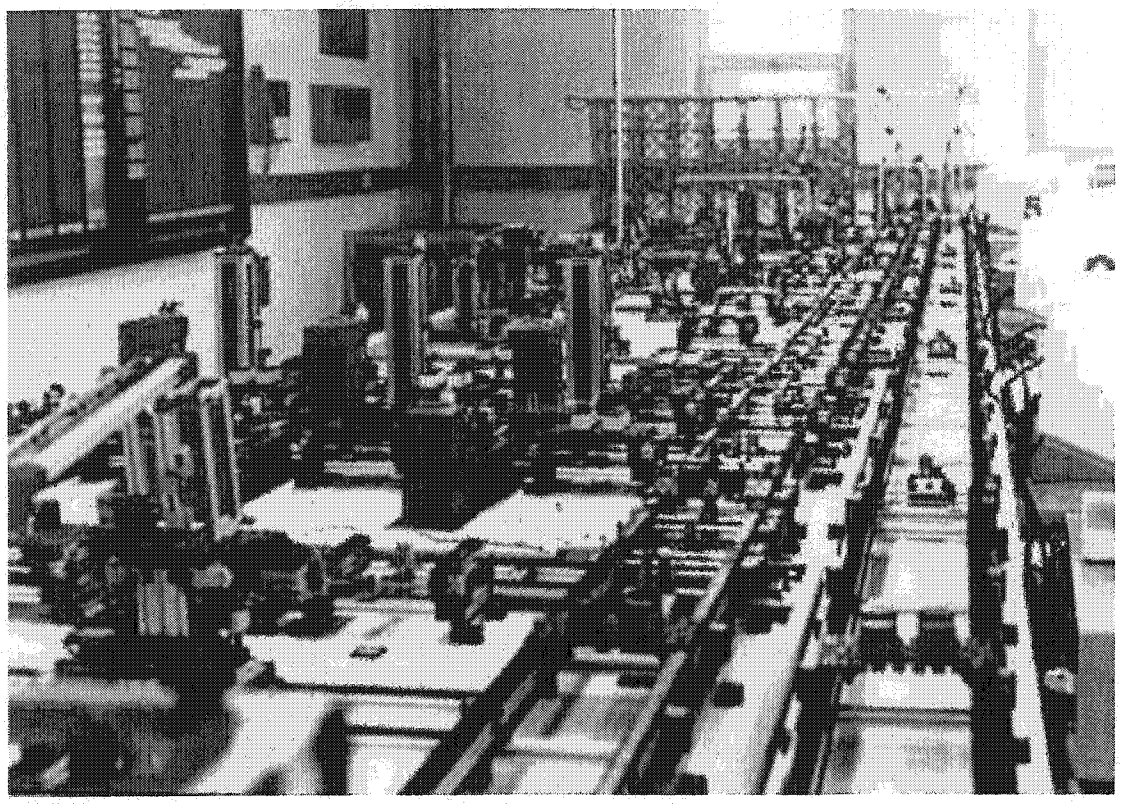


Figure E-9: Complex FischerTechnik Industrial Model, Courtesy Tim King Electronics

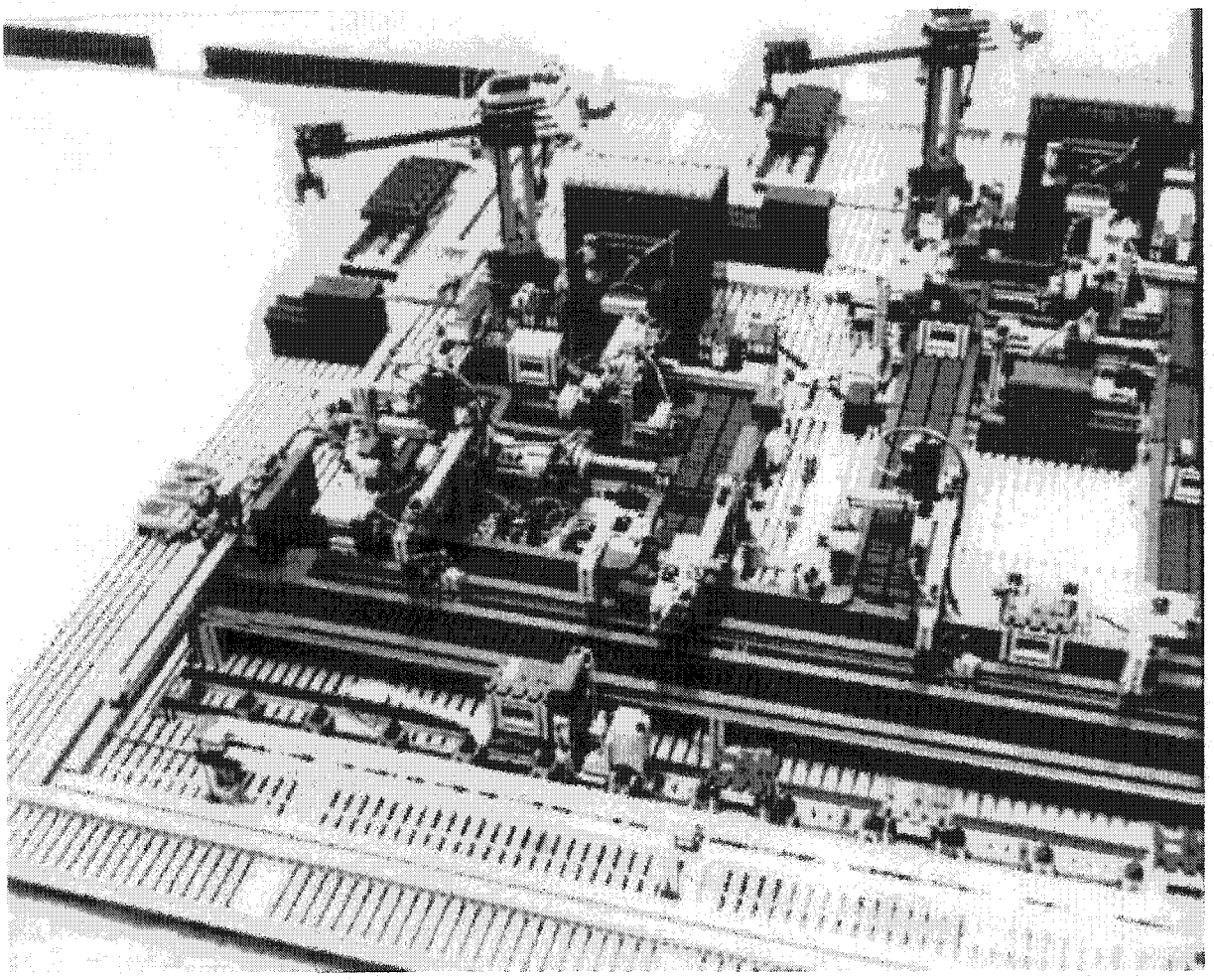


Figure E-10: Complex FischerTechnik Industrial Model, Courtesy Tim King Electronics

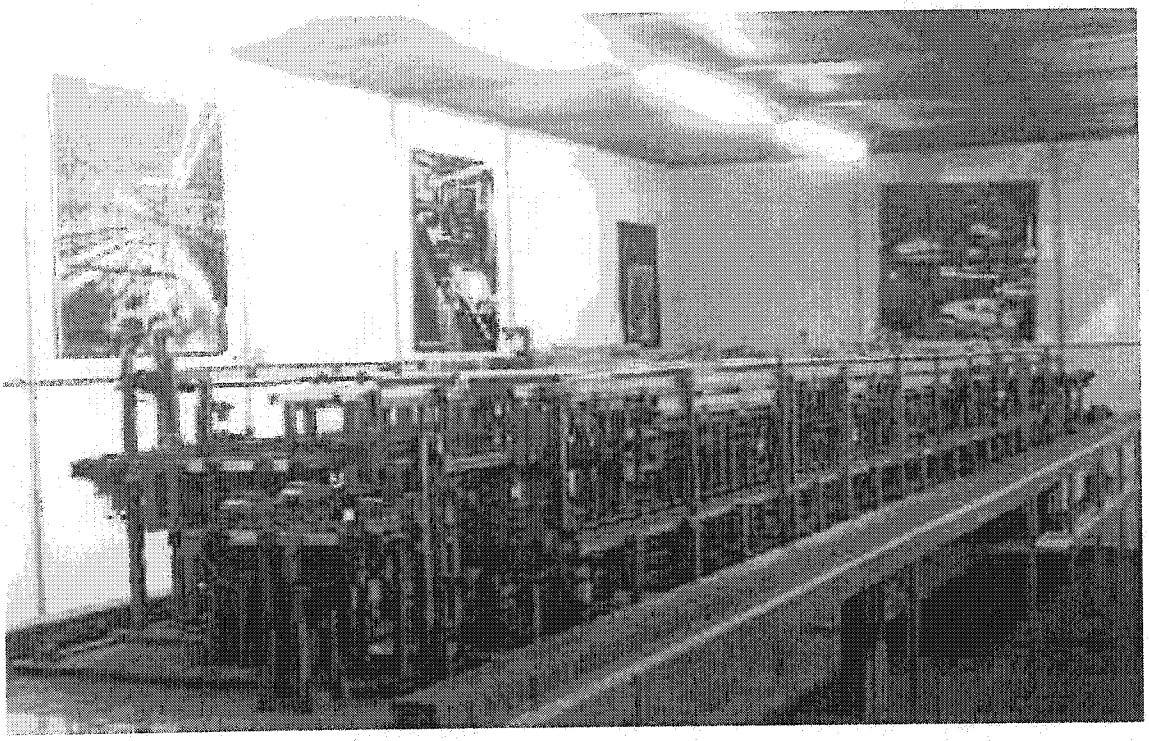


Figure E-11: Complex FischerTechnik Industrial Model, Courtesy Tim King Electronics

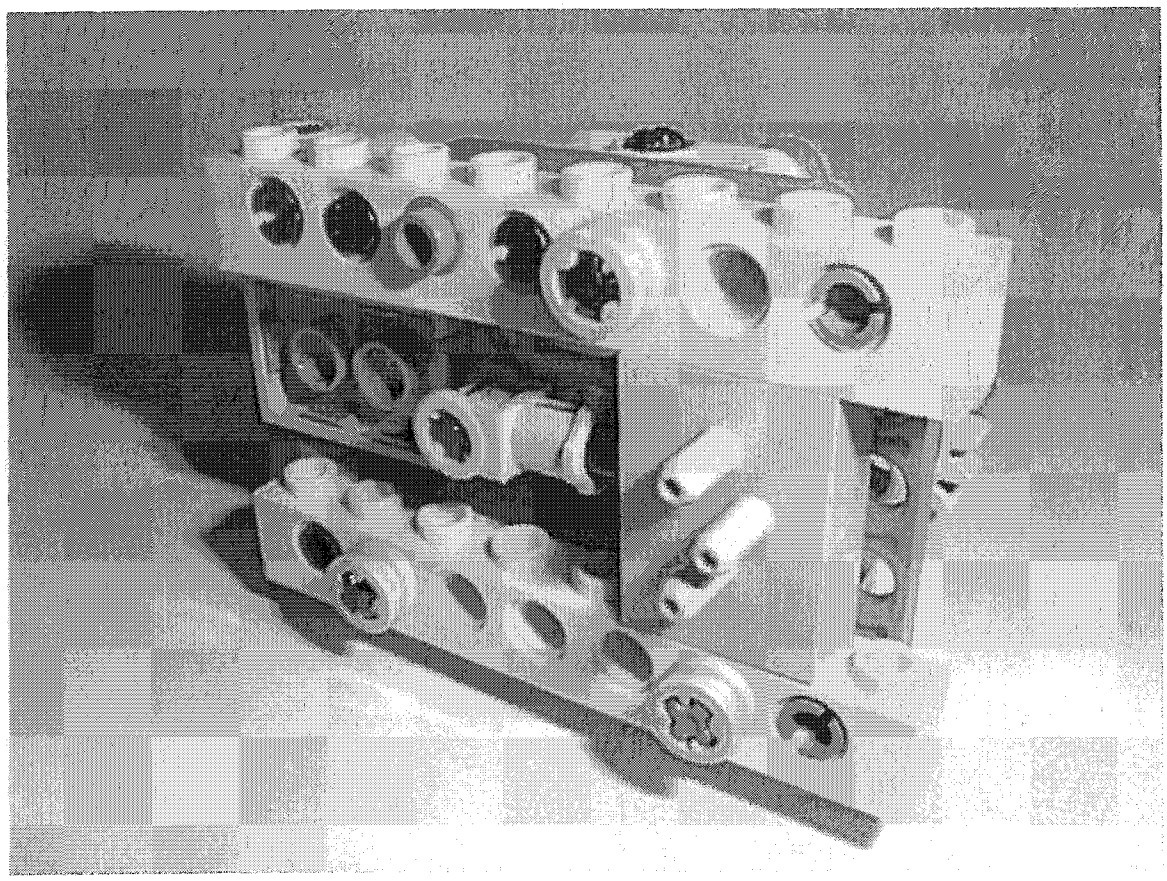


Figure E-12: Motorized LEGO Pneumatic Valve Front View, Taken From [Wil01c]

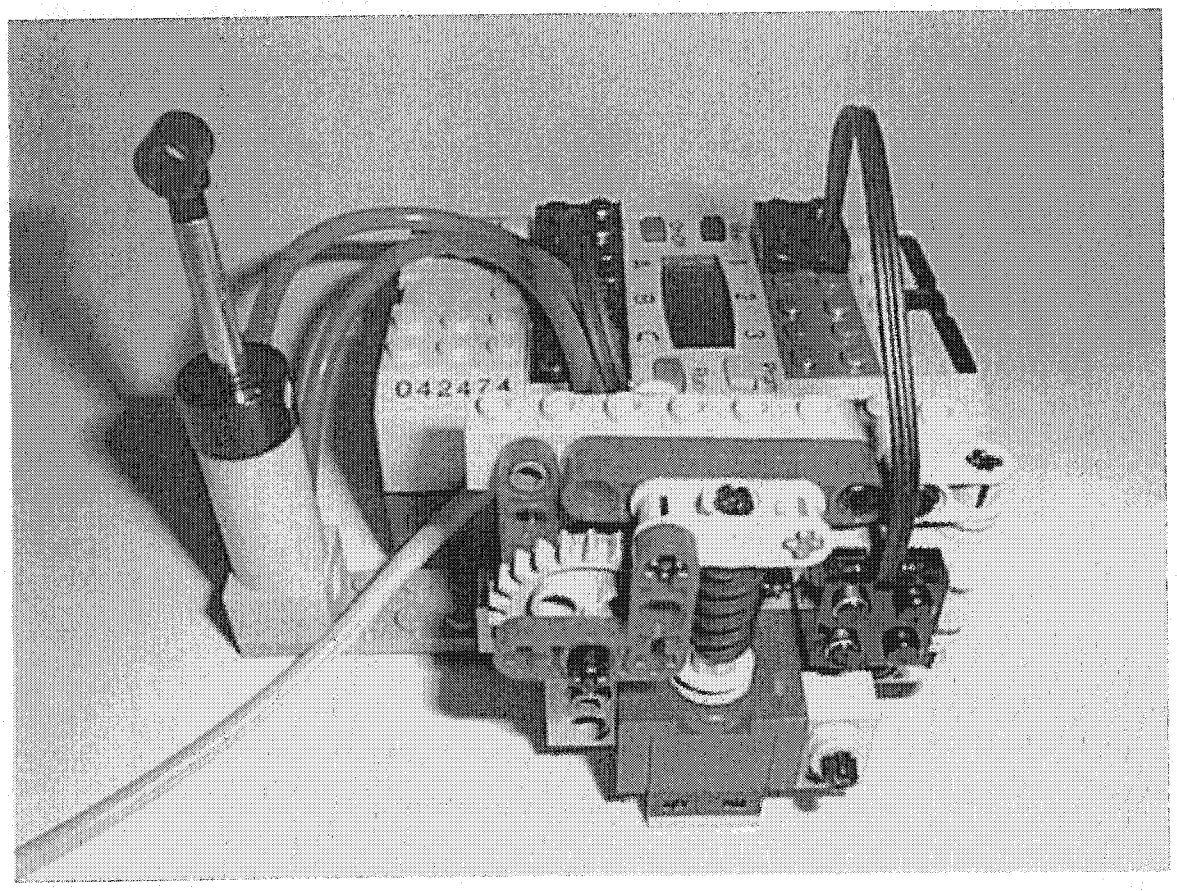


Figure E-13: Motorized LEGO Pneumatic Valve Rear View, Taken From [Wil01c]

