



Pilkington Library

Author/Filing Title Hopkinson

Vol. No. Class Mark T

**Please note that fines are charged on ALL
overdue items.**

LOAN COPY

0402087348



BADMINTON PRESS
UNIT 1 BROOK ST
SYSTON
LEICESTER, LE7 1GD
ENGLAND
TEL: 0116 260 2917
FAX: 0116 269 6639

A New Approach to the Development and Maintenance of Industrial Sequence Logic

by


Peter Hopkinson

A Doctoral Thesis
submitted in partial fulfilment of the requirements
for the award of

Doctor of Philosophy

of Loughborough University
Department of Manufacturing Engineering
November 1998

© by Peter Hopkinson 1998

 Loughborough University	
PL	ary
Date	Oct 99
Class	
Acc No.	040208734

M0000526LB

Acknowledgements

The author wishes to thank:

Dr. Robert Harrison for his friendship, enthusiasm and guidance.

My parents for their love and support.

Chris, Ben and Katie for making everything worthwhile.

Abstract

This thesis is concerned with sequence logic as found in industrial control systems, with the focus being on process and manufacturing control systems. At its core is the assertion that there is a need for a better approach to the development of industrial sequence logic to satisfy the life-cycle requirements, and that many of the ingredients required to deliver such an approach are now available.

The needs are discussed by considering the business case for automation and deficiencies with traditional approaches. A set of requirements is then derived for an integrated development environment to address the business needs throughout the control system life-cycle.

The strengths and weaknesses of relevant control system technology and standards are reviewed and their bias towards implementation described. Mathematical models, graphical methods and software tools are then assessed with respect to the requirements for an integrated development environment.

A solution to the requirements, called Synect is then introduced. Synect combines a methodology using familiar graphical notations with Petri net modelling supported by a set of software tools. Its key features are justified with reference to the requirements. A set of case studies forms the basis of an evaluation against business needs by comparing the Synect methodology with current approaches. The industrial relevance and exploitation are then briefly described.

The thesis ends with a review of the key conclusions along with contributions to knowledge and suggestions for further research.

Contents

CHAPTER 1 INTRODUCTION	10
1.1 BACKGROUND.....	10
1.2 ABOUT THIS THESIS.....	12
1.2.1 <i>Contribution of Each Lobe</i>	15
1.2.1.1 Graphical Method	15
1.2.1.2 Mathematical Model.....	15
1.2.1.3 Software Tool.....	15
1.2.1.4 Method + Tool	15
1.2.1.5 Model + Tool	15
1.2.1.6 Method + Model	16
1.2.1.7 Method + Model + Tool.....	16
CHAPTER 2 THE NEED.....	21
2.1 CLASSIFICATION OF MANUFACTURING PROCESSES	22
2.2 THE BUSINESS CASE FOR AUTOMATION	25
2.2.1 <i>Demands On The Business</i>	25
2.2.2 <i>The Benefits of Automation</i>	26
2.3 AUTOMATION PROJECT BUSINESS DRIVERS.....	27
2.3.1 <i>Cost</i>	27
2.3.2 <i>Timescales</i>	28
2.3.3 <i>Manufacturing Availability</i>	28
2.3.4 <i>Responsive to Change</i>	29
2.3.5 <i>Integration of Shop-Floor to Corporate Systems</i>	29
2.4 WEAKNESSES WITH AUTOMATION PROJECTS.....	30
2.4.1 <i>Software Engineering and System Specification</i>	30
2.4.2 <i>A Typical Project Scenario</i>	32
2.4.2.1 Requirements Definition	32
2.4.2.2 Functional Specification	33
2.4.2.3 Design.....	33
2.4.2.4 Implementation	34
2.4.2.5 Integration	35
2.4.2.6 Commissioning and Acceptance Testing	35
2.4.2.7 Beneficial Operation.....	36
2.4.2.8 Decommission or Upgrade.....	37
2.5 MODULAR AUTOMATION	38
2.6 SUMMARY OF AUTOMATION PROJECT BUSINESS DRIVERS	39
2.7 SUMMARY OF WEAKNESSES WITH CURRENT APPROACHES	39
2.8 OPPORTUNITY FOR CHANGE	41
CHAPTER 3 REQUIREMENTS FOR METHOD, MODEL AND TOOL.....	43
3.1 OUTLINE SOLUTION	44
3.1.1 <i>Requirements of the Method</i>	47
3.1.1.1 Clear, Concise and Complete Specification	47
3.1.1.2 Manage Complexity	47
3.1.1.3 Notations	48
3.1.1.4 Coherent Information.....	48
3.1.1.5 Avoidance of Design Errors.....	48
3.1.2 <i>Requirements of the Mathematical Model</i>	49
3.1.2.1 Visibility	49
3.1.2.2 Graphical Representation	49
3.1.2.3 Ability to Execute and Analyse.....	49
3.1.2.4 Ease of Code Generation.....	50
3.1.3 <i>Requirements of the CASE Tool</i>	50
3.1.3.1 Method Support	50

3.1.3.2	Usability	50
3.1.3.3	Integrated Development Environment	50
3.1.3.4	Rapid Prototyping and Visualisation	51
3.1.3.5	Documentation	51
3.1.3.6	Automatic Code Generation	51
3.2	OPPORTUNITY	52
CHAPTER 4 CONTROL SYSTEM TECHNOLOGY		54
4.1	PLC	55
4.1.1	<i>Background</i>	55
4.1.2	<i>Weaknesses</i>	57
4.1.3	<i>Trends</i>	63
4.2	DISTRIBUTED CONTROL SYSTEMS	64
4.2.1	<i>Implementation of Sequence Logic</i>	64
4.2.2	<i>Comparison with PLC Solutions</i>	65
4.3	INDUSTRIAL COMPUTER	66
4.4	FIELDBUS	67
4.4.1	<i>Echelon LonWorks</i>	70
4.5	SUMMARY	71
CHAPTER 5 STANDARDS		73
5.1	PROGRAMMING	74
5.1.1	<i>IEC 61131-3</i>	74
5.1.1.1	<i>Languages</i>	74
5.1.1.2	<i>Industrial Relevance</i>	77
5.1.1.3	<i>Limitations</i>	78
5.2	DESIGN METHODS	79
5.2.1	<i>EDDI, STEPS and KRAUSE</i>	79
5.2.2	<i>S88.01</i>	80
5.3	COMMUNICATIONS	83
5.3.1	<i>DDE</i>	83
5.3.2	<i>OLE/ActiveX</i>	83
5.3.3	<i>OPC</i>	84
5.3.4	<i>CORBA</i>	84
5.3.5	<i>The Internet</i>	85
5.4	CONCLUSIONS	85
CHAPTER 6 MATHEMATICAL MODELS		87
6.1	INTRODUCTION	89
6.1.1	<i>Communicating Sequential Processes (CSP)</i>	91
6.1.2	<i>Calculus of Communicating Systems (CCS)</i>	91
6.1.3	<i>Z and the Vienna Development Method (VDM)</i>	91
6.1.4	<i>Real Time Logic (RTL)</i>	91
6.1.5	<i>Symbolic Model Checking</i>	92
6.1.6	<i>Theorem Provers</i>	92
6.1.7	<i>Synthesis of Procedural Controllers</i>	92
6.1.8	<i>Petri Nets</i>	93
6.2	MOTIVATION FOR THE ADOPTION OF PETRI NETS	94
6.3	PETRI NETS	95
6.3.1	<i>Ordinary Petri Net</i>	95
6.3.1.1	<i>Modelling</i>	95
6.3.1.2	<i>Analysis</i>	97
6.3.2	<i>Coloured Petri Nets</i>	100
6.3.3	<i>Extended Petri Nets</i>	101
6.3.4	<i>Restricted Petri Nets</i>	101
6.3.5	<i>Timed and Stochastic Petri Nets</i>	101
6.4	WEAKNESSES	102
6.4.1	<i>Expressive Power</i>	102
6.4.2	<i>Analytical Power</i>	103
6.5	SUMMARY	103
CHAPTER 7 METHODS AND TOOLS		105

7.1	METHODS	105
7.1.1	<i>Evaluation Criteria</i>	105
7.1.2	<i>Structured Methods</i>	106
7.1.2.1	Data Flow Diagram	107
7.1.2.2	Entity Relationship Diagram	108
7.1.2.3	State Transition Diagram	109
7.1.3	<i>Object Oriented Methods</i>	109
7.1.3.1	Relevant Features of the Shlaer Mellor Method	112
7.1.3.2	Relevant Features of the Fusion Method	113
7.1.4	<i>Assessment</i>	114
7.2	TOOLS	116
7.2.1	<i>Evaluation Criteria</i>	116
7.2.2	<i>Requirements Capture</i>	117
7.2.3	<i>Design</i>	118
7.2.4	<i>Design Verification</i>	118
7.2.5	<i>Rapid Application Development</i>	119
7.2.6	<i>Visualisation</i>	120
7.2.7	<i>Simulation</i>	120
7.2.8	<i>Prototyping</i>	121
7.2.9	<i>Implementation</i>	122
7.2.10	<i>Testing</i>	122
7.2.11	<i>Auto-Code Generators</i>	123
7.2.12	<i>Assessment</i>	124
7.3	SUMMARY	125
CHAPTER 8 DESCRIPTION OF SYNECT		127
8.1	OUTLINE DESCRIPTION	128
8.1.1	<i>Method</i>	128
8.1.2	<i>Mathematical Model</i>	129
8.1.3	<i>Tools</i>	130
8.2	SYNECT METHOD	131
8.2.1	<i>The Object Hierarchy</i>	131
8.2.1.1	Object Interaction	131
8.2.1.2	Messaging	132
8.2.1.3	Interface with the Controlled System	133
8.2.1.4	Internal Events	134
8.2.2	<i>State Transition Diagram</i>	135
8.2.2.1	State	135
8.2.2.2	Transition	136
8.2.3	<i>Justification</i>	137
8.2.3.1	Clear, Concise and Complete Specification	137
8.2.3.2	Manage Complexity	137
8.2.3.3	Notations	137
8.2.3.4	Coherent Information	138
8.2.3.5	Avoidance Of Design Errors	138
8.3	PETRI NET MODEL	140
8.3.1	<i>Visibility</i>	140
8.3.2	<i>Ability to Execute and Analyse</i>	140
8.3.3	<i>Support for Code Generation</i>	141
8.4	SYNECT TOOLS	143
8.4.1	<i>Application Editor</i>	144
8.4.2	<i>Compiler</i>	145
8.4.3	<i>Analyzer</i>	146
8.4.4	<i>STD Monitor</i>	147
8.4.5	<i>Simulator</i>	148
8.4.6	<i>ANSI C Code Generator</i>	150
8.4.7	<i>Neuron C Code Generator</i>	152
8.4.8	<i>Allen-Bradley PLC Ladder Logic Generator</i>	153
8.4.9	<i>Justification</i>	154
8.4.9.1	Method Support	154
8.4.9.2	Usability	154
8.4.9.3	Integrated Development Environment	154

8.4.9.4	Rapid Prototyping and Visualisation	155
8.4.9.5	Documentation.....	155
8.4.9.6	Automatic Code Generation.....	155
CHAPTER 9 EVALUATION AND INDUSTRIAL EXPLOITATION		157
9.1	EVALUATION.....	158
9.2	CASE STUDIES	159
9.2.1	<i>Integrated Machine Design and Control (IMDC)</i>	159
9.2.1.1	Background	159
9.2.1.2	Synect Modules Used.....	160
9.2.1.3	Results	160
9.2.2	<i>Metal Forming Application</i>	163
9.2.2.1	Background	163
9.2.2.2	Synect Modules Used.....	163
9.2.2.3	Results	163
9.2.3	<i>Ford Rig</i>	164
9.2.3.1	Background	164
9.2.3.2	Synect Modules Used.....	164
9.2.3.3	Results	164
9.2.4	<i>Embedded Control Equipment</i>	165
9.2.4.1	Background	165
9.2.4.2	Synect Modules Used.....	165
9.2.4.3	Results	165
9.3	EVALUATION AGAINST REQUIREMENTS.....	166
9.3.1	<i>Analysis and Design</i>	167
9.3.1.1	Support Seamless Team Working.....	167
9.3.1.2	Support and Encourage Re-use	168
9.3.1.3	Explicit Support for S88.01.....	169
9.3.1.4	Facilitate Clear, Concise and Complete Specifications.....	169
9.3.1.5	Ability to Verify Correctness	170
9.3.1.6	Problem Oriented Approaches and Tools.....	171
9.3.2	<i>Implementation</i>	172
9.3.2.1	Consistent Implementation Architecture	172
9.3.2.2	Automatic Code Generation.....	173
9.3.2.3	Comprehensive Automatic Generation of Diagnostics.....	174
9.3.3	<i>Post-Delivery</i>	174
9.3.3.1	Easily Supportable and Maintainable Control System.....	174
9.3.3.2	Good Documentation.....	175
9.3.3.3	Good Enterprise Integration	176
9.3.3.4	Modular Automation	177
9.4	EXAMPLE WALKTHROUGH.....	177
9.5	INDUSTRIAL EXPLOITATION	179
9.5.1	<i>Relevance</i>	179
9.5.2	<i>Exploitation Results</i>	179
CHAPTER 10 CONCLUSIONS AND CONTRIBUTIONS TO KNOWLEDGE		181
10.1	THE BUSINESS NEED.....	182
10.2	THE REQUIREMENTS FOR A METHOD AND TOOL	183
10.3	INDUSTRIAL AUTOMATION TECHNOLOGY.....	185
10.4	MODEL-BASED AND FORMAL METHODS APPROACHES	185
10.5	METHODS AND TOOLS	186
10.6	CONTRIBUTIONS TO KNOWLEDGE.....	187
CHAPTER 11 SUGGESTIONS FOR FURTHER RESEARCH.....		189
11.1	GRAPHICAL METHOD	191
11.1.1	<i>Human Factors</i>	191
11.1.2	<i>Domain Specificity</i>	191
11.1.3	<i>Object-Oriented and Component-Based Support</i>	192
11.1.4	<i>Sequence Notations</i>	192
11.2	MATHEMATICAL MODEL.....	192
11.2.1	<i>Process Algebras</i>	193
11.2.2	<i>Explicit Support for Time</i>	193
11.2.3	<i>Petri Net Variants</i>	193

11.3	SOFTWARE TOOL	193
11.3.1	<i>Code Generators</i>	193
11.3.2	<i>Support for Enhanced Method</i>	193
11.3.3	<i>Wizards</i>	193
REFERENCES		194
APPENDIX A WALKTHROUGH OF SYNECT APPLICATION DEVELOPMENT		205
A.1	INTRODUCTION	205
A.2	DESCRIPTION OF THE PLANT EQUIPMENT	205
A.3	DESCRIPTION OF THE PROCESS	208
A.4	A SOLUTION USING SYNECT	212
A.4.1	<i>The Object Hierarchy</i>	212
A.4.2	<i>Assembly Cell STD</i>	213
A.4.3	<i>Feed Conveyor STD</i>	215
A.4.4	<i>Machine STD</i>	216
A.4.5	<i>Robot STD</i>	217
A.4.6	<i>Gripper STD</i>	219
A.4.7	<i>Arm Elevation STD</i>	220
A.4.8	<i>Arm Translation STD</i>	221
A.4.9	<i>Using The Synect Tools to Develop The Application</i>	222
A.4.9.1	Specify	222
A.4.9.2	Compile	223
A.4.9.3	Analyse	224
A.4.9.4	Simulate and Animate	227
A.4.9.5	Code Generation	230
APPENDIX B COPIES OF PUBLISHED PAPERS		264
APPENDIX C SYNECT USER GUIDES		309

Chapter 1 Introduction

1.1 Background

Manufacturing and process industries have consistently brought us a better standard of living through the delivery of products ranging from healthcare to consumer goods. Since the industrial revolution, the demand has been for a wider range of products, manufactured more efficiently with respect to labour, material and environmental costs and with progressively shorter product life-cycles.

These industries are consequently under increasing pressure to perform more effectively. This pressure translates into requirements for reduced manpower, shorter elapsed project time, faster and more accurate diagnosis of equipment or plant malfunction, greater flexibility required of the process, shorter production runs and more product variants. Greater levels of integration are required between the shop-floor control systems and the higher level corporate business systems in order to improve the overall business control loop.

Although increasing levels of automation have delivered many benefits to help in achieving these goals, the delivery and maintenance of the control systems are struggling to meet these ever-increasing demands. Control system solutions are based predominantly on Programmable Logic Controller (PLC) or Distributed Control system (DCS) technology which typically offer development environments focussed on implementation rather than analysis or design:

- The PLC was originally introduced to replace relay panels in use in the automotive industry. It was programmed in relay ladder logic for maximum conceptual compatibility with the relay panels. Ladder logic has since been augmented with other control system programming languages such as sequential function chart (for sequence logic) and function block diagram (for continuous control). However, the legacy of decades of familiarity coupled with the available programming environments may explain why many industrial projects are still implemented entirely in ladder logic. As larger and more complex applications are tackled, the resulting solution is prone to being badly structured leading to costly and time-consuming implementation, test and maintenance.

- DCSs have evolved from a continuous process background, initially superseding remote single-loop controllers. Sequencing capabilities have been added to cope with hybrid applications, such as batch control.

To satisfy market demand, flexible manufacturing plants are designed. These require more complex sequence logic but other business needs demand faster development, lower costs and a control system solution which must not compromise operational flexibility. Current approaches struggle to support these needs during both initial development and operational usage:

- Traditionally, there has been little support for the designer attempting to verify whether a proposed design will work as intended. If the design itself contains errors, these may not be found until integration testing or worse still, during commissioning or in the system's operational life. Errors found late in the life-cycle are many times more costly to rectify than errors detected earlier on.
- Control systems tend to be purchased with an anticipated life-span of many years. It is not uncommon to encounter control systems which are fifteen or more years old. During the control system's operational phase of the life-cycle, there may be requests for changes to be made, particularly where the market which the business serves is subject to significant change. Whereas the original developers of the system would have an intimate understanding of the behaviour of the system being controlled and the control system itself, such knowledge tends not to be readily available several years into the operational usage of the system. This leads to risk and uncertainty when design changes are assessed, particularly if a suitable test environment is not available.

1.2 About This Thesis

The core theme of this thesis, shown graphically in figure 1, is that the combination of a graphical method with an appropriate mathematical model, supported by a computer aided software engineering (CASE) tool, can contribute to satisfying the life-cycle business needs associated with the development of industrial sequence logic.

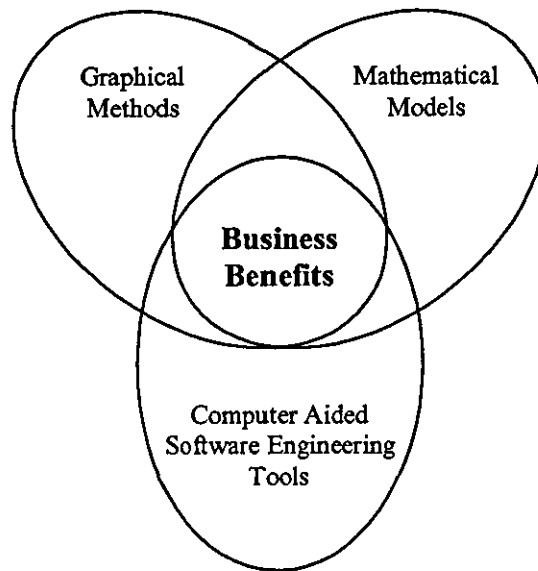


Figure 1 Core Theme Of This Thesis

In particular, the author asserts that increased emphasis should be placed on the early project activities of analysis and specification. Current control system technology is considered to be oriented at implementation and formal methods based approaches have not been offered in an industrially useable form. By integrating a mathematical model with a well-understood and widely-used notation and making these available on a hardware and software platform in regular use by the industrial community, a novel software environment has been developed which addresses current needs and provides the basis for further research.

Figure 2 A Summary of the "Forces" on Automation Projects

Figure 2 summarises chapter two, The Need. Better business performance is demanded of automation projects and this is represented by business drivers attempting to "push" the automation project bubble towards the right of the scale. Deficiencies with current approaches constrain the ability of automation projects to meet these demands. Eliminating these deficiencies will remove an obstacle to improved business performance.

The overall requirements for a solution incorporating a graphical method, mathematical model and software tool are described in chapter three, Requirements for Method, Model and Tool. Figure 3 and section 1.2.1 set this information in context by identifying each lobe's contribution and populates the diagram in figure 1 with examples.

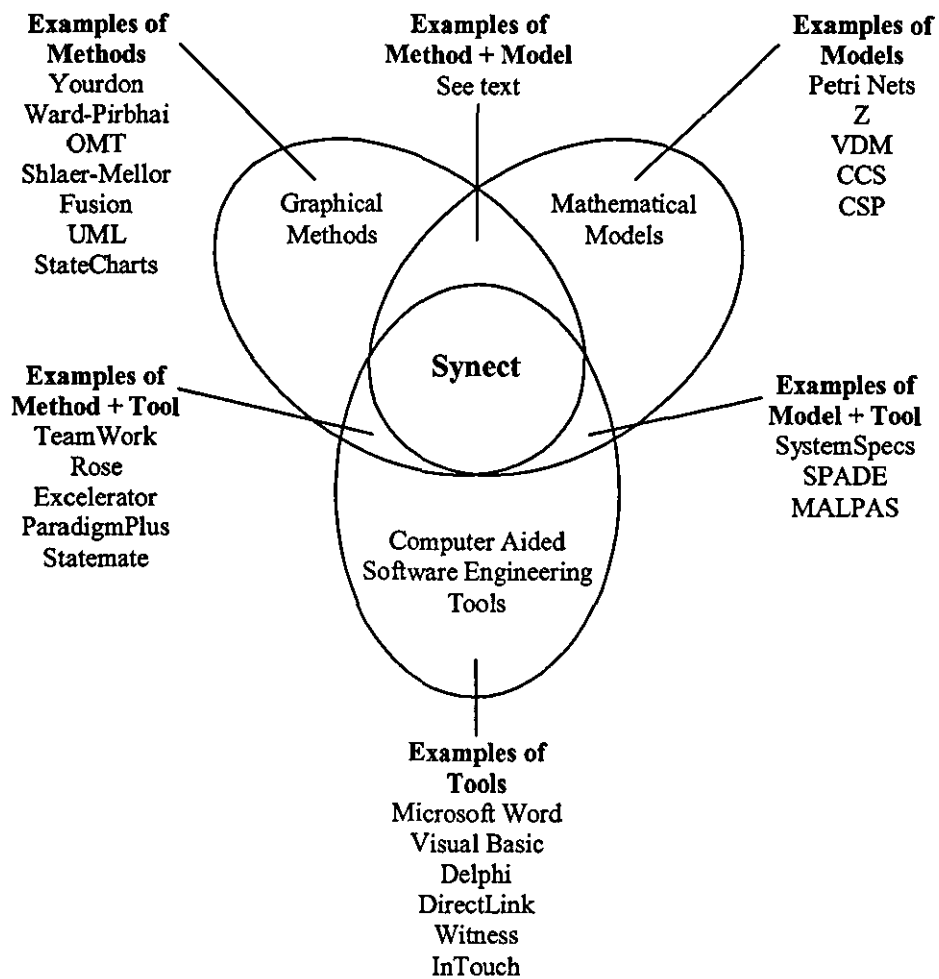


Figure 3 Examples in Each Lobe

1.2.1 Contribution of Each Lobe

1.2.1.1 Graphical Method

The graphical method facilitates effective communication of the requirements and proposed solution by providing a set of expressive notations which are easily understood. It also offers guidance regarding how the application is tackled.

1.2.1.2 Mathematical Model

The model provides the theoretical framework for verifying that the proposed solution is precisely specified and, for the purposes of this thesis, must also be capable of verifying syntactic and semantic properties such as completeness, consistency and intended behaviour.

1.2.1.3 Software Tool

For the purposes of this analysis, industrial software tools range in their applicability from assisting in requirements capture (of which the most common may be the word processor), to the vehicle for implementing a solution via programming language or configuration of pre-supplied functionality.

1.2.1.4 Method + Tool

A method-aware tool often provides an intelligent drawing tool and automated consistency checking, significantly improving productivity compared with a pencil and paper approach.

There appears to be significant industrial activity in this category relating to information systems development but less so specifically relating to sequence-based industrial control systems.

1.2.1.5 Model + Tool

For other than trivial applications, a tool may be essential to the effective use of the mathematical model. For example, manual derivation of a Petri net reachability tree would be impractical for an industrial-scale application.

Although there is significant academic activity in this category, industrial usage is limited and typically relates to high integrity applications such as safety critical systems.

1.2.1.6 Method + Model

Examples in this category would include a graphically expressive set of notations with modelling guidance and complemented by a mathematical model to support reasoning on properties of the specification:

- Although advocates of graphical methods may claim to be in this category due to support for completeness and consistency checks, they have limited support for reasoning on the behavioural properties of the solution.
- Mathematical modellers have extended graphical notations to produce more expressive representations. However, these notations are unfamiliar to the target audience of this research and typically provide weak support for desirable characteristics such as component orientation to promote software re-use.

1.2.1.7 Method + Model + Tool

This thesis derives the requirements for an integrated method, model and tool set and describes a solution which the author has named Synect.

Figure 4 re-phrases the negative statements of deficiency into positive statements of general requirements. Although not intended to be exhaustive, it shows the authors interpretation of the relationship between the general requirements and a set of method, model and tool requirements. Chapter 3, Requirements for Method, Model and Tool, discusses these in more detail and identifies the characteristics demanded of the method, model and tool, also summarised in figure 4. These characteristics are used in later in the thesis as evaluation criteria against which alternatives are assessed.

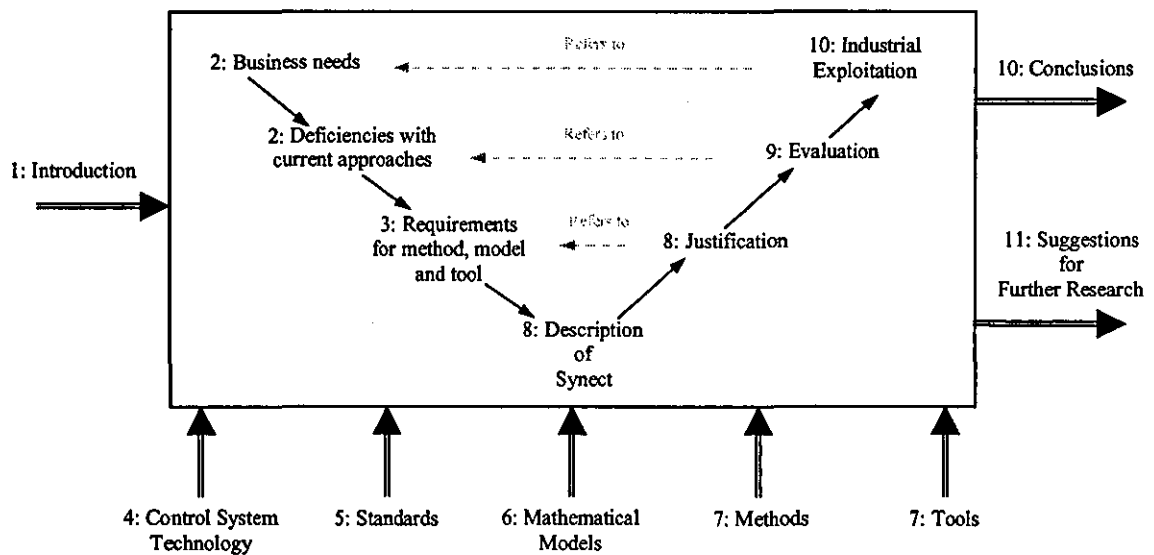


Figure 5 Thesis Map

Figure 5 is a diagram of the structure of the thesis. The chapter number is shown with each subject area.

Chapters four to seven discuss the current state of the art of relevant subject areas, identifying those aspects which can be incorporated into a solution and discussing limitations. These chapters also substantiate the requirements described in chapter three, Requirements for Method, Model and Tool.

The thesis is grouped into four parts:

Part 1

A critical review of industrial practice and the opportunity to satisfy the need.

- | | |
|-----------|---|
| Chapter 1 | is an introduction to the subject area. |
| Chapter 2 | identifies the business needs and deficiencies with current approaches. |
| Chapter 3 | derives a set of requirements for a method, model and software tool. |
| Chapter 4 | considers the strengths and weaknesses of control system technology. |
| Chapter 5 | reviews relevant standards. |
| Chapter 6 | concentrates on Petri net modelling but references alternatives for discrete event modelling. |
| Chapter 7 | considers the contribution which software development methods can offer and software tools which can assist in various phases of the control system life-cycle. |

Part 2

A proposed solution.

- | | |
|-----------|---|
| Chapter 8 | describes the author's method and software tool set called Synect and justifies its characteristics with reference to the requirements. |
|-----------|---|

Part 3

An evaluation of Synect and conclusions along with suggestions for continuing the research.

- | | |
|------------|--|
| Chapter 9 | evaluates the method using case studies as the vehicle to demonstrate how Synect overcomes deficiencies with traditional approaches. |
| Chapter 10 | identifies key conclusions and contributions to knowledge. The industrial relevance and exploitation potential are also considered. |
| Chapter 11 | makes suggestions for further research in this subject area. |

Part 4

Appendices.

- | | |
|------------|---|
| Appendix A | an example walkthrough of application development using Synect, showing the diagrammatic specification and generated ANSI C code. |
| Appendix B | copies of published papers. |
| Appendix C | Synect user guides. |

Chapter 2 The Need

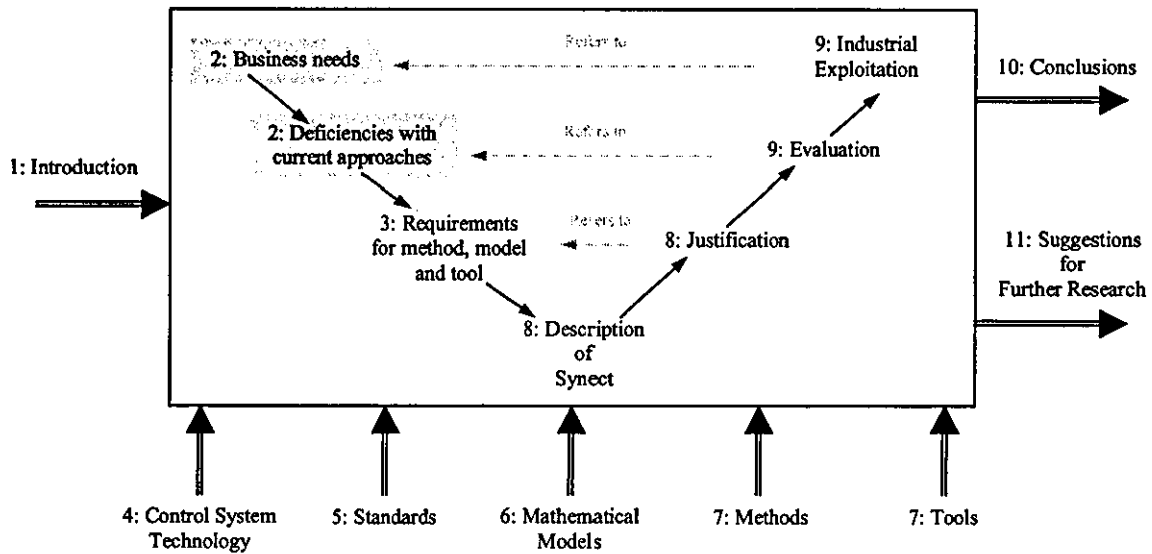


Figure 6 Relationship Between This Chapter And The Thesis Map

Figure 6 shows that this chapter establishes the business needs and weaknesses with current approaches to automation projects, contributing to the identification of a set of requirements against which, in later chapters, a solution is justified, evaluated and its industrial potential assessed.

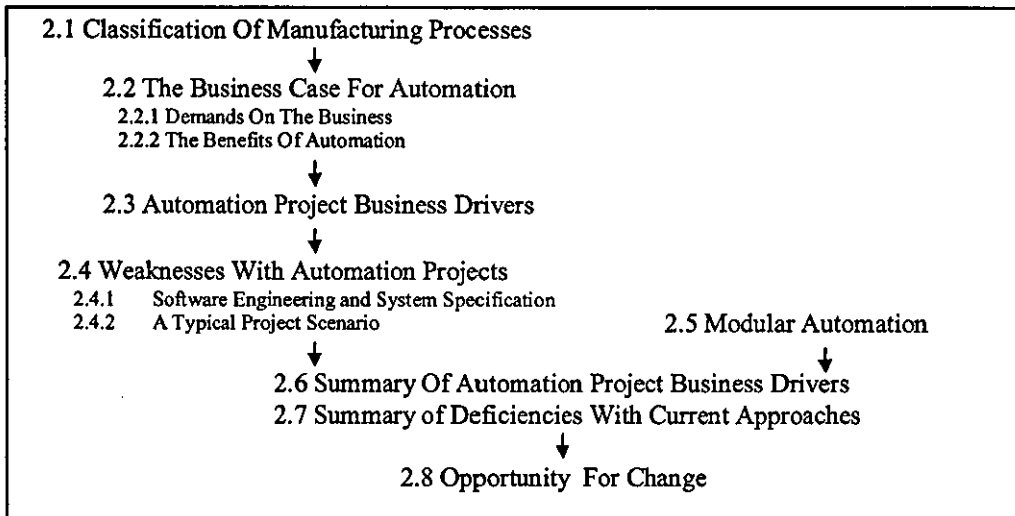


Figure 7 Structure Of This Chapter

As shown in figure 7, the chapter begins with a classification of manufacturing processes to describe the relevance of sequence logic. The contribution which automation can offer is followed by a description of automation project business drivers. Weaknesses with current approaches are then described to show the need for a better approach. Industry is

starting to use modular automation to address these needs and this is considered in the summary of automation project business drivers. Finally, the business demand and technological opportunity for addressing the need is outlined.

2.1 Classification of Manufacturing Processes

Industrial manufacturing processes may be initially classified as continuous, discrete parts manufacturing or batch [1]. The class of manufacturing process can be determined from whether the output of the process appears in a continuous flow (continuous), in finite quantities of parts (discrete parts manufacturing) or in finite quantities of material (batches). A comparison of the three types of process, from a process control perspective, is shown in table 1 [2]:

Characteristic	Continuous	Discrete	Batch
Product frequency	Weeks	Seconds	Hours
Lot sizes	Large	Small	Medium
Labour content	Small	High	Medium
Process efficiency	High	Low	Medium
%discrete / %analog i/o	5:95	95:5	60:40
Typical control system	DCS	PLC	Various

Table 1 A Comparison of Continuous, Discrete and Batch Processes

Batch production now accounts for approximately 50% of chemical production [3].

Analogue control usually requires the control system to drive a plant output such that a measured value tracks a specified setpoint. For example, the controller may be required to maintain a temperature in a vessel to a plant operator specified setpoint value by varying the flow rate of steam to the vessel's jacket. This type of control is often implemented using a PID (Proportional, Integral and Derivative) algorithm to determine the appropriate plant output value from the difference between setpoint and measured value over time.

Discrete control may be decomposed into combinational and sequential logic. Combinational logic is evident when the state of a control system output can be deduced from the instantaneous state of control system inputs, without reference to their history. Sequential logic takes into account the history of events and is characterised by the control system exhibiting memory. For example, assume that the manufacturing cell in figure 8 is

required to drill a pilot hole in the work-piece using the small drill bit, change head to a larger drill bit and then drill out the hole again.

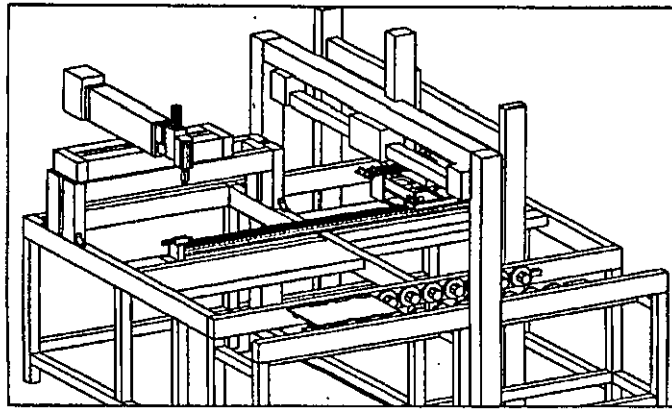


Figure 8 An Example Manufacturing Cell

Assume further that the work-piece is presented to the manufacturing cell on a conveyor and must be clamped whilst being worked upon. The required control of the rotary motion of the drill can be expressed as combinational logic:

Whenever the work-piece is clamped, the drill bit motor is to run.

Sequential logic describes the required control of the remainder of the drill unit:

When the part is clamped, lower the drill unit until it reaches the lower limit of travel

And then

Lift the drill unit until it reaches the upper limit of travel

And then

Change to the larger drill bit

And then

Lower the drill unit until it reaches the lower limit of travel

And then

Lift the drill unit until it reaches the upper limit of travel

And then

Change back to the smaller drill bit.

Finished.

The control of the drill bit motor and the clamp could be integrated with the above sequence definition such that control of the entire manufacturing cell is specified by one sequence and removing the need for any combinational logic. This helps to introduce further classifications within discrete manufacturing:

Classification	Description
Single vs. multi-threaded	The number of concurrent specifications needed to describe the control scheme. Multi-threaded control may require mechanisms for coordinating the activities and arbitration over the use of shared resources.
Autonomous vs. coupled	An autonomous control system operates independently of other control systems whereas coupling introduces the need to share information between control systems.
Repetitive vs. flexible	The repetitive manufacturing example above simply cycles around the statements in the same order every time. The need for alternative paths is a characteristic of flexible manufacturing, although this is to be differentiated from exception handling for dealing with malfunctions.