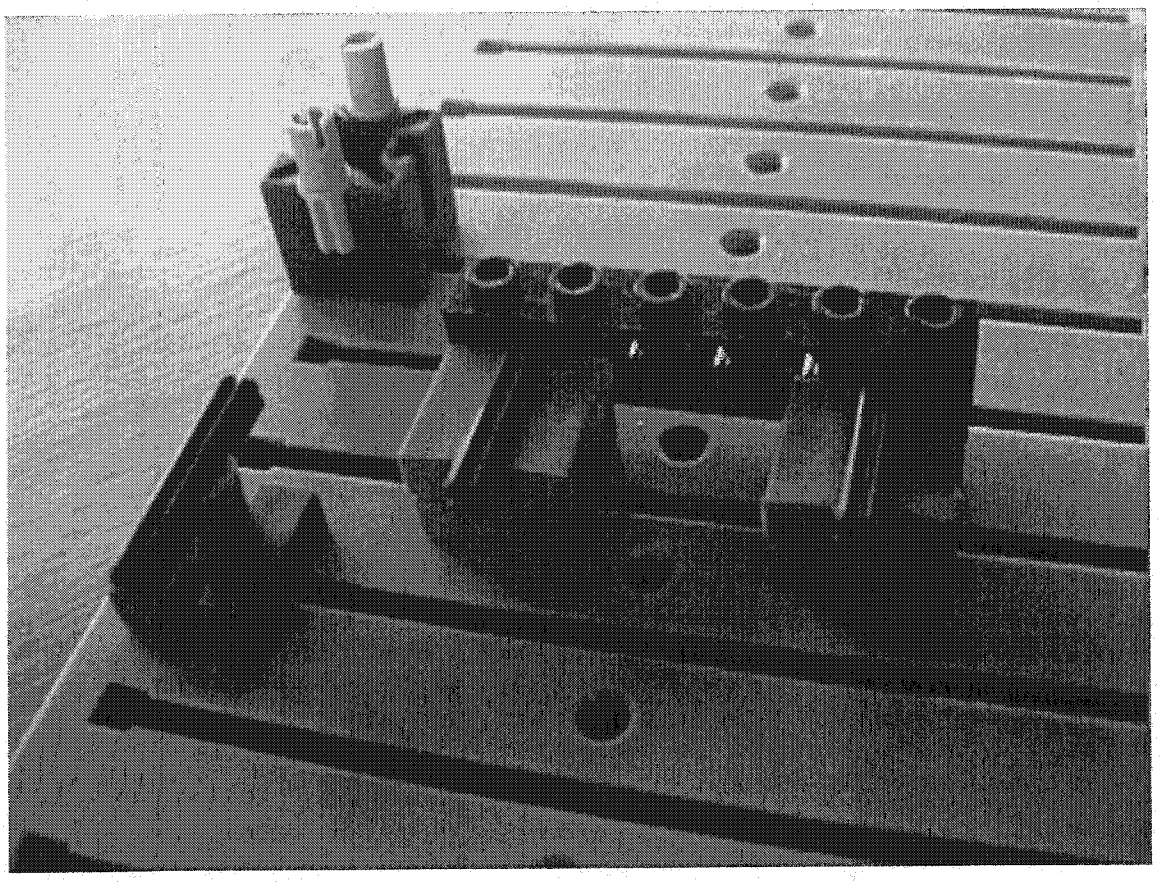


Figure E-14: Integrating LEGO with FischerTechnik

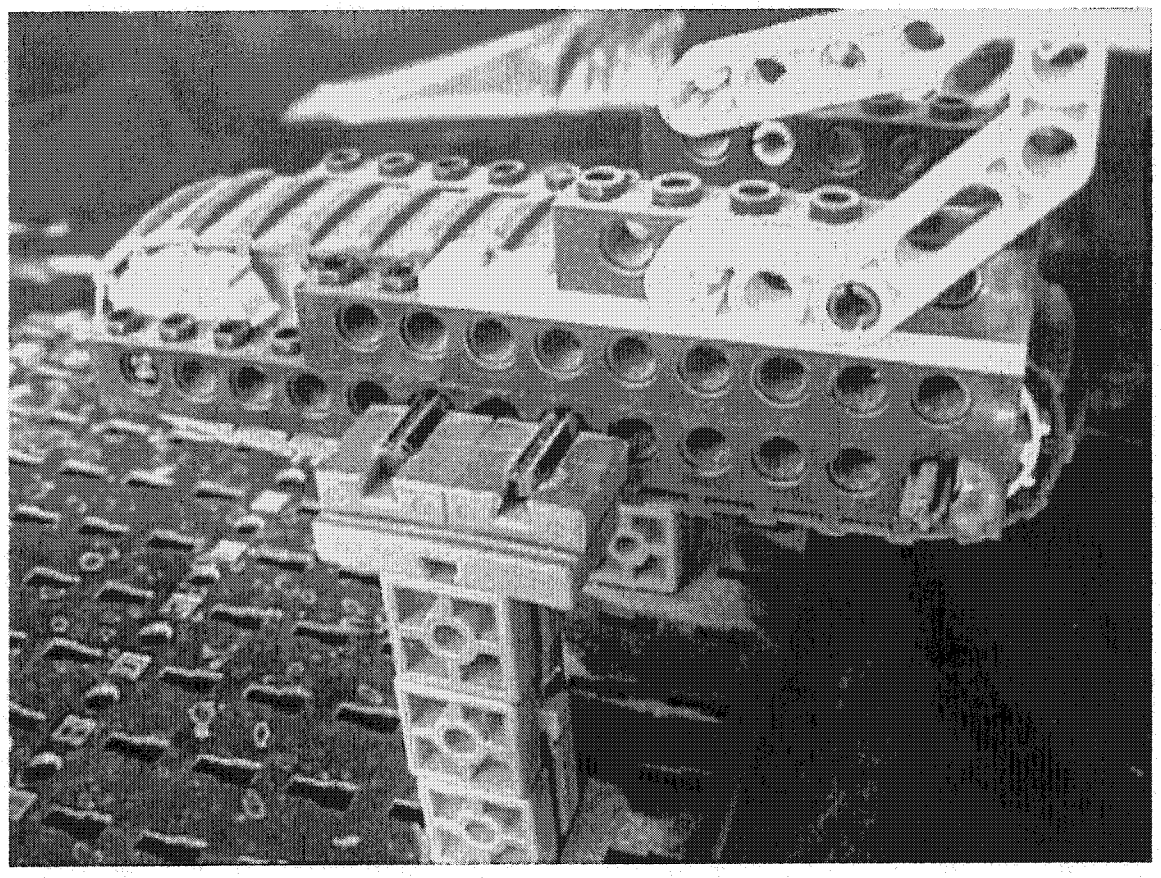


Figure E-15: Integrating LEGO with FischerTechnik

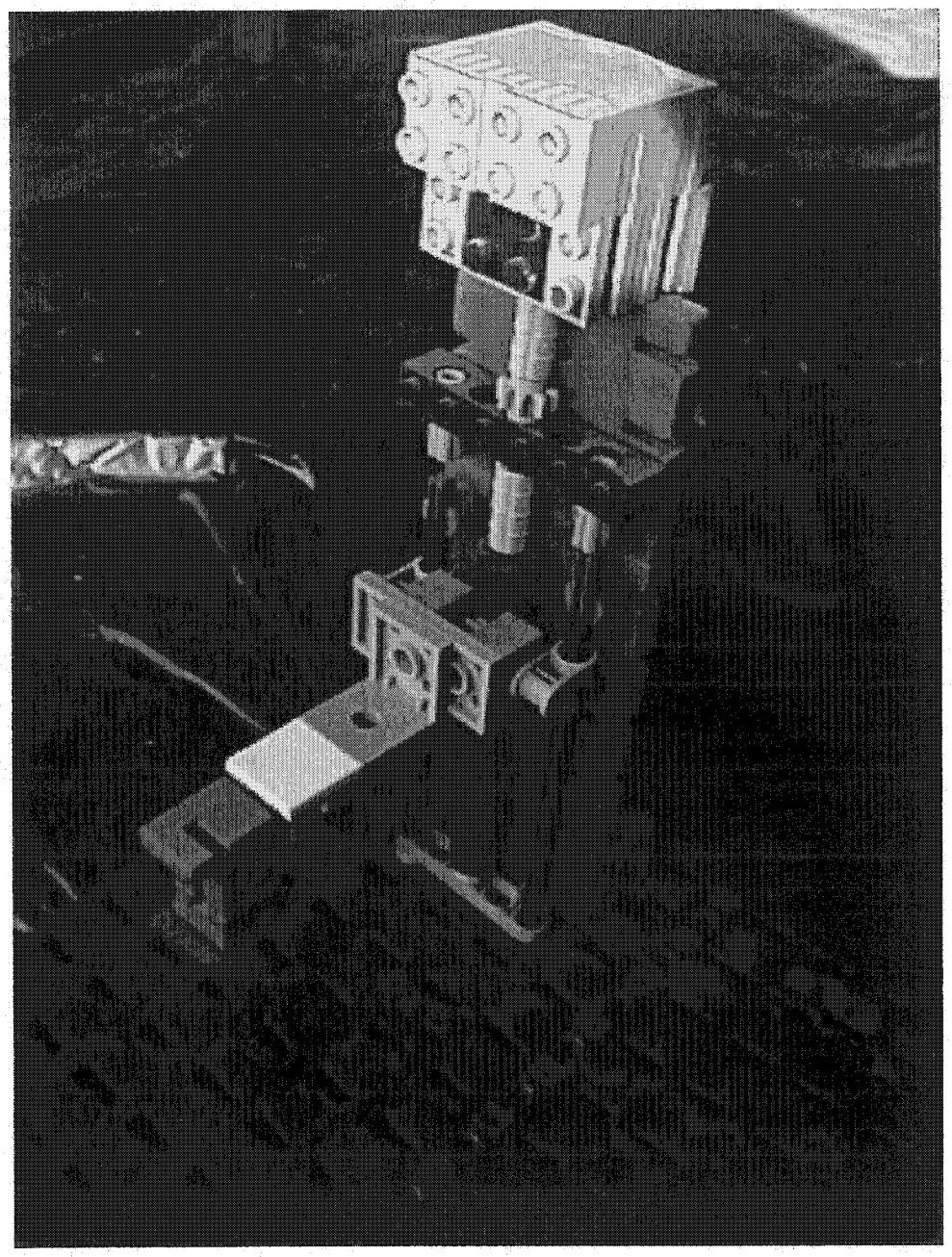


Figure E-16: Integrating LEGO with FischerTechnik

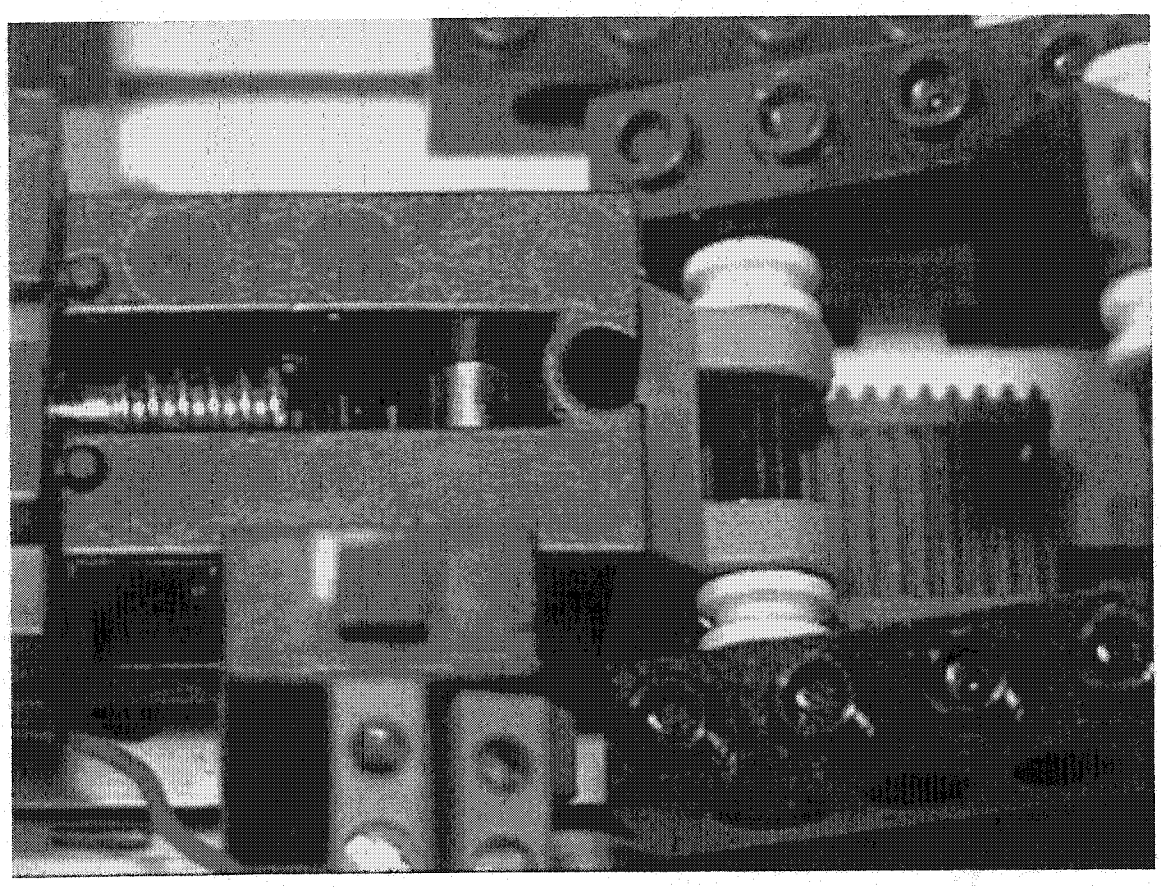


Figure E-17: Integrating LEGO with FischerTechnik

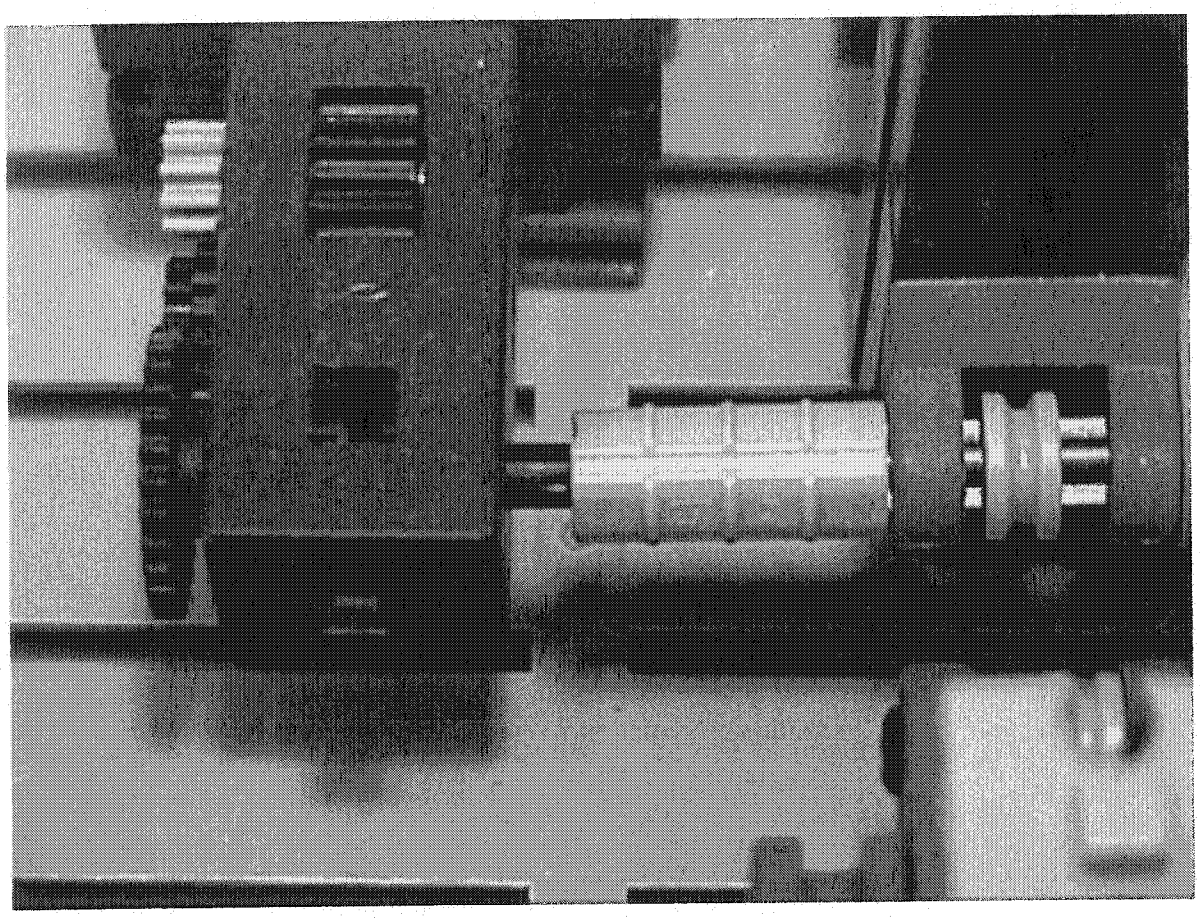


Figure E-18: Integrating LEGO with FischerTechnik

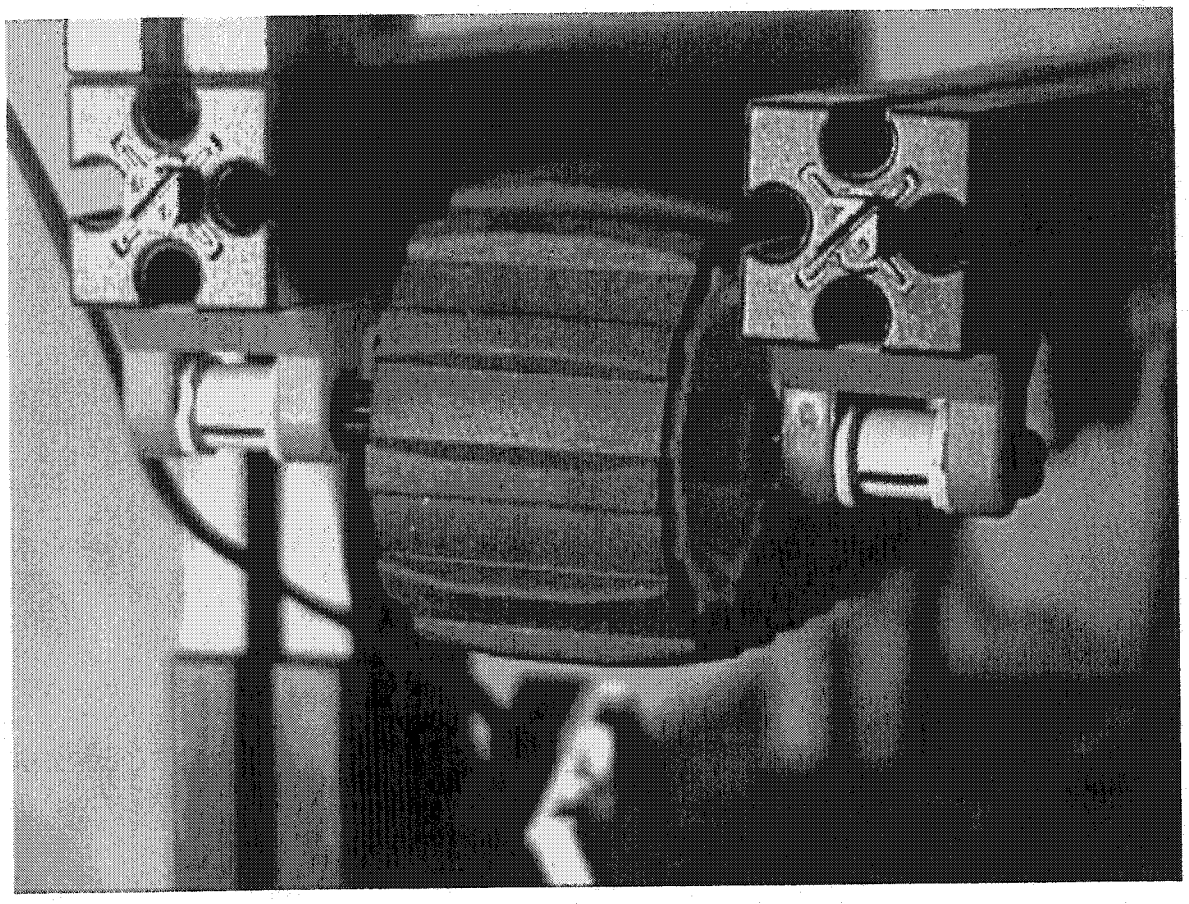


Figure E-19: Integrating LEGO with FischerTechnik

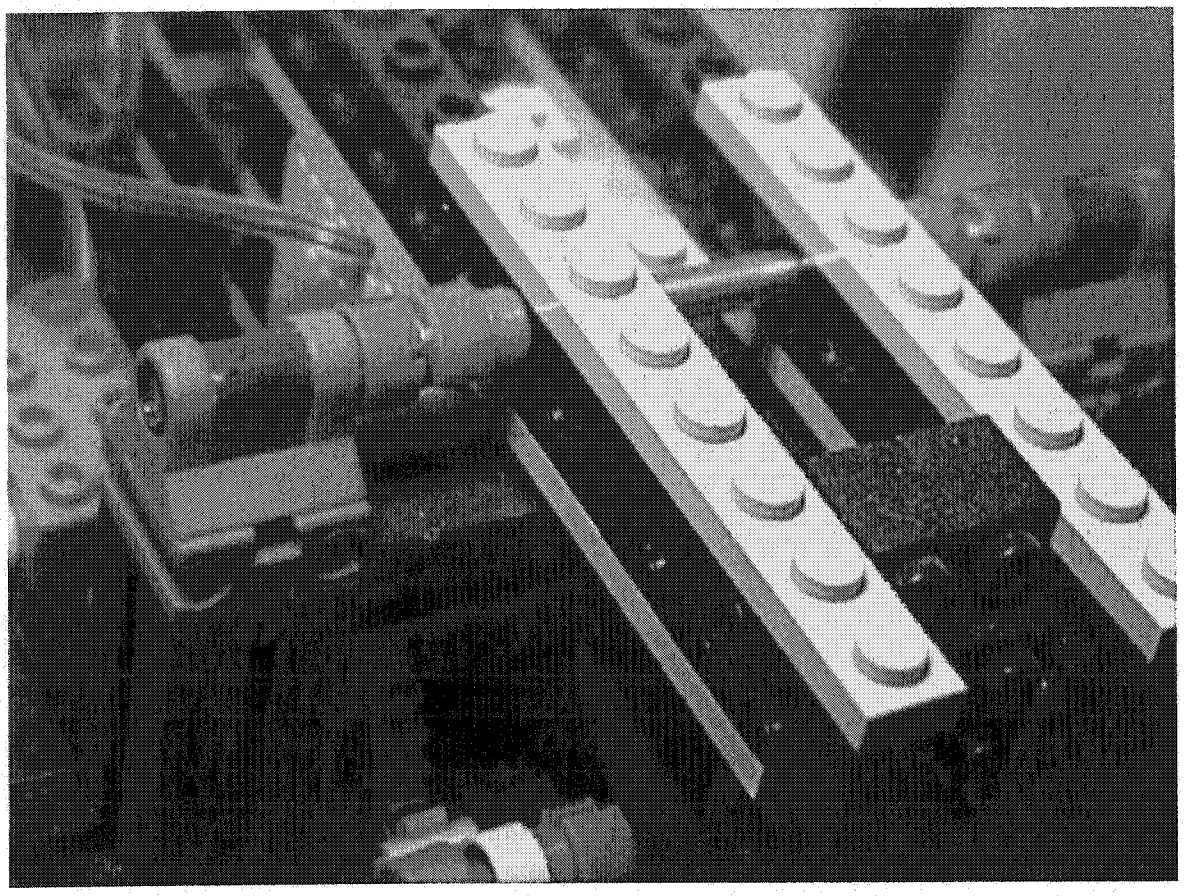


Figure E-20: Integrating LEGO with FischerTechnik

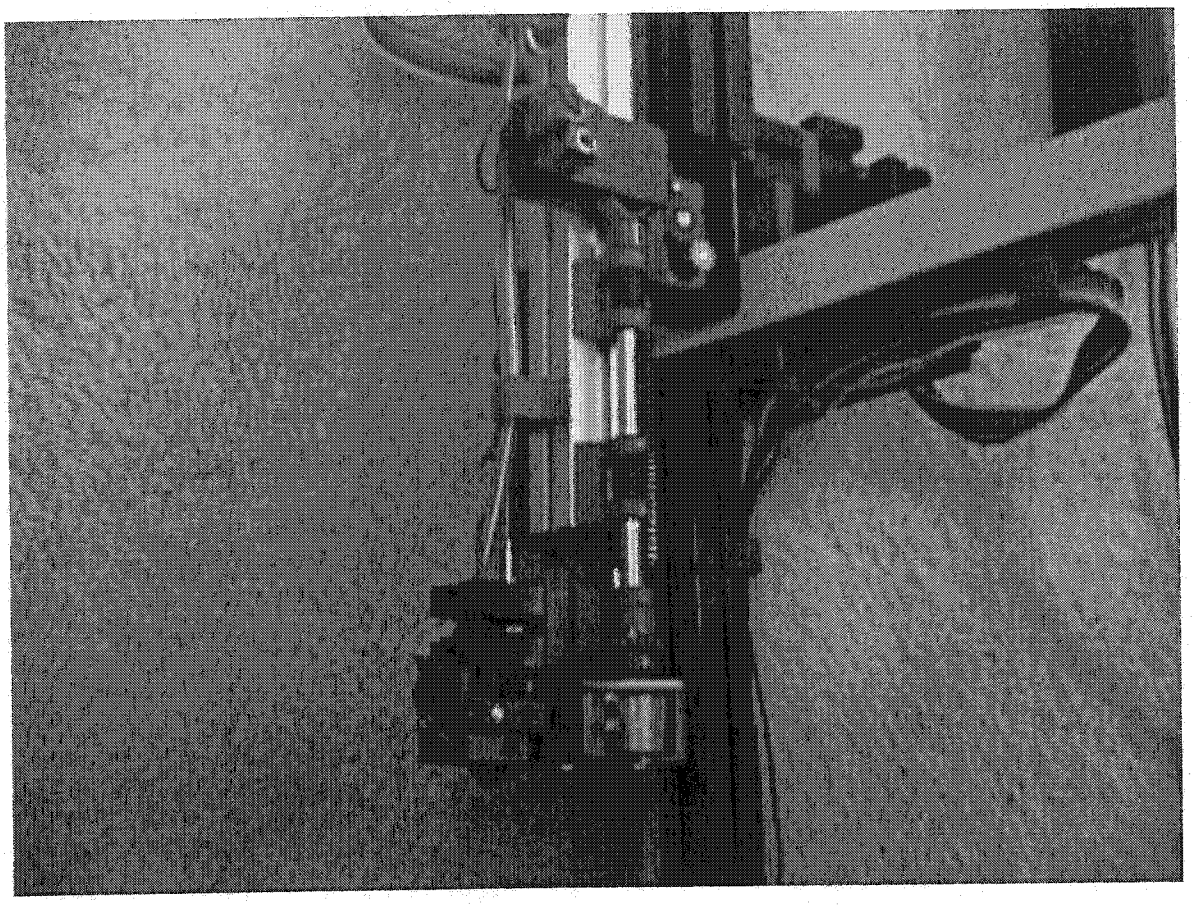


Figure E-21: Gripper, at Top of Stack and Part Present Switch

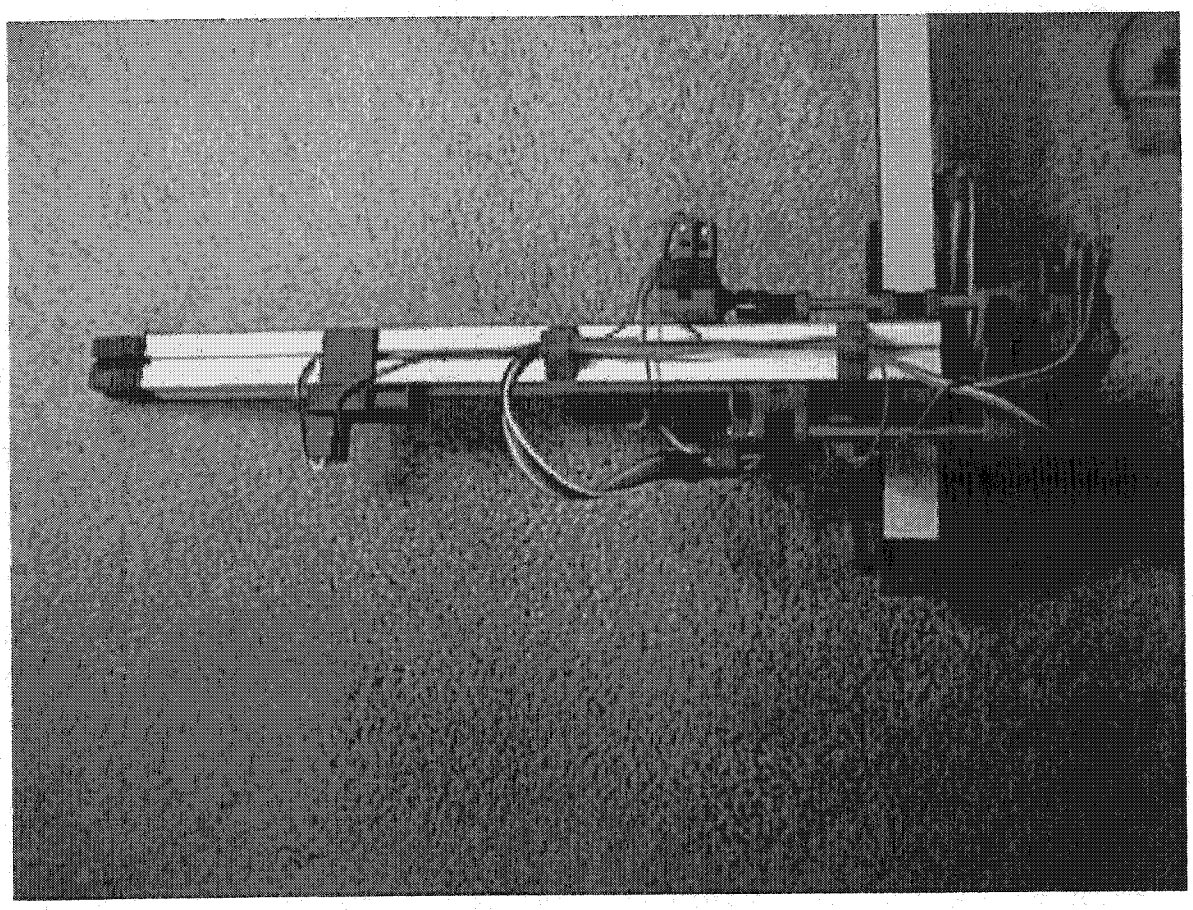


Figure E-22: Aluminium Bar Destacker Arm

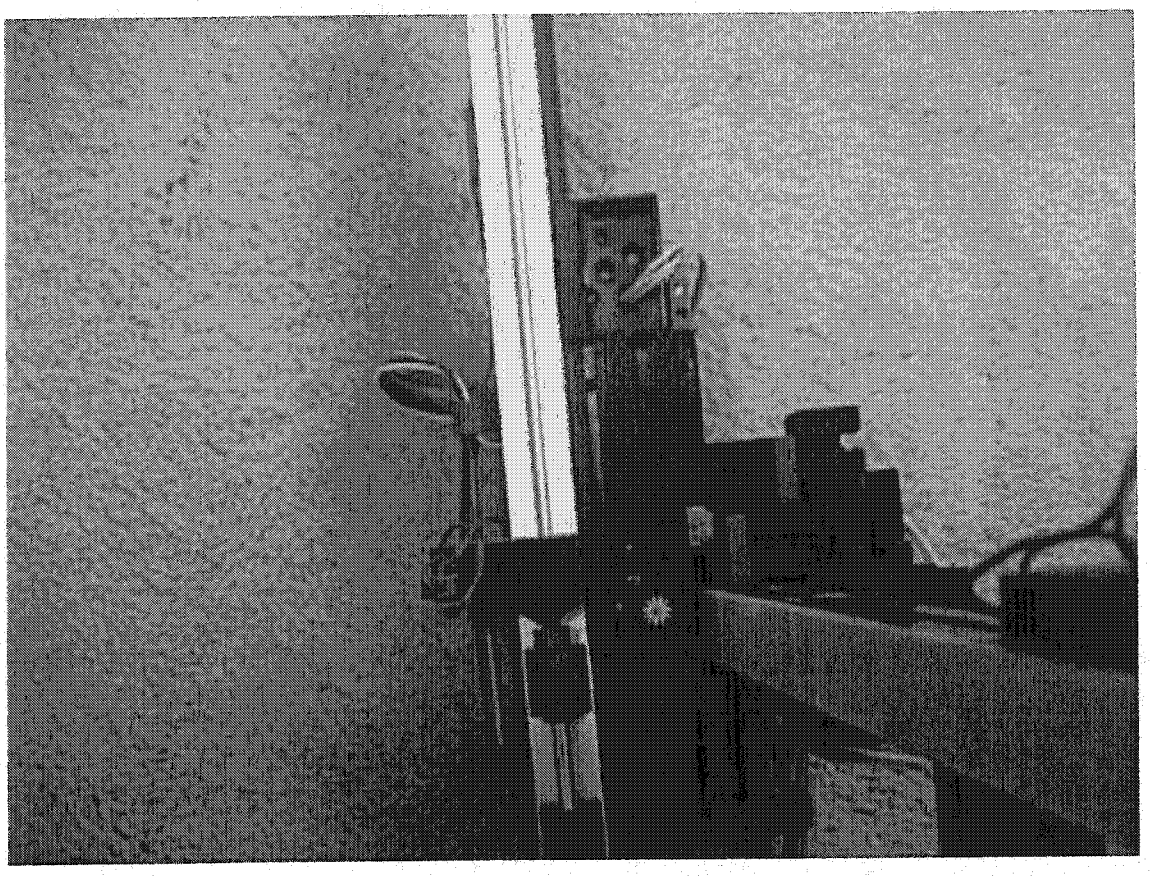


Figure E-23: Vertical Drive Assembly

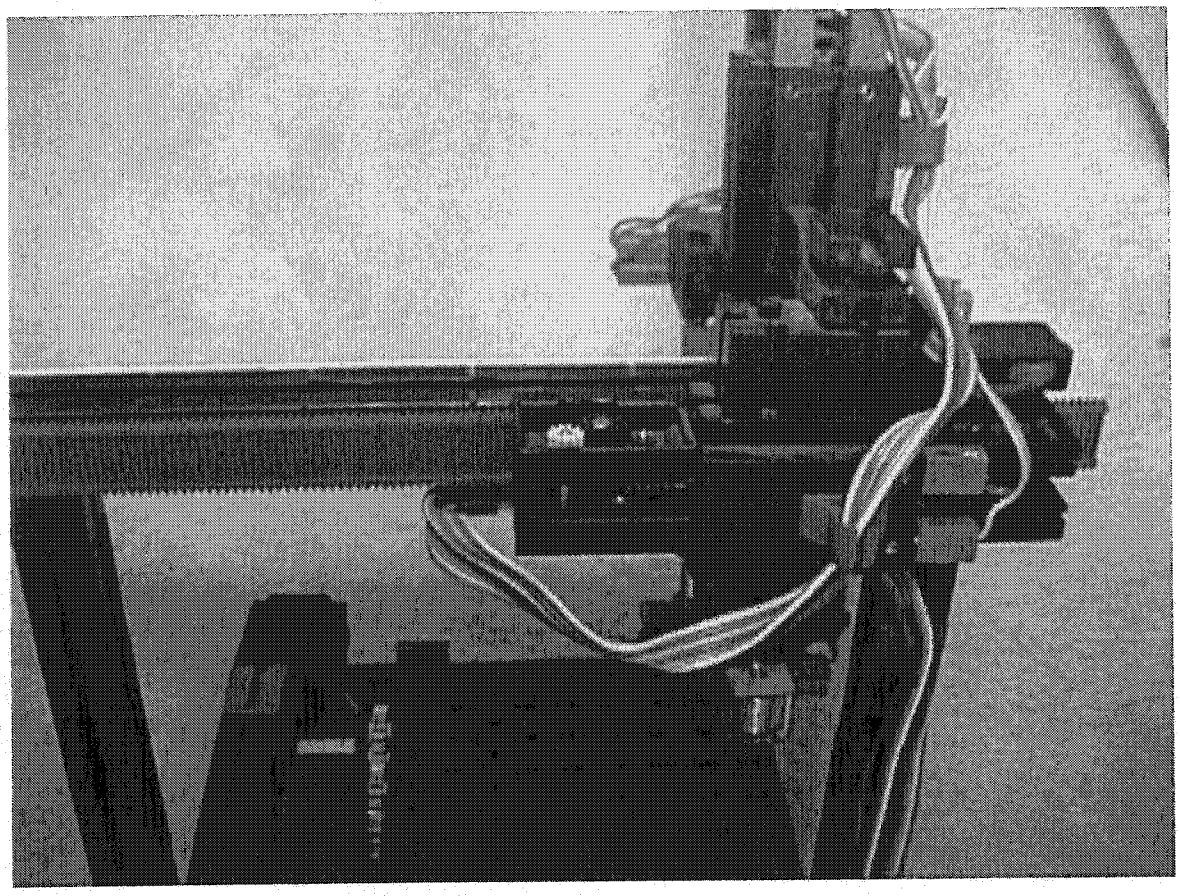


Figure E-24: Horizontal Drive Assembly

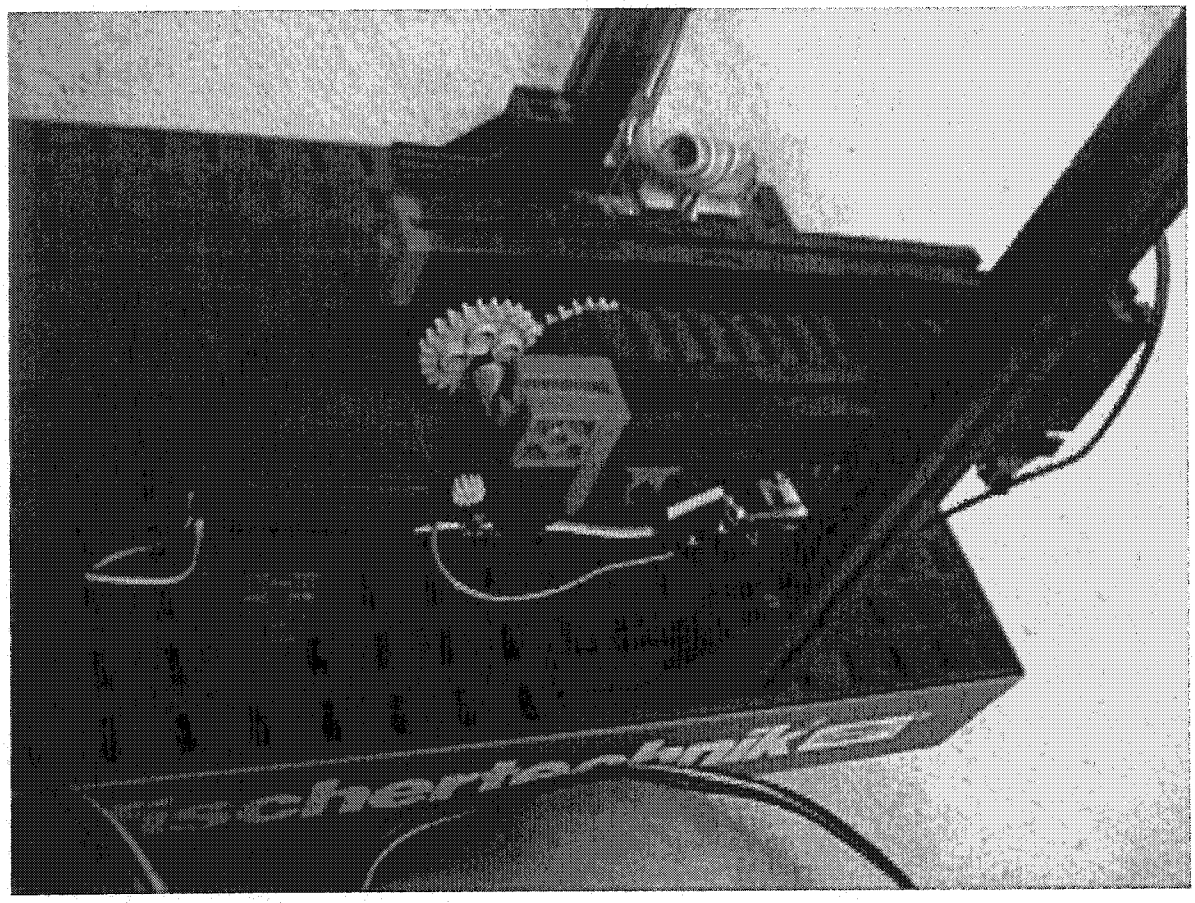


Figure E-25: Press Entry Conveyor

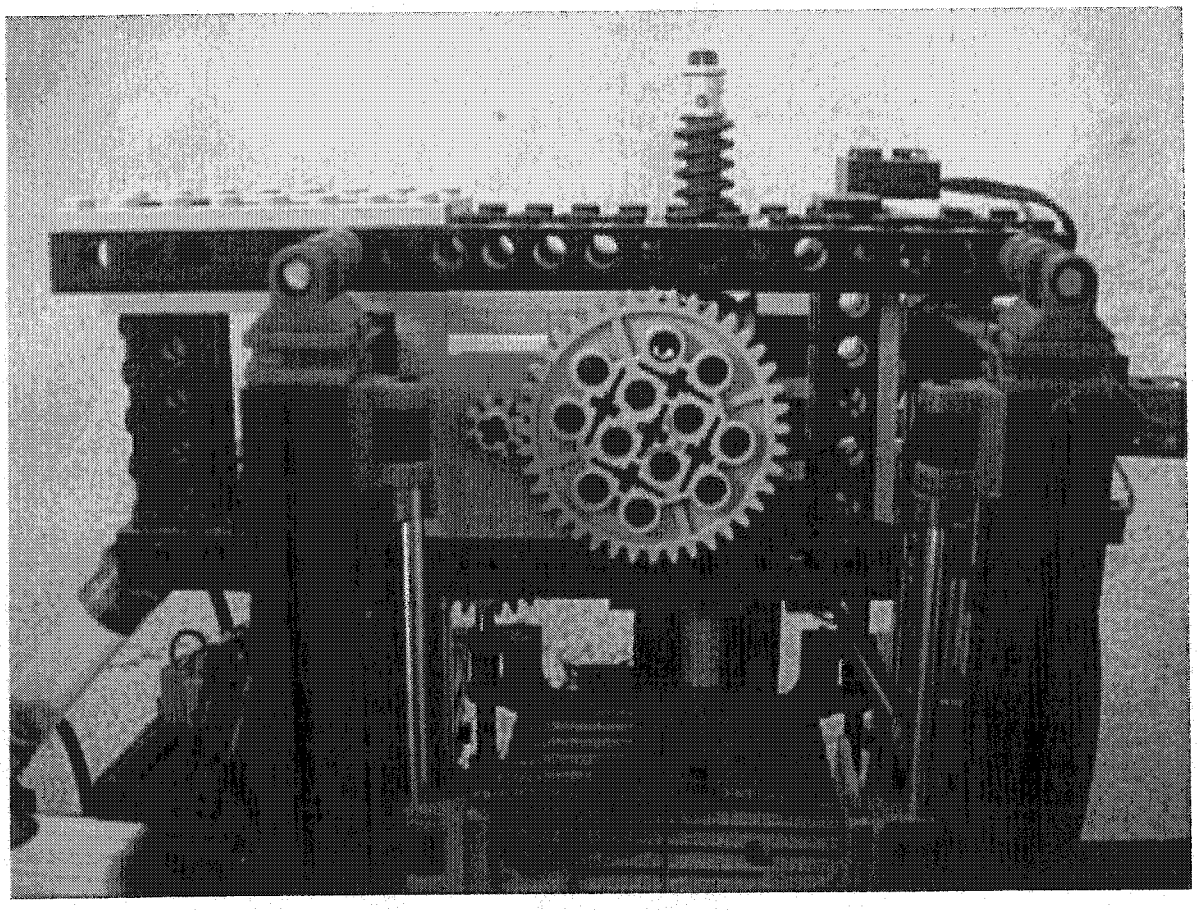


Figure E-26: Press Vertical Drive

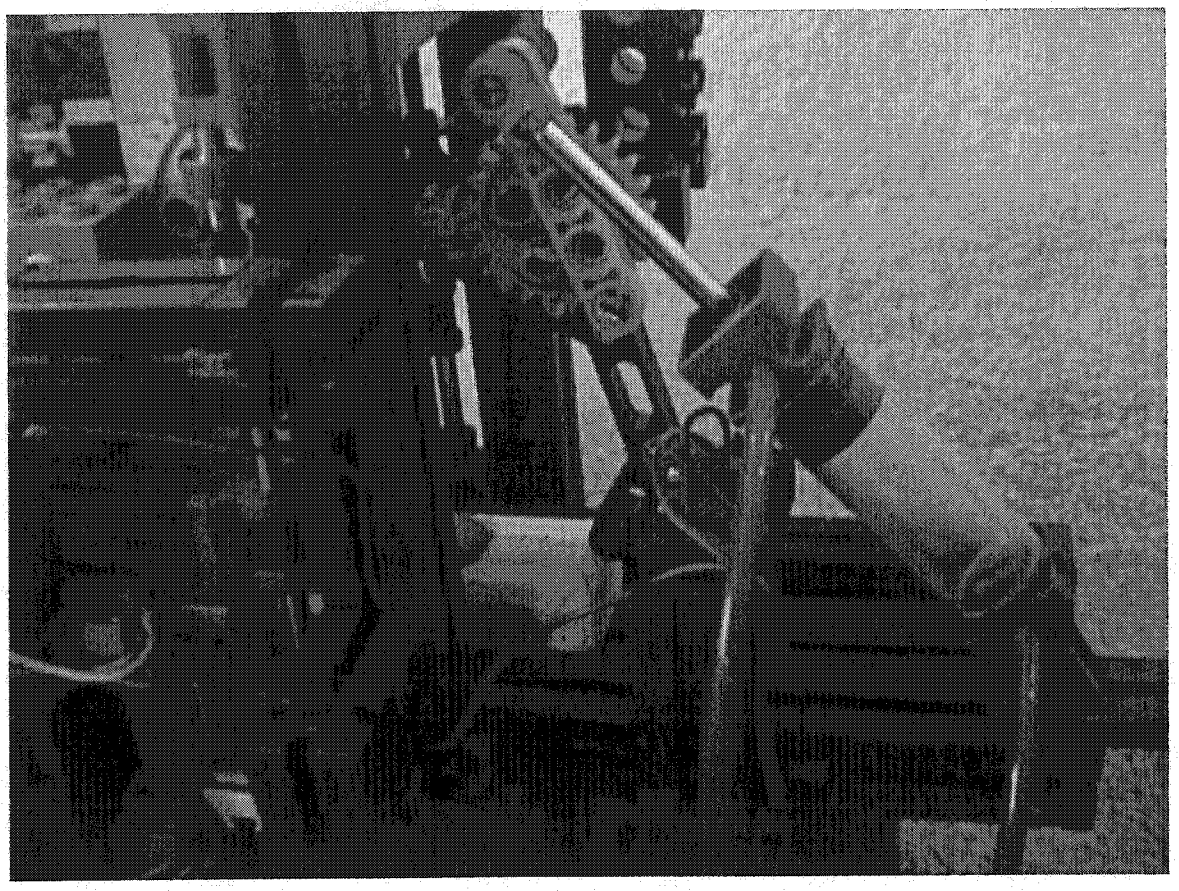


Figure E-27: Press Part Clamp and Ejector

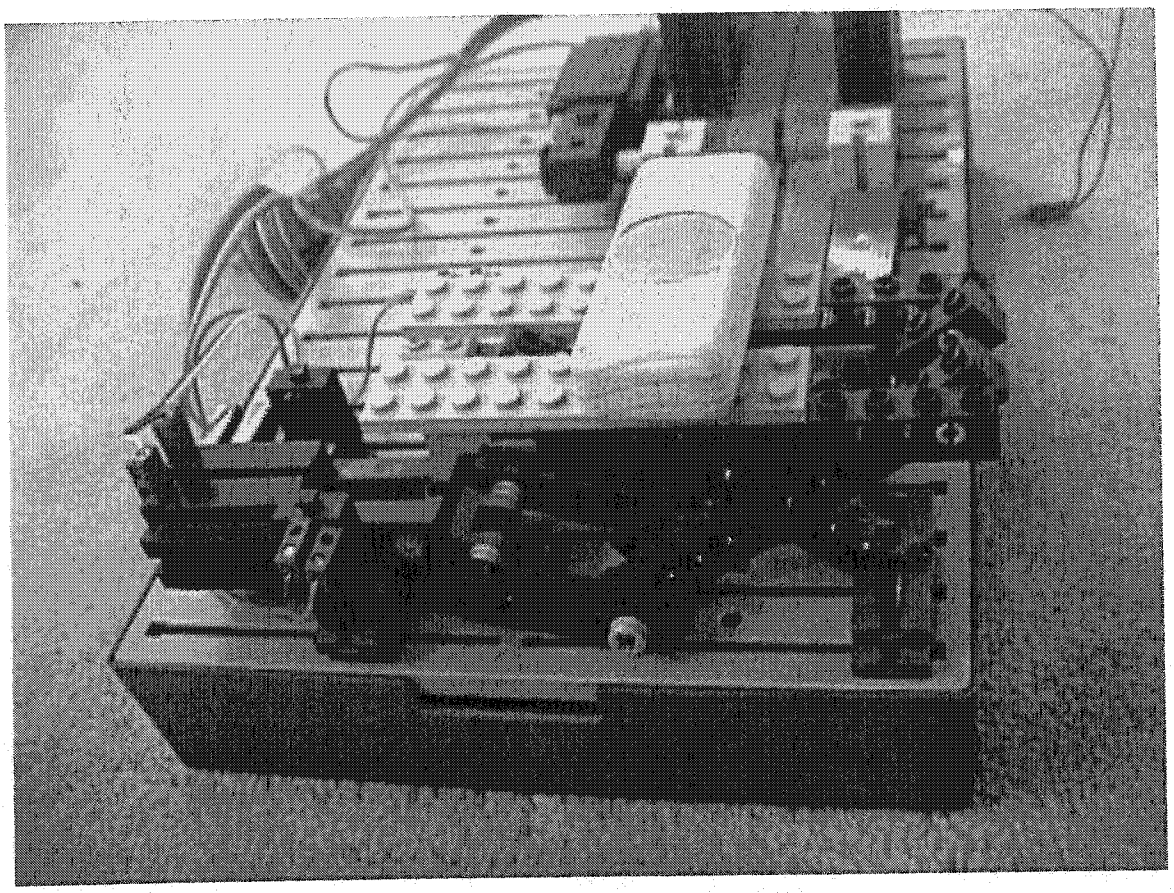


Figure E-28: Scissors Lift

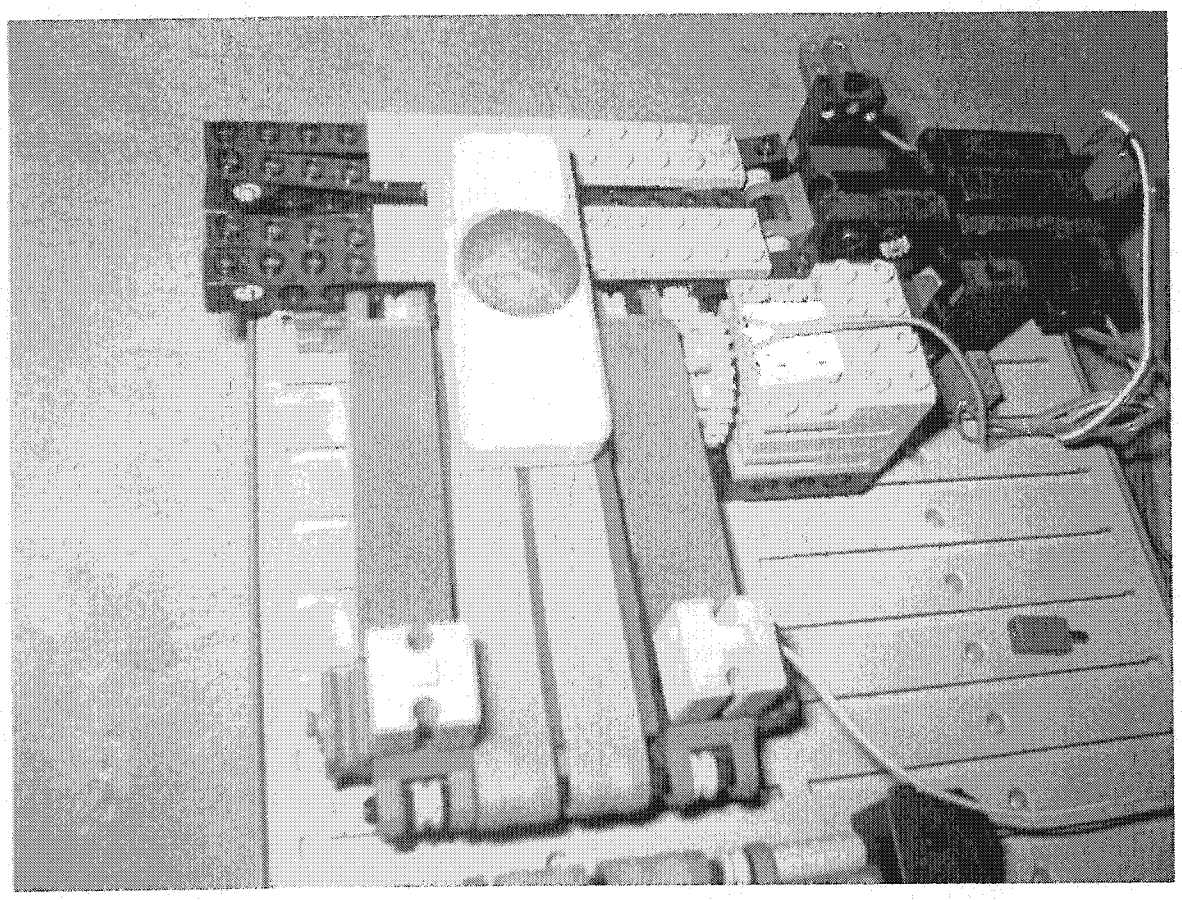


Figure E-29: Roll-Former Entry Conveyor

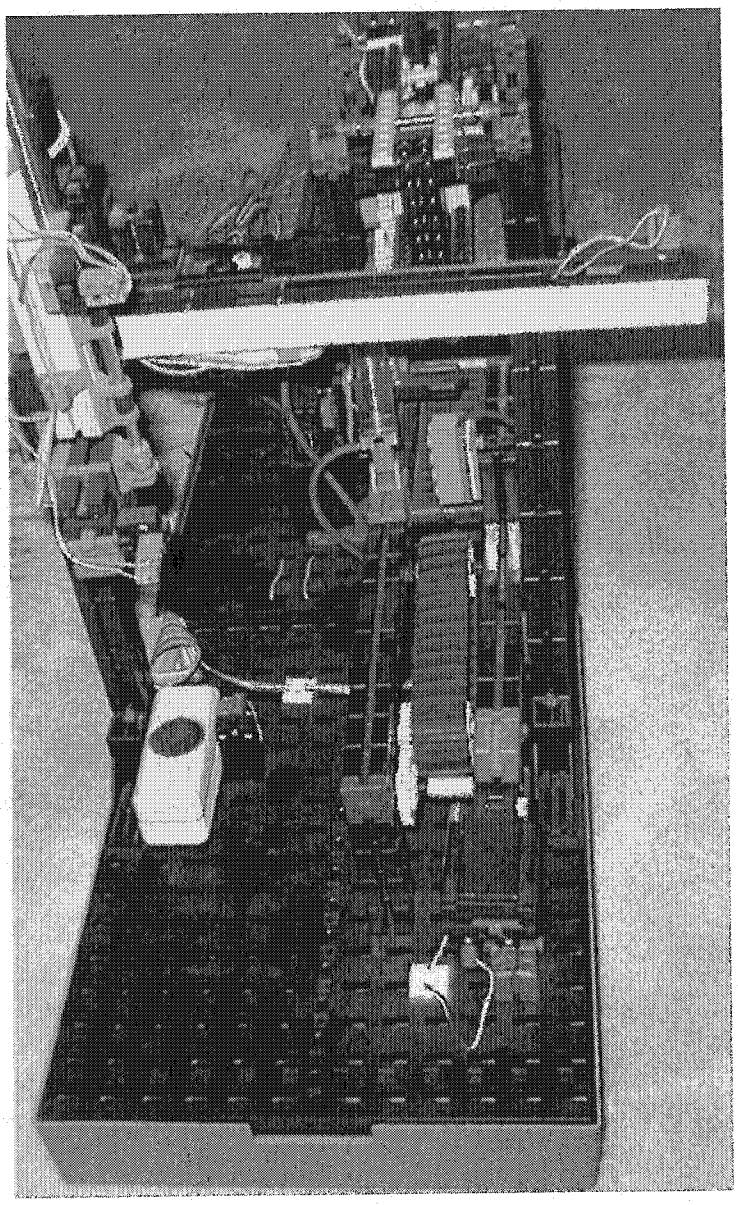


Figure E-30: Wrapper Line Part Stack

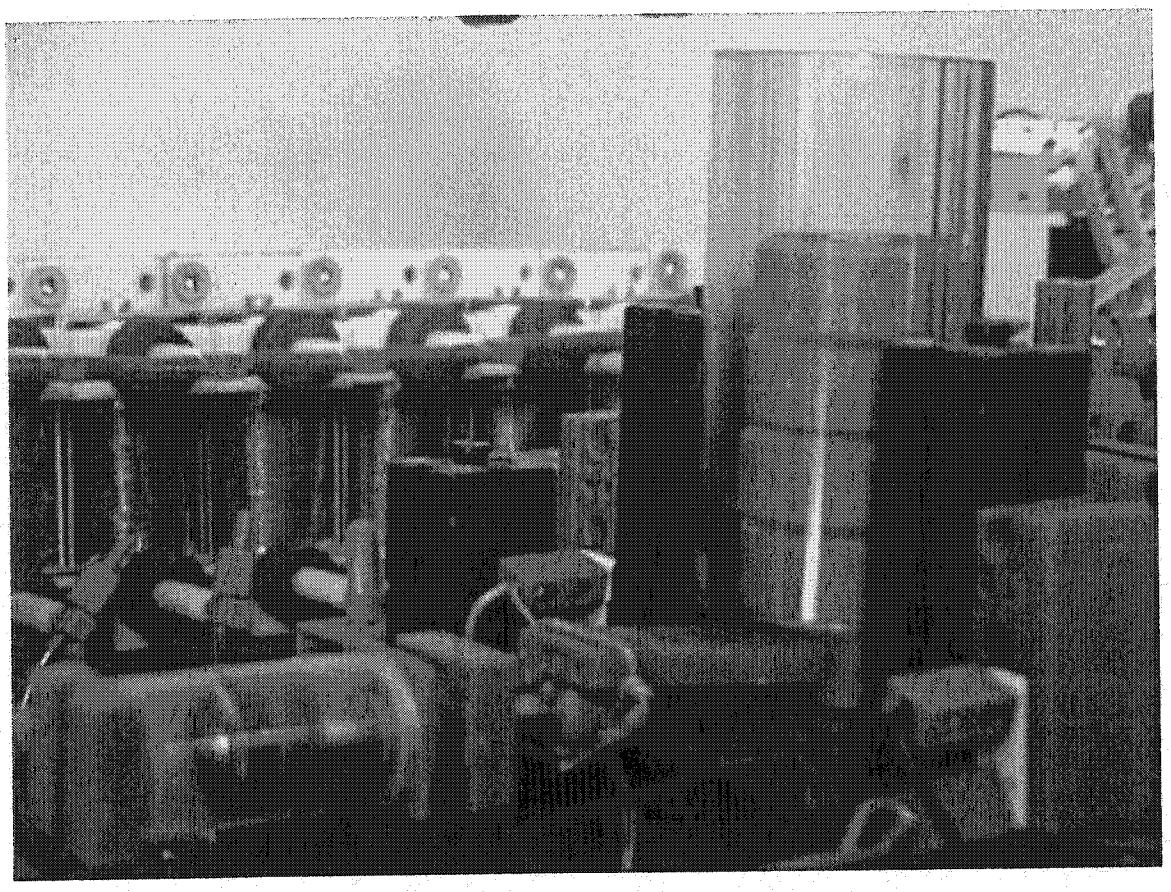