Table 2 Classifications within Discrete Manufacturing

An alternative approach, promoted during the structured analysis and structured design era, is to classify an automated system based on its main role as shown in table 3 (see also chapter seven, Methods and Tools):

- **Data transformation.** This applies to systems which undertake a continuous algorithmic function, such as the transformation of a continuous time-varying signal into an updating frequency spectrum.
- **Entity relationship.** In this type of system, the primary role of the functional aspects is to ensure that the relationships between information entities is maintained. For example, an order management system must maintain the relationships between customers and orders and possibly stock holding and deliveries.
- **Reactive.** A reactive system is considered to be one which is principally concerned with the ability to respond to discrete input stimuli, often referred to as events [4]. They are typically predominantly event-driven in that they may be quiescent until an

event is detected, which then leads to one or more discrete output signals [5]. Transformational systems, by contrast, are more associated with continuous inputs and outputs. In a reactive system, the response to an event may differ depending on the preceding sequence of events. Whereas this can be implemented in a combinational logic form with flags being used on an ad-hoc basis, a clearer specification can often be expressed using a state-based approach, particularly when a visual representation is used. From a modelling perspective, the state-based approach has been found to be a natural medium for describing dynamic behaviour [5], [6].

Characteristic	Data Transformation	Entity Relationship	Reactive
Real-time	Yes	No	Yes
Transaction-based	No	Yes	No
Type of input & output	Continuous	Event	Event
Memory of past events	No	Yes	Yes
Industrial sector	Continuous process	Corporate level in all sectors	Discrete and batch
Primary diagrammatic notation	Data flow diagram	Entity relationship diagram	State transition diagram

Table 3 Characteristics of an Alternative Classification

2.2 The Business Case For Automation

2.2.1 Demands On The Business

The environment in which manufacturing business now operates is being subjected to increasing competition and greater market demands, requiring agile manufacturing with “reconfigurable everything” [7]. The market-place is also becoming increasingly international in both supply and demand:

- A study by UK Foresight [8] anticipated that CAD/CAM, advanced control and low wage economies will directly influence the competitiveness of chemical plants. Labour costs in the Czech Republic, for example, are only 7% of those in the UK [9]. In response, multi-disciplinary teams are seen as helpful in addressing the change in internal business drivers to focus on the need for rapid development of new products and processes.

- Within the next 25 years, the 'emerging' countries will be fully fledged industrial economies. Although published before the recent turmoil in far eastern stock markets, the World Bank forecast that they will overtake the advanced industrial countries in their share of world output by the year 2000 and eclipse them completely 20 years later. China will be the world's largest economy; India and Indonesia will be ahead of all European countries; and Thailand, Taiwan and South Korea will be in the top ten [10].
- ICI predicts that ten years from now, the Asian chemical market will be larger than either America or Europe and that in twenty years, it will be bigger than America and Europe combined [11].

The competitive environment leads to a number of demands on the manufacturing process including:

- Lower operating costs.
- Greater differentiation from the competition.
- Faster response to changing market conditions. This leads to shorter product lifecycles as exemplified by Siemens, who generate 70% of their revenue from products and systems introduced within the last five years [9].

2.2.2 The Benefits of Automation

Selective automation may help to satisfy the business demands listed above [12], [13]:

- Lower operating costs
 - Reduce the number of operations staff required to run the plant.
 - Increase capacity.
 - Increase yield/reduce scrap.
 - Reduce re-work.
 - Reduce inventories.
- Greater differentiation from the competition

- Greater consistency of finished product (tighter specifications) by eliminating variability which is attributable to manual operation.
- Greater individuality of product, the ultimate goal being that each finished product is unique.
- Faster response to changing market conditions
 - Ability to introduce new product variants or increase capacity rapidly. This requirement is particularly apparent in the food and beverage industries. Recipe configuration software and the ability to replicate existing software modules help to satisfy these requirements.

Having identified the benefits of automation, a feasibility study may be established to determine whether a project can be initiated which will deliver the benefits at acceptable cost, in an acceptable elapsed time without undue risk. This is often referred to as a cost/benefit analysis which may then be used to judge the worthiness of the project by considering the payback period - the elapsed time after commissioning when the accumulated savings exceed the project costs. If the project is required to help the manufacturing process meet market demand, elapsed time may be critical to ensure that market share is not lost to competitors.

Historically it may have been infeasible to automate smaller plants, but the large number of small batch plants, increasing business demand for competitiveness and the reduction in scale of entry level automation systems has created the demand for a large number of small automated applications.

2.3 Automation Project Business Drivers

An automation project and the resulting control system are also subject to business drivers which affect activities leading up to beneficial operation and beyond, including:

2.3.1 Cost

There is always a demand to reduce project costs. However, this must be balanced by the need to have the plant operational at the earliest opportunity and the risk of delays.

Typical losses incurred by batch plant downtime are £100,000 to £400,000 per day [14] although this may be in the £1m to £2m range for some pharmaceutical plants. The project cost also affects the viability of the project and hence the scale of application to which automation can feasibly be applied. One of the significant changes in the batch control industry at the moment is the reducing scale of economically viable automation [15].

2.3.2 Timescales

The period between project sanction and ability to enter beneficial operation must be shortened. This lowers project costs, exploits the financial advantage of beneficial operation and may satisfy other marketing objectives, such as maximising market share, by being first to market. In the semiconductor industry, for example, a six month delay in reaching markets can result in a 33% loss of aggregate revenue [16].

In some cases, the control system development is on the critical path. If timescales can be shorted by adopting a simpler or more elegant solution, costs may also be lowered and the solution may be more supportable/maintainable. An example would be where a software platform's directly supported functionality could be used rather than having to undertake custom software or substantial configuration to meet a detailed user requirement.

The timescale can be considered to consist of:

- Development time from project sanction to start of commissioning.
- Commissioning through to startup.
- Rectification of faults and anomalous behaviour during early beneficial operation.

Whilst the focus at the start of a project tends to be on the elapsed time until delivery to site, commissioning overruns are notorious [17] and there is often a period after startup where modifications are made to the control system to correct faults and misunderstandings between the developers and users.

2.3.3 Manufacturing Availability

Unplanned downtime must be minimised and this requires effective support and maintenance of the control system:

- Supportability** The ease with which errors can be identified. The errors might relate to the control system but typically support calls often turn out to have been caused by operator error or plant equipment malfunction. Referring again to the cost of plant downtime, there is clearly a need to be able to rapidly diagnose faults.
- Maintainability** Maintenance is a term which traditionally refers to the need to take preventative action to avoid failure but, in the software industry, is more typically interpreted as the ease with which the system can be modified, usually implying fault rectification or minor enhancement.

2.3.4 Responsive to Change

The control system must be able to exploit the flexibility in the equipment. The specification should be such that the process designer can confidently and correctly re-configure the system rather than needing software engineers to do it. There is evidence of manufacturing facilities being built where the product to be manufactured is unknown until shortly before commissioning [18].

Where the scope of change is such that software modifications are required, for example if additional equipment is added, the change must be capable of being implemented at minimum cost, with minimum loss of availability of the plant and with maximum confidence in its correctness and lack of impact on other aspects of the control system.

2.3.5 Integration of Shop-Floor to Corporate Systems

Ideally the manufacturing facility would be part of the overall business control system. This requires greater ability to integrate the manufacturing control systems with the corporate systems to enable the necessary information to be derived and transferred between them. Information downloaded from the corporate system could include recipe parameters, such as quantities of ingredients. Uploaded information could include machine running hours for maintenance planning and machine utilisation rates and yield for measuring manufacturing performance. However, there are significant differences between the real-time environment of the manufacturing control system and the transaction environment of the corporate system [19]. A standardised manufacturing execution support (MES) architecture helps to bridge the gap, solving immediate concerns but providing extension capabilities [20].

2.4 Weaknesses With Automation Projects

2.4.1 Software Engineering and System Specification

A typical automation project will include a significant proportion of software development and is therefore vulnerable to the problems associated with such projects [21]:

- Poor predictability in attempting to estimate time and effort required to produce a system satisfying the user requirements.
- Low quality programs which either crash or fail to adequately meet the user requirements.
- High maintenance costs. This includes both rectification and enhancement.
- Duplication of effort. The little re-use which does occur tends to be at the cut-and-paste of code level rather than re-use of design.

Although there is a growing awareness of the need for good software engineering practice, many of the problems remain which caused DeMarco to report in 1982 that [22]:

- Fifteen per cent of all software projects never deliver anything; that is, they fail utterly to achieve their established goals.
- Overruns of one hundred to two hundred per cent are common in software projects.

Good software engineering practice is of particular importance in safety-related applications. Reservations regarding the quality of PLC software for PLC-based protection systems lead the DTI and the then Science and Engineering Research Council (SERC – now EPSRC) to sponsor the Software Engineering Methods and Safe Programmable Logic Controllers (SEMSPLC) project. This resulted in the publication of a set of guidelines for the development of PLC application software for safety related applications [23].

Lifecycle models have been adopted to help structure the project activities required from feasibility analysis, into beneficial operation and finally to eventual decommissioning or upgrade. One of the simplest and more widely used is the V Life Cycle Model, alternatively referred to as the Waterfall model (described in section 2.3.2).

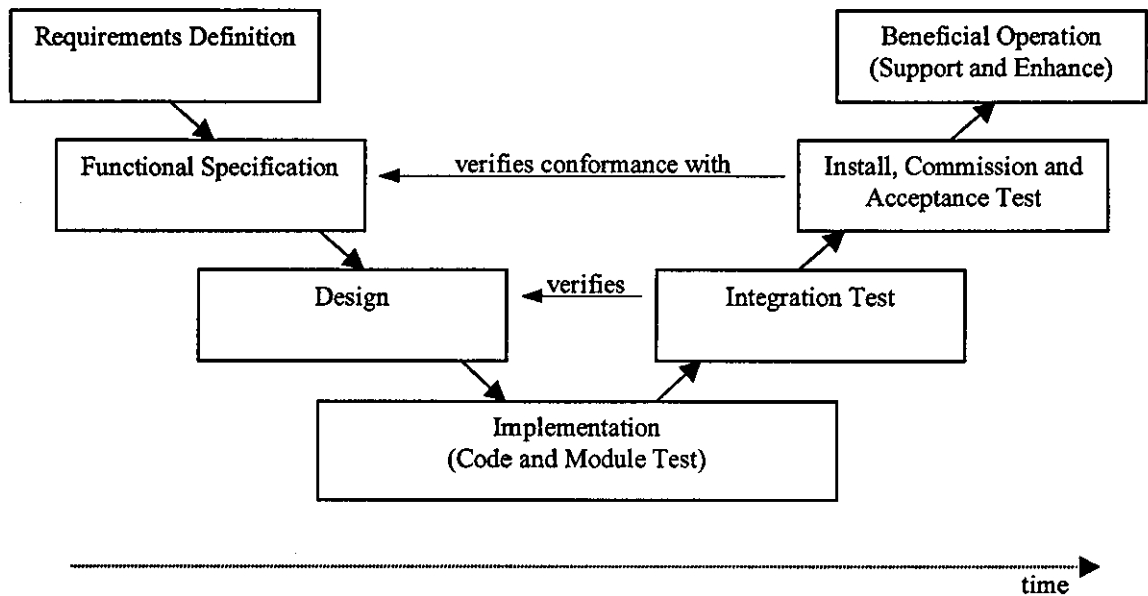


Figure 9 The V Life-Cycle Model

Analyses by life cycle phase suggest that more attention should be paid to the early project activities addressing analysis and design specification:

- In the HSE's Out Of Control publication [24], computerised systems were found to have contributed to serious accidents. The majority of incidents were caused by defects which should have been anticipated rather than by subtle failure modes, 45% of failures investigated being due to specification error.

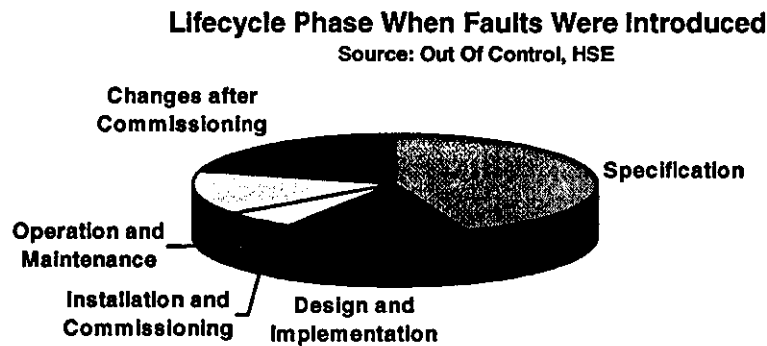


Figure 10 Life Cycle Phase When Faults Were Introduced

- Other studies are reported to show 75% of all in-service software errors are due to specification [25].
- Compared with the cost of immediately correcting an error made at the requirements stage, the error will be one hundred times more expensive to correct if left until the system is in beneficial operation [26].

2.4.2 A Typical Project Scenario

The need for a good specification can be compounded by contractual boundaries between organisations. A typical project scenario is now described by life-cycle phase, along with associated weaknesses.

2.4.2.1 Requirements Definition

This is usually prepared by the manufacturing company or its representatives. It is almost always text-based and attempts to describe the required behaviour of the control system

in general terms. This document is typically issued to a number of potential suppliers for them to respond to. It is not uncommon for requirements to be vague or contradictory.

2.4.2.2 Functional Specification

The supplier defines a functional specification, specifying the functionality to be supplied to meet the user's requirements and highlighting areas of non-compliance. The derivation of the functional specification from the user requirements specification therefore crosses organisational boundaries and increasingly geographic boundaries, due to the globalisation of the market-place. The document tends to be text-based with accompanying diagrams for illustration. It forms the contractual definition of the scope of supply and is the basis from which acceptance test documentation is derived. As such, it should be complete and precise but typically fails on both counts, leading to contractual negotiations about assumed or implied functionality and the implementation of incorrect detailed functionality.

Tool support for functional specification and design have not evolved at the same pace as for implementation [27]. For a batch control system, for example, a proposed solution is typically expressed using the terminology and models defined in S88.01 [1]. However, S88.01 is open to interpretation and although S88.01 aware platforms are becoming available, there is little tool support available at the specification stage of the life-cycle.

2.4.2.3 Design

Having specified the functionality to be delivered, the supplier then designs the mechanisms to implement the functionality. The end-user is typically excluded from this activity on the grounds that they are just interested in the functionality. Design documents tend to be a mixture of text with block diagrams to show modularisation. The modules might then be designed and implemented separately.

Although implementation tools are in widespread usage such as Rockwell's RSLogix5 programming software [28], there are very few design methods or tools in use. Typically there is minimal reuse of either design or implementation code because a supplier is tasked with supplying an integrated system rather than components which can be re-used. Whilst component-oriented languages encourage re-use, IEC 61131-3 offers only weak support for this paradigm (see chapter five, Standards).

The use of formal methods is rare, unless the system is destined for a high integrity application such as the nuclear industry. This is partly attributable to a lack of awareness

of the types of error which formal methods can help reveal. The perception that an advanced level of mathematical ability is required also alienates formal methods from typical implementation staff. Weak and expensive tool support are further disincentives [29]. It is interesting to compare this discontinuity of skill set and approach with the introduction of the PLC where the implementation paradigm was unchanged (see chapter four, Control System Technology). This suggests that advances are more likely to be achieved incrementally.

Although system behaviour should be documented and agreed at the functional specification stage, detailed behaviour is often relegated to the design phase. If poor or unique design structures are used, the solution may be difficult to support, particularly if the design concepts are not apparent from the implementation.

If the system is to be reconfigurable, the functionality of the software modules must be reviewed with a multi-disciplinary team including process, operations and control system experts [30]. Text-based specifications are difficult to review and even printed diagrams can be weak - because they are not animated, there is no easy facility to review how the control system reached the state currently being considered.

2.4.2.4 Implementation

The use of automatic code generation is rare. The implementation phase is often characterised by an enlarged team who hand-code the implementation using the vendor's programming software. Testing will be performed on each module, typically using either simulation software or a hardware box of switches and lights to simulate the plant. The implementation tools tend to be proprietary, imposing a learning curve on the developers whilst they familiarise themselves with the capabilities and idiosyncrasies of the equipment and programming software.

Limited consideration would typically be given to coding standards to increase testability and even less so to the incorporation of diagnostics to aid fault-diagnosis of malfunctions outside the control system, such as sensor failure. Approaches such as STEPS are specifically designed to address this deficiency (see chapter five, Standards, section 5.2.1).

2.4.2.5 Integration

Whereas the testing during implementation focussed on verifying the correct behaviour of individual software modules, the tested modules are now integrated and tested for their ability to work together. This is typically where unforeseen interactions between the modules are identified, although the absence of audit trails can cause substantial difficulties in identifying the cause of anomalous behaviour. Design errors which are only identified during integration can have a significant impact on project costs and timescales. Because the design must be corrected, a number of modules will require change and subsequent testing. Integration is typically the last activity before shipping the control system to site so delays in this phase can directly delay delivery to site. In summary, design errors which propagate through to integration testing are difficult to diagnose, substantially more costly to rectify than if they had been detected during design, and directly impact on delivery to site.

The integrated system must deal with a significant degree of asynchronous behaviour. Whilst the control system is likely to be thoroughly tested for the anticipated sequence of events which would occur when the system behaved correctly, the abnormal behaviour is less completely tested. There may be a considerable number of alternative sequences of events but it is likely that only a subset of these are encountered during integration testing. Even with extended periods of testing, the extent of test coverage will be unknown, leaving the possibility of errors being propagated through to commissioning or live operation of the plant.

2.4.2.6 Commissioning and Acceptance Testing

The supplier commissions the control system, verifying that the equipment is controlled correctly. Errors encountered at this stage may be attributable to the:

- Control system. If the control system does not behave as specified, the root cause may be poor specification or poor design. Such errors should have been trapped during development testing and factory acceptance testing, particularly where plant simulators are employed to mimic the behaviour of plant equipment. However, the number of combinations of sequences of events from asynchronous activities means that there may be an untested sequence of events which causes an error. Typically, errors are more likely to be a result of poor requirements specification, either because the specification was incomplete or vague [31].
- Plant equipment. If the plant equipment behaves differently to expectations, the equipment and control system may be incompatible. Typically this requires that the

control system is modified. In these cases, the control system itself is used as a diagnostic tool for investigating the plant equipment's behaviour. This is a fundamentally different role to that required to control the plant.

- **Process.** In the chemical industry in particular, the manufacturing process specified by the process chemist may be incorrect or incomplete. Only when the control system is on-site and connected to the plant equipment does the process chemist learn whether the process specification was correct (and it often isn't!).

2.4.2.7 Beneficial Operation

After sign-off, the plant will be handed over to production to begin manufacturing. At this time, a support contract is often established with the supplier. Faulty or anomalous behaviour is often poorly reported and again, the control system is being used to diagnose plant equipment malfunction but the lack of an audit trail can make verification and identification of the cause very difficult.

Flexibility may be required to cope with changing business circumstances. This may take the form of:

- Changing the process conditions required to manufacture existing products.
- Making variants of existing products or completely new products.
- Alternative or additional plant equipment in order to tighten specifications or increase throughput, for example.

It would be advantageous for the manufacturer if the process designer were able to make control system changes to reflect required process changes without recourse to its developers. This increases the responsiveness of the automation system to the business needs and lowers overall costs. It also helps to prevent supplier lock-in where a systems integrator knows that income lost during the control system's development can be recovered from requested changes because the manufacturer cannot use competitive tendering to drive prices down. The uniqueness of mapping from functionality to design to code contributes to supplier lock-in so standardisation in this area could offer commercial benefits to end-user organisations.

The ability to support, maintain and enhance traditional control systems is poor. There are many examples of PLC-based systems written in relay ladder logic for which either no

design documentation exists or it is of poor quality. Maintenance of such software, including both changing or enhancing functionality and also fault-finding, is very difficult [32], [33], [34]. Implementations of relay ladder logic solutions typically take the form of a combinational rather than state-based solution. If a suspected logic error is reported, the complexity of the code is such that support engineers cannot easily determine which of many possible sequences of events could have caused the reported behaviour. Tools such as the Symbolic Simulation Based Debugger have been produced which produce a description of all conditions leading to a specified behaviour to help in the detection of logic errors [35]. It is clearly preferable to structure the code and include diagnostics at development time to assist in fault finding.

Considering that enhancement activities may constitute up to 75% of a system's cost over its lifetime, designing for a system's entire life-cycle is imperative [36].

2.4.2.8 Decommission or Upgrade

Ultimately, all control systems are either decommissioned or upgraded. The justification may be business related, such as lack of market demand for the manufactured product or uncompetitive production costs due to the use of a superceded process. Sometimes the reliability and maintainability of the control system itself leads to the requirement to replace it.

In the automotive industry, the life-span of a manufacturing machine may be substantially longer than the life-span of the product being manufactured [37], [38]. Its replacement may perform very similar activities but may not reuse any of the existing equipment or control system software.

Where the business demand is relatively stable, control systems often run for ten, fifteen or more years before a significant upgrade is required. By this time, the original project team is unlikely to be available so the new project team must rely on the available documentation to establish the functionality of the existing system. Before the new control system functionality is specified, a reverse engineering activity may be undertaken to obtain a specification which is complete and unambiguous.

2.5 Modular Automation

Although industrial-strength software is inherently complex [39], which is to say that the complexity is in the problem rather than a by-product of the solution, modularity is a mechanism for managing the complexity by adopting the philosophy of “divide and conquer”.

Modular automation is being driven by the need to control more flexible manufacturing plants and to support reconfiguration of the plant equipment:

- This is currently of major interest to the batch process community and is a key goal of the ANSI S88.01 standard [1]. The benefit to the manufacturer is a control system which supports the inherent flexibility in a plant design [18].
- In the discrete manufacturing environment, modular automation is visible in the use of flexible manufacturing cells and also in the use of “soft” automation where, for example, a machine uses configurable single axis controllers which can be disconnected and used on a different machine.

Modular automation seeks to use well-defined modules to develop an automation system. The goal is to be able to re-use existing modules to lower the design and development costs and to gain from the increased confidence of knowing that the module is already proven (although care must be taken that the new application is not using untested functionality in the module as apparent in the Ariane rocket accident where the trajectory required the use of an algorithm in a previously untested manner leading to catastrophic failure [40]).

Whilst this approach offers the attractiveness of faster development and lower project costs on subsequent projects, the early projects must carry the cost of developing generic modules and their configuration into the required solution rather than the development of a bespoke solution. Considering the highly competitive nature of the automation supply industry and the traditional evaluation criteria placing considerable emphasis on the lowest cost bid to get to beneficial operation, there are clear obstacles to the long term success of this approach.

2.6 Summary of Automation Project Business Drivers

This chapter has established the industrial relevance of a focus on small to medium scale flexible manufacturing systems in the discrete and batch processing industries. In particular, the problem concerns applications which contain concurrent threads of sequence logic.

The business demands on such automation projects can be summarised as follows:

- Reduce the cost from initial conception through to beneficial operation.
- Reduce the overall life cycle costs.
- Reduce the elapsed time from initial conception through to beneficial operation.
- Give the process back to the process designers. Provide a control system which is sufficiently intuitive that the process designers can confidently and successfully implement changes without recourse to the system's suppliers.
- Improve the maintainability of the control system such that plant equipment changes can be accommodated quickly and at low cost.
- Improve availability by incorporating diagnostic capabilities into the control system such that plant equipment malfunctions can be identified and dealt with rapidly in order to minimise plant downtime.
- Integrate the plant control system into the business in order to achieve seamless integration with corporate (business control) computer systems.

2.7 Summary of Weaknesses With Current Approaches

What is preventing manufacturing industries from satisfying the business needs which are so clearly in evidence? The use of a life-cycle approach and the focus on component oriented technologies are two examples of how the industry is attempting to address the business drivers. However, many examples of deficiencies with current approaches were identified in section 2.4, Weaknesses With Current Approaches, and are grouped below by life-cycle phase on which they have the most impact.

Analysis and Design

- Organisational and geographic boundaries cause discontinuities in the project.

- Minimal re-use of proven functionality.
- Explicit support for S88.01 terminology and models is required in a batch control system.
- Requirements specifications and functional specifications suffer from being vague, incomplete and contradictory. In summary, they may offer a poor medium for communication due to being open to misinterpretation.
- Inability to verify correctness of the design, behavioural properties and deepen all parties knowledge of the evolving system until a significant proportion of the software has been designed, coded, module tested and integration tested. In particular, unforeseen interactions may only be revealed late into integration testing.
- Implementation oriented approaches and weak tool support for specifying control system solutions.

Implementation

- Different implementation architecture every time. This leads to inconsistencies which reduces the supportability of the system and is a significant factor in supplier lock-in.
- Manual code generation. At best, the code will faithfully implement the design.
- Poor diagnostics in the implementation. The diagnostics may be necessary to assist in the diagnosis of faulty control system logic but is usually required to assist with troubleshooting plant equipment malfunction. An even greater level of diagnostics are required during commissioning.

Post-Delivery

- Inability to confidently and cost-effectively support the desired degree of flexibility in the plant or process. Flexibility may be required in the form of setpoint changes, changes to the way existing sequence logic is to be coordinated, or changes to plant equipment configuration. In particular, the control system should be capable of reconfiguration without recourse to the original system development staff.

- Limited confidence that documentation reflects the currently live control system. This is of particular concern when the time comes for the system to be replaced.
- Poor integration with corporate systems, partially due to the different organisational environments in which the corporate and manufacturing systems reside.
- Bespoke control system software may prevent the cost-effective re-use of plant equipment when the product it was manufacturing is no longer required.

2.8 Opportunity For Change

The opportunity to meet these needs will be expanded upon in subsequent chapters but may be summarised as:

- **Business demand.** There is a growing awareness of the need to support business goals through the implementation of automation and a recognition of the deficiencies with current approaches. For example, the traditional contractual model has been based on fixed price competitive tendering, concentrating on minimising the cost up to beneficial operation. (Indeed, some organisations consider cost and timescale variations to the original contract to be a measure of the “goodness” of their specification). However, the growing awareness of the need to address life cycle costing [41], [42] is leading to a focus on supply chain management and partnerships. The business benefits of software re-use is also being recognised, particularly in the IT sector [43].
- **Control System Technology (chapter four).** As the base functionality of the target PLC and DCS platforms has increased and their application areas have converged, vendors have responded to market demand and attempted to differentiate their offerings by making them more “open”. This is reflected by the ability to exchange data between the programming environment and third party software and also by the connectivity to the live control system.
- **Familiarisation with the principles of object-oriented modelling, modular automation and the focus on re-use (chapter seven).** Whilst control systems were often seen as bespoke developments, with specifications concentrating on the procedural functionality required, the principle of modelling is gaining acceptance, assisted by the growing awareness of object-oriented methods and languages. Model building

appeals to all engineering disciplines [44], appealing to the principles of decomposition, abstraction and hierarchy.

- The availability of proven and established mathematical models which are sufficiently powerful yet comprehensible by control systems developers (chapter six).
- Internationally agreed and de-facto standards (chapter five). These provide the convergence of opinion and evolution of best practice. They also offer the potential commitment from end-users which gives software tool developers confidence regarding the level of demand for a tool supporting the standard.
- The availability of low-cost powerful PC technology for performing computationally intensive algorithms and to support highly expressive methods which can be quickly learnt (chapter seven).

These factors will be discussed in more detail in the subsequent chapters, highlighting weaknesses and opportunities to contribute to an improved approach by:

- Focussing attention on ensuring that the proposed solution will behave as intended.
- Reducing the craft element of implementation to yield faster and more consistent delivery of the control system.

Chapter 3 Requirements for Method, Model and Tool

Chapter two, The Need, established the need for an improved approach to the development of industrial sequence logic, described deficiencies with current approaches and indirectly identified the opportunity for a method and tool to contribute to a solution by exploiting graphical methods, mathematical modelling and CASE tool technology. As shown in figure 11, this chapter specifies the requirements for a method, model and tool against which the proposed solution will be justified in chapter eight, Description of Synect.

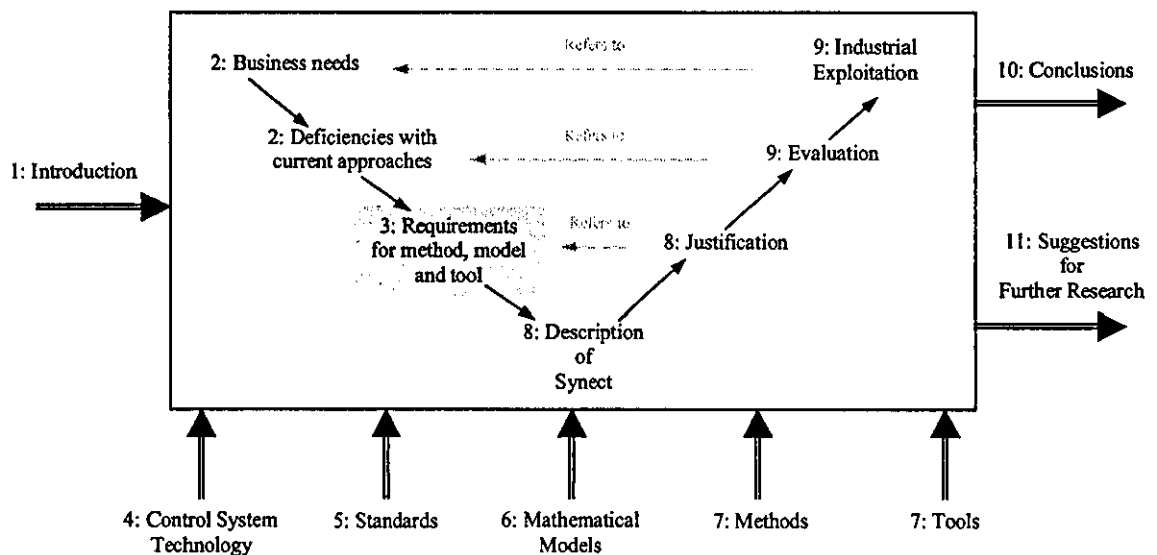


Figure 11 Relationship Between This Chapter And The Thesis Map

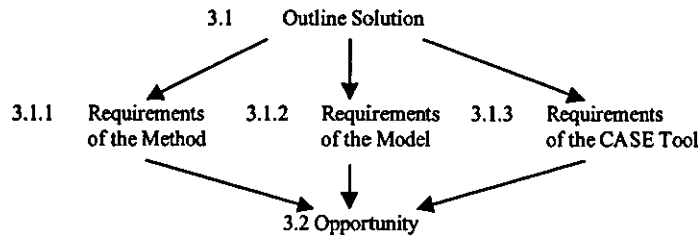


Figure 12 Structure of this Chapter

This chapter is structured as shown in figure 12:

- An outline solution which combines a graphical method, mathematical model and software tool support is justified, establishing the context for the detailed requirements to follow.
- Method requirements, model requirements and tool requirements are then introduced and related to the general requirements as summarised in figure 4 in chapter one, Introduction. The discussion of each set of requirements also identifies desirable characteristics of a solution, which are also shown in figure 4.
- The business motivation, knowledge and technology discussed in earlier chapters is summarised to identify the opportunity to satisfy the requirements.

3.1 Outline Solution

In keeping with the goal of producing research output which has high industrial relevance, the method and tool should be of benefit for small to medium size systems. Whereas academic and major industrial innovations have tended to address highly complex systems [5], many industrial applications are smaller in scope. Such applications can still benefit significantly from a model-based approach with good tool support. The author contends that there is an analogy with best software engineering practice. Once the principles are understood and the benefits recognised, it is unlikely that a project would be undertaken in which the software was implemented with minimal modularity, high coupling etc., although the standards might not be so rigorously enforced. Similarly, whereas formal methods appear to be currently viable only for applications where the cost of error is high, the author believes that once engineers become familiar with the method and tools proposed, they will be reluctant to give them up. Whilst potential users are concerned at taking on a new method or tool for large projects, the risk is considered lower with a smaller project and this helps to reduce one of the obstacles to exploitation. Once the

experience with the method and tool has been established, users may feel more confident about applying such techniques to larger projects.

The target user community includes systems integrators and end-users involved in the specification, development and operational support of small to medium-scale applications. Many of these systems integrators have evolved from electrical rather than software engineering backgrounds [25]. For an innovative product to appeal to early adopters [45], there must be a recognisable potential for benefit with limited investment of time and cost. Consequently the method and tool must be intuitive so that it can be learnt quickly and easily, preferably without the need for training courses or consultancy. Comparing available specification techniques, Mallaband notes that proprietary methods may have advantages in terms of rules for their application and for tool support and, in some cases, automatic code generation [46]. However, he considers that they generally use unfamiliar notations, are expensive and usually need significant training and consultancy to be applied effectively. The tool must therefore guide the analyst towards a good solution. Its emphasis should be on preventing errors being introduced by the analyst rather than providing a mechanism for revealing such errors later. In the comprehensive review of techniques and tools for specifying real-time systems [47], a set of requirements for an ideal tool are listed:

"An easy and intuitible method and tool. Where 'easy' means 'very close' to the analyst mindset. For this reason the tool must be endowed with a graphic user interface, and it must allow both top-down and bottom-up approaches for software specification, as well as a combination of these."

"A model to make easier the reusing of reactive system specifications. This means that the model adopted must provide support for software composition by reusing already defined software components."

"A method for verifying and validating the specified software against critical conditions from the early phases of system specification."

"An executable model to allow the validation of system behaviour by means of simulation".

Communication between humans and between human and control system should be supported:

- Effective human to human communication is essential to ensure that the requirements and proposed solution are clearly understood. Geographic and organisational

boundaries can make such communication more difficult. With reference to batch control systems, practical experience [14] has suggested the desirability of:

- A formal problem decomposition philosophy.
- A standard logic representation, such as the state transition diagram.
- Formal reviews using a multi disciplinary team.
- The solution must be expressed in a form suitable for the target control system, typically using bespoke software and configuration of pre-packaged functionality.

Chapter one, Introduction, described the contribution which a graphical method, mathematical model and software tool can offer. Consequently, an outline solution would provide a development environment consisting of:

- A method to define how the problem should be addressed and to define a set of notations for expressing corresponding concepts.
- A mathematical model to provide the capability to analyse the behaviour of a proposed solution.
- A CASE tool (or suite of tools) platform which supports the method and operates on the model. A platform is considered to be a development environment which helps the developer move closer to the solution by providing a foundation on which to build. In particular in this case, it supports a method for tackling the problem and offers a set of tools to support the method. It also provides a framework to define the relationships between the tools (i.e. so they are an integrated set of tools) and defines the boundary of the platform so they can be integrated with other tools.

The principle of combining these three complementary components should enable changes to be made to an individual component without invalidating the approach. For example, if new research offers an alternative mathematical model with more powerful analytical capabilities, it would be highly desirable to replace or augment the existing model with the new model.

The requirements for each constituent part are now considered in more detail.

3.1.1 Requirements of the Method

3.1.1.1 Clear, Concise and Complete Specification

The method must be easy to learn and apply to help the analyst produce a specification which is clear, concise and complete. It should be problem oriented, being applicable to manufacturing and batch applications and oriented towards the early project activities of requirements analysis and design. Structured methods and object oriented methods have had some success in these regards but are seen as disjoint. However, in comparing structured methods with object orientation, Booch draws attention to the question of which is the correct approach: algorithmic decomposition (structured method) or object-oriented decomposition [44]. He asserts that both views are important. The algorithmic view highlights the ordering of events and the object-oriented view emphasises the agents involved. The author would also assert that the structured method focuses on the required coordination of the agents. Object orientation is particularly associated with the promotion of re-use.

The requirements and functional design specifications typically need to be clear, concise and complete to a range of disciplines and organisations, including:

- Process and operations personnel considering its control and operability aspects.
- Strategic development concerned with longer term supportability, maintainability and flexibility, such as through the adoption of modular automation.
- Systems integrators and control system vendors who are typically contractually bound to the functional design specification.
- Information systems analysts concerned with integrating manufacturing systems with the corporate systems to gain commercial competitive advantage.

3.1.1.2 Manage Complexity

Even with a relatively small application, the asynchronous behaviour of the equipment being controlled results in substantial inherent complexity. According to Dijkstra, "the technique of mastering complexity has been known since ancient times: divide et impera (divide and rule)" [48]. This suggests the design of a software system should be decomposed into smaller parts, each to be independently refined.

The management of complexity is essential in the early project phases to enable multi-disciplinary teams to review coherent but manageable portions of specification. However, it may be vital to understanding a reported malfunction during operational support by staff other than the original developers.

3.1.1.3 Notations

Whereas a typical control system specification currently uses text as the main expression of requirement or implementation, with diagrams as aids to understanding, the use of a formal visual description relegates the text to the role of supplemental information and, with each graphical construct given a precise meaning, is neither vague nor ambiguous [5]. However, the notations must not be an obstacle to adoption of the method. They should therefore be expressive, well understood, in common usage and must be oriented towards event and sequence definition in order to be close to the analysts mind-set [47] and to facilitate effective communication between analyst and end-users and between developers [49]. Furthermore the notations should make coupling between modules explicit in order to facilitate effective re-use.

3.1.1.4 Coherent Information

Related information should be expressed together rather than fragmented. A criticism of some diagrammatic approaches is that related information is spread over many different pages. This may be unavoidable when using a generalised method which must be capable of tackling very large scale applications but is less justifiable when the scope is for small to medium scale applications.

Although software tools may support hyperlinks between related information, coherent printed documentation is necessary for review purposes and contractual reference. Operational support may be provided by personnel without access to the software tools, relying entirely on printed documentation.

3.1.1.5 Avoidance of Design Errors

In producing a clear, concise and complete specification, the analyst should be guided to adopt a structure such that design errors are avoided rather than trapped later by analytical or executable verification. Such a structure may particularly benefit the types of error which would typically be found late in integration testing. It would be unreasonable to

expect that the method could prevent any errors being introduced and consequently should support the identification of such errors by the CASE tool.

There must also be a simple mapping to the mathematical model, as the model provides the basis for verifying correctness of design. The method should therefore have a formal basis, rather than the semi-formal nature of many structured and object-oriented methods.

3.1.2 Requirements of the Mathematical Model

3.1.2.1 Visibility

In order to exploit the benefits of a mathematical model without the obstacle of the perceived learning curve, the casual user should be able to verify the design without needing to understand or directly interact with the model.