Figure E-31: Trainer Model Part Magazine and Feeder

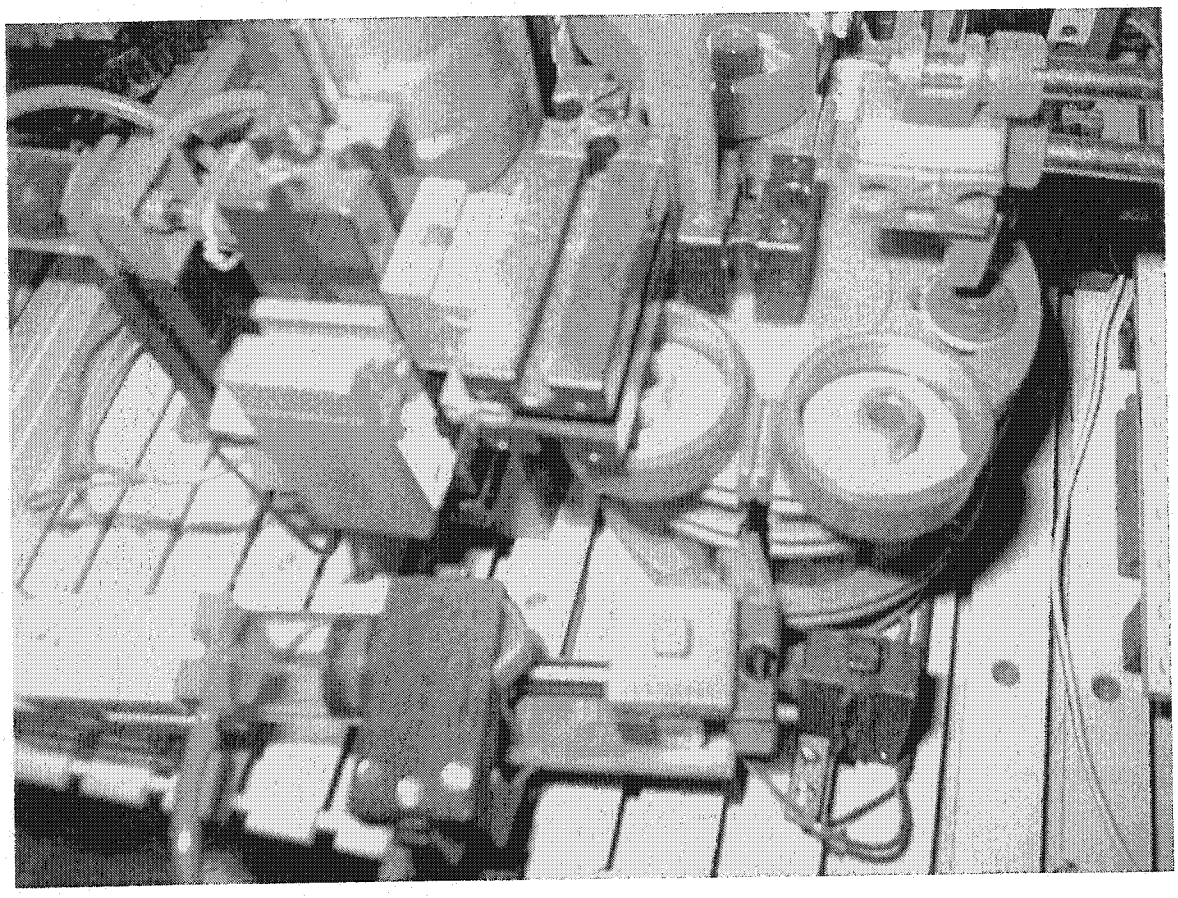


Figure E-32: Pneumatically Actuated Indexing Table

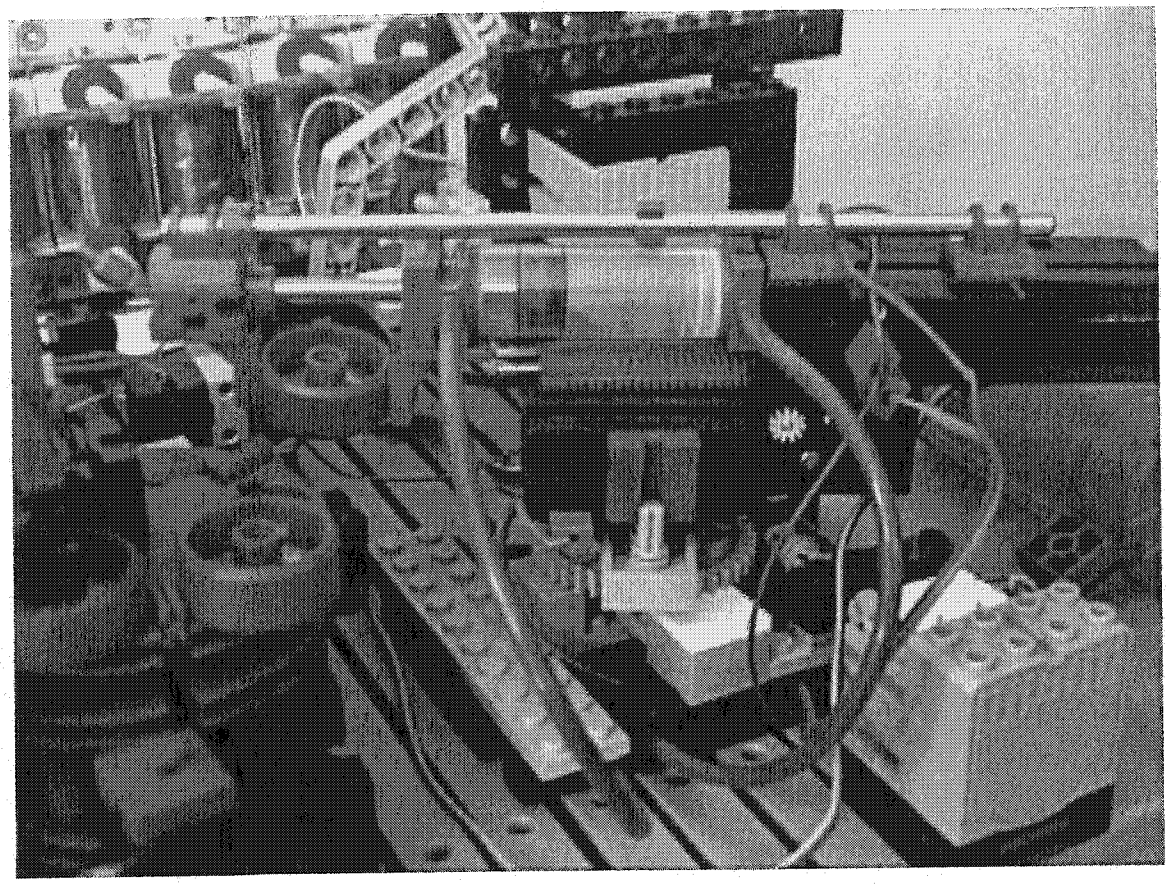


Figure E-33: Pick and Place Robot

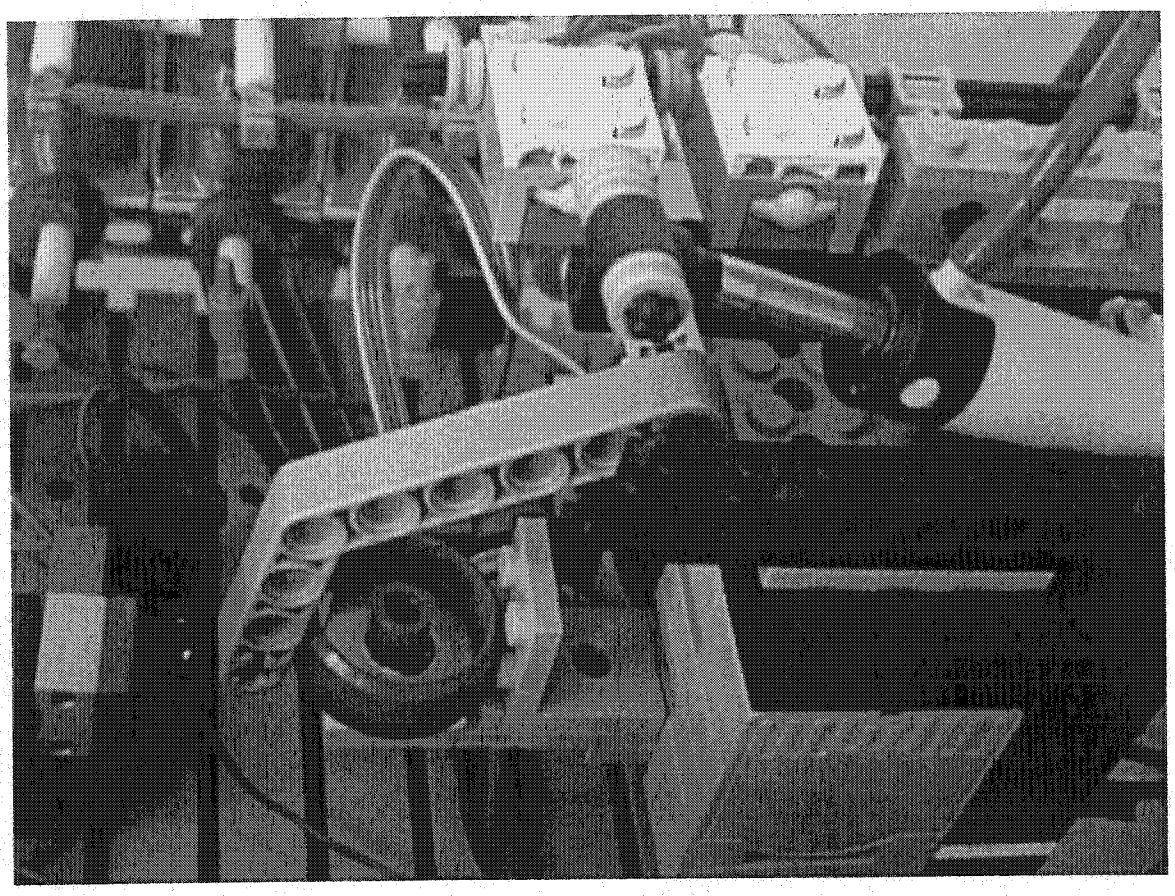


Figure E-34: Switches Used to Locate Clamp Position

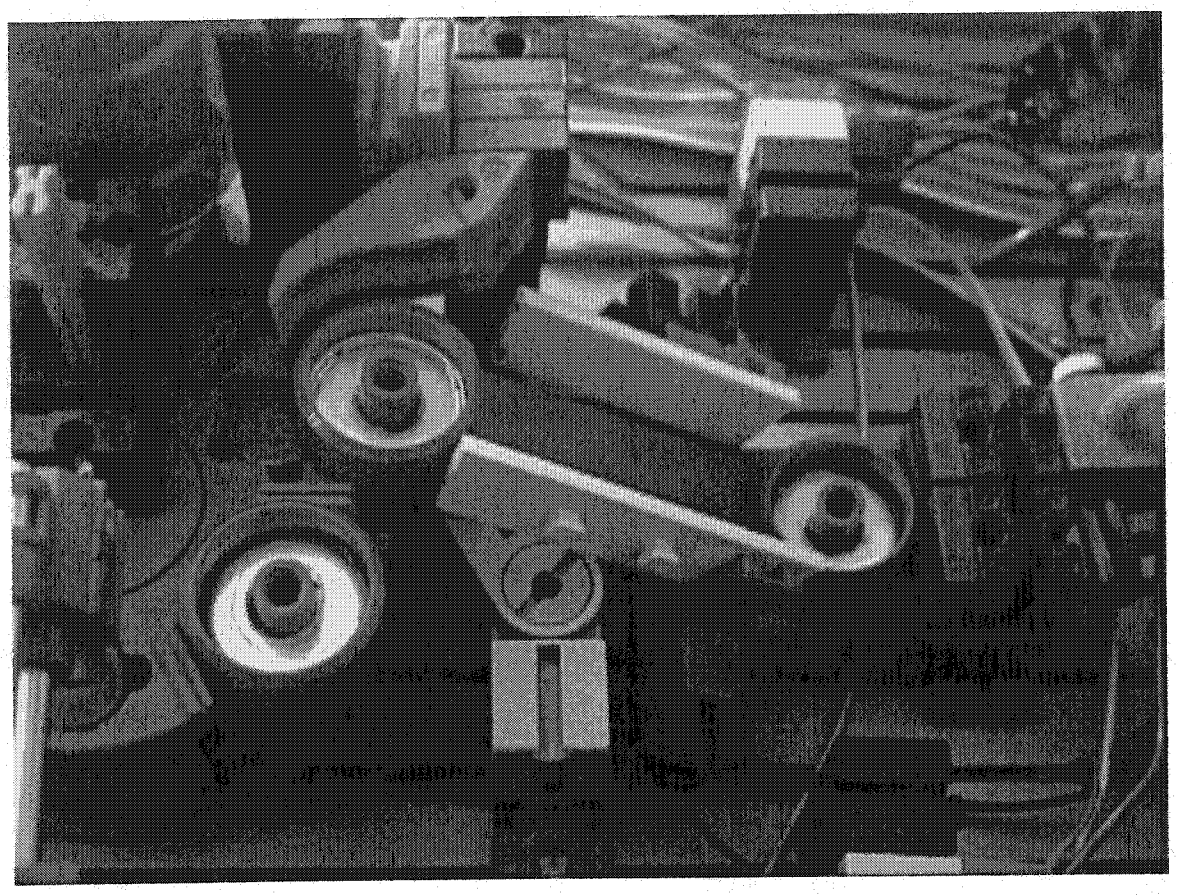


Figure E-35: Part Diverter and Ramp

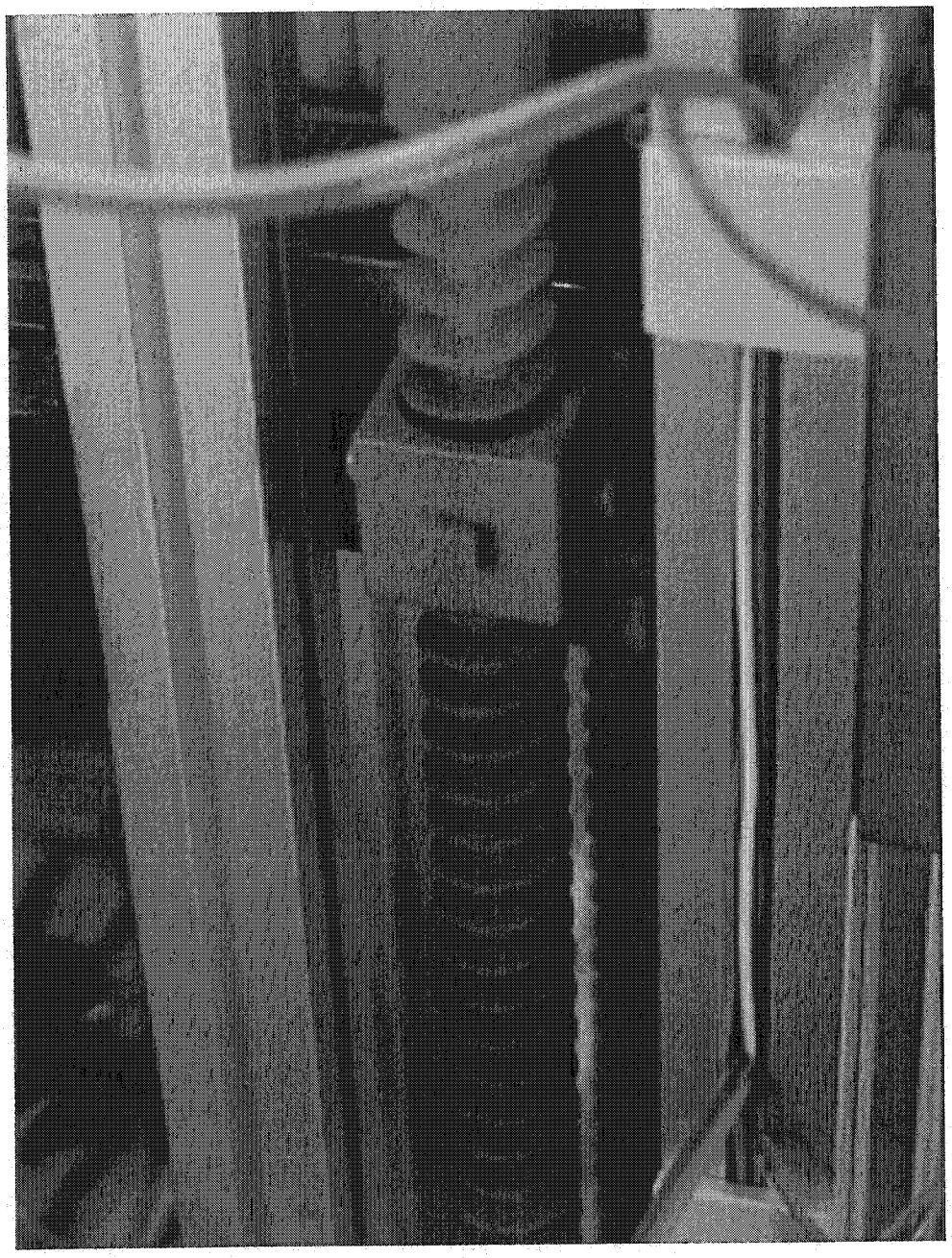


Figure E-36: Lead Screw and Threaded Block Drive

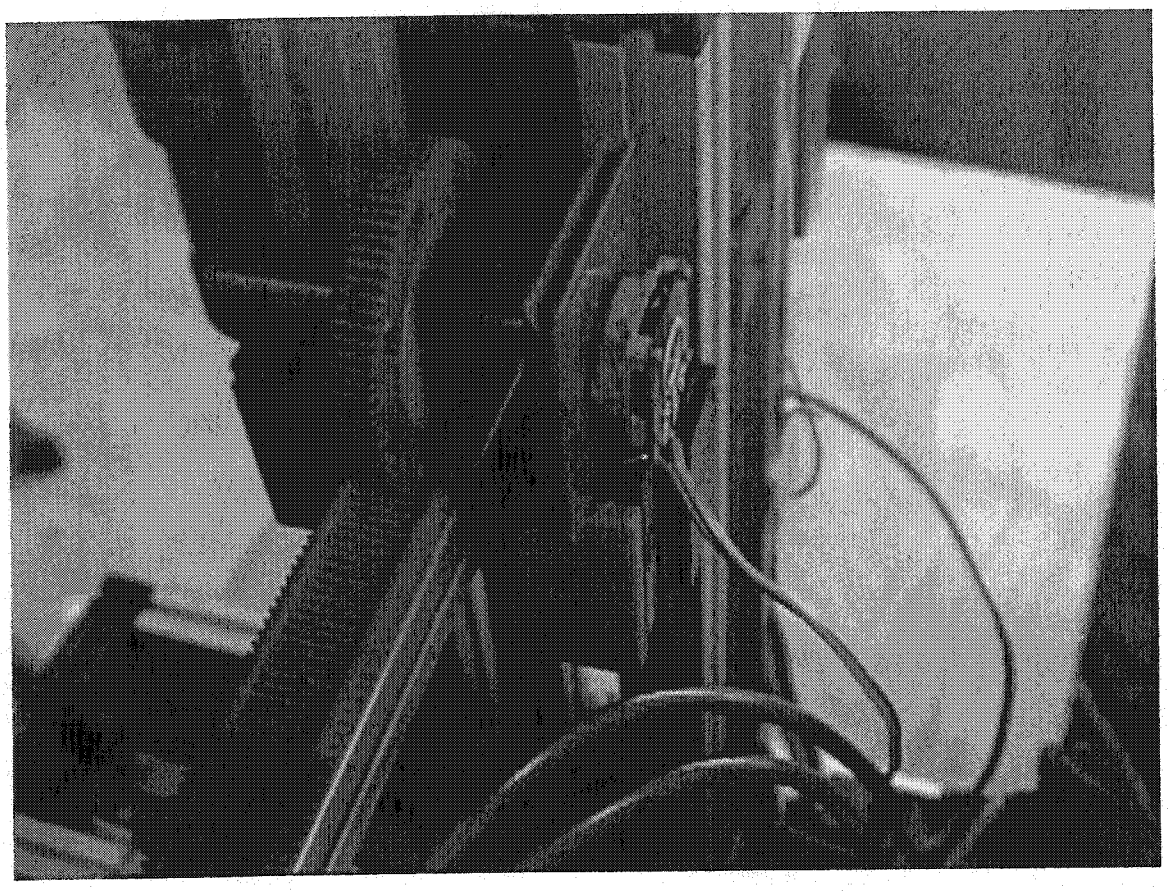


Figure E-37: Potentiometer Position Sensor

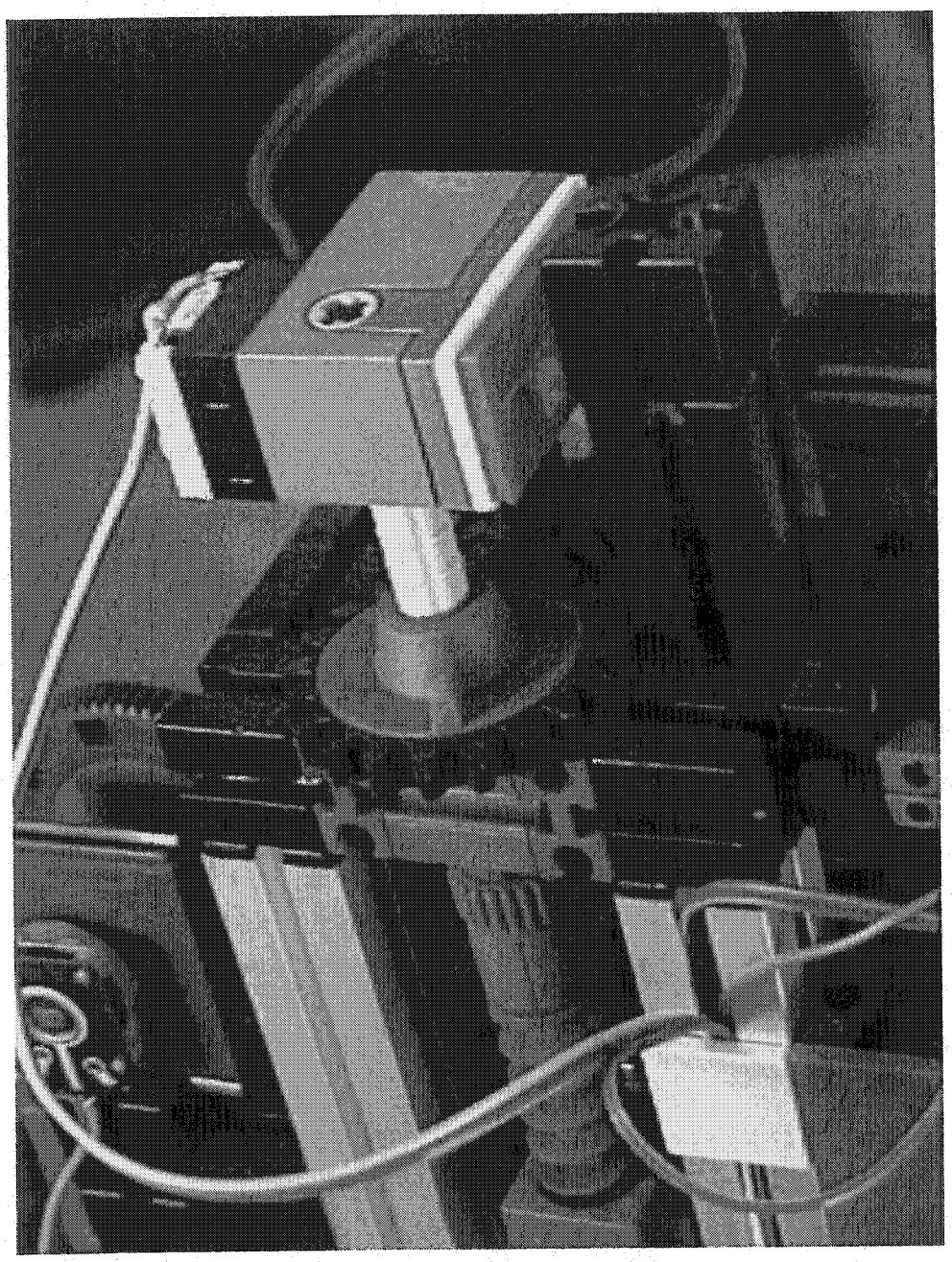


Figure E-38: Shaft Encoder

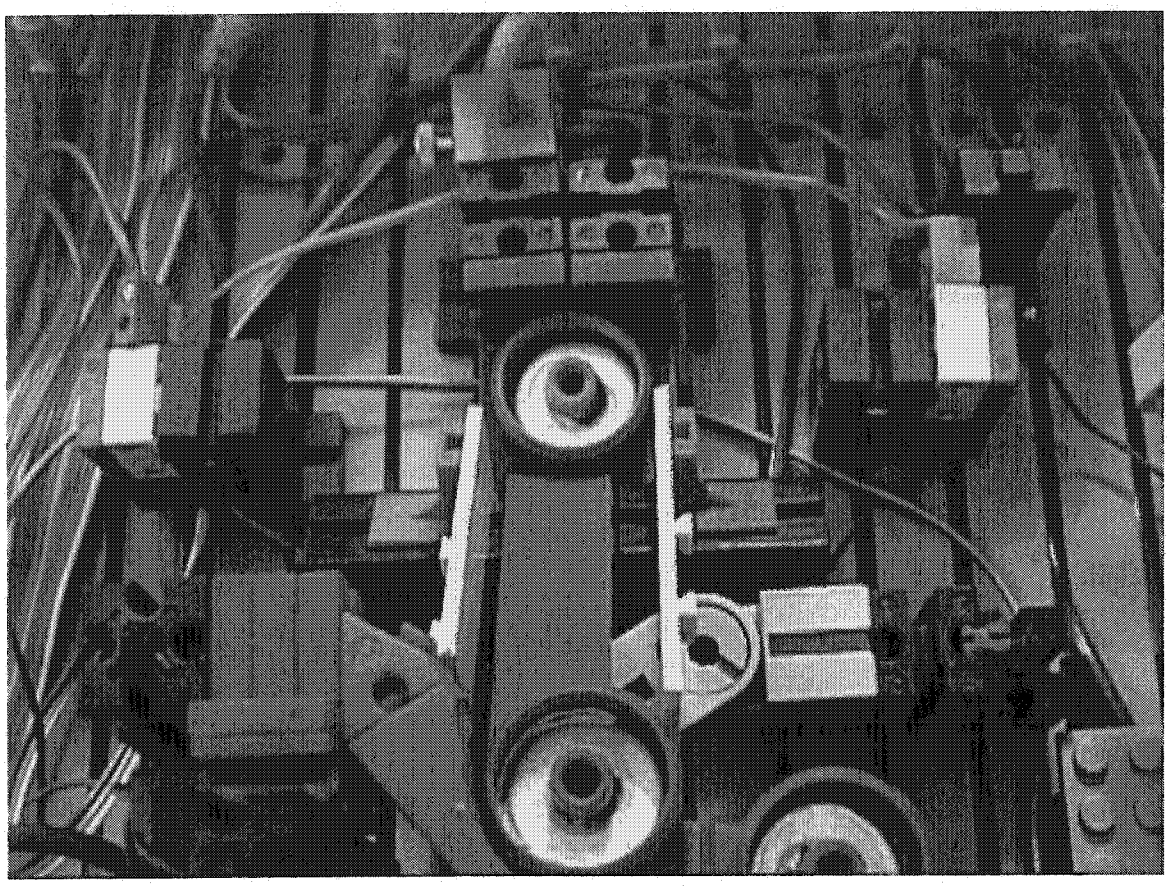


Figure E-39: Part at Bottom of Ramp Locating IR-LED and Phototransistor