There must be a simple mapping from the notations used in the specification to the mathematical model. Support for concurrent threads of control is therefore necessary. Although one view of the purpose of the model is to support the method's notations, an alternative view is that the notations provide a user-friendly front-end to the model, as expressed in [50]:

To be able to apply Petri nets to practical work, software engineers need a means to specify their concepts at a more abstract level with a set of easy-to-use descriptive constructs.

3.1.2.2 Graphical Representation

To provide the necessary level of confidence in design verification and code generation to the more inquisitive user, the model must be established, comprehensible to a typical user and provide clear traceability from the analyst's specification. It should therefore have a non-mathematical representation, preferably using a graphical notation.

3.1.2.3 Ability to Execute and Analyse

For verifying the correctness of the design, the model should be primarily executable but should also be able to support behavioural property querying, such as deadlock and state searches. Users tend to lack confidence in their ability to specify a complete set of desired

and unwanted behaviours whereas an executable model is analagous to an operational control system. Execution of the model and the results of property queries should be presented in familiar terms to the analyst, necessitating references to the original specification. Whilst thorough verification of individual components is important, the ability to test the complete system may reveal errors which would otherwise be found late in integration testing.

3.1.2.4 Ease of Code Generation

In addition to a simple relationship from specification to model, it should also be straightforward to translate the model into a variety of languages for different target platforms with built-in diagnostics. Although a form of code output which was unintelligible to humans would guarantee that the implementation was derived from the specification by preventing modifications of the generated code, the target platforms typically require human-readable code and this also offers traceability from the user specification.

3.1.3 Requirements of the CASE Tool

3.1.3.1 Method Support

Good method support is very important to guide the analyst in applying the method and hence delivering the benefits which the method offers. The tool must hide “technical” concerns whilst allowing full expressivity to capture and represent important information [51]. Whilst many software tools claim to be CASE tools, many offer only programming functionality or are drawing packages with simple consistency checking. Others claim to be multi-method but are limited in their support for a particular method.

3.1.3.2 Usability

To help the analyst apply the method, with minimal training requirements, the tool must offer a user interface which is both familiar and comfortable.

3.1.3.3 Integrated Development Environment

To overcome the discontinuities associated with functionality gaps and overlaps between disparate tools used for specification, simulation, programming and support, the tool should provide an integrated environment supporting the mathematical model and also support connectivity with complementary tools. For widespread adoption by the target user community, it needs to be inexpensive.

3.1.3.4 Rapid Prototyping and Visualisation

Rapid prototyping should be supported such that a potential solution can be verified [39]:

- Perform some analyses automatically for certain classes of design error.
- Deduce performance capabilities.
- Support interaction with the developer for further analysis.
- Provide simulation/animation of the proposed solution, preferably graphically and with the ability to support what-if analyses [52]. This also benefits multi-disciplinary team review and enables the effects of suggested changes to be quickly and effectively evaluated.

The tool should be capable of representing the execution of the model, and the results of property queries, with reference to the analyst's specification. In particular, the tool should be capable of animating the diagrams which the analyst specified. In order to support alternative complementary visualisations, connectivity to other tools is required, such as 3D modellers and process-oriented mimic graphics. This can help prevent saturation with one particular view by offering a different perspective on the system.

3.1.3.5 Documentation

For contractual and reference purposes, the tool must be able to generate comprehensive documentation including the analyst's specification and a definition of the model and traceability between the two.

3.1.3.6 Automatic Code Generation

The author considers automatic code generation to be essential. Commercially, the target user-base is familiar with programming tools but less so with design tools, the output of which must then be manually translated into source code. In addition, there will be users who focus on the implementation effort and for whom the benefit of the tool may be perceived as a documentation aid. The absence of code generation would therefore be a significant obstacle to successful exploitation. More positively, automatic generation of code substantially reduces implementation and corresponding test elapsed time, effort and cost. Confidence in the implementation may also be increased. Maintainability is improved because a consistent style is used, rather than a different design architecture for

different applications. The code should include diagnostics which help accurate reporting of fault symptoms and in troubleshooting plant equipment malfunction. This can help to shorten commissioning periods and minimise plant downtime when in beneficial operation. The code should also facilitate links with external systems, such as corporate systems, so that business statistics may be computed.

Three popular categories of implementation environment which should be supported are:

- PLC running software written in relay ladder logic.
- Real-time executive running scan-based or interrupt-driven software, typically compiled from ANSI C.
- Distributed intelligent nodes, such as Echelon's LonWorks fieldbus architecture with the software for each node written in Neuron C – a modified form of ANSI C.

3.2 Opportunity

The established knowledge, techniques and technology which offer the potential to meet these requirements include:

- The modelling principles of object orientation have become firmly established in the IT sector and are being introduced to the control systems community. Many control system development environments support function block programming, which may be considered as a first step towards object orientation.
- The widespread adoption of the Microsoft Windows graphical user interface increases the acceptability and the shortens the learning curve of a new tool by offering a familiar look and feel.
- Almost universal conformance with Microsoft Windows-based communication mechanisms by PC-hosted tools. This facilitates integration of otherwise disparate tools into an integrated environment.
- The PC is a de facto standard desktop computing environment which is low cost but very powerful in terms of processing speed and memory capacity, offering the computing power to support interactive analyses of a mathematical model.
- Widespread evidence of the application of Petri nets to manufacturing and process industries demonstrating their relevance and value to this problem domain.

To summarise, the essence of this research is to build on what has been achieved to date in order to exploit its value rather than to further academic research knowledge regarding a particular mathematical modelling method or graphical method. As shown graphically in figure 13, which is a copy of figure 1 in chapter one, Introduction, the goal is to apply the combined power of a graphical method, mathematical model and CASE tool in a manner to which an industrial user can relate:

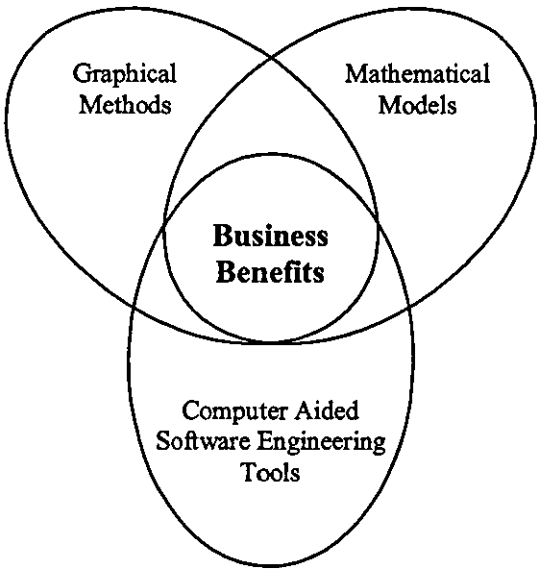


Figure 13 The Goal of This Research

Chapter 4 Control System Technology

This chapter examines the development and run-time technology typically used to implement control system solutions. Trends are identified and deficiencies in meeting the business needs described in chapter two, The Need, are justified.

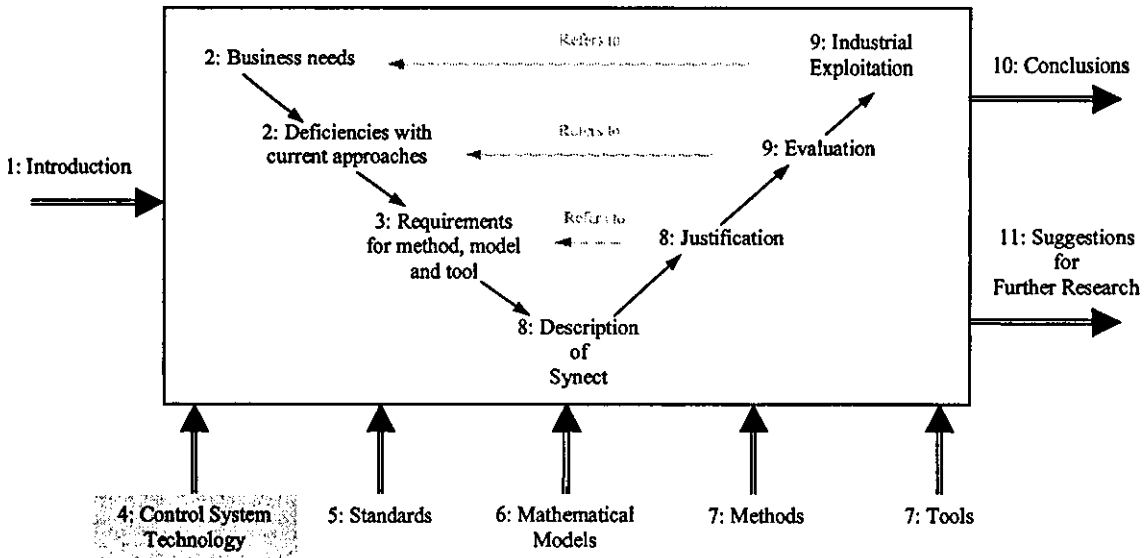


Figure 14 Relationship Between This Chapter And The Thesis Map

As shown in figure 14, the available control system technology influences all aspects of the derivation of requirements for a new approach and the delivery of a solution. Weaknesses with current development environments help determine the requirements for a new approach and the run-time technology will still be required as the target implementation platform for a new approach:

- The development environment refers to the tools and methods which are used to specify and implement the system. These are often closely related to the target platform – the software development for, and configuration of, most platforms relies on software tools which are either part of the platform or sold by the same vendor.
- The run-time technology can be considered to consist of:
 - Computing hardware.
 - Software environment (such as an operating system).
 - Interfaces to plant instrumentation, sensors and actuators.

Considering the dependencies between computing hardware, run-time software environment and development environment to the type of control system platform, these will be described by examining the types of control system platform and then the plant interfaces, namely in the order:

- PLC.
- DCS.
- Industrial computer (including embedded programmable devices such as microcontrollers and single board microprocessors).
- Plant interfaces.

4.1 PLC

4.1.1 Background

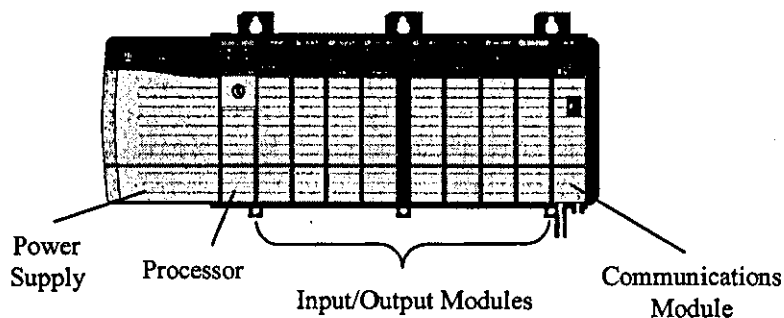


Figure 15 Modular Hardware Structure of a PLC (source: Rockwell Automation)

The programmable controller, or programmable logic controller according to the Allen-Bradley trademark, was originally designed to replace relay panels used in the automotive industry for sequencing of production machinery. Whereas relay panels were, at best, slow and costly to reconfigure, the programmable controller was developed to offer a robust, industrially rugged choice of hardware modules which could be re-programmed using a notation with which electrical and maintenance staff were already familiar. Figure 15 shows a modern example of modular PLC hardware, with the rack containing a power supply, processor and an application-dependent selection of input, output and communications modules.

The dominant programming language in use on PLCs is relay ladder logic. Other programming languages are reviewed in chapter five, Standards. This section examines a number of deficiencies in the traditional use of relay ladder logic for industrial sequence control. Examples have been produced with Rockwell Automation's ladder logic programming software (Rockwell Automation, Allen-Bradley's parent company, has over 50% share of the North American PLC market [53]).

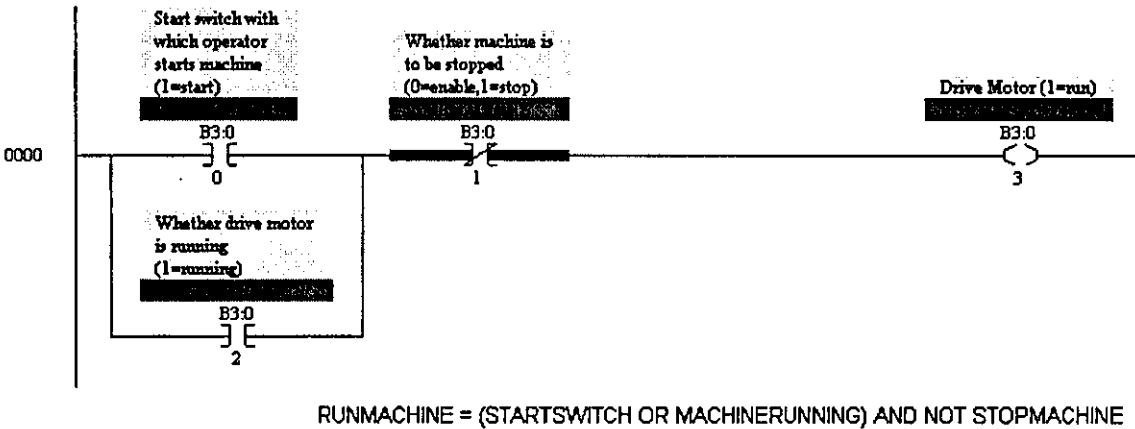


Figure 16 An example of relay ladder logic and its boolean equivalent

Figure 16 is an example of relay ladder logic with equivalent boolean logic in text form. The vertical lines at each side of the diagram represent power rails. Following the model of an electrical circuit, the contacts towards the left of a rung are equivalent to switches and, if all made, supply power to the coil towards the right of the rung. Serially linked contacts represent AND operators and parallel branches represent OR operators.

The programming notation of relay ladder logic allowed simple expressions of combinational logic to be specified in a graphical form, using symbols and terminology such as contacts and coils. Not only was the device programmed in this way, it also modelled the relay panel in terms of providing diagnostic information about the state of the contacts and coils and facilitated the "forcing" of contacts and coils, again mimicking the relay panel. It is interesting that a development which has had such a profound effect on automation started as an incremental evolution from the status quo, i.e. for the end-users of the technology, the change was a small step rather than a gigantic leap. The types of application for which this technology was appropriate were:

- Discrete i.e. no analogue
- Intuitively considered as combinational logic for which an interlock paradigm, such as relay ladder logic, was highly suitable.

Over time, the capability of the PLC hardware increased to offer more memory, faster execution and support for analogue inputs and outputs, high speed counters, communication modules and more. The relay ladder logic instruction set was expanded to handle the new types of hardware module and also to be able to manipulate analogue data. As ladder programs grew larger because bigger applications were tackled, the ability to support software engineering principles of modularisation of the software into subroutines was added. More complex applications were tackled, including multiple concurrent threads of control with alternative branches through the sequence logic.

Today's PLCs provide a rich instruction set and programming tools which typically run on desktop PCs to provide a user-friendly development environment. The trend is also towards smaller PLCs which can be networked together.

4.1.2 Weaknesses

As more complex applications have been tackled, the limitations of current approaches using ladder logic have become evident. These include the lack of an explicit representation of data structures or entity relationships and weak support for algorithmic functionality [54]. Of particular relevance to this thesis, RLL is a poor representation for expressing sequence logic. Indeed, it can be impossible to tell what the intended sequence behaviour is without a knowledge of the equipment to which the PLC is connected. The combinational logic approach leads to a very compact specification and memory-efficient solution but is extremely difficult to maintain and does not provide the level of diagnostics which are typically desired - the control system is often required to assist in the diagnosis of fault conditions in the equipment.

An example of ladder logic will now be considered to demonstrate some typical weaknesses.

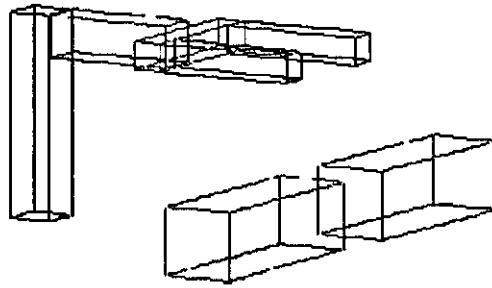


Figure 17 Example Application

In the example application in figure 17, the robot arm moves vertically between the two limit switches LIMIT_SWITCH_ARM_HI (on PLC digital input I:1/0) and LIMIT_SWITCH_ARM_LO (on PLC digital input I:1/1). There are two outputs to the arm motor O:2/0 and O:2/1 as follows:

O:2/0	O:2/1	Arm Movement
0	0	Remains stationary
0	1	Arm raises
1	0	Arm lowers
1	1	Remains stationary

Similar logic exists for the gripper.

Consider a requirement to sequence these devices as follows:

Lower the robot arm and then close the gripper. When the gripper is closed, raise the robot arm.

Figure 18 shows a combinational logic solution expressed in ladder logic to behave as required:

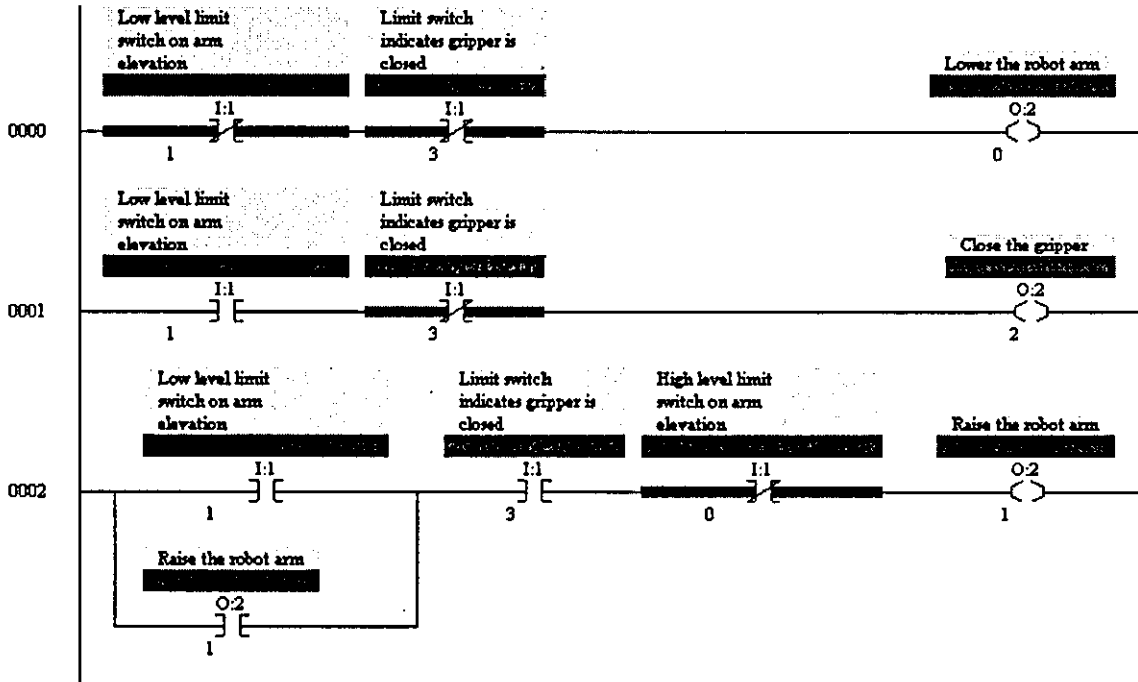


Figure 18 Partial Combinational Logic Solution

Where is the sequence definition? It is partly in the ladder logic and partly in the equipment being controlled. Because the control scheme relies on the plant equipment behaving as required, it cannot trap illegal behaviour or sensor malfunction. Consider the behaviour if the gripper closed sensor fails as the arm is being lifted. The effect is that the GRIPPER_CLOSED contact in rung 2 will prevent power flowing to the RAISE_ARM coil so the arm will not continue upwards. Because the gripper is not closed, power will flow through rung 0 to the LOWER_ARM coil. So the result is that the arm will lower, although this is likely to be undesirable behaviour and is non-obvious in the ladder logic implementation.

The sequence logic requirements for flexible manufacturing systems are more complex. In the above example, the gripper can be assumed to be empty whilst the arm is lowering. In a flexible manufacturing system, however, the robot may be required to retrieve the part for manufacture and then return it later – lowering the arm with the gripper empty on one occasion but holding the manufactured part on the other. This would typically be implemented using a flag to store which mode the robot was in. However, as the requirements grow more complex, more flags are added, some of which are mutually exclusive although, again, this is not apparent from the ladder logic program. It is only with

a good knowledge of the application that this is apparent. At a later stage, if a modification is requested to the program, it is very easy to incorrectly manipulate one of the flags which can cause faulty behaviour under particular circumstances. Alternatively, in the case of a system with decision logic and alternative branches, it is possible that an error could exist in the logic for a long period before a particular sequence of events occurs which causes the system to exhibit faulty behaviour. When that happens, it can be virtually impossible to deduce the root cause of the faulty behaviour.

An additional problem with a combinational logic implementation relates to the use of timers in dependency relationships. For example, a timer might be used to defeat an interlock to allow an actuator time to move away from its end-stop. This example of accidental complexity [39] makes the control intent more difficult to decode and can lead to system malfunction if the type of actuator is replaced, for example during a debottlenecking exercise.

The conclusion of the above discussion is that a combinational logic solution, as encouraged by the ladder logic paradigm, is acceptable for applications which are naturally expressed in terms of permissives or interlocks, but becomes less appropriate as the requirement increases in complexity from a simple repetitive sequencing application to one with more flexibility.

The combinational logic solution could be considered to be a special case of a state-based solution, where transitions are defined from every state to every other state. Usually, the state machine will only define a subset of these transitions so that an illegal transition from one state to another cannot occur. Although the state-based implementation is likely to use more code than a combinational solution for the equivalent functionality, the state-based solution runs faster because the first contact on each rung relates to the current state and there can only be one state active at a time [55]. If a rule-driven process is used to transform the state-based specification into ladder logic, the development effort is moved from implementation to analysis and design.

Ladder logic can be coded to more closely represent the required sequence logic. For example, the implementation of the gripper logic could take the form shown in figure 19, where each state is assigned a unique number and a register stores the current state:

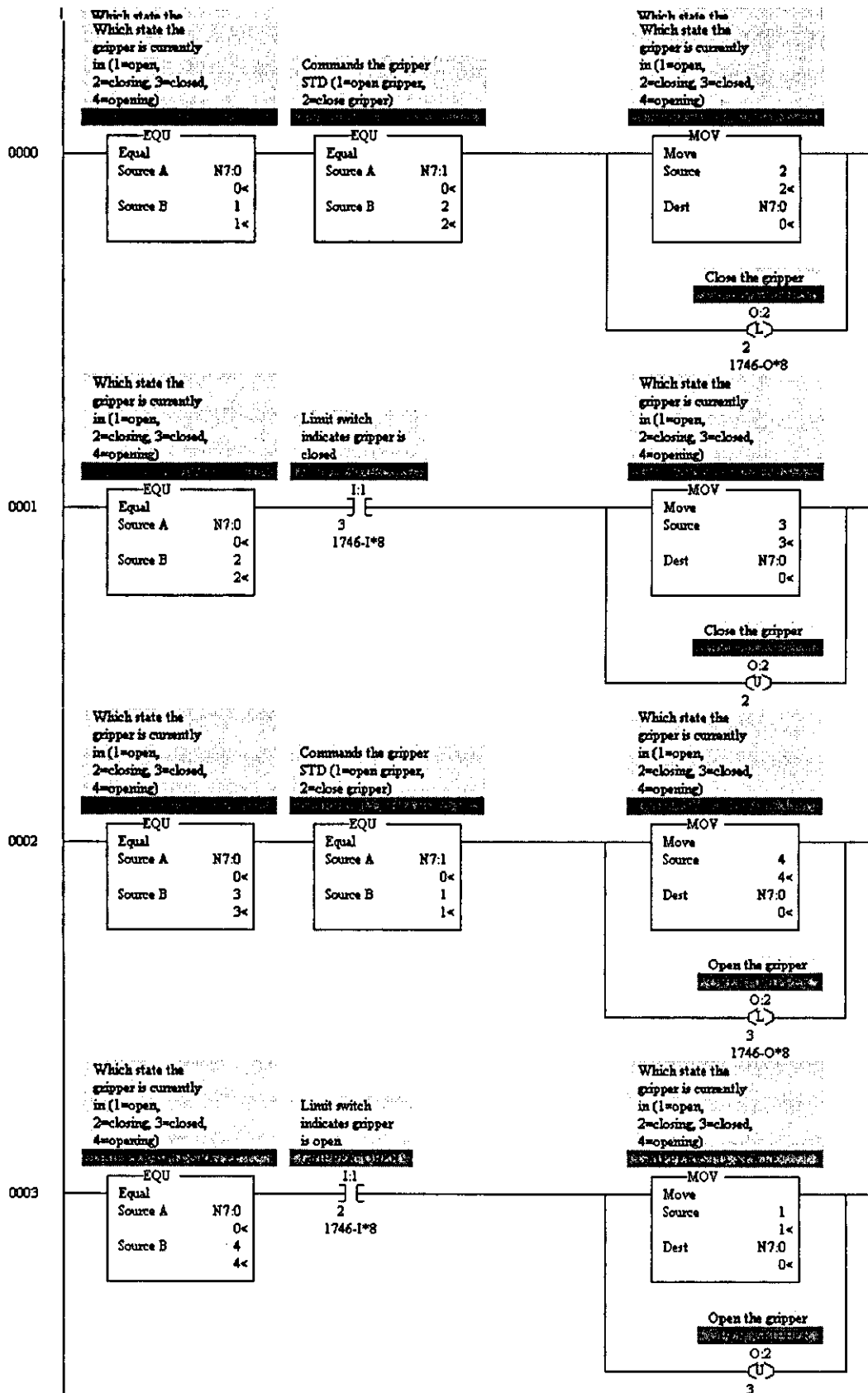


Figure 19 Sequence Logic Implementation of the Example Requirement

This implementation is likely to be more maintainable and less prone to anomalous behaviour because it follows the form of the specification. However, unless an automatic code generator is used, a manual transformation of the specification into code is still required, leading to the possibility of misinterpretation or translation errors. A more appropriate run-time environment would execute the specification directly. Such a specification could be expressed diagrammatically using state diagrams or sequential function charts (see discussion of IEC 61131-3 in chapter five, Standards).

Whilst state diagrams and sequential function charts offer a more intuitive modelling paradigm for sequence definition, there are many applications coded entirely in ladder logic. It should also be noted that many smaller PLCs, such as Rockwell's SLC range, only support ladder logic programming. Although relatively small compared with other PLCs, these smaller PLCs are nonetheless capable of tackling complex applications. For legacy programs, an automated design recovery algorithm may be required to convert the ladder logic into sequential function charts [56].

To generalise, the problem domain is one of software rather than hardware. The software engineering principles of modularity, low coupling and high cohesion are highly relevant to ladder logic implementations [57]. Unfortunately, the language does not guide the developer towards such a structure and, in some variants, can be an obstacle.

Although the demise of relay ladder logic has been routinely predicted over the years, it is still the language of choice for many industrial solutions due to the simplicity of its instruction set and constructs and for its track record with maintenance personnel [58]. Both of these points have a counter argument, however:

- Whilst primitive instructions and constructs (the graphical representation of "and" and "or" and the use of rungs for representing parallelism) ensure a rapid assimilation of the localised functionality over a few rungs of ladder, the absence of more abstract and functionally expressive constructs means that any overall structure is less visible. Combined with the absence of explicit coupling information, showing where or how else a particular variable is used, can lead to a localised code modification impacting on other unforeseen areas of the software.
- History has many examples of techniques being adequate for a particular purpose and, consequently, lowering the motivation for better solutions [59]. Whilst ladder logic enables maintenance personnel to see the state of plant inputs, plant outputs and internal memory variables, a more comprehensive approach would offer higher

level diagnostics, such as deducing that a particular sensor had failed. In fact, many of the claimed maintenance benefits of ladder logic may be more associated with the capabilities of the programming environment than the notation itself. The ability of the PLC programming environment to monitor and force plant inputs and outputs is considered essential although the inability to take a coherent snapshot or log a historical sequence of events is a more significant limitation when debugging sequence logic compared with combinational logic.

4.1.3 Trends

Although a mature concept, the PLC still has a role in future industrial automation solutions:

- The range of application which can be addressed using PLC technology is widening. Smaller and cheaper PLCs are becoming available, such as Siemens Logo [60].
- Partly in response from the automotive automation market, vendors are offering open modular architecture controllers (OMACs) [61]. These are typically PC-compatible processors which can be programmed in C, such as Rockwell's OpenController [28], Siemens M7 [62] and Foxboro's Micro I/A [63].
- Whereas PLC vendors started in the discrete sector, they are now competing directly with DCSs for hybrid applications, such as batch control, which contain both analogue and discrete control requirements [64].

A market research survey in 1997 predicted that the European PLC market would grow by 20% between 1997 and 2002 [65]. However, this market is under pressure from softlogic – PLC functionality on a PC. Indeed, Honeywell predict that discrete manufacturing systems will be increasingly controlled by PCs rather than PLCs, with PCs taking 50% of the market by the year 2000 [66].

4.2 Distributed Control Systems

4.2.1 Implementation of Sequence Logic

Distributed Control Systems (DCSs) are widely used in the process industries. Whereas the PLC grew out of the discrete equipment control market place, DCSs were developed to satisfy continuous control requirements. In particular, the early DCSs had a separate processor for each PID loop (hence the term distributed control). Sequencing capabilities have since been added to facilitate their application to batch processes, using a variety of notations including:

Type of DCS	Sequence Language
Honeywell TDC 3000	Text language (Honeywell CL)
Yokogawa Centum	Sequence tables
Moore APACS	Sequential function chart and relay ladder logic
Fisher Delta V	Sequential function chart and relay ladder logic

Table 4 Sequence Language Available in Popular DCSs

Historically, a DCS application involved more configuration of vendor-supplied functionality whereas a PLC application would require more programming. DCSs are usually supplied with a library of "function blocks" which are "software wired" together to define continuous control transforms and interlock functions.

4.2.2 Comparison with PLC Solutions

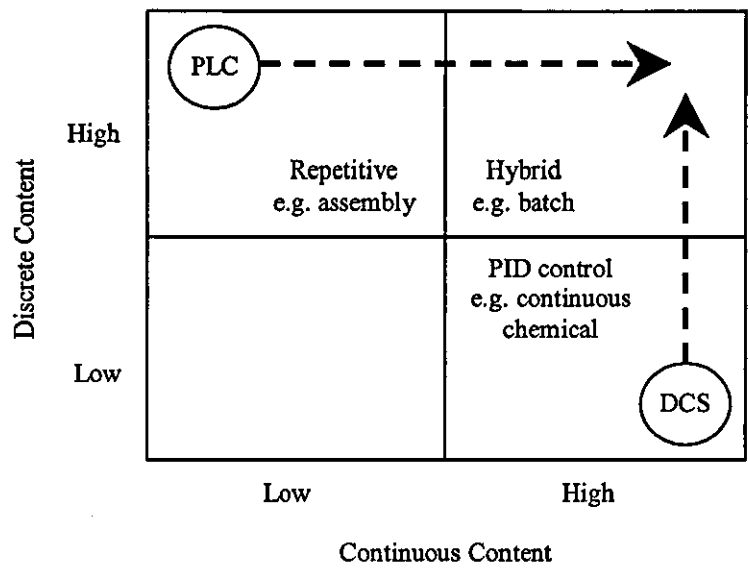


Figure 20 Trends in PLC and DCS Capability

Figure 20 shows that although the PLC and DCS have different origins, their capabilities have increased such that both types of solution are now capable of automating batch processes with batch applications accounting for 31.5% of the 1997 European DCS market [67]. DCS and PLC platform technology is therefore converging with commercial considerations being a greater differentiator. The DCS vendor tends to sell the whole development environment and target platform including human machine interface (HMI) whereas the PLC vendors sell a ruggedised panel or SCADA as the HMI. Another important difference, commercially, is that the DCS vendor tends to sell the whole package including configuration whereas the PLC vendor sells via system integrators.

Whereas the vendors of PLC technology and SCADA vendors have promoted the use of Microsoft Windows as a basis for viable control systems for a number of years, this was resisted by DCS vendors until recently. In the last few years, however, the major DCS vendors have also launched Windows offerings [68], [69].

Although batch offerings are converging on the S88.01 standard, significant sequence logic is required in the phase definitions, leading to similar problems confirming that it will behave as intended.

In summary:

- DCSs are relevant to this thesis because they are widely used to implement batch process automation which contains significant sequence logic.
- Although there is more configuration rather than programming compared with a PLC solution, many of the characteristics associated with the development of industrial sequence logic are similar to PLC solutions, with correspondingly similar weaknesses.
- The DCS is a relevant target platform for a solution generated by a new method and software tool, which is the subject of this thesis. The support for Microsoft Windows connectivity and editing techniques, such as cut and paste, facilitate the integration of DCS configuration tools with a third party software tool.

4.3 Industrial Computer

Although PLCs and DCSs dominate as target implementation platforms, there have been many different types of industrial computer platforms with development environments more oriented at the traditional software engineer. Some of these have been proprietary whilst others have facilitated integration of many different vendors hardware:

- Digital Equipment Corporation's PDP range of computers were particularly popular in process industries. Digital offered the RSX operating system but UNIX and third party operating systems were also used.
- In contrast to the proprietary offering, VME-based systems have enabled control systems developers to build a system from many different vendors hardware. There are many examples of VME-based control systems running the OS/9 multi-tasking operating system with applications developed in C.

The recent promotion of "softlogic" is increasingly relevant. Softlogic is the term being used to describe the implementation of PLC functionality in a PC. It is estimated that there are more than 800,000 PCs in use in UK manufacturing industry which is an average of one PC for every six people [70]. American research organisation ARC believes this acceptance of PC-based technology has advanced the adoption of PC based logic control software as an alternative to conventional PLC hardware. They forecast a doubling of sales in 1997 compared with 1996 and an exponential growth in sales over the following

four years [71]. Softlogic is being promoted by both traditional control system vendors and packaged software suppliers:

- ASAP is supplying Wonderware and GE Fanuc with its Windows NT SoftLogic product. GE Fanuc will be incorporating it into their Cimplicity SCADA software [72].
- Rockwell Automation's SoftLogix product is based initially on an enhanced PLC5 instruction set [73].
- Intellution believe that SoftLogic such as their Paradym-31 on a Windows NT machine will soon start to take over from PLCs on the factory floor [74].

Even without the real-time element, PC technology is increasingly in use in a supervisory capacity, either through custom SCADA configuration or, in the batch control industry, as the batch execution engine. S88.01-aware products such as OpenBatch [75] and InBatch [76] run on the Windows NT operating system.

In summary:

- The development environment is typically that of general purpose programming language software rather than offering particular support for the development of concurrent sequence logic.
- Although less popular than PLCs and DCSs, automation solutions hosted on industrial computers represent an implementation platform which should be supported by a new method and tool. In particular, Microsoft Windows hosted applications are increasingly relevant as the human machine interface and as the logic engine.

4.4 Fieldbus

Fieldbus is the generic name given to the use of digital communication technology for plant monitoring and control [77]. It carries an implication of greater intelligence in the instrumentation, relieving the burden on the main processor. This has a significant impact on the control system software architecture and hence on its analysis and design. With reference to a PLC solution, the trend has been as follows:

- 1) Traditional applications would wire each plant device individually to the I/O module in the PLC rack. In order to detect a fleeting signal from the device, a maximum processor scan-time must not be exceeded. As the complexity or size of the

application increases, a faster PLC is required or segmentation of the program such that some modules are not executed each scan.

- 2) To reduce wiring, the devices could be connected to the PLC via a network with each node allocated a unique address. The device might be limited in computation capability to transforming the raw signal into engineering units.
- 3) The device could be capable of executing continuous control, such as 3-term control loops.
- 4) Devices can exchange data independently of the PLC.
- 5) Devices have intelligence such that the PLC is no longer necessary. A fleeting signal is captured by the local processor and information transmitted to other relevant nodes. These are sometimes referred to as control networks and a thorough description of the differences, application areas and popular systems is provided in [78]. Two important implications of the use of the distributed intelligence approach are:
 - The computation power required on each node may be small. So a small amount of local intelligence removes the bottleneck of a central processor.
 - Network traffic is substantially reduced because information is only transmitted when a significant change has occurred.

The author's perception of the relative impact of the above is shown in figure 21.

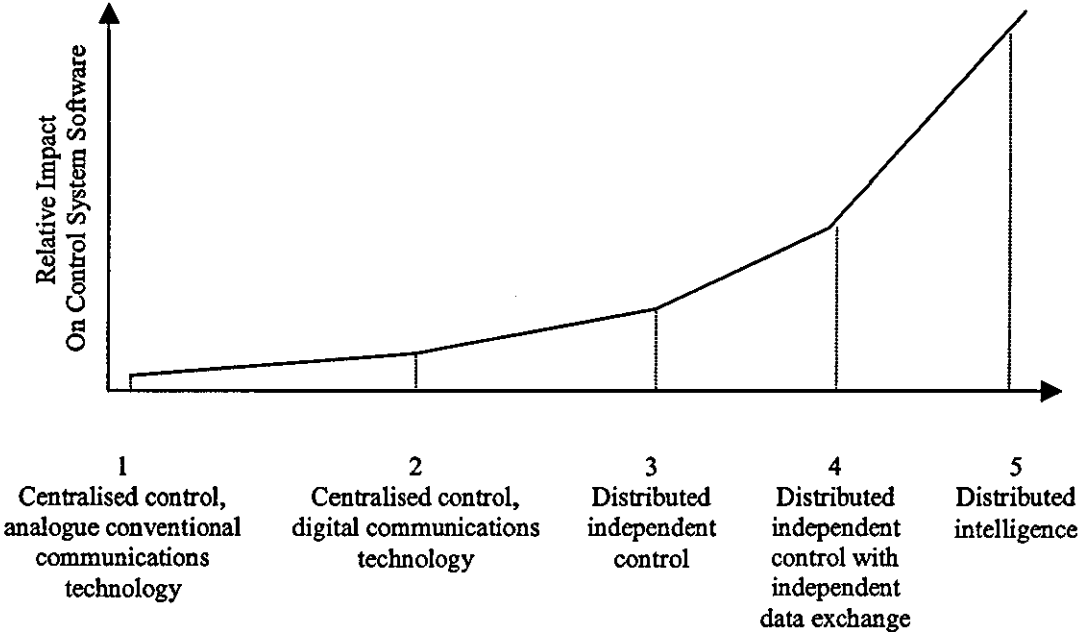


Figure 21 The Evolution of Digital Communications Technology and its Impact

Two independent surveys predicted realistic savings of 25% using fieldbus technology instead of conventional 4-20mA loops [79]. Other studies from the Fieldbus Foundation suggest a 78% reduction in field wiring, a 46% reduction in equipment costs and a 25% reduction in the person hours for system verification and configuration [77]. The process industries are considered to be lagging behind manufacturing industry in utilising fieldbus, possibly due to major plant re-instrumentation occurring typically every 10 years in process industry compared with every 5 years in manufacturing industry [79].