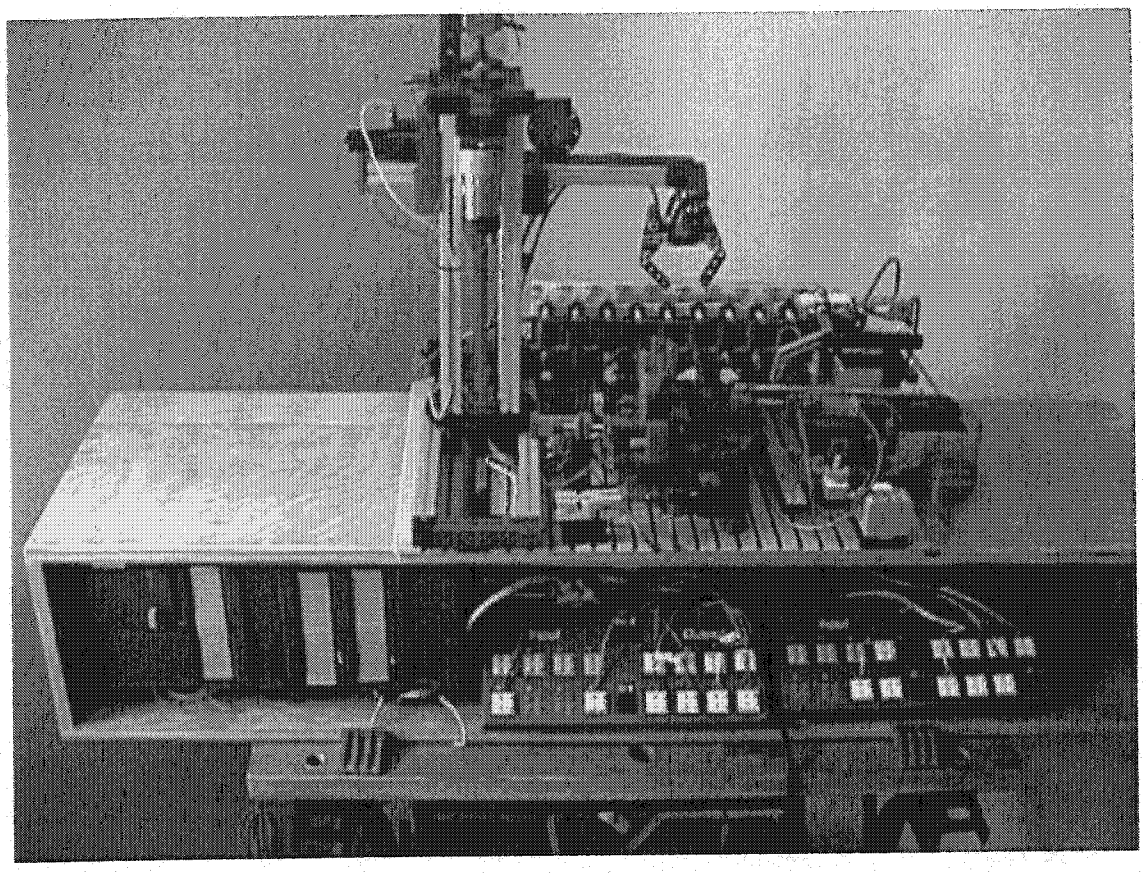


Figure E-40: Portable Experimental Platform

Appendix F

Lego I/O Interface Communication Protocol

Command		Affect on Motor	Affect on Lamp	Affect on Sound
Turn On	10010001	Turns it on	Turns it on	Turns it on
Turn Off	10010000	Turns it off	Turns it off	Turns it off
Reverse	10010101	Reverses Rotation	none	Changes sound played
Set Left	10010100	Rotates Clockwise	none	Plays sound #1
Set Right	10010011	Rotates Counter-CW	none	Plays sound #2
Set Power	10110nnn	Sets speed to n	Sets brightness to n	Sets volume to n

Table F-1: Output Port commands.

Byte numbers	Meaning
1-2	Zeros
3-4	Status of device on LEGO port 4
5-6	Status of device on LEGO port 8
7-8	Status of device on LEGO port 3
9-10	Status of device on LEGO port 7
11-12	Status of device on LEGO port 2
13-14	Status of device on LEGO port 6
15-16	Status of device on LEGO port 1
17-18	Status of device on LEGO port 5
19	Checksum byte

Table F-2: Input frames.

The sum of the values of the 19 frames must add up to 0xFF or an error has occurred in communication.

The interpretation of the two-byte status code corresponding to a port in each frame depends upon the kind of device connected to the port. It is impossible to know, from the input, exactly what kind of device is attached to a certain port. Thus, the software must process the data from each port for each kind of input device. The interpretation of the two-byte status codes for each device is as follows:

- For the touch sensor, if the first byte is 0x2E, then the sensor is pressed in, otherwise it is not.
- For the light sensor, the rightmost 12 bits are a value that describes the intensity of light received by the device.
- For the thermometer, the rightmost 12 bits can be converted into an approximation of the Fahrenheit temperature by using the formula T=(760-value)/4.4+32.

For the angle sensor, the third bit from the right is a Boolean value specifying the direction of rotation of the sensor. The rightmost two bits specify the amount of change in the angle since the last frame. This value, divided by 16, gives the fraction of a circle that the sensor rotated through since the last update. By remembering the last angle and using the direction and rate of change, it is possible to track the angle of rotation.