A transmissions plant in Melbourne, Australia was automated on one line using a PLC with conventional wiring and on the other using Honeywell's Smart Distributed System (SDS) approach [80]. The costs are shown in figure 22 below.

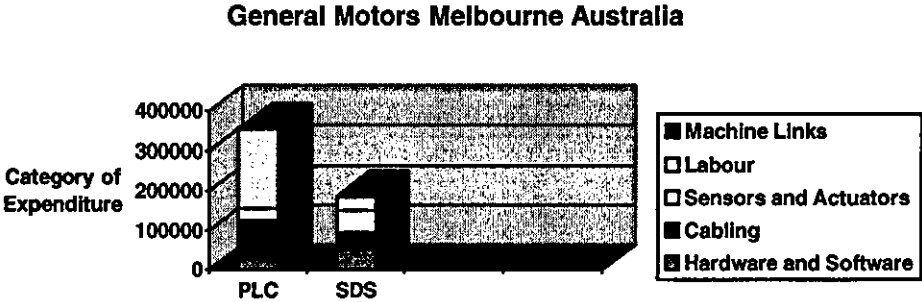


Figure 22 Comparison of PLC and SDS Implementations

4.4.1 Echelon LonWorks

In terms of the categories shown in figure 21, Echelon's LonWorks is an example of distributed intelligence, being a low-cost bus system using peer to peer communication between asynchronous computing nodes. Each node on the network contains a Neuron chip which incorporates three processors. Two processors are for network communications and the third is user-programmable in a modified form of C for applications functionality [81]. The essential difference between LonWorks and other fieldbusses is the integration of distributed intelligence on location in the controller through the application CPU [82].

LonWorks is data driven rather than command driven. For example, a temperature sensor could make its temperature available on the network but without knowing which other devices will use it or what they will do with the information [83]. Whereas a typical PLC will spend most of its processor time scanning I/O which has not changed since the previous scan, a LonWorks node waits to be notified that an event has occurred or a data value has changed and then reacts accordingly.

The LonWorks architecture is intuitively straightforward for continuous data applications but sequence logic applications require additional care. LonWorks is oriented at wide, flat architectures with any node being able to communicate with any other node. However, the absence of an explicit hierarchy and fragmentation of the sequence logic across

several processing nodes require an extra degree of discipline. Well-defined coordination and cooperation mechanisms must be specified and adhered to.

4.5 Summary

- Although control system technology is improving and converging, vendors are still offering proprietary tools which are highly implementation-oriented rather than problem-oriented and consequently offer minimal support in satisfying the business needs identified:
 - They offer minimal ability to support requirements capture, rapid application development or verification of correctness, either through analytical methods or simulation.
 - They tend to be intimately coupled with the target platform.
- Implementation languages are moving towards IEC 61131-3 but there is limited standardisation of analysis and design models (S88.01 may be one of the better examples in this regard). Consequently, the ability to maintain or re-use existing functionality, even when implemented using standard languages, is hindered.
- Whilst relay ladder logic is unlikely to be the language of choice for software developers, end-user demand ensures that it cannot be ignored. Although relay ladder logic is oriented at combinational logic, more robust state-based implementations are possible.
- Including a PLC, industrial computer and Echelon LonWorks as target platforms for a new method and tool would cover the major types of implementation environment.

Table 5 summarises the relevant characteristics of the target environments discussed:

Characteristic	PLC	DCS	Industrial Computer	LonWorks
Strongest application type	Simple repetitive discrete	Continuous	Computationally intensive	Low-cost distributed
Proprietary software development environment	Yes	Yes	No	Yes
Primary programming paradigm	Relay ladder logic	Configuration and software wiring of function blocks	3GL language development	Neuron C
Typical sequence programming language	Relay ladder logic, sequential function chart	Relay ladder logic, sequential function chart, text language, state table	C	Neuron C

Table 5 Summary of Target Environments

Chapter 5 Standards

This chapter describes relevant standards (including de-facto standards) influencing sequence control in industrial automation. Although many device-level and plant-level communications standards exist, they are not considered in this section because their relevance has already been discussed in section 4.4, Fieldbus. The focus is therefore on the development environment and includes:

- Programming.
- Design methods.
- Communications.

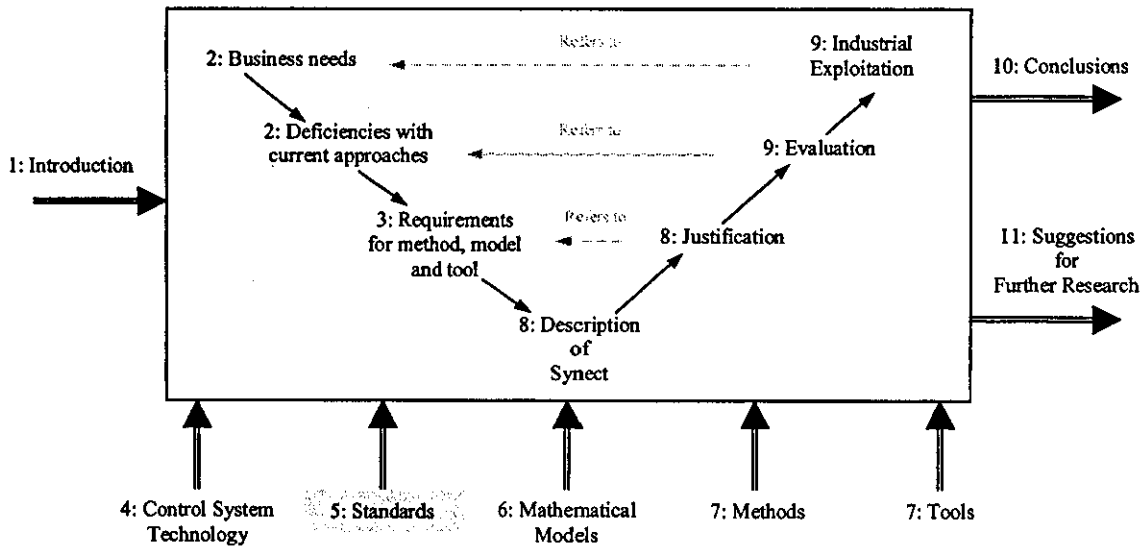


Figure 23 Relationship Between This Chapter And The Thesis Map

As depicted in figure 23, standards influence many aspects of this thesis. Relevant standards are reviewed, identifying their limitations but also their contribution in exploiting available technology and maximising user acceptability.

5.1 Programming

5.1.1 IEC 61131-3

IEC 61131-3 is the international standard for PLC programming, providing a reference software model, shown in figure 24, and covering the printed and displayed representation of programming languages for programmable controllers (the term PLC, for programmable logic controller, is a registered trademark belonging to Allen-Bradley). It also specifies the syntax and semantics of the languages [84], [32].

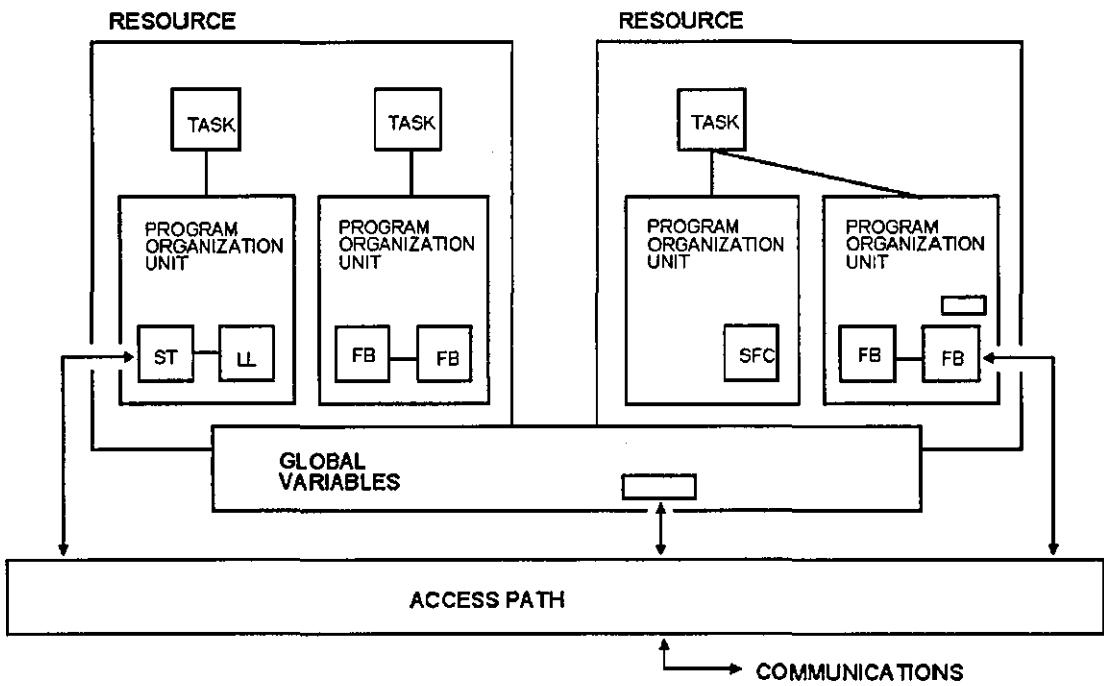


Figure 24 IEC 61131-3 Software Model of a Configuration (Source: Moore Products)

5.1.1.1 Languages

The IEC 61131-3 standard states that it specifies the syntax and semantics of two textual languages (instruction list and structured text) and two graphical languages (ladder diagram and function block diagram). Interestingly, sequential function charts are considered to be for the purpose of structuring the internal organisation of programmable controller programs and function blocks rather than a language in their own right [84]. Also, whilst it has been suggested that the use of function block language brings the

benefits of object oriented design to PLC applications [85], there are major omissions such as lack of support for a class hierarchy (such as “an air-fail closed valve is a type of valve”) or method invocation. This is not object orientation by Stroustrup's criteria [86].

Relay ladder logic has already been introduced in chapter three, Requirements for Method, Model and Tool, so the following examples cover the other three languages and sequential function charts.

```
START:      LD          grip_command  (* GRIPPER COMMAND *)
            JMPC        OPEN          (* COMMAND=OPEN *)
            RET          (* NOTHING TO DO THIS TIME *)
```

Figure 25 Example of Instruction List

Figure 25 shows an example of instruction list (IL), which is a low level language, similar to assembler and can be used to write tight, optimised code for performance critical operations [32].

```
IF grip_command THEN
    gripper_state:= OPENING ;
ELSE
    gripper_state := CLOSING ;
END_IF ;
```

Figure 26 Example of Structured Text

Structured text, shown in figure 26, is a high level language with strong data type checking and a formal syntax, similar to PASCAL

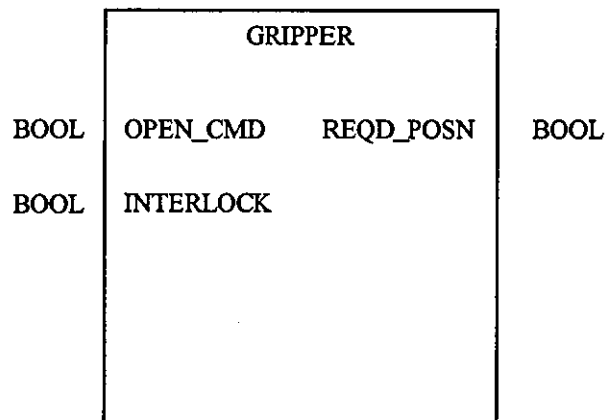


Figure 27 Example of a Function Block Declaration

Function blocks are declared as demonstrated in figure 27 and “wired” together in a function block diagram. Function blocks have been a popular approach used in process control and manufacturing, typically stored in libraries of proprietary controls [87]. National and international work on a standard function block notation, and work by the Fieldbus Foundation on the use of function blocks in process control based on fieldbus, has resulted in recommendations for incorporation into IEC 61131-3.

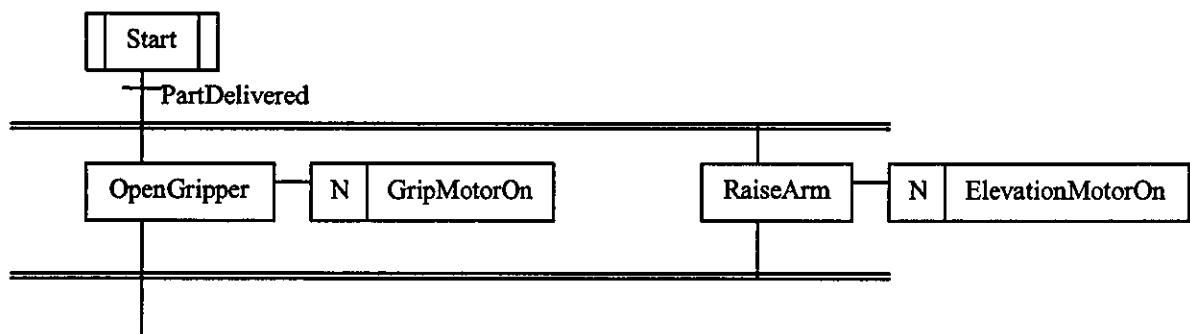


Figure 28 Example of Sequential Function Chart

As the name implies, sequential function charts (SFCs) were designed for implementing sequence logic. The example in figure 28 shows the diagrammatic representation of concurrent threads – the OpenGripper step and RaiseArm step would execute concurrently. The box to the right of the step name contains the actions associated with the step and the action qualifier – N in this example denoting that the action is to be performed while the step is active.

5.1.1.2 Industrial Relevance

The major PLC vendors have either adopted IEC 61131-3 in their programming software or claim to be moving towards compliance [88], [89], [90], [91]. However, for consistency with previous offerings and to differentiate themselves from competitors, vendors often offer functions and facilities over and above the demands of IEC 61131-3 [89]. This is clearly an obstacle to the goal of being able to port software directly from one vendor's PLC to another's [92]. Another obstacle is that the standard does not define configuration storage formats. So even if two vendors complied strictly with the standard, it is exceedingly unlikely that software could be ported from one programming environment to another.

There are examples of vendor neutral programming tools, such as IsaGraph [93] and CADEPA [94]. IsaGraph is an IEC 61131-3 based tool which includes code generation for PC-based industrial computers running real-time executives such as OS-9 and can also generate code to run on an Echelon LonWorks node's coprocessor.

PLCOpen is a vendor and product neutral organisation aiming to bring greater value to users of industrial control systems through the use of IEC 61131-3 [95]. PLCOpen

promotes the concept of a program support environment (PSE) to enable users to "move between different makes [of PLC] without training, exchanging applications with minimal effort" [96].

5.1.1.3 Limitations

Although IEC 61131-3 has many benefits, it has a limited model of communication which is a weakness when applying it to distributed systems [97]. This is being addressed by standard IEC 1499, currently in draft, which describes distributed function blocks [87], [98], separating event communication from continuous data communication.

Although the IEC 61131-3 SFC offers a clear diagrammatic representation of concurrent logic, the support for an object-based approach is weak:

- Hierarchical decomposition is supported whereby an action's logic can be expressed in another SFC. However, there may then be several low-level SFCs which manipulate a particular device, although the object-based paradigm would require that this functionality is encapsulated rather than fragmented.
- The SFC may be embedded in a function block which is connected to another function block encapsulating the control of the device. This approach requires event communication between function blocks which is not differentiated from continuous signal flows in the function block diagram notation. Compatible multi-valued variables must be defined and a handshaking mechanism adopted. This requires significant skill and therefore would not satisfy Booch's criteria for claiming support for an object-based approach [86].

None of the IEC 61131-3 languages could be considered to be state of the art, with the world-wide recognition or user base of C or C++ [99].

5.2 Design Methods

5.2.1 EDDI, STEPS and KRAUSE

EDDI (Error Dynamic Diagnostic Indicator) [100], STEPS (Structured Transfer-Machine EDDI Programming System) [101] and Krause [102] were introduced to impose structured programming on PLC applications implementing sequence logic requirements:

- Diagnostics are integrated with the control logic for faster and more reliable trouble-shooting (see below).
- A template-based approach is used to increase productivity (see below).
- The use of a skeleton application and rules for applying the method help to promote consistency between systems integrators, machine builders, etc..

These design methods have been successfully used in the automotive industry and other sectors, such as cement manufacture.

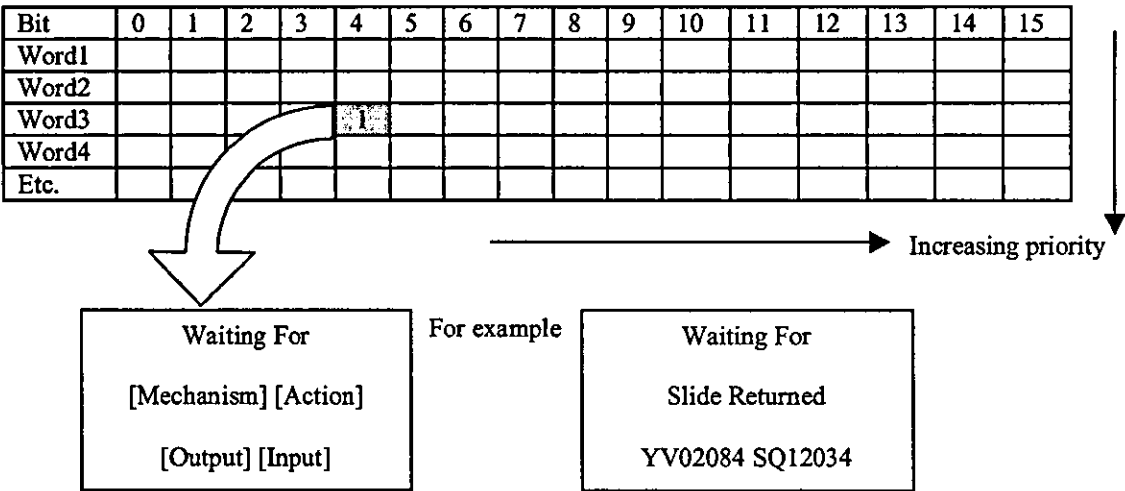


Figure 29 Example Matrix in STEPS

In STEPS, for example, machine control and diagnostics centre around a matrix, as shown in figure 29 above. Each cell in the matrix corresponds to a message. When an action occurs, the corresponding bit in the matrix is set, switching on the message and

inhibiting the machine from entering another step. When the action completes, the bit is turned off and the machine may step on. Several actions may occur in parallel but all bits in the matrix must be off for the machine to step on.

Figure 30 shows a populated example of the template used for specifying coordinated sequence behaviour. This machine consists of a transfer bar and two machine heads. Transfer bar steps 1 to 5 are performed with no head function being undertaken. At transfer bar step 6, heads 1 and 2 both begin executing step 1 of their respective sequence logic.

Transfer		Head 1		Head 2	
Step No.	Function	Step No.	Function	Step No.	Function
1	Unclamp				
2	Transfer return				
3	Check loader				
4	Raise transfer				
5	Clamp				
6	Cycle heads and await	1	Supports advance	1	Head engaged
		2	Supports clamp	2	Depth
		3	Head engaged		

Figure 30 Example of a STEPS Sequence Overview Diagram

5.2.2 S88.01

In 1989, the Instrument Society of America (ISA) established Standards and Practices committee number 88 (SP88) to produce a set of terms and models for batch control which would be applicable from the most complex to the simplest batch process, whether fully automated or entirely manually operated. In October 1995, the document *Batch Control, Part 1: Models and Terminology* was approved as document number *ANSI/ISA-S88.01-1995*.

S88.01 is therefore a relatively new standard offering a common terminology and model for batch control. One benefit of the standard is to focus analysis on the production of a highly flexible automation system. This ensures that capabilities of the plant can be fully utilised to meet market demands for variations of existing products and for the manufacture of additional products. The S88.01 analysis separates the modelling of the plant equipment from the definition of recipes.

In figure 31 below, control modules have been grouped into equipment modules as shown by the dotted lines. This results in phases ChargeIngredientA, ChargeIngredientB, Agitate and Drain being available to the process chemist when configuring recipes.

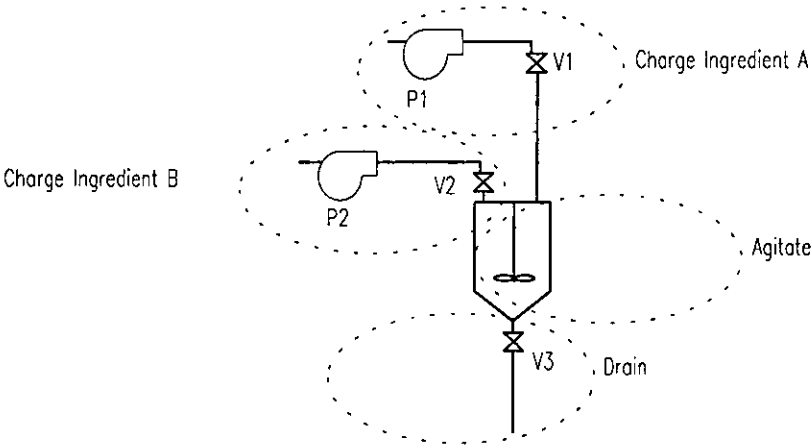


Figure 31 An Example of Equipment Modules

The S88.01 standard is becoming the predominant vehicle for descriptions and implementations of batch control and has been very widely accepted - practically all batch control vendors have aligned their offerings with the standard. Some of the key features of the standard are:

- A standard terminology in the field of batch control.
- Modular structure of process and equipment.
- Product independent plant design.
- Equipment independent recipes.
- A structured way of operating batch plants.

An S88.01-aware platform such as OpenBatch [75] or InBatch [76] can benefit productivity and maintainability by reducing the volume of software to be developed [103]. These products move implementation effort towards configuration rather than custom software development.

For a successful project, it is crucial that the relevant parties communicate effectively. S88.01 provides the terminology and models to facilitate effective review of proposed solutions [30]. If the S88.01 analysis has been carefully planned, subsequent similar projects should be able to re-use proven blocks of logic, from basic control (e.g. PID loops, valve drivers) to phase logic responsible for carrying out a useful process activity. In a regulated environment such as the pharmaceutical industry, this can substantially reduce validation time and cost as new products are brought to market [104], [105].

Much of the effort in implementing an S88.01-aware batch control system, however, is associated with the phase sequence logic, as shown in figure 32:

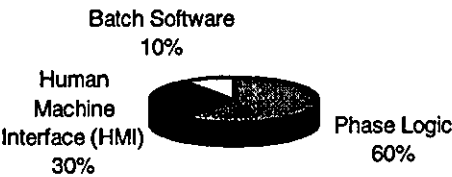


Figure 32 Relative Effort for a New S88.01-Aware Batch Control System (Source: Sequencia Corporation)

Although the S88.01-aware platforms are predominantly configured rather than programmed, the phase sequence logic tends to require significant analysis, design and implementation effort. However, currently available tools tend to offer only general purpose programming functionality, such as IEC 61131-3 language support supplemented, in the case of DCSs, with pre-built control module driver types. Also, whilst S88.01 assists the review process via a common approach to modularisation, it offers no guidance for ensuring that the phase logic behaviour is well understood, although changes here have a far greater cost impact than configuration changes to recipes.

In conclusion, S88.01 offers significant benefits to the batch control community by encouraging effort to be focused on the analysis and design activities. However, the standard offers minimal guidance regarding the development and verification of phase

sequence logic which is responsible for most of the cost and key to the correct functioning and future flexibility of the manufacturing process.

5.3 Communications

In order to maximise the benefits which a CASE tool can offer, it is important that it should be capable of interacting with other software products. For example, the CASE tool can then exploit the visualisation capabilities of 3D modellers, process industry mimic diagrams and custom applications which can be developed by third parties. Communications standards are an important first step in the ability of products to interact. An alternative, if possible at all, would be to define and develop software for each different type of interface to be supported.

Given the predominance and acceptance of Microsoft Windows as the host environment for control system software development, this section concentrates on Microsoft communications standards but also references open standards.

5.3.1 DDE

Dynamic Data Exchange (DDE) is a mechanism for Microsoft Windows applications to exchange data. NetDDE enables DDE to be used over a Microsoft network. Variations such as FastDDE and BlockDDE have been implemented to overcome performance limitations with DDE. DDE can be considered to be the lowest common denominator in Microsoft Windows communications standards – if an industrial software product supports communication with other Windows-based software, it is likely to support DDE.

5.3.2 OLE/ActiveX

Whereas DDE enables a Windows program to obtain data from another program, it still leaves the recipient with the task of interpreting and manipulating the data. For example, a drawing tool could make its graphic data available but if the recipient needs to display and print the drawing, functionality must be duplicated.

Object Linking and Embedding (OLE) was introduced in 1990 and followed shortly by OLE2 to provide the ability to embed one document type inside another. OLE2 also provided:

- Drag and drop - allowing graphical entities to be dragged from one program to another.
- OLE automation - functionality to enable scripts to control OLE-aware applications.

Components in Visual Basic or C++ which drove applications via OLE automation were referred to as OLE controls or OCXs.

OLE2 used a single machine communications model called Common Object Model (COM). Microsoft is now shipping Distributed COM (DCOM) with Windows NT 4 and it is also available for Windows 95. The term ActiveX is now being used and covers COM/DCOM/OLE/OCX generally [106].

5.3.3 OPC

OLE for Process Control (OPC) is a standard established in 14 months by Fisher-Rosemount, Intellution, Intuitive Technology, Microsoft, Opto 22 and Rockwell Software [107], [108]. OPC is a communication standard based on OLE that fosters greater interoperability between automation/control applications, field systems/servers, and business/office applications. OPC defines standard objects, methods, and properties built on OLE component technologies for servers of real-time information like DCSs, PLCs, smart field devices, and analysers to communicate the information they contain to standard OLE-enabled clients [109].

5.3.4 CORBA

The Object Management Group (OMG) is an organisation established in 1989 representing a collection of companies concerned with the development of an architecture for distributed component-based object computing. The architecture is referred to as the Common Object Request Broker Architecture (CORBA) and is vendor-neutral. There are now more than 100 CORBA-related products on a wider range of platforms than DCOM/ActiveX [106]. Whereas ActiveX controls have evolved from a single node environment, CORBA was designed from the start for large-scale distributed applications. As a consequence, it is more capable with regard to issues such as security [110].

However, according to Forrester Research in the USA, CORBA has gained acceptance in only 14% of the Fortune 1000 companies and this is even declining [111]. The reasons are said to be:

- Programmers are frustrated by the lack of progress with the architecture.
- Too complex and cumbersome for the average developer.
- Availability of Microsoft's COM/DCOM model (on which OLE depends) and JavaBeans [112].

5.3.5 The Internet

The Internet is the global, open, public computer network currently linking 64 million users [113]. An intranet is identical in technology, including the use of the same TCP/IP protocol, but is on a closed network thereby limiting access to, for example, a group of employees or companies.

Monitoring, and possibly even control, via the Internet is anticipated to be a major growth area by vendors of industrial control software [114]. This approach can be used via the World Wide Web for global access, or via an intranet for more restricted user access. Wonderware, for example, has launched a product called Scout VT which is a view-only client using configurable OLE browsers, graphs, charts and trend components to provide the process view [76].

5.4 Conclusions

Other than S88.01, there is minimal standards support for analysis and design which would tackle the business needs such as more re-usability and greater confidence in the correctness of the logic:

- IEC 61131-3 defines a reference PLC software model and standard programming languages but, in practice, the major control system vendors are more interested in compatibility with previous versions of their products and differentiation from their competition than they are in adhering strictly to the standard. Although claims have been made that the standard is object-oriented, excellent for structuring large applications which may involve sequence logic and, possibly by virtue of the claimed support for object-orientation, ultimately able to lower life-cycle engineering costs through the re-use of proven function blocks, tangible evidence is difficult to find and

anecdotal evidence de-emphasises its relevance. For the purposes of this thesis, it is therefore considered to be of value in encouraging graphical specification of sequence logic and the promotion of good software engineering practice in general to the control systems community. It is primarily implementation oriented rather than problem oriented, being focussed on the software development phase of the life-cycle with minimal contribution to the analysis and software architectural design phases.

- IEC 1499 will address the weaknesses of IEC 61131-3 with regards to logic distributed across intelligent nodes. This is achieved through the specification of distributed function blocks which, once again, is implementation oriented.
- Approaches such as EDDI, STEPS and KRAUSE are target-environment dependent which reduces their opportunity for widespread adoption. They do not support the component-based approach to system development.
- S88.01 contributes significantly to the analysis and design phases of batch control system applications and enjoys widespread international support from user and vendor communities. A domain-specific CASE tool which directly supported the life-cycle specification, development and support of phase logic would complement the standard and could help reduce the cost of change management, particularly in the pharmaceutical industries [104]. A significant first step would be to support the verification of phase sequence logic behaviour and facilitate effective review between analyst, process and operations personnel.

There has been an evolution of communications mechanisms for linking applications on the same computer and between computers. However, whilst Microsoft Windows industrial software packages typically support DDE, support for other communications mechanisms is not as widespread. An industrial CASE tool should therefore currently support DDE communication with other software packages.

Chapter 6 Mathematical Models

There is a considerable volume of research material available which is concerned with mathematical modelling and analysis techniques. Significant levels of research activity are still ongoing in many complementary and diverse approaches, applying a particular modelling approach to new application areas and tackling existing problem domains with new techniques.

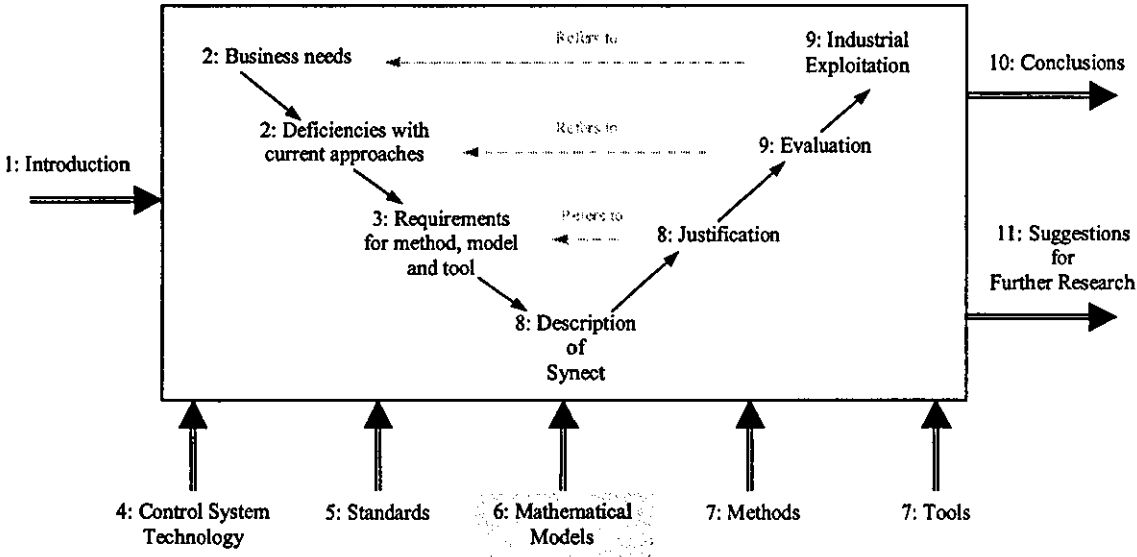


Figure 33 Relationship Between This Chapter And The Thesis Map

As depicted in figure 33, mathematical models pervade many aspects of this thesis. Figure 34, a copy of figure 1 from chapter one, Introduction, shows that mathematical models are considered by the author to be an integral part of an improved approach to automation projects.

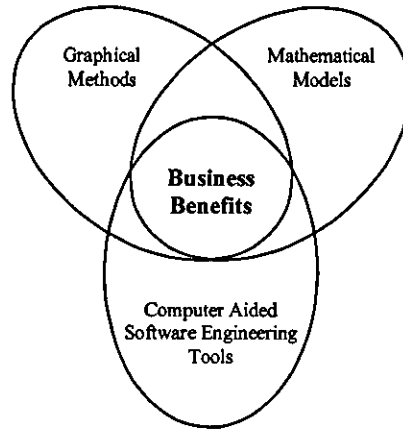


Figure 34 The Relevance of Mathematical Models

This chapter evaluates alternative mathematical models against the criteria established in chapter three, Requirements for Method, Model and Tool, and summarised in figure 4 in chapter one, Introduction:

- Executable
- Support behaviour queries
- Simple mapping from method
- Simple mapping to implementation
- Graphical representation
- Support concurrency
- Established

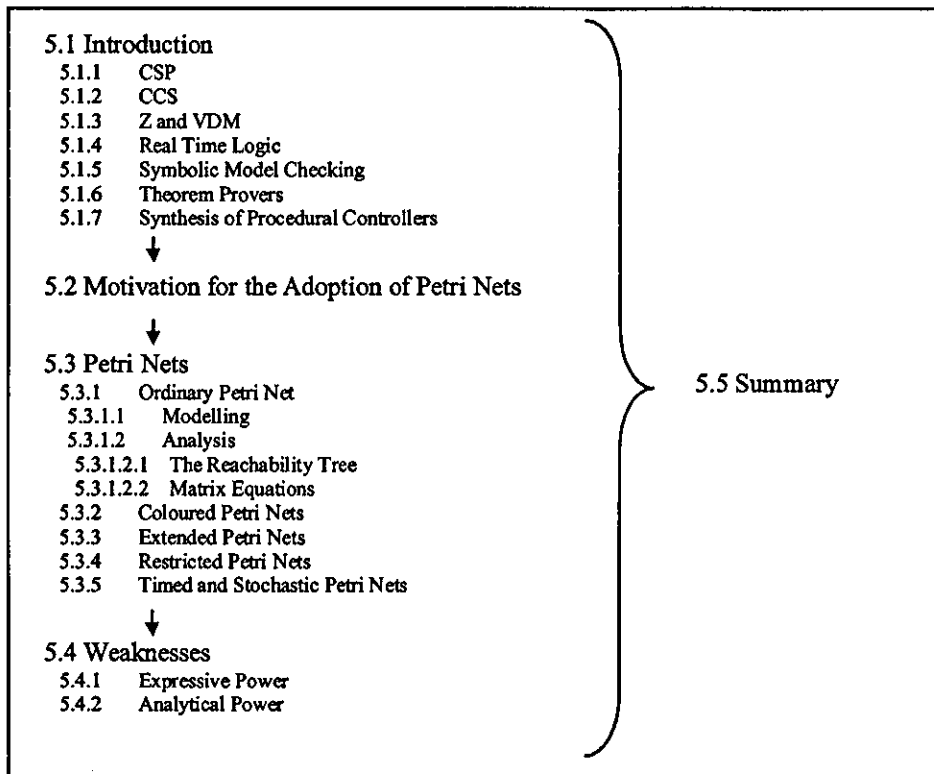


Figure 35 Structure of this Chapter

Figure 35 shows how the information in this chapter is organised. After introducing a selection of popular mathematical modelling techniques, Petri nets are justified as the chosen technique to complement the visual and CASE tool aspects of a solution and consequently are described in greater depth than other techniques. Known weaknesses in the application of Petri nets alone are then discussed before the key points from the chapter are summarised.

6.1 Introduction

Formal methods are an approach to software engineering based on a method with a sound basis in mathematics [115]. Their detractors claim they are expensive to introduce, partly due to their mathematical nature with consequent skill requirements of their practitioners. As a result, they tend to be applied to applications where the impact of a defect is very high. A literature survey and industry survey in 1992 investigated why formal methods are not widely accepted in industry, including considerations such as the benefits, limitations and barriers associated with them [29]. One of the findings was that people were not clear that cost benefits could be gained through the use of formal

methods but approximately 50% of respondents considered that lack of tool support was a serious problem. Although substantial research output would suggest that formal methods could contribute to the success of industrial applications, it has been suggested that they are a failure in the market-place [116].

Many techniques have been proposed for the formal specification of real-time systems and a large number of tools, from research prototypes to commercially available products, have been developed. There are numerous ways of classifying these techniques and tools, one of which in a review specifically oriented at real-time systems [47] considers:

- Mathematical support for reasoning on communicating concurrent processes, such as CSP (Communicating Sequential Processes) [117] and CCS (Calculus of Communicating Systems) [118].
- Operational approaches which describe the system by means of an executable model. These can be further divided into:
 - Languages and methods based on transition-oriented models, such as state machines and Petri nets.
 - Methods based on abstract notations, usually used for system analysis and design. Being semi-formal, these models cannot be directly used for system simulation and specification execution. Examples and corresponding tools are considered in chapter seven, Methods and Tools.
- Descriptive approaches focussing on the behaviour (what must be done by the system) rather than the structure (how it must be done). These approaches usually provide a means of verifying completeness and consistency. Examples include Z [119], VDM [120] and RTL [47].
- Dual approaches which aim to offer the formal verifiability of descriptive approaches with the executability of operational approaches.

Along with the refinement of the individual techniques and their application to new industrial requirements, new and hybrid approaches have been developed and attempts made at standardisation. For example, the Esprit SEDOS (Software Environment for the Design of Open Systems) project assisted in establishing two formal description techniques, Estelle [121] and LOTOS [122] (based on CCS), as international standards

and also developed prototype tool support [123]. Recently proposed approaches include symbolic model checking, theorem provers and the synthesis of procedural controllers.

6.1.1 Communicating Sequential Processes (CSP)

As its name suggests, CSP considers a system in terms of separate sequential processes which communicate via channels [117]. Although the communication mechanism is synchronous such that both sender and receiver must be ready for the communication to proceed, intermediate buffers can be introduced to facilitate asynchronous behaviour.

CSP supports the concepts and provides notations for specifying sequential (\gg), parallel (||) and interleaved (|||) sequencing.

6.1.2 Calculus of Communicating Systems (CCS)

CCS [118] and CSP are often described together. Along with CSP, CCS is concerned with the verification of concurrent systems although the CCS model includes descriptions of the model's actions, making it executable. The CCS communication mechanism is asynchronous.

6.1.3 Z and the Vienna Development Method (VDM)

Z [119] is a modular language based on set theory. VDM [120] is a state-based language using pre and post conditions to specify operations. Z and VDM provide the means for defining an explicit, if abstract, model of the system's state space and the operations which transform the current state, but there is no explicit representation of concurrency.

Extensions have been proposed which support the definition of timing constraints and move the models towards the object-oriented philosophy. Examples include VDM++ [124], Object-Z [125] and Z++ [126].

6.1.4 Real Time Logic (RTL)

Whereas many discrete event methods have difficulty coping with the concept of time, which may be important in a real-time reactive system, RTL has its focus on the formal description of temporal relationships relating to events and actions. In particular, all

language constructs are defined in terms of the current value of time on the occurrence of an event.

6.1.5 Symbolic Model Checking

Symbolic model checking is a verification technique which was developed to verify circuits and communications protocols but has been applied to safety and operability of chemical process systems [127]. Given a finite state model of a system and a set of temporal logic specifications, the symbolic model checking algorithms can verify whether or not the specifications hold on the model.

6.1.6 Theorem Provers

Prover is a tool which uses the patented Stalmarck theorem proving algorithm to test for properties of a system modelled in propositional logic [128]. The algorithm claims to be able to tackle full-scale industrial applications being dependent on a new measure of hardness which relates to the complexity of the proof rather than the size of input [129], [130]. Application areas such as railway signalling, aircraft landing equipment and nuclear power generation seem natural targets to which Prover has been applied but research output is also available regarding the application of the algorithm to PLC programs in general [131]. Programming cost savings of 30% and testing cost savings of 60% have been claimed [132].

6.1.7 Synthesis of Procedural Controllers

In contrast to techniques which facilitate expression of a proposed solution and then provide analysis methods for testing properties of the solution, an alternative approach is to derive the control requirements as a subset of all possible behaviours of the controlled system.

Ramage and Wonham modelled the plant as an automaton and used language theory to design a controller which forces the plant to exhibit behaviour consistent with given objectives [133].

In the context of batch processes, this approach forms the basis of a formal framework for the analysis and generation of provably correct control code, including both normal and

abnormal process operation [134]. This is referred to as synthesis of procedural controllers. Further work is required, however, before this approach could be exploited industrially, such as support for modularisation through hierarchical modelling and control structures as defined in the S88.01 standard for batch control.

6.1.8 Petri Nets

Petri nets began with Dr Carl Adam Petri's PhD dissertation in 1962, entitled "Kommunikation mit Automaten" (Communication with Automata), designed specifically to address systems with interacting concurrent components [135]. Petri nets are attractive for research into industrial sequence logic because they offer both a graphical notation for expressing attributes of interest, such as concurrency and dependency, along with a mathematical formalism. So they appear to satisfy the requirements to be able to both model and analyse the system under consideration. As the adopted mathematical model, Petri nets will be described in more detail in section 6.3, Petri Nets.

6.2 Motivation for the Adoption of Petri Nets

The following table assesses the above mathematical models against the evaluation criteria referenced in chapter three, Requirements for Method, Model and Tool, and summarised in figure 4 in chapter one, Introduction.

Criterion	CSP	CCS	Z	VDM	RTL	Symbolic Model Checking	Theorem Provers	Synthesis of Procedural Controllers	Petri Nets
Executable	x	✓	x	x	x	x	x	✓	✓
Support behaviour queries	✓	✓	x	x	✓	✓	✓	✓	✓
Simple mapping from method	x	x	x	x	x	x	x	x	✓
Simple mapping to Implementation	✓	✓	x	x	x	x	x	✓	✓
Graphical representation	x	x	x	x	x	x	x	✓	✓
Explicit support for concurrency	✓	✓	x	x	x	✓	✓	x	✓
Established	✓	✓	✓	✓	x	x	x	x	✓

The author consequently adopted Petri nets as the mathematical formalism to underpin the solution to the business needs identified in earlier chapters, although there are examples in the literature of benefits attributable to each individual method outlined above and examples of hybrid methods to overcome weaknesses. Further confidence was gained from:

- The maturity of the approach [115]. There is a substantial volume of published research material on Petri nets and their application to various domains, including manufacturing and process industry:

- A paper on a Petri-net tool called UNISON [136] references fourteen papers which relate to the application of Petri nets to the simulation and control of hardware systems.
- Petri nets have been applied to discrete manufacturing [137], [138], [139], [140], [141], [142], [143], [144], [145], [146] and to the discrete control and monitoring of process systems [147], [148].
- A paper examining the use of a Petri net approach to sequential industrial control systems compared with a traditional approach using relay ladder logic reports that several companies in Japan have achieved significant savings in system development time [149].
- Evidence that there is still considerable Petri net research activity ongoing, including their application to manufacturing industry, demonstrating their relevance to problems in this domain.
- Published research showing how different classes of Petri net such as safe or free-choice, can be derived from CSP, CCS, FSM and other models [150].

Based on the above criteria, Petri nets are the most appropriate choice but it is important to appreciate that the approach advocated embeds the mathematical model into the CASE tool rather than involving the analyst directly in its use (reviewed in section 6.4, Weaknesses and 6.5, Summary). The approach could therefore utilise alternative formal methods, or use several complementary methods, to offer the desired functionality. However this is beyond the scope of this thesis (see chapter eleven, Suggestions for Further Research).

6.3 Petri Nets

6.3.1 Ordinary Petri Net

6.3.1.1 Modelling

The graphical representation of the Petri net consists of places and transitions. A place is represented by a circle and a transition by a bar. Directed arcs (lines with an arrow head at one end) show the relationship between places and transitions. A directed arc from a place to a transition defines the place as an input place to the transition. A directed arc from a transition to a place denotes that the place is an output place of the transition.

In the example in figure 36, place p1 is the only input place to transition t1. Places p2 and p3 are the output places of transition t1. Place p2 is the only input place to transition t2 but is connected via 2 arcs. Place p4 is the only output place of transition t2.

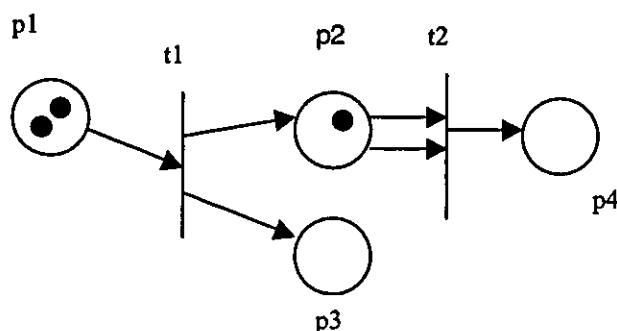


Figure 36 An Example of a Petri net

Whereas the preceding description relates to the *structure* of the Petri net, it is the *marking* of the net with *tokens* which is of real value. A token is represented by a black dot - figure 36 shows two tokens in place p1 and one token in place p2. When a transition fires, it removes tokens from its input places and adds tokens to its output places. More specifically, the transition is said to be *enabled* if each input place has at least one token for each arc to the transition. When it fires, tokens are removed from the input places corresponding to each arc to the transition. Using the marking shown in figure 36 as an example, transition t1 is enabled. When transition t1 fires, one of the tokens in place p1 is removed and a token is added to places p2 and p3. Place p2 now has two tokens and place p3 has one token. Transition t2 is now enabled and, when it fires, two tokens are removed from place p2 and one token is added to place p4.

From the above simple description, Petri nets can be seen to offer an intuitive representation for different approaches to modelling. To model the flow of material through a production system, the token could represent material and the structure of the net represent the paths through the manufacturing process. Alternatively, considering a place to represent a mode of operation of a machine, the token would denote the current mode of operation. Peterson [135] describes the use of Petri nets in modelling a diverse range of systems, including computer hardware and chemical reactions.

The Petri net is particularly attractive for modelling concepts such as concurrency, conflict and synchronisation. Considering a flexible assembly cell as an example, concurrency would be demonstrated by a lathe and drill each performing a sequence of operations on a different workpiece in parallel. The lathe and drill operate asynchronously - each can proceed through its sequence without regard to the other equipment. If a robot is servicing the cell, an example of conflict would be apparent if the lathe and drill both completed their operations and wanted the robot to remove the workpiece. Synchronisation would be required when the robot picks the part from the lathe to ensure that the lathe does not release the workpiece until the robot has hold of it.

6.3.1.2 Analysis

The Petri net also offers the ability to analyse the behaviour of the system. Characteristics of interest include:

- Given an initial marking, can a particular marking be reached? In the example of the flexible assembly cell, this could be used to ensure that the lathe could not start until the robot has passed it a workpiece.
- Can the system deadlock? This would occur if a design error enabled the system to reach a point from which it could not proceed. For example, if the robot picked a new workpiece to introduce to the assembly cell when the lathe and drill were already operating on workpieces, the system would reach a point from which it could not proceed:
 - The lathe and drill need the robot to remove their workpieces which it cannot do because it already holds a workpiece.
 - The robot cannot pass its workpiece to the lathe or drill because they are already occupied.

Many other types of query can be envisaged, such as safeness (maximum number of tokens in any place never exceeds one), boundedness (maximum number of tokens in any place never exceeds a fixed integer number), liveness (transition liveness relates to whether a marking can be reached in which a particular transition can fire - the net is live if all transitions are live), and the reader is referred to [135], [151] for a description.

6.3.1.2.1 The Reachability Tree

A straightforward analysis approach involves the generation of the reachability tree [135]. The initial marking of the Petri net in figure 36 has two tokens in place p1 and a token in place p2 but no token in place p3 or p4. This is represented as $(2, 1, 0, 0)$. From this marking, transition t1 is the only transition which can fire and this leads to the marking $(1, 2, 1, 0)$. From this new marking, transition t1 could fire again to give marking $(0, 3, 2, 0)$ or transition t2 could fire to give marking $(1, 0, 1, 1)$. The complete reachability tree is represented diagrammatically as follows:

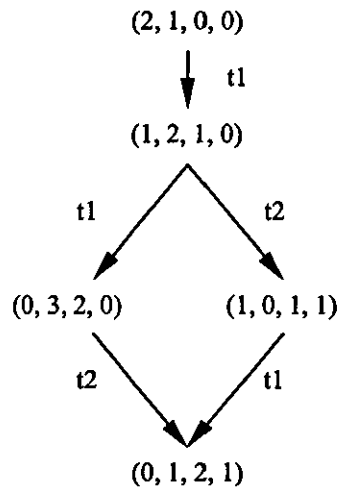


Figure 37 The Reachability Tree Corresponding to Figure 36

To see if a particular marking can be reached, the marking at each node in the tree is examined. For example, it is clear from the reachability tree in figure 37 that there is no marking where each place contains exactly one token.

To identify whether the system can deadlock, the reachability tree is examined to identify nodes which have no successor nodes. Figure 37 shows a deadlock at the node with marking $(0, 1, 2, 1)$. With this marking, none of the Petri net's transitions are live.

In both of the above types of search of the reachability tree, if a node is found which satisfies the query, it is a simple task to identify a sequence of transitions which, when fired, transform the marking from the initial marking to the node's marking. Considering a

practical example, this could be used to reveal to a designer of a manufacturing cell, not only that a deadlock exists, but a sequence of events which could cause it to happen.

One of the major disadvantages of the reachability tree is that its size and the computation time to generate it increase non-linearly with increasing size of Petri net [135]. In practical terms, the reachability tree cannot be generated for industrial scale applications.

6.3.1.2.2 Matrix Equations

An alternative analysis approach involves the treatment of the Petri net as a pair of matrices to define the relationships between transitions and their input and output arcs. The D^- matrix defines, for each transition, the tokens which are “consumed” when the transition fires. The matrix has a row per transition and a column per place. Referring to figure 36, when transition t_1 fires, it consumes one token from place p_1 but no tokens from any of the other places. Transition t_2 consumes two tokens from place p_2 but no tokens from any other place. The D^+ matrix similarly defines, for each transition, the tokens which are “generated” when the transition fires. The D^- and D^+ matrices corresponding to figure 36 along with their combined form, the D matrix ($D = D^+ - D^-$) are:

$$D^- = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

$$D^+ = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D = \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & -2 & 0 & 1 \end{bmatrix}$$

Consider the sequence of transition firings $\sigma = t_1, t_2, t_1$. Counting the number of times each transition fires, this transition sequence can be represented as firing vector $f(\sigma) = (2, 1)$. The relationship between the initial marking μ , a subsequent marking μ' , the composite change matrix D and the firing vector $f(\sigma)$ is as follows:

$$\mu' = \mu + f(\sigma).D$$

From the example in figure 36, the initial marking μ is (2, 1, 0, 0) and the final marking μ' is (0, 1, 2, 1). This can be expressed in matrix form as follows:

$$(0, 1, 2, 1) = (2, 1, 0, 0) + (2, 1) \cdot \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & -2 & 0 & 1 \end{bmatrix}$$

More usefully, a test to determine whether a marking is reachable from a given start marking can be performed by searching for a solution, in nonnegative integers, for the firing vector.

Although the matrix approach circumvents the difficulty with the non-linear growth of the reachability tree, it has weaknesses [135]. Specific limitations applicable to this research are:

- The firing vector contains only the number of times each transition fires rather than the sequence in which they fire. Consequently, there would be no ability to show an analyst the path from the initial system state to the sought state.
- There is no ability to search for deadlocks, although subsequent research has claimed a simplified ability to detect deadlocks through the use of modified equations [152].

6.3.2 Coloured Petri Nets

Jenson [153], [154] describes the use of coloured Petri nets. Coloured Petri nets are effectively a means of overlaying one Petri net on top of another, using coloured tokens and coloured arcs to achieve a more compact representation than would otherwise be possible. Design/CPN is a tool package developed by Meta Software Corporation to support hierarchical coloured Petri nets [155].

Updated Petri nets are an enhanced version of Coloured Petri nets, developed and used for information systems, such as MRP II [156].

6.3.3 Extended Petri Nets

In order to increase the modelling power, various extensions have been proposed to the primitive Petri net mechanisms, including inhibitor arcs, prioritising of transitions and constraints to name but a few [135]. Unfortunately, these compromise the ability to analyse the net [135], [157], [158].

6.3.4 Restricted Petri Nets

In contrast with extended nets, restricted nets reduce the modelling power in order to increase the analytical capabilities. Two approaches of interest are the marked graph and the free-choice Petri net [159].

In a marked graph, each place is an input for one and only one transition and an output of one and only one transition. Marked graphs can model concurrency and synchronisation but not conflict (where there is a choice of which enabled transition to fire). A marked graph may therefore be adequate for simple repetitive concurrent sequencing but would be incapable of explicitly modelling the decision making necessary for a flexible manufacturing cell.

A free-choice Petri net can model concurrency and synchronisation but imposes restrictions on the modelling of conflict. If a place is an input to more than one transition, it is the only input place to each of those transitions.

6.3.5 Timed and Stochastic Petri Nets

Associating a time delay with each transition enables performance characteristics to be derived. A practical application of such a timed Petri net [160] models a pallet handling system [161]. The time delay is expressed in terms of two times, T_{\min} and T_{\max} . T_{\min} defines the minimum time for which the transition must be enabled before it can fire and T_{\max} defines the maximum time for which the transition can be enabled before it must fire.

In a Stochastic Petri net [162], each transition has a random firing delay. Stochastic Petri nets with geometrical or exponential delay distribution are appropriate for modelling non-deterministic processes for performance statistics. To overcome size and complexity

constraints of stochastic Petri nets representing non-trivial systems, the Generalized Stochastic Petri Net divides transitions into immediate and timed [163].

6.4 Weaknesses

6.4.1 Expressive Power

The primitive notations of the Petri net are initially very attractive because they offer considerable flexibility and minimise the learning curve. However, the lack of more expressive notations becomes a significant obstacle to clarity when wishing to model more abstract concepts, such as queues, or for non-trivial systems. Whilst the issue of translating a Petri net into relay ladder logic has been tackled, the primitive Petri net notations lack the expressive power to be useful in an industrial scale application [164]. An example of the use of Petri net models for a cellular manufacturing system show how unreadable a model can become without the use of more expressive constructs [165]. Many alternatives have been proposed to address these limitations, providing notations to model more abstract representations, including Petri nets with Objects [166], Control-nets [167], control Petri nets [168], Process Translatable (PROT) nets which encourage top-down structuring [169] and Hierarchical Petri nets which have been used in the automatic generation of ladder logic [33] and as a means of tackling the non-linear explosion in the size of the reachability tree [170], [171]. Other forms of Petri net include Continuous and Hybrid (continuous and discrete) nets [172] and the Extended Modified Petri net (with graphical and textual language representations) [173]. Unfortunately, from an industrial perspective, the proliferation of so many alternatives may have hindered the uptake of any particular approach.

Whereas the above approaches have adopted more expressive Petri net constructs, an alternative approach is taken with Hierarchical Graph modelling and its language extension, Parallel Flow Graphs [174]. The modelling notation makes no reference to Petri nets but the control structure and flow is translated to a timed Petri net, which exactly represents the controller's logical structure and provides the basis for analysis.

The approach taken in [175] to use hierarchical time extended Petri nets (H-EPN) with notations for different types of place, such as action place and subnet place, results in a diagram which both incorporates non-standard notations and is still visually complex.

6.4.2 Analytical Power

Many approaches have been advocated which attempt to tackle the complexity problem associated with the analysis of Petri nets. Methods have been advocated based on the dynamics of the net and the structure of the net [176], the combination of sub-nets [145], hierarchically organised state spaces [170] and algorithms which produce a reduced reachability tree via concurrent execution semantics [177], [178]. Techniques which attempt to partition the total net are dependent on how intuitively the system can be mapped to the partial nets. Although techniques which produce a reduced reachability tree may be capable of analysing larger nets than would otherwise be the case, they are still likely to fail as the net increases in size to that required for an industrial-scale application. They also fail to exploit the modularity inherent in the analyst's design of the system.

For performance evaluation, enhanced formalisms have included the Extended Place/Transition Net (EPTN) with additions to Ordinary Petri net token definition, transition definition and transition firing rules [179].

6.5 Summary

The Ordinary Petri net is a suitable choice as the mathematical model to be combined with a graphical method and supported with a software tool because it has the following characteristics:

- Good support for:
 - Sequence
 - Concurrency
 - Event driven applications as exhibited by reactive systems
- Simple to understand via its graphical representation
- Simple to understand its analysis capabilities obtained by deriving its reachability tree

Used in isolation, however, without a graphical method and software tool, Petri nets have limitations for use in the development of industrial sequence logic. In particular:

- The primitive graphical notations tend to produce large and unstructured specifications [180] and provide inadequate expressive power to model more abstract or aggregate entities such as a machine consisting of a collection of axes and tools. Although more powerful notations have been developed to address this problem, approaches have diverged and they still represent unfamiliar notations to a typical engineer in industry. IEC 15909 is a draft standard for high level Petri nets but offers neither modularity nor notations which would be familiar to an engineer [181].
- The non-linear growth in size of the reachability tree with increasing size of Petri net limits the scale of application which can be tackled using an Ordinary Petri net. Whilst techniques have been published for tackling this problem, they have not been in an industrially usable form supporting an intuitive modularisation of the application and facilitating analysis of well-bounded components.

Chapter 7 Methods and Tools

Methods and tools can be considered independently, as shown in figure 38, but overlap where a method is explicitly supported by a software tool:

- There are many different software development methods in use, particularly in the information technology sector, with some examples of their application to industrial control applications [182]. The key aspects relevant to the development of industrial sequence logic are discussed in this chapter.
- In addition to software development tools which directly support one or more methods, tool support is also available for individual phases of a control system life-cycle. The benefits which such software tools offer are described to identify characteristics which a tool supporting a proposed new method should offer.

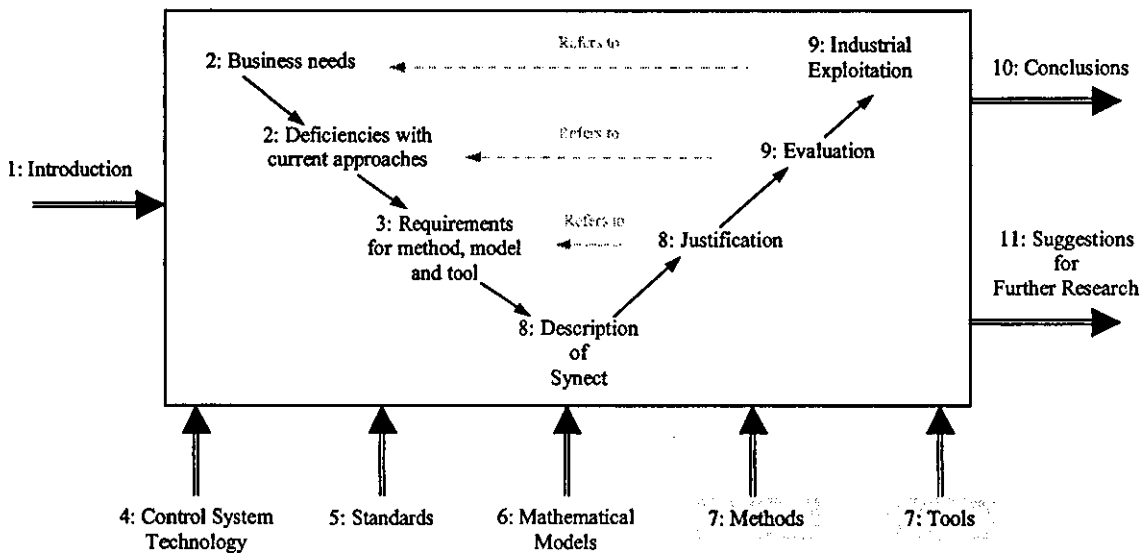


Figure 38 Relationship Between This Chapter And The Thesis Map

7.1 Methods

7.1.1 Evaluation Criteria

The desired characteristics identified in chapter three, Requirements for Method, Model and Tool, and summarised in figure 4 in chapter one, Introduction, are as follows:

- Graphical
- Manage complexity
- Expressive notations
- Well-understood notations
- Encourage object-oriented view regarding behaviour of agents
- Encourage structured method view of ordering of events and coordination of agents
- Support decomposition
- Support sequence and event behaviour
- Coherent information
- Easy to learn
- Discrete (manufacturing) applications
- Hybrid (batch) applications
- Oriented towards analysis and design activities
- Formal definition

Structured and object-oriented methods will now be discussed, followed by an assessment of their conformance with the above criteria.

7.1.2 Structured Methods

Ed Yourdon pioneered the use of the structured method [183] and this was enhanced for the real-time community by Ward and Mellor [184] and Hatley and Pirbhai [185]. These methods have been widely applied, including examples at the PLC level [25].

Structured methods use three sets of diagrams to express the required behaviour of the system:

- Data flow diagram (DFD) for showing data flows between activities.

- Entity relationship diagram (ERD) for showing relationships between stored information.
- State transition diagram (STD) for defining sequence behaviour.

7.1.2.1 Data Flow Diagram

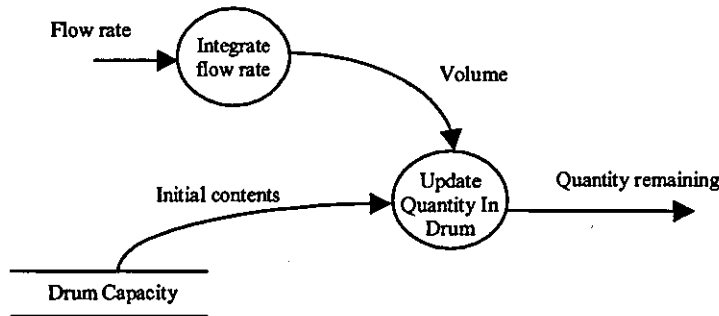


Figure 39 A Partial Example of a Data Flow Diagram

Figure 39 is a partial example of a data flow diagram (DFD). The circles represent data transforms, the arrows represent data flows and the parallel lines represent data stores. Real-time extensions add dotted circles and arrows to represent control transforms and control signals.

The specification is hierarchically ordered. The content of each data transform is defined on another page as either another DFD or a text-based specification describing how the data transform is accomplished. Control transforms are defined using state transition diagrams (STDs). Whilst this provides a very effective means of modularising the specification, a reviewer will often find the need to reference many pages simultaneously in order to understand the derivation or usage of a data flow or control signal.

The DFD is functionally oriented. Each data transform should be named using a “verb noun” phrase, being as explicit as possible. In a batch control system, for example, “Process Material” would be less explicit than “Heat Additive”. The contents of this transform would coordinate all necessary activities to heat the additive. Assume that one of the outputs of the transform is a command to control a steam inlet valve. In such a functionally oriented specification, it would be quite reasonable for another transform, such as “Pre-Heat Reactor” to also control the steam inlet valve. This will be considered further in the subsequent discussion regarding object-oriented approaches.

7.1.2.2 Entity Relationship Diagram

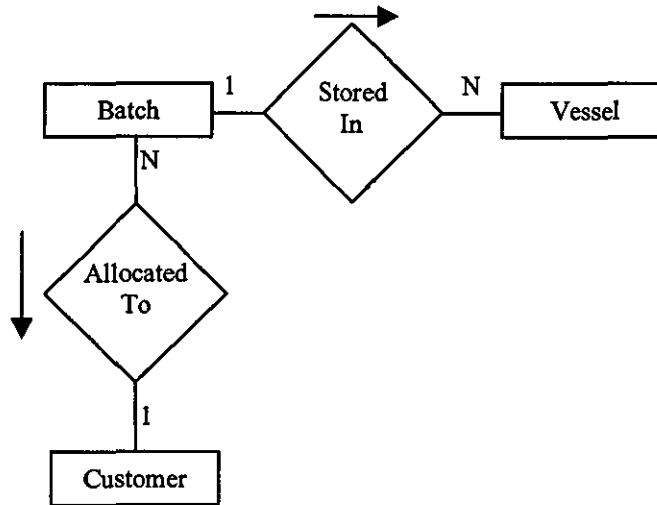


Figure 40 An Example of an Entity Relationship Diagram (ERD)

The entity relationship diagram in figure 40 contains two types of symbol. The rectangle represents an entity type and the diamond represents a relationship. The diagram graphically represents the following relationships:

- A batch is stored in many vessels.
- Many batches are allocated to a customer.

In an application in which information relationships are of primary importance, such as order and stock management, the ERD may be the key to understanding the requirements, with the DFD and STD considered to be responsible for maintaining the integrity of the relationships in the ERD.

7.1.2.3 State Transition Diagram

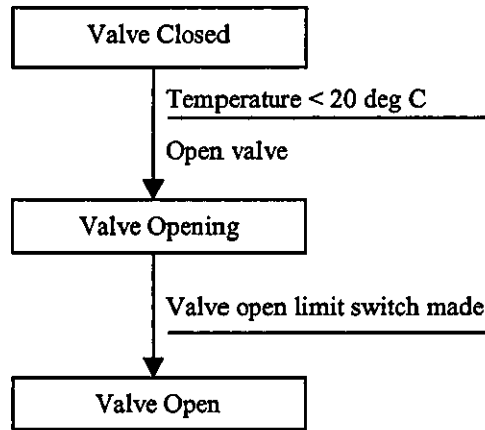


Figure 41 An Example of a State Transition Diagram (STD)

Figure 41 shows a fragment of a state transition diagram (STD). The rectangles represent recognisable states of the system and the arrows show, from any given state, which other states are reachable. The horizontal line beside the arrow separates the conditions which must be satisfied for the transition to fire, shown above the line, from the actions which will be taken if the transition fires, shown below the line. This is the Mealy STD representation. An alternative form, the Moore STD, associates actions with the state rather than with the transition.

The STD is specifically oriented at graphically representing sequence logic. It is simple to understand and hence review in multi-disciplinary teams yet formally defines the intended behaviour.

7.1.3 Object Oriented Methods

Object oriented methods have their origins in modelling and simulation. Booch points out that "the fundamental ideas of classes and objects first appeared in the language Simula 67" [44]. The emphasis initially is on modelling items in the physical world. These are referred to as *objects* and they communicate by sending each other *messages*, resulting in the invocation of *methods* [21].

For the purposes of clarity, it is useful to be able to distinguish between object-oriented and non object-oriented programming. Stroustrup asserts that a language cannot claim to support a technique "if it takes exceptional effort or skill to write such programs" [86]. Of

particular relevance to this research is the difference between object-based and object-oriented languages. Cardelli and Wegner refer to a language as object-based rather than object-oriented if it does not support "kind of" relationships, such as an apple is a kind of fruit and fruit is a kind of food.

There are now many different object oriented methods, of which the more established are:

- Shlaer Mellor
- Booch
- Object Modelling Technique (OMT)
- Objectory
- Class Responsibility Collaborator (CRC)

In the absence of an agreed means of comparing or evaluating alternative methods, any comparison must be judged in the context of the requirements to be satisfied. Several comparisons have, however, been performed [186], [187], [188], [189], [190]. One of the authors of the Shlaer-Mellor OOA/RD method has compared their method with OMT and Booch [191], [192].

There are now so-called second generation methods, such as Fusion [21] which attempt to integrate and extend the first-generation methods. Fusion builds on existing methods and incorporates an element of formal methods:

- | | |
|------------------|--------------------------|
| ■ OMT | object model and process |
| ■ CRC | object interaction |
| ■ Booch | visibility |
| ■ Formal methods | pre and post conditions |

With a similar objective, UML (Unified Modelling Language) "fuses the concepts of Booch, OMT and OOSE" (Object-Oriented Software Engineering) [193] and was developed by major industry organisations including Microsoft, Oracle, Hewlett-Packard, IBM and others. It is considered to be applicable to real-time systems, client/server and other kinds of "standard" software development [194].

On its own, object orientation does not overcome the deficiencies with automation projects identified in chapter two, The Need. Indeed, successful deployment of object orientation has been found to intensify the need for an organised and disciplined approach to software development [195].

Neither is the object oriented paradigm necessarily intuitive. The philosophy of modelling objects which exist in the real world as encapsulated entities which communicate by messages invoking an object's methods, is vulnerable to criticism. "If you drink a cup of tea, you do not invoke the drink operation on the cup any more than the cup invokes the drink operation on your lips, or, indeed, anything invokes an operation on anything else" [196]. This criticism reflects the focus of object orientation on the agents rather than the coordination, in contrast with structured methods which focus on coordination.

Although variants of the state model are popular in object oriented methods for specifying sequence logic, Fusion takes a different approach. The relevant aspects of the Shlaer Mellor method and Fusion method will therefore now be compared.

7.1.3.1 Relevant Features of the Shlaer Mellor Method

7.1.3.1.1 Object Communication Model

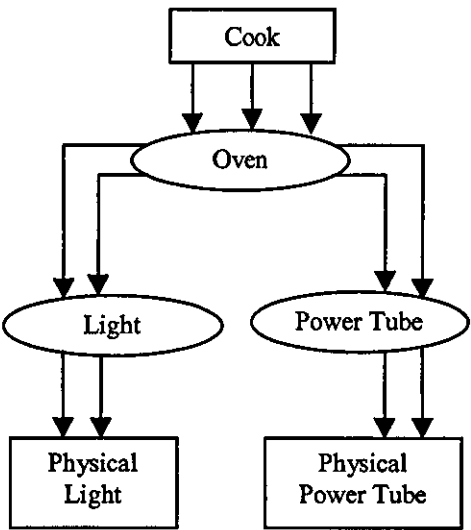


Figure 42 Shlaer Mellor Object Communication Model (Source: Object Lifecycles - Modelling The World In States)

The Shlaer Mellor method shows communication between objects on an Object Communication Model, as shown in figure 42 [6]. Patterns of behaviour where more intelligent objects delegate work to their subordinates and coordinate their progress are regarded as typical in larger models and a convention is adopted whereby more knowledgeable and powerful objects are towards the top of the diagram. However, such a convention is not enforced, nor is the strict hierarchy of communication which prevents an object at the top of the diagram from communicating directly with an object towards the bottom of the page.

7.1.3.1.2 State Transition Diagram

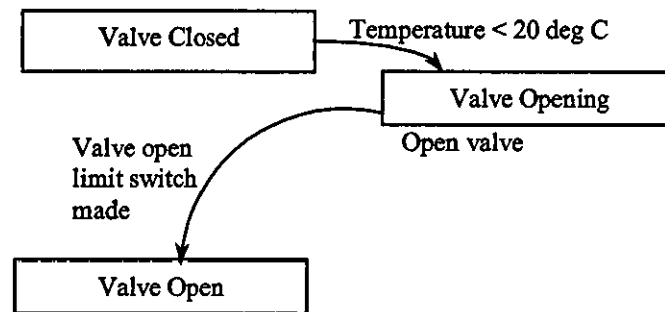


Figure 43 Moore Form of State Transition Diagram

The Shlaer Mellor method uses the Moore form of state transition diagram for specifying sequence logic as shown in figure 43, associating actions with states rather than transitions. This can lead to the printed form of the specification being more difficult to review if the logic is fragmented due to actions being printed on a separate page from the diagram. Alternatively, if the actions are printed on the same page as the diagram, more space may be required to denote the same logic compared with the Mealy form of state transition diagram, particularly if different actions are required depending on the route taken to a given state.

7.1.3.2 Relevant Features of the Fusion Method

7.1.3.2.1 Pre-Conditions and Post-Conditions

The principle underlying Fusion is that analysis should be concerned with specifying what behaviour is required rather than how it is achieved [21]. Fusion considers the state model to be a definition of how the behaviour is achieved and is therefore avoided in the analysis. Instead, the system response to an event is defined by specifying the state of relevant objects before the event and their state after the event. Although this successfully avoids specifying how the functionality is achieved, weaknesses related to its use for industrial sequence logic applications include the following:

- The pre-conditions and post-conditions refer to named states which imply a knowledge of valid object states which are not explicitly defined.
- The transition directly from one state to another may be illegal but this is not apparent, nor is the intermediate path taken to effect the transition. For example, assume a vessel is under vacuum when a fault condition is detected and that the reaction on

fault detection is to vent the vessel. It may be necessary to first bring the vessel to atmospheric pressure in a controlled manner before venting to atmosphere.

- **Non-intuitive review procedure.** To ensure effective and accurate requirements specification, a multi-disciplinary team is typically required. Adopting a “bottom-up” approach to review has been found to be effective, whereby the required functionality of lower level objects is reviewed and agreed before higher level objects are reviewed. This helps to gain the confidence of the team because it starts with plant equipment with which they are familiar. Using the Fusion approach, this definition would not be available when considering the system’s reaction to events.

7.1.3.2.2 Scenarios of Use

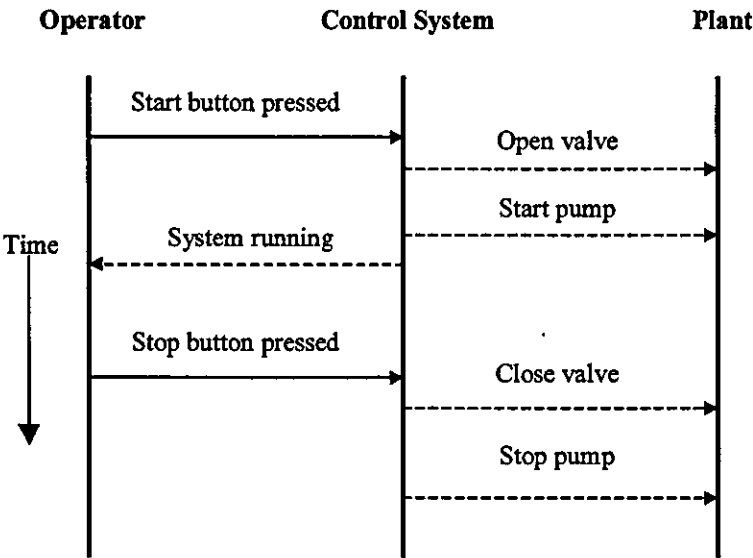


Figure 44 An Example of a Scenario of Use

Figure 44 shows an example of a “scenario of use” which is considered useful for establishing the interface boundary [21]. It is also a useful technique for reviewing outline behaviour and as a definition of requirement against which the specified object behaviour can be verified.

7.1.4 Assessment

Criterion	Structured Method	Object-Oriented Method
Graphical	✓	✓
Manage complexity	✓	✓
Expressive notations	✓	✓

Well-understood notations	x	x
Encourage object-oriented view regarding behaviour of agents	x	✓
Encourage structured method view of ordering of events and coordination of agents	✓	x
Support decomposition	✓	✓
Support sequence and event behaviour	✓	✓
Coherent information	x	x
Easy to learn	x	x
Discrete (manufacturing) applications	✓	✓
Hybrid (batch) applications	✓	✓
Oriented towards analysis and design activities	✓	✓
Formal definition	x	x

To summarise:

- Structured methods and object-oriented methods satisfy many of the criteria required but lack the necessary formal definition.
- They are not specifically focussed towards reactive systems and consequently present the user with the requirement to address information relationship and continuous data models. Although there are examples of their application to industrial control systems [182], there is additional learning required above that which could be considered to be essential. This applies to both the notations and the approach to be taken.
- Their scope includes very large and complex systems and the diagramming conventions reflect this. Consequently, related information is not always presented coherently. For example, several layers of data flow diagram may need to be studied to understand a message flow.
- A subset of simple notations and models with formalised interpretation would satisfy the criteria i.e. the Synect method will utilise a subset of structured and object-oriented method notations and models.