Appendix G

Visual Basic Classes Supporting I/O Interface

G.1 Three Position Selector Switch Class

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
  Persistable = 0  'NotPersistable
  DataBindingBehavior = 0  'vbNone
  DataSourceBehavior  = 0  'vbNone
  MTSTransactionMode  = 0  'NotAnMTSObject
END
Attribute VB_Name = "class_3_Position_SSW"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
Option Explicit

Private int_PLC_Input_Byte_Number(1 To 3) As Integer
Private byte_PLC_Bit_Number(1 To 3) As Byte
Private optb_Position_Selection(1 To 3) As OptionButton
Private str_Selected_Message(1 To 3) As String
Private rgb_Selected_Colour(1 To 3) As Long
Private str_Not_Selected_Message(1 To 3) As String
Private rgb_Not_Selected_Colour(1 To 3) As Long
Private pbs_Selector_Position_State(1 To 3) As Push_Button_State
Private Sub Class_Initialize()
  Dim i As Integer
  For i = 1 To 3
    pbs_Selector_Position_State(i) = PB_Not_Initialized
  Next i
End Sub
Public Sub Initialize(optb_Position_Selection_Parameter() As OptionButton, _
                      int_PLC_Input_Byte_Number_Parameter() As Integer, _
                      byte_PLC_Bit_Number_Parameter() As Byte, _
                      str_Selected_Message_Parameter() As String, _
                      rgb_Selected_Colour_Parameter() As Long, _
                      str_Not_Selected_Message_Parameter() As String, _
                      rgb_Not_Selected_Colour_Parameter() As Long)
  Dim i As Integer
  For i = 1 To 3
    Set optb_Position_Selection(i) = optb_Position_Selection_Parameter(i)
    int_PLC_Input_Byte_Number(i) = int_PLC_Input_Byte_Number_Parameter(i)
    byte_PLC_Bit_Number(i) = byte_PLC_Bit_Number_Parameter(i)
    str_Selected_Message(i) = str_Selected_Message_Parameter(i)
    rgb_Selected_Colour(i) = rgb_Selected_Colour_Parameter(i)
    str_Not_Selected_Message(i) = str_Not_Selected_Message_Parameter(i)
    rgb_Not_Selected_Colour(i) = rgb_Not_Selected_Colour_Parameter(i)
    optb_Position_Selection(i).Caption = str_Not_Selected_Message(i)
    optb_Position_Selection(i).BackColor = rgb_Not_Selected_Colour(i)
    pbs_Selector_Position_State(i) = PB_Not_Pressed
  Next i
End Sub
Public Sub Position_1_Click()
  Call Interface_With_PLC.Set_PLC_Input_Bit(int_PLC_Input_Byte_Number(1), _
                                            byte_PLC_Bit_Number(1))
  optb_Position_Selection(3).Enabled = False
End Sub
Public Sub Position_3_Click()
```

```
  Call Interface_With_PLC.Set_PLC_Input_Bit(int_PLC_Input_Byte_Number(3), _
                                            byte_PLC_Bit_Number(3))
  optb_Position_Selection(1).Enabled = False
End Sub
Public Sub Position_2_Click()
  Call Interface_With_PLC.Clear_PLC_Input_Bit(int_PLC_Input_Byte_Number(1), _
                                              byte_PLC_Bit_Number(1))
  Call Interface_With_PLC.Clear_PLC_Input_Bit(int_PLC_Input_Byte_Number(3), _
                                              byte_PLC_Bit_Number(3))
  optb_Position_Selection(1).Enabled = True
  optb_Position_Selection(3).Enabled = True
End Sub
```

G.2 Bi-Directional Motor Class

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
  Persistable = 0  'NotPersistable
  DataBindingBehavior = 0  'vbNone
  DataSourceBehavior  = 0  'vbNone
  MTSTransactionMode  = 0  'NotAnMTSObject
END
Attribute VB_Name = "class_Lego_Bidir_Motor"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
Option Explicit
Private bool_Forward_PLC_Output As Boolean
Private bool_Reverse_PLC_Output As Boolean
Private byte_Motor_Speed As Byte
Private byte_Lego_Output_ID As Byte 'bit mapped
Private int_Forward_PLC_Output_Byte_Number As Integer
Private byte_Forward_PLC_Bit_Mask As Byte
Private int_Reverse_PLC_Output_Byte_Number As Integer
Private byte_Reverse_PLC_Bit_Mask As Byte
Enum type_Bi_Directional_Motor_State
  Bi_Directional_Motor_Not_Initialized
  Bi_Directional_Motor_Stopped
  Bi_Directional_Motor_Turning_Forward
  Bi_Directional_Motor_Turning_Reverse
End Enum
Private Bi_Directional_Motor_State As type_Bi_Directional_Motor_State
Public Sub Process_PLC_Output(vrnt_PLC_Outputs_Parameter As Variant)
  Dim byte_Output(2) As Byte
  If ((vrnt_PLC_Outputs_Parameter((int_Forward_PLC_Output_Byte_Number)) _
      And byte_Forward_PLC_Bit_Mask) <> 0) Then
    bool_Forward_PLC_Output = True
  Else
    bool_Forward_PLC_Output = False
  End If
  If ((vrnt_PLC_Outputs_Parameter(int_Reverse_PLC_Output_Byte_Number) _
      And byte_Reverse_PLC_Bit_Mask) <> 0) Then
    bool_Reverse_PLC_Output = True
  Else
    bool_Reverse_PLC_Output = False
  End If
  Select Case Bi_Directional_Motor_State

  Case Bi_Directional_Motor_Stopped
    If (bool_Forward_PLC_Output = True) _
        And (bool_Reverse_PLC_Output = False) Then
```

```
      byte_Output(0) = &H94  'set direction to  rotate clockwise
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      byte_Output(0) = &H91  'turn motor on
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      Bi_Directional_Motor_State = Bi_Directional_Motor_Turning_Forward
    ElseIf (bool_Forward_PLC_Output = False) _
        And (bool_Reverse_PLC_Output = True) Then
      byte_Output(0) = &H93  'set direction to rotate counter clockwise
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      byte_Output(0) = &H91  'turn motor on
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      Bi_Directional_Motor_State = Bi_Directional_Motor_Turning_Reverse
    End If
  Case Bi_Directional_Motor_Turning_Forward
    If (bool_Forward_PLC_Output = False) _
        And (bool_Reverse_PLC_Output = False) Then
      byte_Output(0) = &H90  'turn motor off
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      Bi_Directional_Motor_State = Bi_Directional_Motor_Stopped
    ElseIf (bool_Forward_PLC_Output = False) _
        And (bool_Reverse_PLC_Output = True) Then
      byte_Output(0) = &H93  'set direction to rotate counter clockwise
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      Bi_Directional_Motor_State = Bi_Directional_Motor_Turning_Reverse
    End If
  Case Bi_Directional_Motor_Turning_Reverse
    If (bool_Forward_PLC_Output = False) _
        And (bool_Reverse_PLC_Output = False) Then
      byte_Output(0) = &H90  'turn motor off
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      Bi_Directional_Motor_State = Bi_Directional_Motor_Stopped
    ElseIf (bool_Forward_PLC_Output = True) _
        And (bool_Reverse_PLC_Output = False) Then
      byte_Output(0) = &H94  'set direction to  rotate clockwise
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      Bi_Directional_Motor_State = Bi_Directional_Motor_Turning_Forward
    End If
  Case Else
      byte_Output(0) = &H90  'turn motor off
      byte_Output(1) = byte_Lego_Output_ID
      form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
      Bi_Directional_Motor_State = Bi_Directional_Motor_Stopped
  End Select


End Sub



Private Sub Class_Initialize()
  Bi_Directional_Motor_State = Bi_Directional_Motor_Not_Initialized
End Sub
Public Sub Initialize(byte_Lego_Output_ID_prm As Byte, _
                      byte_Motor_Speed_prm As Byte, _
                      int_Forward_PLC_Output_Byte_Number_prm As Integer, _
                      byte_Forward_PLC_Bit_Number_prm As Byte, _
                      int_Reverse_PLC_Output_Byte_Number_prm As Integer, _
                      byte_Reverse_PLC_Bit_Number_prm As Byte)
  Dim byte_Output(2) As Byte
```

```
  byte_Lego_Output_ID = byte_Lego_Output_ID_prm
  byte_Motor_Speed = byte_Motor_Speed_prm
  int_Forward_PLC_Output_Byte_Number = int_Forward_PLC_Output_Byte_Number_prm
  byte_Forward_PLC_Bit_Mask = 2 ^ byte_Forward_PLC_Bit_Number_prm
  int_Reverse_PLC_Output_Byte_Number = int_Reverse_PLC_Output_Byte_Number_prm
  byte_Reverse_PLC_Bit_Mask = 2 ^ byte_Reverse_PLC_Bit_Number_prm
  byte_Output(0) = &H90  'turn motor off
  byte_Output(1) = byte_Lego_Output_ID
  form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
  byte_Output(0) = &HB0 + byte_Motor_Speed_prm 'set motor speed
  form_PLC_to_Model_Interface.comp_Lego2.Output = byte_Output
  Bi_Directional_Motor_State = Bi_Directional_Motor_Stopped
End Sub
```

G.3 Retentive Push-Button Switch Class

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
  Persistable = 0  'NotPersistable
  DataBindingBehavior = 0  'vbNone
  DataSourceBehavior  = 0  'vbNone
  MTSTransactionMode  = 0  'NotAnMTSObject
END
Attribute VB_Name = "class_Retentive_PB"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
Option Explicit
Enum Push_Button_State
  PB_Not_Initialized
  PB_Pressed
  PB_Not_Pressed
End Enum
Enum Contact_Type
  Normally_Open
  Normally_Closed
End Enum

Private int_PLC_Input_Byte_Number As Integer
Private byte_PLC_Bit_Number As Byte
Private cmdb_Push_Button As CommandButton
Private str_Pressed_Message As String
Private rgb_Pressed_Colour As Long
Private str_Not_Pressed_Message As String
Private rgb_Not_Pressed_Colour As Long
Private pbs_Button_State As Push_Button_State
Private udt_Contact_Type As Contact_Type


Private Sub Class_Initialize()
  pbs_Button_State = PB_Not_Initialized
End Sub
Public Sub Initialize(cmdb_Push_Button_Parameter As CommandButton, _
                      int_PLC_Input_Byte_Number_Parameter As Integer, _
                      byte_PLC_Bit_Number_Parameter As Byte, _
                      str_Pressed_Message_Parameter As String, _
                      rgb_Pressed_Colour_Parameter As Long, _
                      str_Not_Pressed_Message_Parameter As String, _
                      rgb_Not_Pressed_Colour_Parameter As Long, _
                      udt_Contact_Type_Parameter As Contact_Type)
```

```
  Set cmdb_Push_Button = cmdb_Push_Button_Parameter
  int_PLC_Input_Byte_Number = int_PLC_Input_Byte_Number_Parameter
  byte_PLC_Bit_Number = byte_PLC_Bit_Number_Parameter
  str_Pressed_Message = str_Pressed_Message_Parameter
  rgb_Pressed_Colour = rgb_Pressed_Colour_Parameter
  str_Not_Pressed_Message = str_Not_Pressed_Message_Parameter
  rgb_Not_Pressed_Colour = rgb_Not_Pressed_Colour_Parameter
  cmdb_Push_Button.Caption = str_Not_Pressed_Message
  cmdb_Push_Button.BackColor = rgb_Not_Pressed_Colour
  udt_Contact_Type = udt_Contact_Type_Parameter
  pbs_Button_State = PB_Pressed
  Call Just_Pressed
End Sub
Public Sub Just_Pressed()
  Select Case pbs_Button_State
  Case PB_Not_Pressed
    pbs_Button_State = PB_Pressed
    Select Case udt_Contact_Type
    Case Normally_Open
      Call Interface_With_PLC.Set_PLC_Input_Bit(int_PLC_Input_Byte_Number, _
                                                byte_PLC_Bit_Number)
    Case Normally_Closed
      Call Interface_With_PLC.Clear_PLC_Input_Bit(int_PLC_Input_Byte_Number, _
                                                  byte_PLC_Bit_Number)
    End Select
    cmdb_Push_Button.Caption = str_Pressed_Message
    cmdb_Push_Button.BackColor = rgb_Pressed_Colour
  Case PB_Pressed
    pbs_Button_State = PB_Not_Pressed
    Select Case udt_Contact_Type
    Case Normally_Closed
      Call Interface_With_PLC.Set_PLC_Input_Bit(int_PLC_Input_Byte_Number, _
                                                byte_PLC_Bit_Number)
    Case Normally_Open
      Call Interface_With_PLC.Clear_PLC_Input_Bit(int_PLC_Input_Byte_Number, _
                                                  byte_PLC_Bit_Number)
    End Select
    cmdb_Push_Button.Caption = str_Not_Pressed_Message
    cmdb_Push_Button.BackColor = rgb_Not_Pressed_Colour
  End Select
End Sub
```

Appendix H

State Based and Sensor Based Code Samples

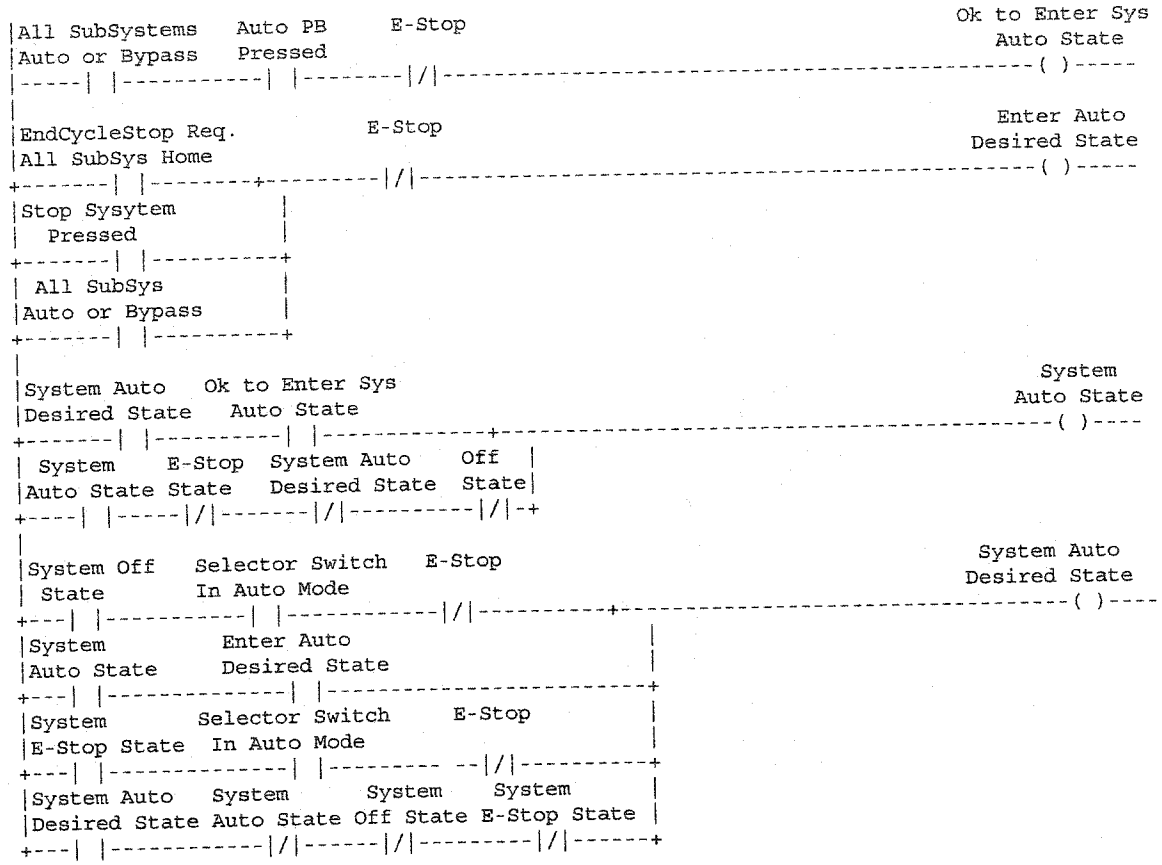


Figure H-1: State Based Code

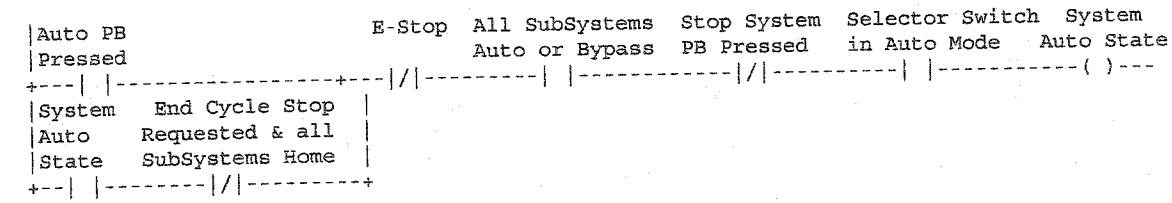


Figure H-2: Sensor Based Code