7.2 Tools

7.2.1 Evaluation Criteria

The desired characteristics identified in chapter three, Requirements for Method, Model and Tool, and summarised in figure 4 in chapter one, Introduction, are as follows:

- Support graphical method
- Support mathematical model
- Visualisation
- Code generation - ANSI C
- Code generation - relay ladder logic
- Code generation - Echelon Neuron C
- Code generation - built in diagnostics
- Inexpensive
- Full life-cycle support
- Support rapid prototyping
- Connectivity

There are many different software tools which have a contribution to offer in the development of industrial automation systems. Although the boundaries are not rigid, for the purposes of discussion, the different types of industrial software tool support are considered with reference to the need they address:

- Requirements capture
- Design
- Proving the design satisfies specified constraints (via property verification or executable model)
- Rapid application development, particularly for visual interface design

- Visualisation
- Simulation (for performance analyses – useful for identifying process bottlenecks)
- Prototyping
- Implementation
- Testing
- Auto-code generators

7.2.2 Requirements Capture

These visual modelling [197] tools concentrate on helping the analyst and knowledge holder to capture the requirements and to help verify that they are complete, consistent, clear and unambiguous. Structured analysis methods and object oriented analysis methods fall into this category and tend to be supported by method-specific or multi-method tools. Examples in this category are:

Yourdon, Ward-Mellor, Hatley-Pirbhai	Promod-PLUS [198], TeamWork [199]
UML	Rational Rose [200]
OOA (Shlaer-Mellor)	OOA-CASE [201]
Fusion	Paradigm Plus [202]

In order to guide the analyst and prevent unnecessary errors from being introduced, the tool should offer good support for the method. For example, the tool should be able to verify data flows between diagrams. Ideally, the tool should offer the analyst the information to prevent such incompatibilities being specified rather than reporting the error later. A good tool frees the analyst from the details of applying the method in order to concentrate on the essence of the problem to be solved.

In contrast with method-specific tools, some of the most commonly used, but primitive, requirements capture tools are word processors, spreadsheets and drawing tools such as VISIO [203].

7.2.3 Design

Structured methods and object oriented methods include a design activity which defines how the required behaviour will be achieved. Shlaer Mellor's method refers to design by elaboration meaning that it is a mechanistic derivation from the analysis.

Tools in this category include CASE tools supporting a structured or object-oriented method and component-oriented tools, such as Rockwell's RSFrameworks which offers library management of pre-built function blocks.

7.2.4 Design Verification

These tools help in checking that a design is correct with reference to defined criteria. These take various forms, some relating to testing properties of a system where the system specification and the property are specified mathematically, as described in chapter six, Mathematical Models, and others provide an executable model which can either be driven interactively or via a script. There are many examples of tools which are available as research prototypes but also a number which are commercially available:

Property checking	SPADE [204], MALPAS [205], Prover [128]
-------------------	---

Executable	StateMate [206], Stateflow [207], BetterState Pro [208]
------------	---

STATEMATE, for example, is a set of tools for the specification, analysis, design and documentation of large and complex reactive systems, such as real-time control systems [206]. It uses three graphical languages to capture the specification of the structure, functionality and behaviour of the system. The behavioural view is specified using statecharts, which are an extended form of state transition diagram [209]. An example application of STATEMATE in the avionics industry is given in [210].

Using an executable model, the analyst can gain a deeper understanding of how the system will operate, for example simulating external events and observing the system's reaction. However, the ability to execute the model does not, in itself, fully satisfy the need:

- It does not completely verify system behaviour. For example, assume that a particular sequence of events would cause the system to deadlock. This will only be revealed if that particular sequence of events is tested.

- The degree of test coverage cannot be ascertained.

In contrast, a property checking tool could confirm behavioural properties without having to verify individual scenarios. However, this approach requires that all behavioural properties of interest are specified and does not offer the confidence gained through observing the system execute. Whilst it may be infeasible to analyse an industrial-scale control system in its entirety, analyses of subsets may be practical and useful.

These approaches need not be mutually exclusive. An ideal tool would offer both of these complementary approaches.

7.2.5 Rapid Application Development

The component oriented model of software development is being widely adopted and seems to have its populist roots in the Microsoft Visual Basic language. The paradigm is based on specifying components with defined behaviour (e.g. a button has two states and changes from up to down when the mouse is clicked over it) and properties (such as the text on the button, its colour, etc.). There is a clear correspondence with object orientation in this paradigm. More importantly, many pre-defined components are supplied with the programming environment which can then be "soft-wired" together. So a novice need undertake only minimal programming, developing applications using only available components, whereas the expert can define new components and make them available in a form which is indistinguishable from the pre-supplied components. So the language environment caters for both the novice and the expert. Typical examples of component-oriented packages are:

Visual Basic [211]	General purpose language development, particularly good for HMI applications.
HP Vee [212]	HP VEE is a component-based approach from Hewlett Packard and is now also available via I/O company Amplicon, for use in conjunction with their test and measurement and data acquisition products. It is considered to be especially suited to the fast development and maintenance of user and instrument interfaces and displays [213].

LabVIEW [214]	National Instruments LabVIEW graphical programming environment also uses the approach of defining components and "wiring" them together. An interface is available for linking LabVIEW with Data Translation's acquisition hardware.
Matlab [215]	Matlab is a graphical tool for programming, analysis, modelling and simulation with the emphasis on support for matrix manipulation.

7.2.6 Visualisation

Visualisation tools which can be used as a user-friendly perspective on the behaviour of a sequence logic specification include:

- 3D modelling tools, such as Workspace [216], are common-place in the robotics industry. These tools enable the user to see how the equipment will move and can also perform crash-detection.
- Supervisory Control and Data Acquisition (SCADA) tools, such as Wonderware's InTouch [76] and Rockwell's RSView32 [28], are common in process and manufacturing industries with current plant equipment state represented visually by graphics symbols.

7.2.7 Simulation

Simulation is a term commonly associated with performance analyses. For example, a manufacturing line could be specified in terms of the manufacturing stages which must be performed and then a statistical distribution assigned for the time taken at each stage. The model could then be run to find out the overall throughput, machine utilisation rates and to determine the effect of malfunction on buffer capacities etc.. These packages usually contain a mimic-style visualisation module, often including animation, to present the results in a more user-friendly manner.

Examples of tools in this category include Witness [217] and AutoMod [218].

7.2.8 Prototyping

Prototyping has been advocated in a variety of forms and for a variety of reasons, of which better user involvement is the most often cited [219], [220], [221]. However, a major challenge is keeping the scope of the project under control:

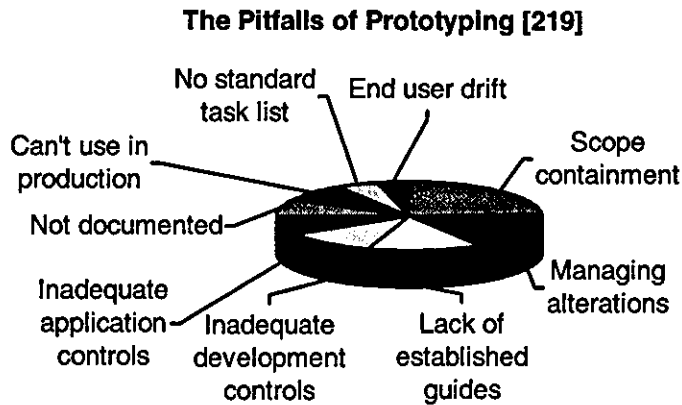


Figure 45 The Pitfalls of Prototyping

Requirements specification and design are the life-cycle phases which typically benefit from prototyping. During requirements definition, the system operability may be reviewed through prototyping of the user interface. Design phase prototyping may include verification of the performance characteristics of a proposed software architecture. In general, the value of prototyping is in the early phases of a project, hence the prototyping motto "if you are going to fail, fail early".

The use of target software platforms, such as S88.01-aware batch control systems, can have a dual effect on prototyping:

- The availability of pre-built functionality can facilitate cost-effective prototyping because less effort is required to construct the prototype.
- Software platforms usually impose constraints which are most clearly expressed through prototyping. The alternative may be to spend more effort specifying and implementing alternative mechanisms to implement similar functionality.

7.2.9 Implementation

There are many language development tools for general purpose software development, of which Microsoft C++ is an example. Implementation tools oriented at manufacturing and process industries tend to be closely allied to the vendor's control system, although the goal of IEC 61131-3 is to enable a program developed on one vendor's software to be used on another vendor's control system. Software tools in this category include Rockwell's RSLogix5 [28] and CJ International's IsaGraph [93].

The Advanced Productivity Tool (APT) was developed by Texas Instruments (now part of Siemens) and marketed as a CASE tool for their top of the range PLCs. APT supports languages similar to those defined in IEC 61131-3. One estimate was that it could reduce the time required to program and make changes to a PLC program by 75% compared with relay ladder logic [222].

7.2.10 Testing

Another area of tool support includes automated testing tools. One claim was that users could reduce development costs by up to 80% through the use of computer aided software testing (CAST) tools [223]. Considering that testing is estimated to consume 30% to 50% of software development effort and budget, this would appear to be a very promising area.

However, whilst computer aided test tools are available for the general IT market, there is limited usage by industrial automation developers. The nearest tends to be the use of software to simulate the behaviour of plant equipment, such as DirectLink [224]. DirectLink can communicate with a PLC and for example a valve simulation component would look for the PLC output to move the valve, wait a pre-configured time and then give the PLC the input to signal that it was open. Tools such as these also allow fault conditions to be simulated, such as a valve failing to open, so that the behaviour of the control system can be verified. Although these tools are an improvement over switch boxes and lamps, they inadequately fulfill the role of a test tool because they are unable to capture evidence of the control system behaviour or automatically manage a series of tests.

An alternative approach, offered by Rockwell, is to provide the ability to run the target PLC ladder logic on a PC-hosted emulator. The control program includes debug files to

simulate the plant equipment which run on the emulator but are not downloaded to the PLC.

7.2.11 Auto-Code Generators

These tend to be associated with particular tools because they need to translate a design into software code. For example, StateMate [206] and SystemSpecs [225] can generate C and VHDL. Other than vendor-independent programming tools, such as CADEPA [94], ladder logic generation is not supported.

7.2.12 Assessment

The following table compares examples from the above categories against the criteria referenced in chapter three, Requirements for Method, Model and Tool, and summarised in figure 4 in chapter one, Introduction.

Criterion	Rational Rose	Ilogix StateMate	NPL Prover	Rockwell RSFrame Works	Microsoft Visual Basic	Rockwell RSLogix5	CJ Inter- national IsaGraph	Rockwell RSView32
Support graphical method	✓	✓	✗	✗	✗ (1)	✗ (1)	✗ (1)	✗
Support mathematical model	✗	✓(2)	✓(3)	✗	✗	✗	✗	✗
Visualisation	✓	✓	✗	✓	✓	✓	✓	✓
Code generation – ANSI C	✓	✓	✗	✗	✗	✗	✓	✗
Code generation – relay ladder logic	✗	✗(4)	✗	✓	✗	✓	✓	✗
Code generation – Echelon Neuron C	✗	✗	✗	✗	✗	✗	✓	✗
Code generation – built in diagnostics	✗	✗	✗	✗	✓	✗	✗	✗
Inexpensive	✓	✗	✗	✓	✓	✓	✓	✓
Full life-cycle support	✗(5)	✗(5)	✗(5)	✗(6)	✗(6)	✗(6)	✗(6)	✗(6)
Support rapid prototyping	✓	✓	✗	✗	✓	✗	✗	✓
Connectivity	✗	✗	✗	✗	✓	✗	✗	✓

Notes:

1. These programming tools support graphical notations but not a graphical method oriented at analysis and design.

2. The Statemate model is executable but does not support behavioural queries.
3. Prover supports behavioural queries but does not have an executable model.
4. A research example of the generation of monolithic instruction list code from a subset of statechart notations using StateMate is described in [226].
5. Good design documentation is necessary for effective support and maintenance but these tools do not directly link the implementation with the tool environment by animating the analyst's graphical specification from the live control system or supporting the replay of the live control system's event log.
6. Weak support for the early project phases such as requirements capture.

7.3 Summary

The key points from this chapter to be considered in satisfying the requirements for a new approach are:

- Structured methods emphasise coordination whereas object oriented methods emphasise participating agents. Both perspectives are valuable and are complementary.
- To ensure a consistent structure regarding component interaction, an organised approach to internal communication is required.
- Sequence definition should use a graphical notation. The state transition diagram is a widely-used notation.
- For effective use of a method, a tool which strongly supports the method is highly beneficial. This helps to prevent syntax and semantic errors from being entered as the application is specified rather than being trapped later.
- The new tool should be capable of being used with other tools, such as 3D or process mimic visualisation software.
- Analysis of system behaviour before the system is constructed is very valuable but there are limitations to simply executing the specification because a sequence of events which would expose an error may not be tested. Complementing an executable model with the ability to test for behavioural properties would overcome this weakness.

- Analysis of subsets of a system would be advantageous, particularly if the system's components are to be re-usable.

Chapter 8 Description of Synect

This chapter describes a solution to the requirements, introducing the method, the mathematical model and software development platform which the author has trademarked as Synect. Figure 46 shows how this chapter relates to the other information in this thesis.

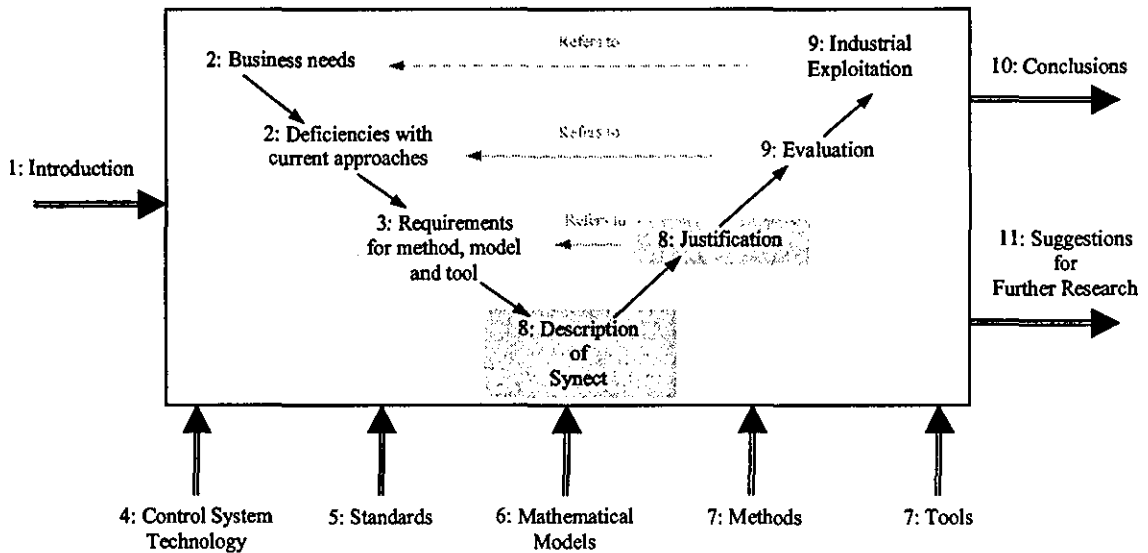


Figure 46 Relationship Between This Chapter And The Thesis Map

Synect is a commercially available product and consequently the algorithms used by the tools and their internal data structures are not discussed in this thesis. The detailed usage of the tools is described in the user guides [227] included in Appendix C, Synect User Guides.

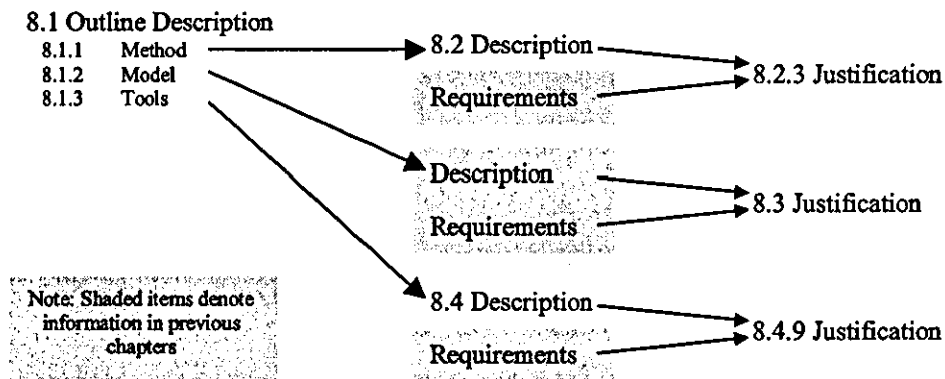


Figure 47 Structure of this Chapter

Figure 47 shows the structure of this chapter. The development environment is first introduced in general terms to provide a foundation for the subsequent detailed description of the method and CASE tool, Petri net modelling having been described in chapter six, Mathematical Models. Each section concludes by justifying the solution against the requirements specified in chapter three, Requirements for Method, Model and Tool.

8.1 Outline Description

Synect consists of a method, a mathematical model and a set of tools to support the method and model.

8.1.1 Method

The method uses an object-based decomposition to hierarchically organise the components of the solution. This is referred to as an Object Hierarchy. Objects communicate explicitly by sending messages or implicitly by referring to the internal state of another object. The interface between the control logic and the plant equipment is also represented on the Object Hierarchy. The messages which the object sends and receives, together with the plant inputs and outputs, are referred to collectively as the object's external interface.

An object's sequence logic is specified by one or more state transition diagrams (STDs). The conditions and actions associated with an STD transition are directly related to the object's external interface, the state of another object's STD or to one of the object's

variables. Together, the Object Hierarchy and STDs provide a formal specification of the application.

8.1.2 Mathematical Model

The mathematical model is an Ordinary Petri net:

- STD states are mapped to Petri net places.
- STD transitions are mapped to Petri net transitions.
- Plant input and plant output functions are associated with Petri net transitions.

8.1.3 Tools

The platform consists of a suite of tools which run on PC hardware under the Windows operating system. Each tool is a separate program.

Tool	Function
Application Editor	This behaves as an intelligent drawing tool, to enable the analyst to specify the object hierarchy and state transition diagrams. It produces graphical data for use by the STD Monitor application and a definition of the logic for use by the Compiler.
Compiler	Takes the formal control logic definition in terms of objects and STDs from the Application Editor and generates a Petri net, which the remaining tools use as the definition of control logic.
Analyzer	Provides the ability to generate the Petri net's reachability tree, report the existence of deadlock and enables the user to perform simple searches for specified combinations of STD state. If a deadlock or specified state is found, it will generate a list of the sequence of events which would take the system from its initial state to the target state. This is written into an event log which can then be replayed via the Simulator.
Simulator	This is a Petri net engine, capable of executing the Petri net and logging the transitions fired into an event log. The analyst can replay the event log to review how the system reached its current state. The Simulator is controlled using a control panel analogous to a cassette tape player. The analyst can interact with the Simulator to simulate real-world events and can see the actions which the control system would take. The Simulator is often used with the STD Monitor. However, it can also be linked to external packages (via Windows DDE) for detection of conditions and for visualisation.
STD Monitor	Animates the diagrams which were used to specify the logic, taking input from either the Simulator or the live control system.
Code Generators	Generate software for the target environment, using the Petri net as the definition for the logic and configuration files to define how the code is to be produced and I/O translations. There are currently three supported code generators: ANSI C Generates scan-based or interrupt-driven C code. The scan-based code can be either data driven (the Petri net is effectively loaded into arrays) or code-oriented where each transition is translated to inline code. The latter variant was included for a user targeting the code at micro-controller applications where RAM is at a premium. Built-in event-logging in the generated code provides diagnostics which can be used in conjunction with simulation/visualisation for fault finding. Neuron C Generates Neuron C code to map the logic to nodes on an Echelon LonWorks network. Ladder Logic Generates relay ladder logic for an Allen-Bradley PLC5 PLC.

8.2 Synect Method

8.2.1 The Object Hierarchy

An application is considered to consist of a strict hierarchy of objects. The topmost object is referred to as the root object and is the most abstract view of the application. This root object will typically consist of other objects, referred to as child objects (the root object is the parent of these children).

An object typically models an item in the system being controlled. For example, a manufacturing application might have separate objects for a machine, a robot and a conveyor system. An object which has no children is called a primitive object. An object with children is called a composite object.

An object bounds the functionality of the item of interest by defining:

- The messages which it can be sent and which it will return.
- The interface with the controlled system (real world inputs and real world outputs).
- The sequential logic within the object (using state transition diagrams).

8.2.1.1 Object Interaction

Objects interact in one of two ways. The first is by sending and receiving messages from other objects. This is described in more detail in the next section. The other type of interaction allows an object's STD to use the state of another STD as a condition of a transition. For example, if a light bulb is controlled by a switch, the first method would require that the switch object sends the light bulb object a message when the switch changes state from "off" to "on". The second method would require that the light bulb object monitors the state of the switch object and illuminates when it detects that the light switch is in state "on".

If centralised control code is to be generated, either method can be used. If, however, the logic is to be distributed across multiple processing nodes, such as when using the Synect Distributed Neuron C Code Generator for use with Echelon's LonWorks technology, the state-reference method must be used. Synect requires that two objects which are to be assigned to different nodes must not use message-based communication between them. This enforces the LonWorks philosophy that a node shares its information with other nodes on the network, to which the receiving node can react as appropriate, rather than

one node explicitly commanding another node. Objects which are to be assigned to the same node can use either or both of the message-based and state-reference methods.

8.2.1.2 Messaging

As stated in the previous section, parent and child objects can communicate with each other by sending messages. A message from a parent to a child is referred to as a command. A message from a child to its parent is referred to as a response. An object manages its children on behalf of the rest of the application - in figure 48, for example, object A can only communicate with objects D and E via object B. Similarly, objects F and G can only communicate with object B via object D.

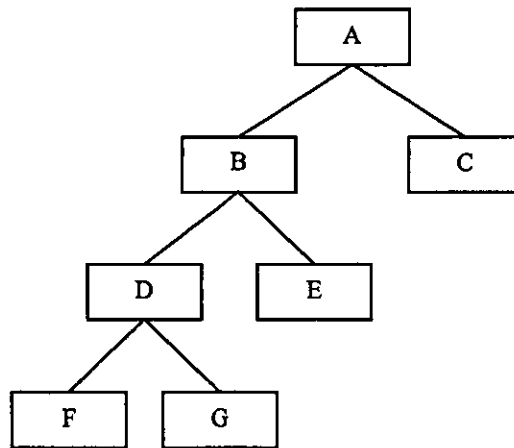


Figure 48 Object Messaging Visibility

A message can be either synchronous or asynchronous. A synchronous message is one which causes the sending transition and the receiving transition to fire as one. It therefore synchronises the two transitions. If the receiving STD is not in a position to receive the message, the transition in the sending STD cannot fire. A synchronous message is identified by the square brackets around the message name, for example:

[start_motor]

An asynchronous message is one which can be sent irrespective of whether the receiver is yet in a position to act on it. The message is placed in a buffer and is then said to be pending. If the message is already pending (i.e. the buffer is full), the transition in the sending STD cannot fire. Use of asynchronous messages can cause a substantial increase in the number of combinations of state that the application can reach, possibly to

the extent of preventing the use of the Analyzer. It can also make the behaviour of the application more difficult to follow.

8.2.1.3 Interface with the Controlled System

Real world inputs enable an object to read the value of sensors in the system being controlled (such as whether a switch is closed). Real world outputs enable an object to instruct the system being controlled to take some action (such as starting a motor).

A real world input is an input into the application from the controlled system. A real world input can be thought of as a boolean function which the application calls. For example, the following real world inputs may be available for a motor:

- motor_stopped
- motor_running
- motor_running_full_speed

Synect is independent of the target control system hardware. As such, the means by which a real world input function determines the boolean state to be returned is only defined at the code generation stage.

A real world output is an output from the application to the controlled system. A real world output can be thought of as a function which the application calls. For example, the following real world outputs may be available for a motor:

- stop_motor
- start_motor
- run_motor_at_full_speed

Again, because Synect is independent of the target control system hardware, the means by which a real world output function causes the controlled system to take the required action is also defined at the code generation stage.

8.2.1.4 Internal Events

An object may also have internal events. An internal event is either an internal command, a variable test or a variable operation.

8.2.1.4.1 Internal command

An object may contain several STDs to define the required sequential logic. These STDs may communicate with each other just as objects do - by sending messages. These internal messages are called internal commands. Internal commands may be synchronous or asynchronous.

8.2.1.4.2 Variable

An object may contain variables. A variable has the following configuration attributes:

- name
- minimum value
- maximum value
- initial value

The values are subject to the following constraints:

- minimum value \leq maximum value
- minimum value \leq initial value \leq maximum value

The variable has an integer value which can be changed by a variable operation and tested against by a variable test.

8.2.1.4.2.1 Variable test

A set of tests may be defined which are applicable to each variable, where each test is one of $<$, \leq , $=$, \geq or $>$. A variable test can then be used as a condition on a transition. A transition cannot contain more than one condition referring to any particular variable.

8.2.1.4.2.2 Variable operation

A set of operations may be defined which are applicable to each variable, where each operation is one of:

INCR	increment the value by 1 provided that the variable value is less than its maximum.
DECR	decrement the value by 1 provided that the variable value is greater than its minimum.
RESET	reset the variable to its initial value.

A variable operation can then be used as an action on a transition. A transition cannot contain more than one action referring to any particular variable.

8.2.2 State Transition Diagram

The logic within each object is defined by one or more state transition diagrams (STDs). Each STD typically consists of several states and several transitions.

8.2.2.1 State

Each state is represented by a unique name which is displayed in the STD window in a rectangle. Each state represents an identifiable mode of operation of the item being modelled. For example, a gripper could have states denoting "open", "closing", "closed" and "opening". One of the states is designated as the initial state and is drawn with a thicker border in the STD window. The STD's current state will change to another state if a transition fires which starts at the current state and ends at a different state. An STD is said to be in a particular state - for example, the gripper STD is in state "closing".

A state may be either a primitive state or a macro state. A macro state is drawn with a very thick border and is further decomposed into a sub-sequence STD.

8.2.2.2 Transition

Transitions define, for each state, which other states are directly reachable. A transition is represented by an arrowed line where the arrow shows the direction from start state to end state. In the following example of a kettle control system, the STD starts in state "Empty". The only other state reachable from state "Empty" is "Off Not Boiled" (presumably when the kettle has been filled with water). From state "Heating", the STD may return to state "Off Not Boiled" (probably because the user has decided to abort the sequence) or state "Boiled" (boiling water detected). From state "Boiled", the STD may return to state "Emptying" (probably because the user has decided to empty the kettle) or state "Empty" (boiling water detected).

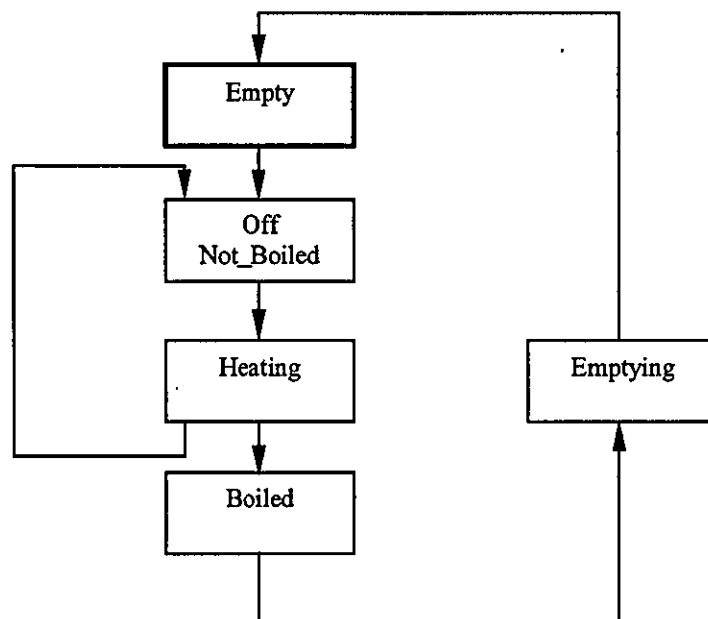


Figure 49 States and Transitions

Associated with each transition is a (possibly empty) set of conditions. If these conditions evaluate to true (an empty set always evaluates to true), a transition which has the current state as its start state is said to be enabled and can fire.

Also associated with a transition is a (possibly empty) set of actions. When a transition fires, the current state changes to the transition's end state and the actions are invoked. The transition is considered to fire instantaneously such that the STD is always in one of the defined states.

8.2.3 Justification

This section justifies the method with respect to the requirements specified in chapter three, Requirements for Method, Model and Tool, which are summarised as subsection headings.

8.2.3.1 Clear, Concise and Complete Specification

Graphical notations express the decomposition of the system in terms of a hierarchy of objects along with state transition diagrams for sequence definition. The use of appropriate notations helps to ensure clarity without verbosity. The scope of the system is recognisable from the object hierarchy, helping to identify omissions. Although the method cannot automatically identify incompleteness in the detail, it facilitates an effective means of review. Considering the current method as a common denominator for sequence oriented systems in general, domain specific methods capable of revealing incompleteness in the detail could be developed as additional layers. For example, an S88.01-aware variant might expect to find logic for running, holding and stopping phase states.

The use of a hierarchy of objects encourages the analyst to express both the coordination and independent behaviour required of the system.

8.2.3.2 Manage Complexity

The analyst decomposes the overall solution into manageable parts by modelling the solution in terms of a hierarchy of communicating objects. The specification can proceed bottom-up, top-down or middle-out. As proven re-usable objects become available at the lower levels, the analyst has even less complexity to deal with.

8.2.3.3 Notations

Most structured and object oriented methods show data or event relationships between either modelled entities or functional entities. The object hierarchy diagram shows how the analyst considers the objects to be related, the message communication paths implicitly (parent to child and child to parent but not grandparent to grandchild, for example), and the specific messages sent and received. It also shows the inputs from the plant and outputs to the plant. This external interface uses the four sides of the object rectangle to ensure that the representation is explicit and unambiguous.

The object hierarchy shows message coupling between objects explicitly although a weakness is that coupling through state-references are not shown. Explicit representation of coupling helps to promote re-usability. However the CASE tool identifies illegal state-reference coupling, preventing design errors being propagated into the Petri net model.

The state transition diagram (STD) was selected for expressing sequence behaviour in preference to sequential function chart (SFC) because it is widely understood and offers a more compact representation than afforded by SFCs. The STD is also perceived as more design oriented whereas the SFC is perceived as more implementation oriented. The author is unaware of any structured method or object oriented method which uses SFCs although several use variants of the STD.

8.2.3.4 Coherent Information

Where possible, related information should be shown on the same diagram. This helps to reinforce the context of the item under consideration and makes the paper representation more manageable. In accordance with the method and tool being oriented at small to medium scale applications, the object hierarchy shows the entire hierarchy on one diagram. The use of the Mealy model of state transition diagram provides the most compact representation of sequence definition, preventing the need to fragment the definition over several diagrams or pages.

8.2.3.5 Avoidance Of Design Errors

Whereas the Synect method will allow the analyst to specify a "wide flat" hierarchy and use state-reference communication between any and all objects, the method encourages a more structured approach with coordination and arbitration of child objects being managed by their parent object.

The restriction that messages can only be used for communication between adjacent levels in the object hierarchy facilitates analyses of subsets of the whole system. An object can be made the root object of a subsystem by discarding its parent, grandparent etc.. Similarly, an object's children can be discarded to restrict the scope of the subsystem. The absence of grandparent to grandchild communication ensures that this operation does not remove circular dependencies which would otherwise cause deadlock. In fact, analysis of the subsystem may reveal a deadlock which was not apparent in the whole system, because the ordering of events in the whole system prevented the subsystem from following a path which lead to deadlock. Subsystem analyses are therefore valuable in order to produce well-behaved re-usable objects.

One of the limitations of Petri net reachability analysis is the non linear growth in computation time and tree size with increasing size of Petri net. This limitation tends to render reachability analysis unusable for industrial scale applications. Chapter six, Mathematical Models, describes approaches which have been taken to address this problem. The isolation and independent analysis of small subsystems [228] offers another approach - because the subsystem is less complex than the whole system, it follows that the reachability tree of the corresponding Petri net is smaller and hence may be of manageable size.

The use of a message event rather than raising or lowering of a flag warrants explanation, having been chosen so that the CASE tool, through the use of the mathematical model, can identify design errors which would otherwise be missed. Two forms of message are supported – synchronous and asynchronous:

- For a transition in an STD to fire which sends a synchronous message, the transition in another STD which receives the message must also fire (these are mapped to one Petri net transition by the Synect compiler). If the receiving STD is not in the required state, the transition in the sending STD cannot fire.
- When an STD sends an asynchronous message, it is placed into its “pending” buffer, of capacity one, from which the receiving STD consumes it. If the pending buffer is full (i.e. message not yet consumed), the transition in the sending STD cannot fire.

In both of these cases, a design error which prevented the receiver from consuming messages would cause the sender to hang. Considering a subset of the total system, this typically leads to deadlock which can be identified from the Petri net's reachability tree. Alternatively, the anomalous behaviour is apparent when the model is executed by the Synect Simulator.

A disadvantage of this type of messaging is that it can initially appear to be more difficult to construct error-free applications. Although this reflects the need to invest more effort early in the project which is recovered by less fault rectification later, the perception that the method is difficult to use could deter a potential user of the method.

In addition to being useful in motion control applications, where the profiles of two or more independently controlled axes must be coordinated, synchronous messages also remove the need for handshake messages to indicate that the receiver has acted on the instruction.

Having justified the benefits of a highly structured communication mechanism, the support for state-reference communication between objects anywhere in the hierarchy appears anomalous. It was introduced primarily to fulfill the expectations of developers of Echelon LonWorks applications [229]:

- Sibling communication is the norm instead of a hierarchy.
- Objects make their internal state available for other objects on the network to react to rather than explicitly sending the other object a command.

8.3 Petri Net Model

This section justifies the selection of an Ordinary Petri net model with respect to the requirements specified in the previous chapter, which are summarised as subsection headings. In general, the Ordinary Petri net model was chosen because it is well established and there is a substantial body of research output on which to build. These factors offer the optimum likelihood of industrial acceptability and capability to support the required analyses.

8.3.1 Visibility

The analyst can use the method without needing to understand the structure or behaviour of the Petri net model because it can be derived from the object and STD definition without additional specification. However, the simple constructs which are required to define an Ordinary Petri net, coupled with the intuitive graphical representation of places and transitions, enable mathematically averse analysts the opportunity to understand the underlying formalisms and how the model is generated.

8.3.2 Ability to Execute and Analyse

The Petri net model is inherently executable by following the rules for firing of transitions. The close relationship between the Petri net model and the analyst's specification enables the state evolution to be represented to the analyst in terms of the original diagrammatic specification.

Generation of the Petri net's reachability tree facilitates analysis of many different behavioural properties of which two have been pursued in Synect:

- **Deadlock detection.** There is anecdotal evidence of deadlocks having been found in industrial applications late into integration testing. Considering the relative costs of errors found at this stage compared with errors found early in a project, deadlock detection could be highly beneficial.
- **State search.** The ability to verify that the application cannot reach a particular combination of states, or alternatively to show how the application can reach that state combination, provides a behavioural view of the system which complements the structural view on which the object-based specification focuses.

8.3.3 Support for Code Generation

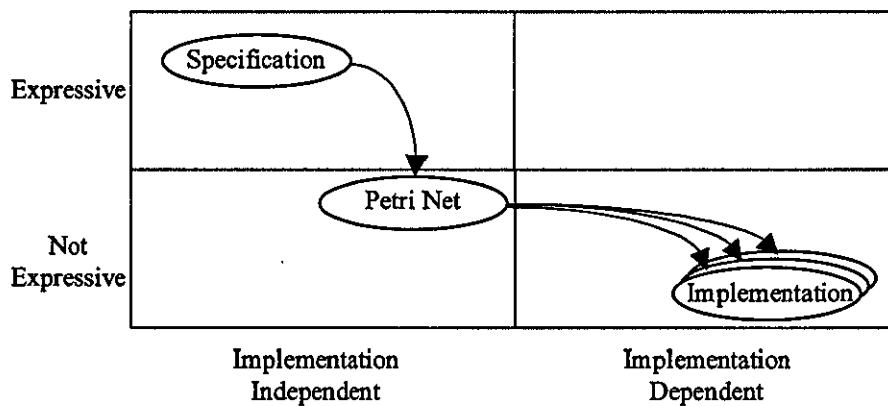


Figure 50 Petri net as an Intermediate Stage between Specification and Implementation

Figure 50 shows that a Petri net model provides a suitable intermediate representation between a highly expressive, target-independent specification and target-specific implementations.

The ease with which a Petri net execution engine can be defined translates into ease of code generation strategies. Having verified that the model accurately reflects the analyst's intent, the translation into code must be as simple as possible to maximise confidence in the generated code and to increase the viability of supporting many different target control architectures.

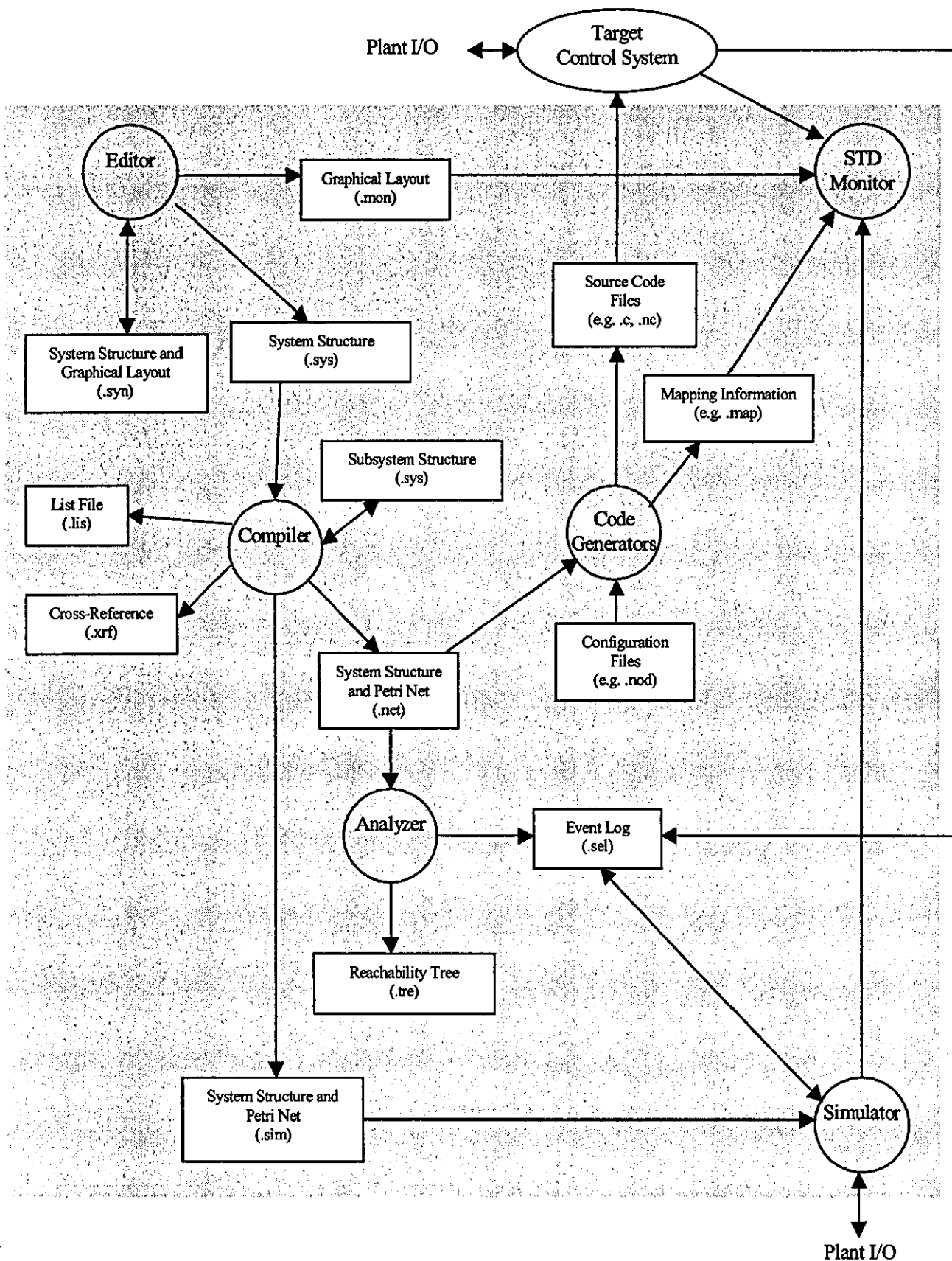


Figure 51 Dependencies between Synect Tools

8.4 **Synect Tools**

This section describes the Synect tools in more detail. Full details of the use of the tools is given in the Synect User Guides [227]. This section concentrates on an overview of each tool's capabilities and salient features.

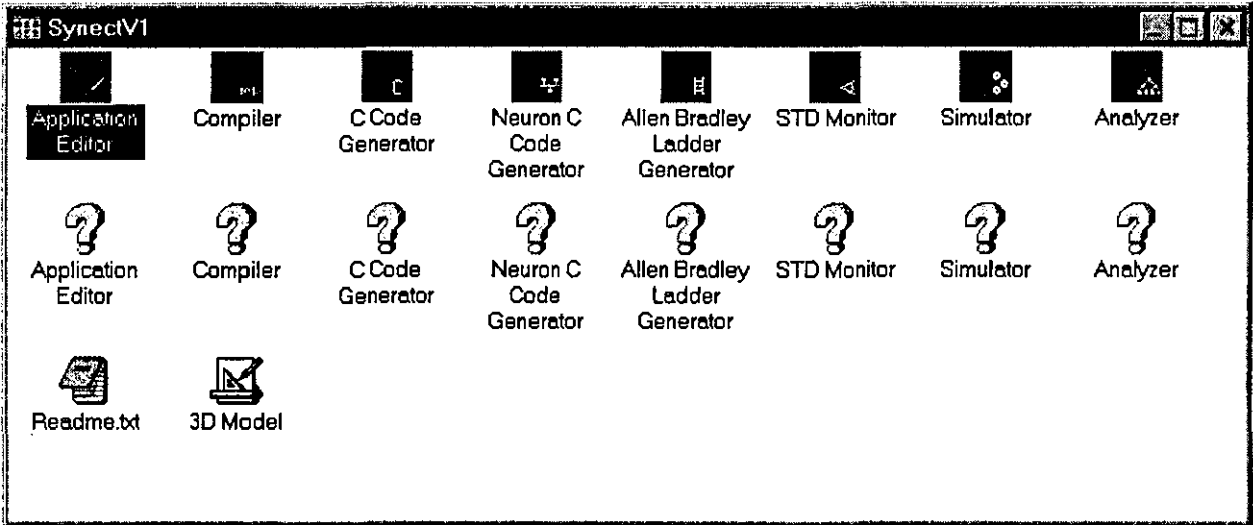


Figure 52 Screenshot of Program Manager Group showing Synect Tools and On-Line Help.

Figure 52 shows the Windows program group after installation of Synect. Each tool has its own icon and associated on-line help. There is also a simple wire-frame 3D modeller and a readme file which do not constitute part of the Synect software.

Figure 51 shows the Synect tools and the information dependencies between the tools. Each tool will now be described, using figure 51 to provide the context in which the tool is used.

8.4.1 Application Editor

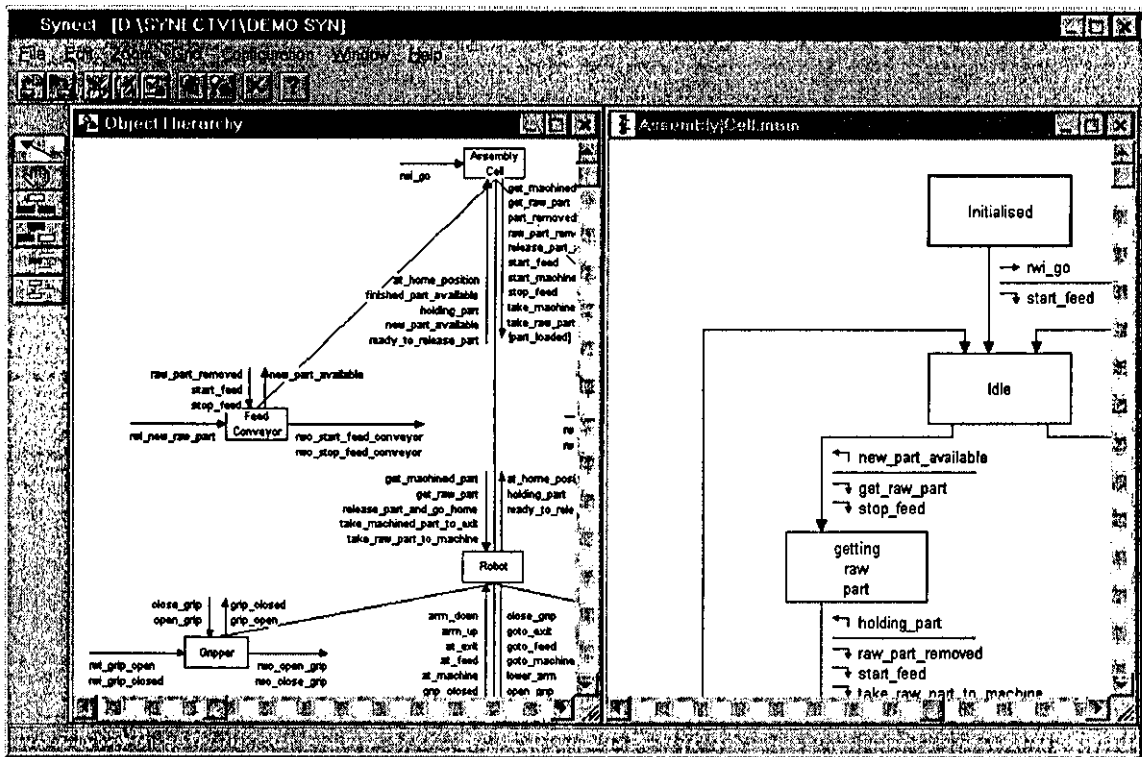


Figure 53 Screenshot of Application Editor

The Application Editor is the intelligent drawing tool which supports the Synect method through its knowledge of the syntax of object hierarchy diagrams and state transition diagrams. This helps to guide the analyst to follow the method and prevents, as far as possible, the introduction of errors which must be found and removed at a later stage. For example, a transition has a start state and an end state, a set of conditions and a set of actions. If a state is moved on the diagram, the transition will be moved accordingly. Wherever possible, pick-lists are used to prevent the need for information to be typed more than once in order to prevent typographical errors. Lists of real-world inputs and real-world outputs can be defined in a text file and loaded into the Application Editor to minimise further the need to re-type information which has already been typed. This also supports the goal of developing a tool to increase productivity - using traditional drawing tools, engineers still spend significant amounts of time simply driving the software rather than expressing their design. The Application Editor supports the standard Windows facilities of cut and paste, printing, on-line help etc..

The Application Editor produces three files:

- All of the information necessary to reload the application (with file extension “.syn”). This is stored in a proprietary format.
- A definition of the application's structure (i.e. without the graphical layout information) for the Compiler (file extension “.sys”). For maximum openness, this is produced in a text file format so that alternative formal methods derivations could be added by third parties.
- The graphical information for the STD Monitor tool.

8.4.2 Compiler

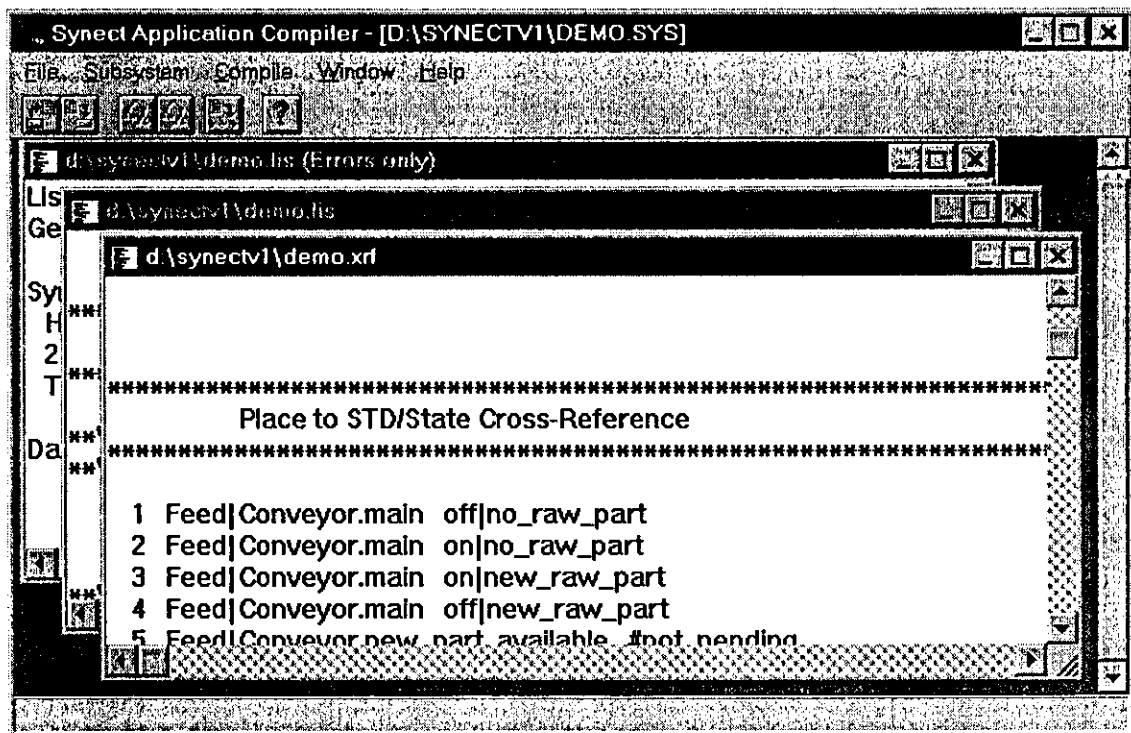


Figure 54 Screenshot of Compiler

The Compiler takes the definition from the file generated by the Application Editor (file extension “.sys”) and, provided it is error-free, derives an Ordinary Petri net (file extension “.net”). A flat Petri net is used because it offers more analysis potential than is currently available for hierarchically organised nets [230].

A list file containing errors and warnings is produced (file extension “.lis”) along with a cross-reference listing of Petri net to user specification (file extension “.xrf”) for traceability. The cross reference listing and the derived Petri net are written to text files to make the tool open such that alternative forms of Petri net analyses may be added by third parties.

The Compiler also enables the user to generate Petri nets corresponding to sub-sets of the whole application. This is provided so that sub-system analyses can be performed on a correspondingly smaller reachability tree to provide more confidence in the behaviour of individual components and component clusters. This circumvents the state explosion problem inherent in attempting to generate the reachability tree for an industrial scale application.

The Simulator information is written to an additional file (file extension “.sim”).

8.4.3 Analyzer

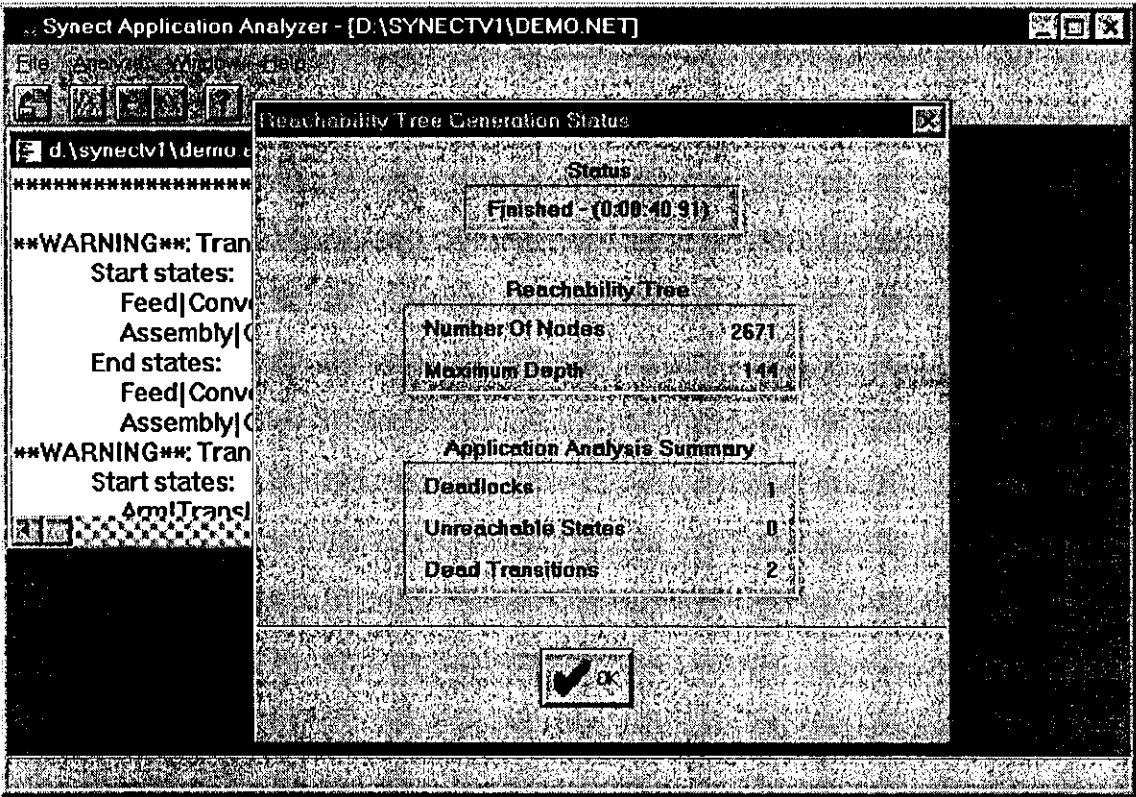


Figure 55 Screenshot of Analyzer

The Analyzer takes the Petri net (file extension “.net”) and derives the reachability tree (file extension “.tre”). It also gives feedback regarding whether any deadlocks have been found and enables the user to specify state combinations for which the reachability tree will be searched. If any are found, the Analyzer will generate a Simulator event log containing the transitions to take the Petri net from its initial state to the reachability tree node found (file extension “.evl”). The Analyzer can also identify dead transitions (transitions which can never fire).

8.4.4 STD Monitor

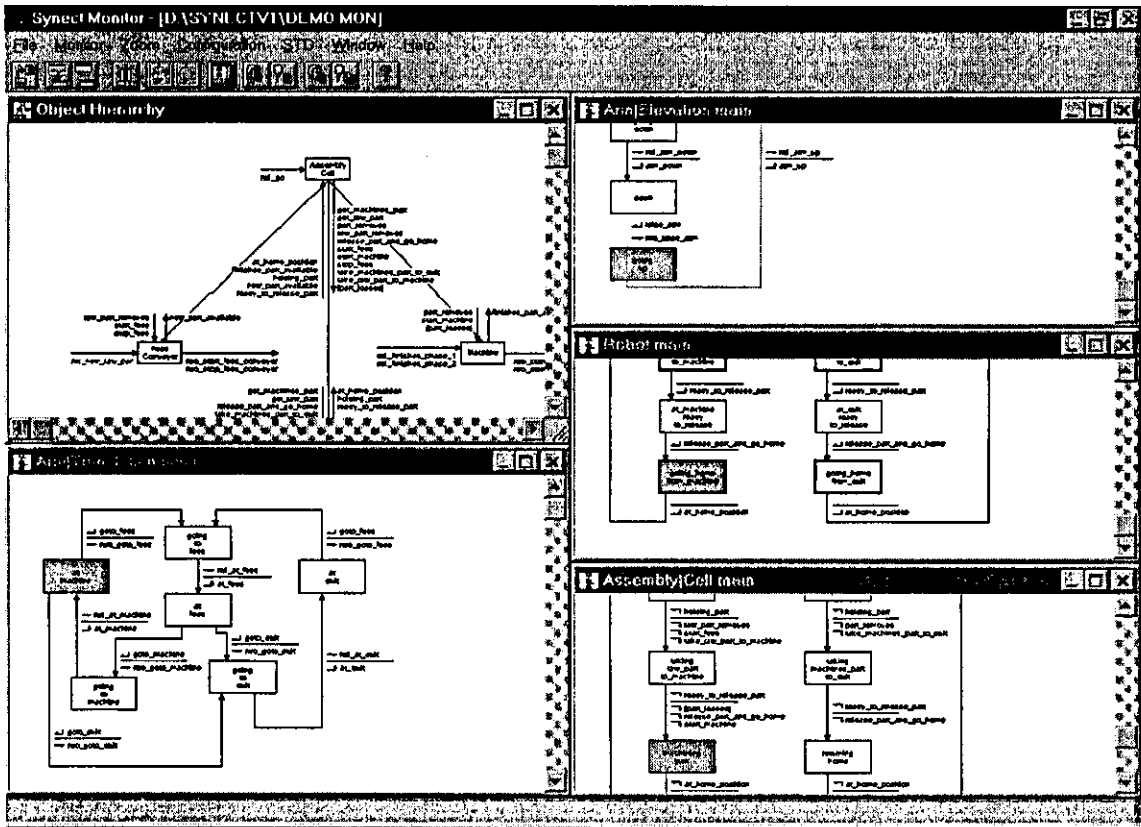


Figure 56 Screenshot of STD Monitor

The STD Monitor animates the object hierarchy and STD diagrams with which the developer specified the control system logic. It obtains its data about the control system behaviour from either the Simulator or the live control system via Windows DDE. If monitoring a LonWorks system, for example, the STD Monitor tool loads a mapping file generated by the code generator which defines where the necessary information resides in the control system.

The displays are colour-coded. The object hierarchy window shows the status of real-world inputs (blue for not-being called, red for being called and returning false, green for being called and returning true) if data is being sourced from the Simulator. The real-world inputs on STD transitions are similarly coloured. The STD also shows the current state in grey and transitions which are enabled in the Petri net in green. The STD Monitor can also be configured to pan the window over the STD so that the current state is always visible.

8.4.5 Simulator

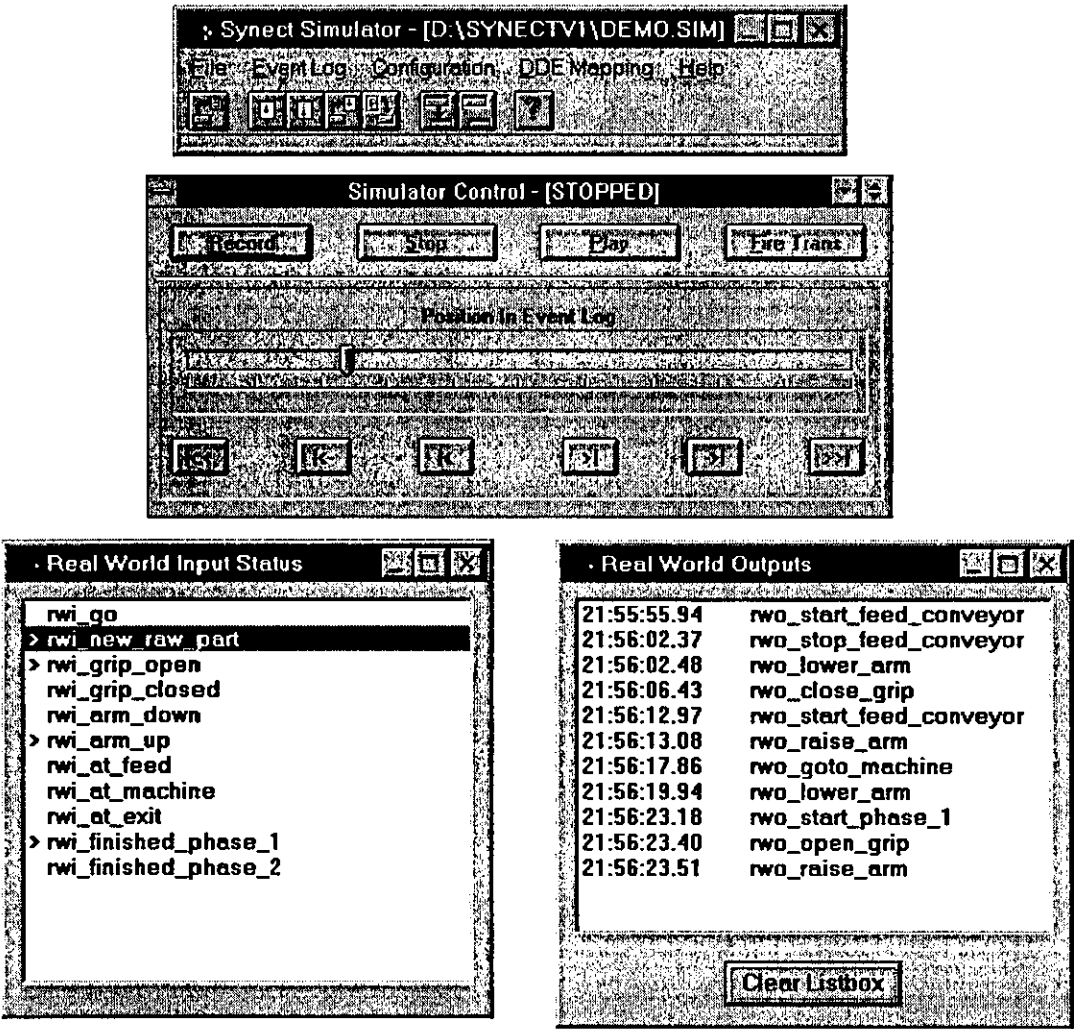


Figure 57 Screenshot of Simulator

The Simulator executes the Petri net model produced by the Compiler (file extension ".sim"). The Real-World Inputs listbox shows all real-world inputs. The user simulates their status by selecting or de-selecting entries in the listbox. As a real-world output is invoked,

a timestamped entry is appended to the Real-World Outputs listbox. The user controls the Simulator via a control panel using buttons modelled on a tape recorder:

- **Record** – instructs the Simulator to execute the model, recording transitions fired in an event log.
- **Stop** – stops the Simulator
- **Play** – replays an event log
- **Fire Trans** – the Simulator may be configured to pause between firing transitions. In this case, the user instructs the Simulator to continue by means of the Fire Trans button.

The event log generated by the Simulator can be saved to disk to be reloaded and replayed at a later date. An event log may also have been created by the control program written by one of the Synect code generators for diagnostic purposes. In this case, the Simulator enables the user to visualise the behaviour of the control system up to the time when the event log was copied. Event logs generated by the Analyzer can also be replayed.

The Simulator offers DDE connectivity to source the status of real-world inputs and also to invoke real-world outputs. This can be used for:

- Alternative visualisations during the design process, such as process plant mimic drawn on a SCADA product or 3D model.
- Training by means of the SCADA interface.
- Performance analyses, such as cycle times, machine utilisation calculations. This requires the other end of the DDE link to contain timing data.
- Testing the control system hardware. The Simulator can be used to drive the live hardware, offering a superior debugging environment to that normally available.

8.4.6 ANSI C Code Generator

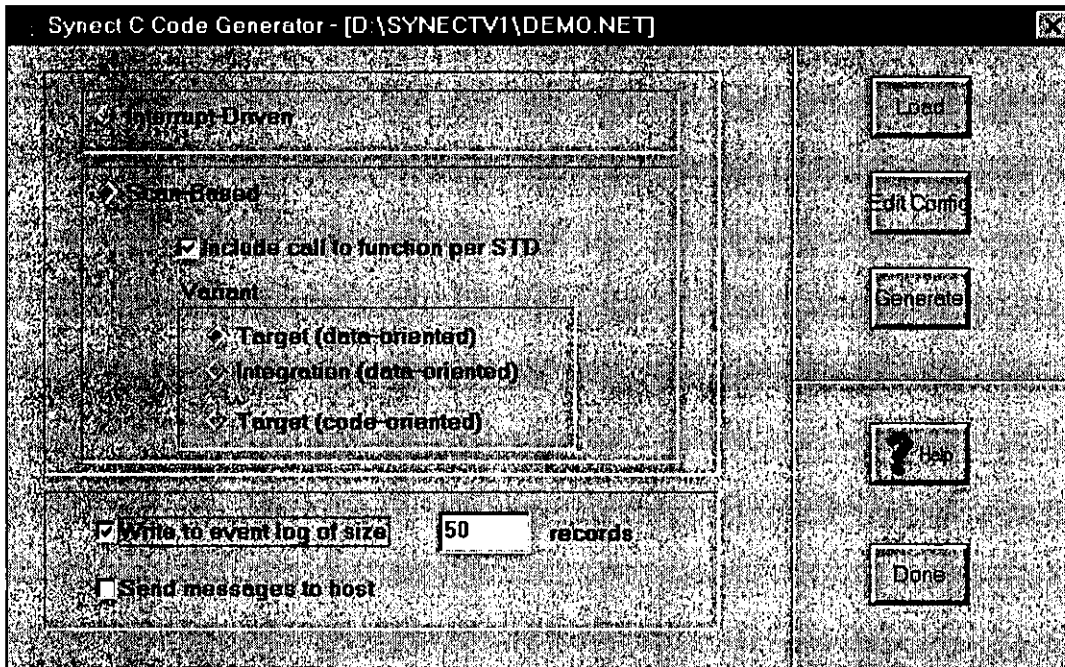


Figure 58 Screenshot of ANSI C Code Generator

The ANSI C code generator writes files containing ANSI C code to implement the control logic defined by the Petri net (loaded from file with extension “.net”). The user specifies the form the code is to take:

- Scan-based – the Petri net engine tests each transition, firing those which can be fired, and calls a user-supplied function at the end of the scan to pause for a user-controlled period.
- Code oriented – inline code is generated to implement the Petri net, with an “if” statement per transition. This variant was developed for a microcontroller target platform where RAM is at a premium – the code element being ROMable. This variant provides the user with the facility to use a configuration file to define C code for each real-world input and output to avoid the need for calls to functions which would be an unacceptable overhead for the microcontroller environment.
- Data oriented – the generated code contains arrays defining the relationship between transitions, places, real-world inputs and real-world outputs. The

generated code contains a Petri net engine to execute the net as defined by the data in the arrays.

- **Target** – the generated code contains no diagnostic statements and is suitable for the target control system platform, such as an OS-9 hosted system.
- **Integration** – the generated code contains “printf” statements to produce an audit-trail of the execution of the control system. It is referred to as “integration” because it was envisaged that this option would be used as the various hardware elements of the control system were being integrated.
- **Interrupt-driven** – by default, the Petri net engine is paused. A real-world input is required to generate an interrupt and call an interrupt-level real-world input function. This in turn, tests the transitions in which the real-world input is referenced and, if enabled, fires the transition.
- **Diagnostics** – the generated code can include event logging and host messaging. Event logging causes the control program to record, in a circular file, when a real-world input is tested and a transition fired. This file can then be copied to the development PC for replay via the Simulator. Host messaging causes the control program to call a user-supplied function to notify the host (such as a development PC) when a transition is fired.

8.4.7 Neuron C Code Generator

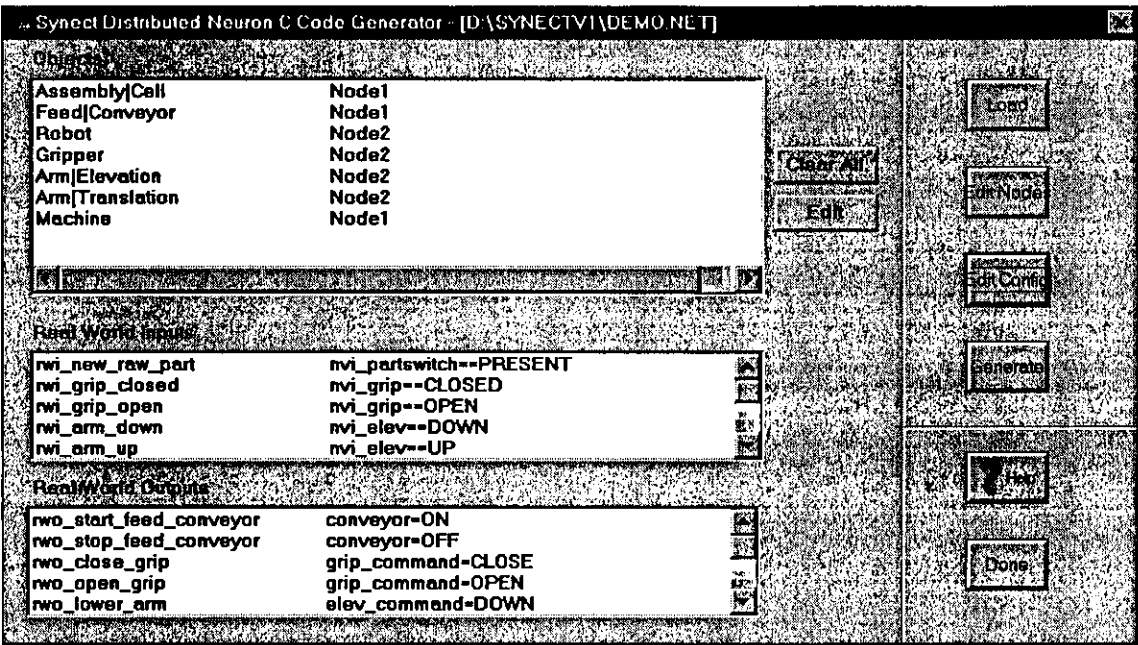


Figure 59 Screenshot of Neuron C Code Generator

The Neuron C Code Generator loads the “.net” file produced by the Compiler and writes code which is suitable for an Echelon LonWorks target environment, reading implementation-specific configuration data from a manually prepared text file. Each node on a LonWorks network is programmed in Neuron C which, as its name suggests, is a modified form of C. One of the most significant differences is the use of the “when” statement which is similar to the C “if” statement but adds the condition to the Neuron chip’s scheduler. The code can therefore be considered to be interrupt driven.

The code generator requires the user to define the mapping of Synect objects to LonWorks nodes. Objects which communicate by sending messages must be on the same node. This avoids the complication of a node being reset but leaving a message pending in a buffer which is not apparent to the developer. It also avoids the additional network traffic which would be required for the receiver to notify the sender that the message had been “consumed” and conforms with the LonWorks philosophy of objects making their state publicly available for others to act upon rather than instructing other objects to perform particular actions.

In addition to files containing the control program, the code generator also generates a file which can be read by the Synect STD Monitor application for monitoring the state of the live control system.

8.4.8 Allen-Bradley PLC Ladder Logic Generator

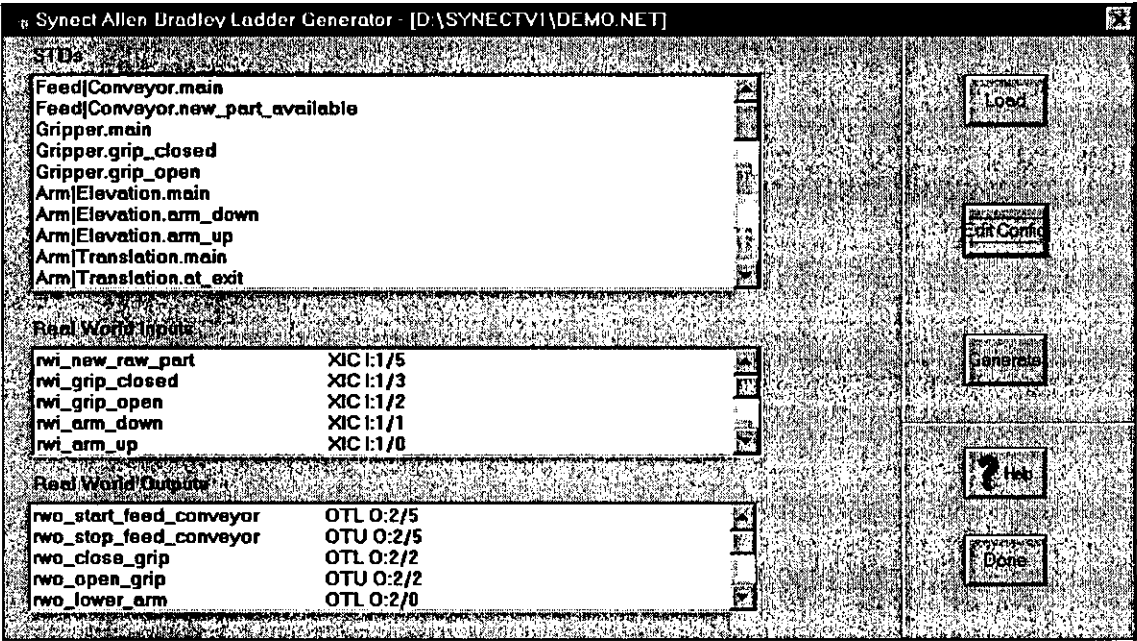


Figure 60 Screenshot of the Ladder Logic Generator

The Ladder Logic Generator loads the “.net” file produced by the Compiler and writes text files for import into Allen-Bradley programming software. A manually prepared configuration file defines the integer register to hold the current state of each STD. The tool can automatically generate state values for each state or can be instructed, via the configuration file, to parse the state name to decode the required state value. This eases the compatibility of the generated ladder logic with S88.01 batch control system platforms, which can show the current step index (or state value) of each phase.

The configuration file defines all necessary information for the code to be generated, such as the state register, prefix for decoding state values from state names, etc.. It is anticipated that such a file would be automatically generated from a central project database which would also contain the entire configuration data for all software components, such as the batch control system software, SCADA database, PLC database, etc..

The Ladder Logic Generator can include additional rungs which implement breakpoint and single step functionality for debugging purposes.

In addition to files containing the control program, the code generator also generates a mapping file which can be read by the Synect STD Monitor application for monitoring the state of the live control system.

8.4.9 Justification

This section highlights how the tools meet the requirements specified in the chapter three, Requirements for Method, Model and Tool, which are summarised as subsection headings.

8.4.9.1 Method Support

The tools in general, and the Application Editor particularly, explicitly support the method. Wherever possible, the editor prevents syntactically illegal constructs from being specified, such as a transition without a start or end state. Where usability would be compromised by rigidly enforcing this approach, such as following the use of cut-and-paste functionality, the editor makes inconsistencies apparent on the specification. However, it is still possible for the analyst to specify an error which the editor ignores but is reported by the compiler.

8.4.9.2 Usability

The tools run on low-cost PC hardware under the Microsoft Windows operating system using the Windows look and feel. In conjunction with the design principle of always providing the user with maximum information, the method and tool can be self-taught with the aid of comprehensive user manuals.

8.4.9.3 Integrated Development Environment

The tools are designed to work together, using the analyst's specification for graphical representation and the Petri net model as the formal definition of the application. Support for all phases of the life-cycle is therefore available without the discontinuities which disparate tools would impose.

The overall Synect functionality is split between several focussed tools to provide a simple and effective means of access control to be implemented. For example, a user organisation could provide an engineer account which offered access to all tools and an operator account which could only use the STD Monitor. Synect can be used to satisfy a range of needs including:

- Drawing tool for developing and documenting sequence logic.
- Development of prototype sequence logic with visualisation provided by a 3D model. This does not require the use of the STD Monitor or a code generator.
- Code generation of very simple sequence logic for a particular target platform. Visualisation may not be required and only one of the code generators is required.

8.4.9.4 Rapid Prototyping and Visualisation

The tools support rapid development and iteration of design. Verification of correct behaviour is facilitated through the analytical detection of design errors such as deadlock and unreachable states, complemented by the ability to execute the model with animation of the analyst's specification. Alternative visualisations, which are available due to the connectivity offered by the Synect Simulator, increase the effectiveness of review by supporting alternative perspectives of the application's behaviour. Operator training can also benefit from this functionality.

Performance characteristics can be derived by linking an external model of plant equipment behaviour, for example mimicking the length of time a valve takes to open, to the Synect Simulator.

8.4.9.5 Documentation

The Application Editor is able to print the analyst's specification and supports cut-and-paste so that the diagrams may be incorporated into documentation of the analyst's choosing.

The tools also produce text files containing traceability information which can be printed for documentation. Examples include the Petri net model, the reachability tree and cross-reference listings.

8.4.9.6 Automatic Code Generation

The specification and Petri net model are completely independent of target control system platform. The translation from target independence to generation of specific control system software is implemented by the various code generators, with the mapping defined as configuration data either supplied interactively or via a text file.

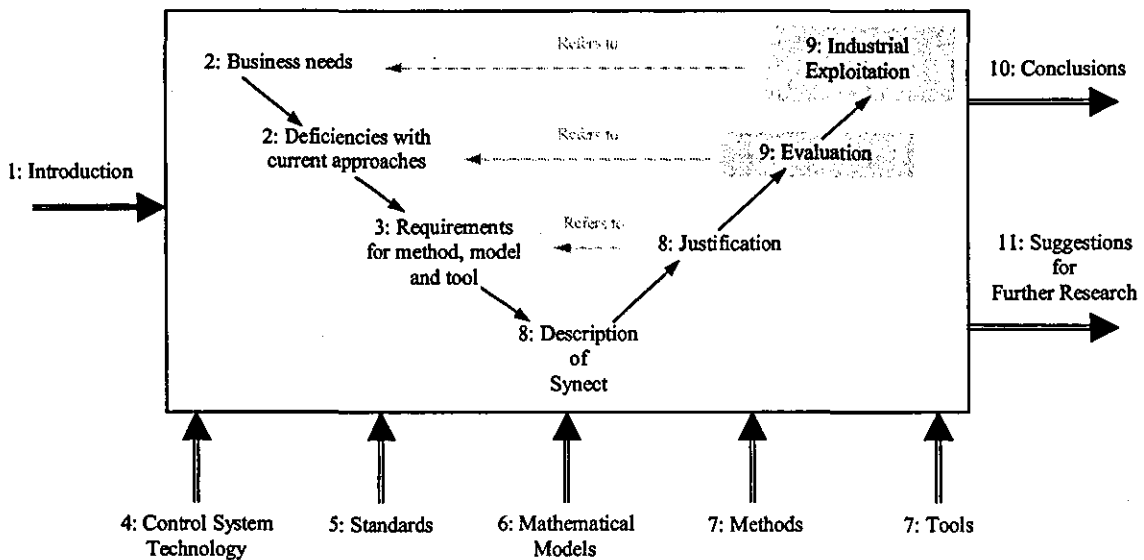
The ease with which the Petri net model can be translated into target control system code ensures that the most significant difficulty is in establishing a generally acceptable structure for the generated software. This is of particular relevance to the PLC community where different end-users would require different structures for the same application running on the same PLC. Algorithms for executing the Petri net on a PLC, such as matrix or list-based, are described in [231]. Synect is also capable of generating interrupt-driven code, for use with operating systems such as OS-9 or Microsoft Windows, resulting in more efficient use of the processor.

The code generators support a range of options to assist with diagnosis and troubleshooting of the live control system, including:

- Automatic event-logging, where the target control code writes an audit-trail of transitions fired. The audit-trail can be replayed via the Synect Simulator with the analyst's diagrammatic specification being animated via the Synect STD Monitor.
- Generation of cross-reference data so that the Synect STD Monitor can display the current state of the control system by animating the analyst's specification. This also facilitates derivation of maintenance and performance statistics for use by corporate computer systems.
- Debug options to prevent transitions from firing automatically so that sequences can be single stepped.

Chapter 9 Evaluation and Industrial Exploitation

This chapter is a critical appraisal of Synect with reference to the business needs. Whereas the previous chapter described the Synect method, model and tools and correlated attributes of each of these aspects with the requirements stated in chapter three, Requirements for Method, Model and Tool, the focus now returns to the business drivers and examines how effectively Synect addresses the deficiencies which were identified with current approaches. For a more detailed example of application development with Synect, the reader is referred to the walk-through in appendix A.



The information herein is structured as follows:

- In order to demonstrate the applicability of Synect to a variety of real needs, the salient characteristics of a selection of applications are described.
- The deficiencies with current approaches, identified in chapter two, The Need, are listed and the corresponding strengths and weaknesses of Synect are described.

- To show how the Synect tools might typically be used, an example application is developed.
- The dissemination and exploitation of this research is described.

9.1 Evaluation

Against evaluation criteria established for assessing methods:

Criterion	Synect
Graphical	✓
Manage complexity	✓
Expressive notations	✓
Well-understood notations	✓
Encourage object-oriented view regarding behaviour of agents	✓
Encourage structured method view of ordering of events and coordination of agents	✓
Support decomposition	✓
Support sequence and event behaviour	✓
Coherent information	✓
Easy to learn	✓
Discrete (manufacturing) applications	✓
Hybrid (batch) applications	✓
Oriented towards analysis and design activities	✓
Formal definition	✓

Against evaluation criteria established for assessing mathematical models:

Criterion	Synect
Executable	✓
Support behaviour queries	✓
Simple mapping from method	✓
Simple mapping to implementation	✓
Graphical representation	✓
Support concurrency	✓
Established	✓

Against evaluation criteria established for assessing tools:

Criterion	Synect
Support graphical method	✓
Support mathematical model	✓
Visualisation	✓
Code generation – ANSI C	✓
Code generation – relay ladder logic	✓
Code generation – Echelon Neuron C	✓
Code generation – built in diagnostics	✓
Inexpensive	✓
Full life-cycle support	✓
Support rapid prototyping	✓
Connectivity	✓

9.2 Case Studies

9.2.1 Integrated Machine Design and Control (IMDC)

9.2.1.1 Background

The Manufacturing System Integration (MSI) Research Institute has established leading edge engineering concepts to enable the design and control of manufacturing machines. The integrated environment for machine design and control (IMDC (EPSRC ref. GR/J/57827)) project has proved the feasibility of utilising modelling technology to support consistency and correctness in the design, synthesis, visualisation, simulation and analysis, construction and configuration, distributed runtime execution and management of change in machine systems. Synect has been utilised as the application logic design and analysis tool on this project and its capabilities have been proven over across a range of applications. On the IMDC project it was used to control a complex PCB handling system with a large number of concurrent application tasks, illustrating the capability of Synect to cope with large and complex applications

IMDC was a three year EPSRC funded collaborative project in the UK, involving the Manufacturing Systems Integration (MSI) Research Institute and a consortium of software vendors, industrial control system suppliers and machine builders. The collaborating companies involved in this work were GSM Syntel Ltd, Hopkinson Computing Ltd, OO Technology Ltd, SHS Ltd and Quin Systems Ltd.

A number of follow on projects have resulted from the IMDC work. All the following industrial projects have used Synect for implementation of application specific machine control logic:

- Forming machine (and process) visualisation and control for CMB, as part of a BRITE project “New Manufacturing Processes for Thin Walled Components”, BREU2-CT94-1024.
- The capture and use of dynamic models of machines and their application in the design and control of wheel balancing machinery for Cirrus Technology, as part of a DHSM LINK integrator project with Aston University “Integrated Approach to the Design of Control Systems for High Speed Machines”, GR/K 38694.
- A distributed control system for agricultural vehicles within DHSM LINK project number GR/H/53187, entitled “Integration of Control Systems on Agricultural Tractors and Implements”.

9.2.1.2 Synect Modules Used

All Synect modules were used other than the Echelon LonWorks and PLC code generators. The target environment was a multi-tasking operating system running interrupt-driven control software which was compiled from ANSI C source code.

9.2.1.3 Results

To show the non-trivial scale of application being developed, an object hierarchy for an example IMDC application is shown in figure 61.

Synect was considered to be easier to integrate into the workbench than the other tools evaluated. This applied to both ability to link with visualisation tools, such as 3D modellers, and also to code generation capability.

Synect was also considered to be more effective at preventing errors being specified. For example, the Application Editor ensures that, wherever possible, an identifier is only typed once. Thereafter, the analyst selects the identifier from a pick-list. This minimises the risk of typographical errors.

In general, the ability to make a change to the design via the Application Editor and see the corresponding control software running on the target Universal Machine Control (UMC) rig within a small number of minutes was considered to be a strength.

The visualisation capabilities were considered to be a particular strength. The ability to interactively execute the model whilst animating the analyst's STDs gave increased confidence in the design. Driving the 3D modeller from the simulator before any code was generated added to that confidence and was a useful demonstration aid.

Although the analytical capabilities were considered useful, these were perceived as less important than the visualisation and code generation capabilities. This was due to the perceived simplicity of application and the limited consequences of an error being propagated through to implementation.

In the context of the IMDC, Synect is described in [34]. The use of Synect in the interactive visualisation of sequence logic for machine design is described in [232].

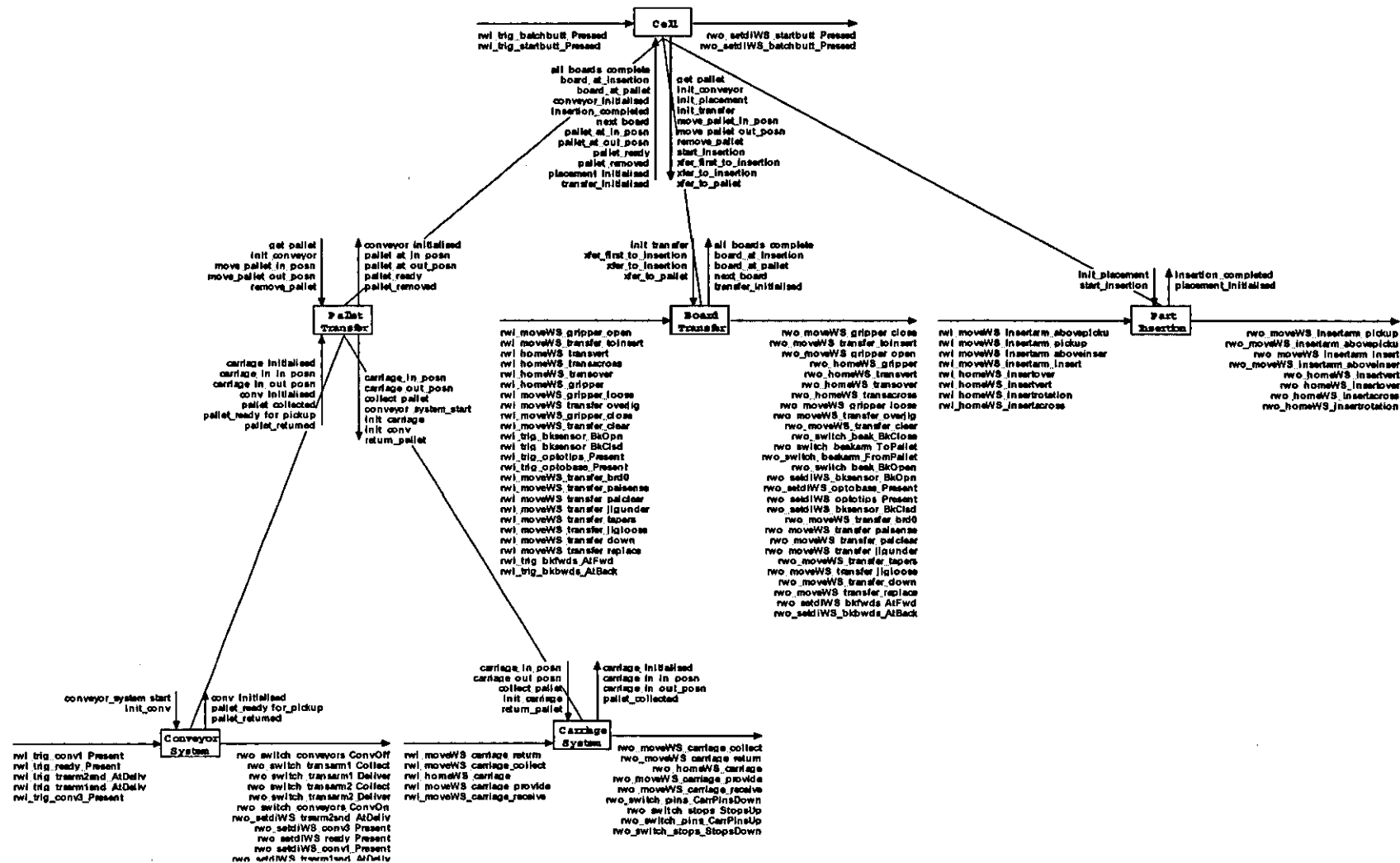


Figure 61 An Example IMDC Application's Object Hierarchy

9.2.2 Metal Forming Application

9.2.2.1 Background

This was a Brite-EuRam II funded project (number BREU2-CT94-1024) undertaken by the MSI Research Institute at Loughborough University in conjunction with a metal forming manufacturer. The project was to investigate a new manufacturing process for forming metal containers. Due to confidentiality constraints, minimal details can be provided of the application.

9.2.2.2 Synect Modules Used

Synect was used for prototyping sequence logic. A 3D modelling package, developed at Loughborough, was used for visualisation of synchronised motion control. The sequence logic for the application was specified via the Synect Application Editor with the derived Petri net model executed by the Simulator and the 3D modelling package being driven by the Simulator.

9.2.2.3 Results

A number of projects have confirmed the short learning curve associated with adopting Synect, of which this was one. A research assistant was assigned to the project and provided with the Synect software and associated manuals. Within a few days and with minimal assistance from colleagues, this analyst produced a solution and used Synect to verify its behaviour.

The ability to quickly and easily develop and refine sequence logic and the Simulator's ability to take input from and send output to the 3D modeller were considered to be strengths of Synect in this project.

This project also demonstrated the potential for an application developed by one organisation to be supported by another. During the project, the author paid a visit to the MSI Research Institute and, by examining the specification defined using the application editor, was able to describe the intended behaviour of the application to a visitor without having had any prior exposure to the application.

9.2.3 Ford Rig

9.2.3.1 Background

Ford Motor Company established a project to investigate Echelon LonWorks technology, drawing on the expertise of the MSI Research Institute. The requirement in general was to develop a control system to control a mechanical rig on the Ford Motor Company's stand at the 1998 AutoTech exhibition at the NEC, Birmingham.

9.2.3.2 Synect Modules Used

This was a short timescale project requiring the prototype development of distributed sequence logic whilst the rig was being built. The Synect Neuron C code generator was then used to produce code to run on the nodes on the LonWorks network.

This project used all Synect modules other than ANSI C and PLC code generators. In particular:

- The Application Editor was used to specify the sequence logic.
- The Analyzer was used to verify absence of deadlock and to test for reachability of specified combinations of system state.
- The Simulator and STD Monitor provided the ability to interactively drive the application to gain confidence in the behaviour of the application.
- The Neuron C Code Generator was used to generate control code corresponding to the analyst's specification.
- The STD Monitor was used to observe the evolving state of the control system through animation of the analyst's diagrammatic specification.

9.2.3.3 Results

In addition to the pressures of a short timescale project with minimal elapsed time for trialling a solution on the target control system driving the rig prior to the exhibition, the analyst was required to undertake international travel during the project whilst his assistant remained in the UK. This provided a good test of Synect's ability to support cooperative working across geographically remote sites. The application was developed iteratively with first the analyst developing the specification and testing it via the Analyzer and Simulator, followed by handover to his assistant for refinement by sending the

specification by email. After a small number of iterations, the control code was generated and the rig demonstrated as planned at the exhibition.

The application of Synect to the distributed control of manufacturing machines using LonWorks technology is described in [233].

9.2.4 Embedded Control Equipment

9.2.4.1 Background

An American company which develops embedded controllers for equipment manufacturers, including agricultural equipment, were already specifying their control logic using a hierarchy of state diagrams. They identified Synect after searching the World Wide Web for CASE tools which might improve their productivity. Of particular appeal was the ability to analytically verify the absence of deadlock and test the logic before generating control code.

The target control system was a microcontroller, programmed in ANSI C, with limited RAM but relatively more ROM. At the time, the Synect ANSI C code generator produced a data driven program, with the structure of the Petri net in arrays. This was unsuitable because it required considerable RAM but limited ROM. A new variant was therefore developed to produce inline code for each transition instead of being data driven.

9.2.4.2 Synect Modules Used

All of the Synect modules were used other than the Neuron C and PLC code generators.

9.2.4.3 Results

Once again, the ability to verify that the application would behave as required, through analytical tests and interactive execution of the model, was considered to be of significant value.

This type of application is an example of the recognised need for an appropriate method and tool in small to medium size applications. It also offers some reassurance that the use of a hierarchy to structure the application, and STDs for specifying the sequence logic, are an intuitive means for expressing the design.

9.3 Evaluation Against Requirements

Previous chapters have described current typical approaches to the development of industrial sequence logic and have identified better approaches:

- A more consistent implementation including designed-in diagnostics, such as the STEPS method.
- A graphical method supported by a CASE tool, such as Rose for general purpose object oriented modelling using UML or StateMate for complex reactive systems.
- A mathematical model supported by a CASE tool, such as SystemSpecs.
- The combination of a graphical method with a mathematical model supported by a CASE tool, such as Synect.

Requirement	Traditional Approach	STEPS	Rose	StateMate	SystemSpecs	Synect
Support seamless team working			✓✓	✓	✓	✓✓
Support and encourage re-use		✓	✓✓	✓		✓
Explicit support for S88.01			✓	✓		✓
Facilitate clear, concise and complete specifications			✓	✓		✓
Ability to verify correctness of design			✓	✓	✓	✓✓
Problem oriented			✓	✓	✓	✓✓
Consistent implementation architecture		✓✓		✓	✓	✓✓
Automatic code generation				✓	✓	✓✓
Comprehensive automatic generation of diagnostics		✓				✓✓
Easily supportable and maintainable control system		✓	✓	✓	✓	✓✓
Good documentation		✓	✓✓	✓✓	✓✓	✓✓
Good enterprise integration			✓✓	✓		✓
Modular automation			✓	✓		✓✓

Table 6 Comparison of Approaches against Requirements

Table 6 is the author's interpretation of the strengths of these approaches in satisfying the requirements identified in chapter two, The Need. An empty box indicates no support or weak support for the requirement, one tick denotes some support and two ticks denotes good support. A textual interpretation now follows.

9.3.1 Analysis and Design

9.3.1.1 Support Seamless Team Working

9.3.1.1.1 Traditional Approach and STEPS

These approaches rely on potentially large and text-based documentation in order to communicate across organisational and contractual boundaries. STEPS uses manually completed tables and templates to achieve consistency from specification to specification.

9.3.1.1.2 Rose

Rose supports the UML object oriented method, offering expressive notations and being particularly oriented to requirements capture. Although it is a general purpose method with minimal guidance regarding its application to sequence-based reactive system design, the standardisation of notations should facilitate common understanding.

9.3.1.1.3 StateMate and SystemSpecs

These methods both provide rich notations, capable of modelling complex reactive systems. However, this richness would be daunting to the unfamiliar, particularly with SystemSpecs, which uses the unfamiliar notations of high level Petri nets.

9.3.1.1.4 Synect

As described in the case studies above, Synect supports seamless team working through the use of simple notations. It can be effectively applied across geographic boundaries because the formal definition is clear and unambiguous and can be easily transmitted around the world using standard internet mechanisms.

A weakness of the Synect method is the inability to relate explanatory text to portions of the diagram. Whilst this facility could be abused, for example by making the text specification the master definition in the event of a contradiction with the diagrams, it could help to explain the behavioural intent whereas the diagrams define system structure.

Clarity of expression may be helped by supporting more abstract concepts such as different types of STD, such as cyclic, linear, arbitration, etc.. Also an STD “wizard”, as now found in many PC software packages, including control system packages, could enable greater standardisation of style, offering more rapid assimilation of the design intent.

9.3.1.2 Support and Encourage Re-use

9.3.1.2.1 Traditional Approach and STEPS

A traditional approach would typically re-use little design or code, re-use being limited to cut-and-paste of lower-level software routines.

STEPS facilitates re-use of common functionality, such as diagnostic display drivers, but is otherwise very application specific. However, in projects where an almost identical machine is required, the entire application may be copied and then relatively minor modifications made.

9.3.1.2.2 Rose

Object-oriented methods strongly promote the concept of re-usable objects. CASE tools such as Rose assist in its management by supporting version control and configuration control tools.

9.3.1.2.3 StateMate

Although StateMate provides a library management tool, the method does not guide the analyst towards structuring the solution as re-usable components.

9.3.1.2.4 SystemSpecs

SystemSpecs is oriented more at the modelling and analysis of application-specific solutions than the development of re-usable components.

9.3.1.2.5 Synect

The object-based approach whereby a piece of plant equipment is modelled should facilitate re-use on subsequent projects, particularly at the lower levels where primitive components such as an on/off motor controller are concerned. It is less likely that more abstract components, such as an assembly cell, could be re-used until the common requirements were established. However, when the value of re-use of lower-level objects is realised, there may be increased commitment towards standardisation at a more abstract level.

A weakness of the Synect method in terms of re-use relates to object communication via messaging. For example, if a child object sends a message to its parent, the parent must consume that message otherwise the child will hang if it attempts to send it again. This places a requirement on the behaviour of the parent by the child which may not be appropriate for a different application.

Tool support for re-use could be strengthened through the support for component libraries with version control and change control. Change control relates to the ability to manage the evolution of a component, often by controlling access to the library. Version control manages the record of which version of a component was used in a particular application.

9.3.1.3 Explicit Support for S88.01

9.3.1.3.1 Traditional Approach, STEPS and SystemSpecs

These methods pre-date the batch control standard S88.01 and consequently do not support its models or terminology.

9.3.1.3.2 Rose, StateMate and Synect

In an S88.01 environment, these methods would be appropriate for the development of equipment control but less so for more supervisory activities, such as recipe management and batch scheduling. Although some of the S88.01 terms and models can be used, the methods and tools provide minimal guidance in this respect. These methods would still be preferable to typical industrial development and programming environments due to their support for re-usable components containing sequence logic.

Domain-specific variants of Synect could be offered which would overcome such weaknesses and help overcome an obstacle to successful exploitation. Considering that up to 60% of a typical batch control system development could be associated with phase sequence logic, Synect could be of substantial benefit to batch control applications.

9.3.1.4 Facilitate Clear, Concise and Complete Specifications

9.3.1.4.1 Traditional Approach, STEPS and SystemSpecs

Traditional approaches and STEPS typically rely on textual requirements documents which may be unclear and are very difficult to check for completeness or consistency.

SystemSpecs uses the unfamiliar notations of high level Petri nets which would impair communication in a typical multi-disciplinary team responsible for an industrial control system.

9.3.1.4.2 Rose, StateMate and Synect

These tools facilitate good communication in the early phases of a project through the support for expressive notations and support for consistency checking between diagrams.

The intuitive graphical notations in Synect are suitably expressive whilst being simple and consequently easy to review with a multi-disciplinary team. The ability to execute the model whilst animating the STDs and supporting a process view via 3D modellers or process mimics further enhances the effectiveness of communication. Pre-built event logs can be replayed via the simulator to demonstrate scenarios of use.

Once again the method and tools could be extended to offer improved support for concepts which should be addressed to ensure completeness. For example, older control systems used to manage normal operation only with abnormal behaviour, such as due to plant equipment malfunction, being addressed by plant operational staff as and when it arose. Modern control systems should be expected to adopt a philosophy for management of abnormal behaviour and the explicit support for the expression of these philosophies in Synect could improve consistency between applications in addition to ensuring completeness of the specific application.

Similarly, concepts in the analyst's mind-set such as permissive and fault interlocks, trips and alarms could be explicitly supported.

9.3.1.5 Ability to Verify Correctness

9.3.1.5.1 Traditional Approach and STEPS

Traditional approaches and STEPS provide no support for verifying the correctness of the proposed solution.

9.3.1.5.2 Rose, StateMate and SystemSpecs

These tools facilitate manual verification of correctness by enabling the review of clear but formal diagrammatic specifications and by supporting executable models which can be subjected to tests representing different scenarios.

9.3.1.5.3 Synect

Support for verification is a particular strength of Synect through the combination of a method, mathematical model and CASE tool set. As described in the case studies above, the ability to execute the model whilst animating the diagrammatic specification and additional visualisations via third party products, is perceived as highly valuable in establishing confidence in the design. The ability to perform analytical tests, such as automatic deadlock detection and state searches, is considered to be useful but of less value than the executable capability. This may be due to the lack of confidence in the ability to specify the behavioural property queries or could reflect a natural familiarity with system execution.

Synect could be improved by automatically managing the derivation of sub-system analyses although it is unclear whether this would significantly affect user perceptions.

A highly desirable approach would be to identify rules or guidelines for constructing applications which could be intuitively recognised to avoid particular types of design error. These should be explicitly supported by the Application Editor.

9.3.1.6 Problem Oriented Approaches and Tools

9.3.1.6.1 Traditional Approach and STEPS

As described in chapter four, Control System Technology, traditional approaches are highly implementation oriented. The required behaviour is specified in the language of the target control system rather than the language of the problem domain.

STEPS is an implementation oriented approach, aiming to ensure consistency of coding structures and styles across different projects.

9.3.1.6.2 Rose, StateMate and SystemSpecs

These tools support the development of discrete and batch process applications although they are not specifically targeted at these types of application. Rose supports general purpose object oriented analysis and design, StateMate is predominantly used in high integrity applications such as nuclear and aviation industries, and SystemSpecs has been used in the finance sector.

9.3.1.6.3 Synect

Synect offers support for the lifecycle from requirements capture through to operational support via an appropriate method embodied in an integrated suite of tools. This avoids the discontinuities with existing approaches, delegating implementation to the role of a translator of a target-independent representation to a target-specific representation. This causes effort to be focused on analysis to help ensure that the correct problem is solved and to support agile manufacturing through support for rapid application development and rapid exploitation of flexibility. The method is oriented at small to medium size discrete and batch process applications, using simple notations and offering guidance in how the solution should be structured.

9.3.2 Implementation

9.3.2.1 Consistent Implementation Architecture

9.3.2.1.1 Traditional Approach

There is little consistency of implementation architecture in traditional solutions, with variations due to the personal styles of different programmers and variability from the same programmer on different occasions.

9.3.2.1.2 STEPS

STEPS is specifically aimed at ensuring consistency of implementation architecture so that maintenance personnel can rapidly understand the structure of a PLC program.

9.3.2.1.3 Rose

Rose is oriented at IT systems development rather than control applications and consequently does not support automatic code generation for the required languages,

such as ladder logic. Consistency of implementation architecture would therefore require strict coding rules.

9.3.2.1.4 StateMate and SystemSpecs

Consistency of implementation architecture will only be achieved through the use of the automatic code generators for supported languages, such as C or VHDL, or by enforcing strict adherence to manual coding rules.

9.3.2.1.5 Synect

The use of automatic code generation ensures a unique mapping from specification to code. This avoids style variations between developers and ensures that a support engineer can rapidly understand how to make necessary changes to the implementation. (Changes should not be made directly to code which reflects the structure of the Petri net model - the Application Editor should be used and the code re-generated.)

The consistency of implementation structure also reduces the risk of supplier lock-in because the corresponding learning curve is significantly reduced.

9.3.2.2 Automatic Code Generation

9.3.2.2.1 Traditional Approach, STEPS and Rose

No support for automatic code generation.

9.3.2.2.2 StateMate and SystemSpecs

Code generation to specific languages such as C and VHDL supported. No support for generation of relay ladder logic, distributed architectures or required variants of 3GL as required for different types of target environment, such as micro-controllers or interrupt-driven real time executive.

9.3.2.2.3 Synect

Synect supports automatic code generation to a variety of languages and could be easily enhanced to support many more. Due to the elegance of the Petri net model for the use to which Synect puts it, new code generators can be developed with high confidence that

the generated code will faithfully reflect the Petri net model. Once tested, new control system code can be implemented at a fraction of the elapsed time or cost of manual coding and with significantly higher confidence in its correctness.

9.3.2.3 Comprehensive Automatic Generation of Diagnostics

9.3.2.3.1 Traditional Approach, Rose, StateMate and SystemSpecs

No automatic generation of diagnostics.

9.3.2.3.2 STEPS

Although code generation is a manual activity, a key goal of the STEPS approach is to support fault-diagnosis by incorporating diagnostics with the control logic. This also ensures that the control logic and associated diagnostics do not diverge.

9.3.2.3.3 Synect

Synect's ability to automatically generate diagnostic options built-into the code could be of significant value in diagnosing plant equipment malfunction or operator misuse. Typical Synect diagnostic capabilities include the ability to animate the analyst's graphical specification and the automatic recording of an audit trail which can be replayed via the development tools to determine the sequence of events leading up to a malfunction.

9.3.3 Post-Delivery

9.3.3.1 Easily Supportable and Maintainable Control System

9.3.3.1.1 Traditional Approach

Traditional approaches tend to be difficult to support and difficult to maintain and enhance. The lack of built-in diagnostics and the use of combinational logic can lead to a sensor malfunction triggering anomalous behaviour from which the cause is difficult to identify. A poor software structure leads to difficulty in predicting the side-effects of enhancements or modifications.

9.3.3.1.2 STEPS

Although providing excellent support for diagnosing malfunctions such as sensor faults, STEPS still suffers from a discontinuity between the specification and implementation phases. It can be difficult to determine the design intent from the implemented ladder logic.

9.3.3.1.3 Rose, StateMate and SystemSpecs

These tools are good for requirements capture, design and verification and provided that the mapping from specification to implemented code is clear and consistent, the resultant control system should be enhanceable. However, the lack of diagnostics automatically built into the control system results in limited supportability.

9.3.3.1.4 Synect

The control system is capable of being modified with confidence by site-based personnel. The Synect tools support the lifecycle from specification to implementation ensuring that the implementation corresponds with the specification. Consequently, to understand how to effect the required change, the engineer refers to the specification which is readily understood because of the use of appropriate notations. The impact of changes can be evaluated using the Analyzer and Simulator before the new control code is generated. This is a significant change to current practice where often changes must be made to the live control system.

9.3.3.2 Good Documentation

9.3.3.2.1 Traditional Approach

Traditional approaches tend to yield poor documentation which suffers from the same weaknesses as other text documentation, being potentially incomplete, inconsistent and open to interpretation.

9.3.3.2.2 STEPS

The emphasis on producing a control system which conforms to standard structures results in associated documentation which clarifies the organisation of the software. However, due to the discontinuities between requirements, design and implementation

activities, the production of good implementation documentation does not necessarily imply that the requirements and design documentation will be good.

9.3.3.2.3 Rose, StateMate and SystemSpecs

In general, all method-aware tools produce good documentation because they print the diagrams and cross-reference information. However, the usefulness of the documentation depends on its value to the target audience. The notations used by these tools may therefore be less useful to developers of discrete and batch process applications than for general purpose software development.

9.3.3.2.4 Synect

The ability to automatically generate highly effective documentation ensures that it is produced as the project progresses which also increases the motivation to express clearly the design intent, such as through the use of meaningful state names.

9.3.3.3 Good Enterprise Integration

9.3.3.3.1 Traditional Approach and STEPS

Good enterprise integration depends in part on the effectiveness of communication between the control systems personnel and the corporate systems personnel. Traditional approaches and STEPS use structures and languages which are likely to be unfamiliar to IT development staff.

9.3.3.3.2 Rose, StateMate and SystemSpecs

As a general purpose software development tool, Rose is likely to support notations and views which are familiar to IT staff. The notations and organisation of StateMate and SystemSpecs solutions are typically unfamiliar to IT staff but provide a rigorous definition of the intended behaviour of the control system.

9.3.3.3.3 Synect

Many information system methods use a variant of the state diagram. Information systems analysts should therefore be comfortable with the structure and behaviour of a control system specified using Synect and should correspondingly be able to express their

requirements for derivation of maintenance or performance statistics, for example. Whereas vendors are still purporting to be solving the “shop-floor to top-floor” integration problem through providing transport-level compatibility, Synect could help to ensure that the systems communicate at the information level layer.

9.3.3.4 Modular Automation

9.3.3.4.1 Traditional Approach, STEPS and SystemSpecs

Traditional approaches, STEPS and SystemSpecs do not encourage the identification of re-usable modules, being more concerned with meeting functional requirements.

9.3.3.4.2 Rose and StateMate

Rose and StateMate specifications could be organised as components which directly mapped to plant equipment, although there is minimal guidance in this regard. The absence of automatic relay ladder logic code generation means that code re-use would depend on the rules for transforming the specification to code being clear and consistently applied.

9.3.3.4.3 Synect

As described in section 9.3.1.2, Support and Encourage Re-use, Synect supports the modelling of a hierarchy of plant equipment. A modular machine comprised of mechatronic components would have corresponding Synect objects. When re-using mechatronic components, the corresponding Synect objects would be easily identified, saving development effort and increasing confidence in the control logic. Due to the support for automatic generation of relay ladder logic, the benefits of support for modular automation would still be gained even if the code generator produced different code corresponding to a particular Synect object each time executed, provided that its behaviour remained unchanged.

9.4 Example Walkthrough

In order to demonstrate how the Synect method and tools would benefit an automation project, this section describes typical tool usage throughout a control system's life-cycle. Clearly, the dependencies shown in figure 51 impose some constraints on the order in which the tools can be used. An iterative development process and personal preference introduce variations. Appendix A offers a more detailed walkthrough.

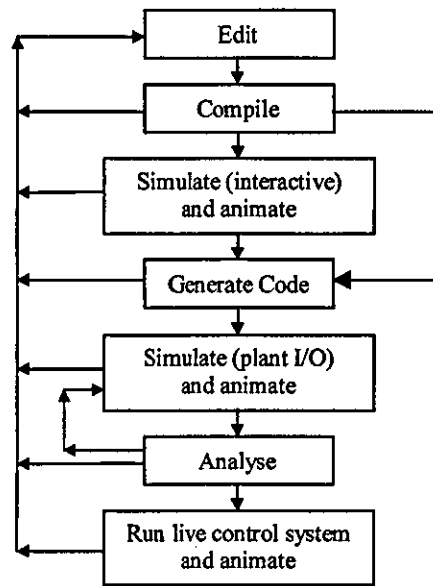


Figure 62 Example Order of Tool Usage

The example order of usage shown in figure 62 is as follows:

- The application is specified using the Application Editor. Printouts of the diagrams may form the basis of a review between the analyst, the process team and the operational users. Revisions may be required before agreement is reached.
- The application is compiled by the Compiler, identifying an incompatible messaging error (the use of pick-lists minimises the risk of this type of error but cannot prevent it altogether). The error is corrected by editing the application again and re-compiling it.
- The analyst interactively drives the application using the Simulator, observing its behaviour via the diagrams being animated by the STD Monitor. As the Simulator executes the model, an event log is recorded. If an unintended system state is observed, the analyst can step the Simulator backwards and forwards through the event log to help identify the cause of the problem. Event logs may be saved to disk and reloaded for replay to illustrate scenarios of use to the process and operations team. Undesirable behaviour may prompt a revision of the logic using the Application Editor and recompilation.
- After interactive testing, the Analyzer is used to verify the absence of deadlocks and to confirm that particular combinations of system state cannot be reached. For a large application, this may be comprised of a set of analyses on subsets of the overall system. Faults will require the application to be re-edited using the Application Editor.

- The plant equipment may be exercised by using the Simulator in conjunction with a DDE interface application provided by the target control system, such as the Echelon LonWorks DDE Server. Once again, the evolving system state can be observed using the STD Monitor. Incorrect assumptions regarding the operation of plant equipment may require the control logic to be modified via the Application Editor.
- Control code is generated for the target platform, with target-specific details specified via a text-based configuration file.
- The live control system is monitored using the STD Monitor to show the current state by animating the analyst's diagrammatic specification.

Overall, the above example shows how the method, model and tool help the analyst to iterate rapidly towards a solution. Effective communication with the process and operations personnel was supported and automatic code generation prevented the introduction of coding errors.

9.5 Industrial Exploitation

9.5.1 Relevance

Synect is relevant to discrete and batch applications (and the start-up and shut-down phases of continuous processes), whether implemented on PLC, DCS, computer-based or control network targets. The opportunity for exploitation will grow more rapidly as the size of application which can be feasibly automated reduces, as encouraged by control system vendors with smaller cost and capability hardware products coupled with software licence break-points to lower entry costs.

Although most control system vendors now claim to offer IEC 61131-3 compliance, many control system applications are still developed entirely in ladder logic. For example, Rockwell Automation's SLC range can only be programmed in ladder logic. A CASE tool, such as Synect, which automatically generates the ladder logic, is therefore highly relevant to developers of sequence logic.

9.5.2 Exploitation Results

Exploitation of the research to date has included:

- Published papers.

- Real-world applications (a selection of which were described as case studies above).
- Incorporation of the tool into research addressing a wider context, such as:
 - Synect will form the "core" of the software tools to be used on a new IMI project (COMPonent based Paradigm for AGile Automation (COMPAG)) targeted at distributed control in the automotive industry. This project is jointly funded by industry and the EPSRC. It involves, Ford, Jaguar, Mazda, Giddings & Lewis, Krause, Rexroth Group and Parker Hydraulics and will result in the creation of full scale transfer line machines for assembly and machining applications.
- Product promotion:
 - Demonstrations and presentations associated with the MSI Research Institute's projects.
 - Exhibition stand at an Echelon LonUser's conference.
 - Article in Computing [234].
 - World Wide Web site.
 - Cold calling (by telephone) with follow-up visits.

The evidence of successful product sales from limited test marketing demonstrates that there is a commercial market for such a product. The next stages of product exploitation are to:

- Revise the product in line with user feedback.
- Identify innovators and early adopters [45] to generate case studies in the target markets.

Chapter 10 Conclusions and Contributions to Knowledge

Addressing the need to lower the life-cycle costs associated with the sequence logic aspects of industrial control systems has required the author to examine many complementary subject areas, as shown in figure 63. Indeed the author would assert that this thesis is driven by the opportunity to support the target user community by:

- Utilising the power of mathematical modelling without burdening the user with its notations.
- Providing an intuitive design approach and expressive but simple set of notations.
- Offering a CASE tool environment.

The novelty and major contribution to knowledge is the amalgamation of the above in a consistent and integrated manner. In particular the author has:

- Examined the business need for improvements and identified the types of application likely to derive the greatest benefit.
- Reviewed current technology and the associated trends in the industrial automation market-place.
- Described the opportunity for model-based approaches and formal methods to assist the development of industrial sequence logic.
- Reviewed general purpose methods and tools which are in greater use in the software engineering community than in the control systems community.
- Analyzed the industrial need, available academic knowledge, current technology and the exploitation-related factors to derive a set of requirements for a method and tool.
- Developed a method and tool satisfying these requirements.

- Assessed and published findings relating to the development of industrial sequence logic.

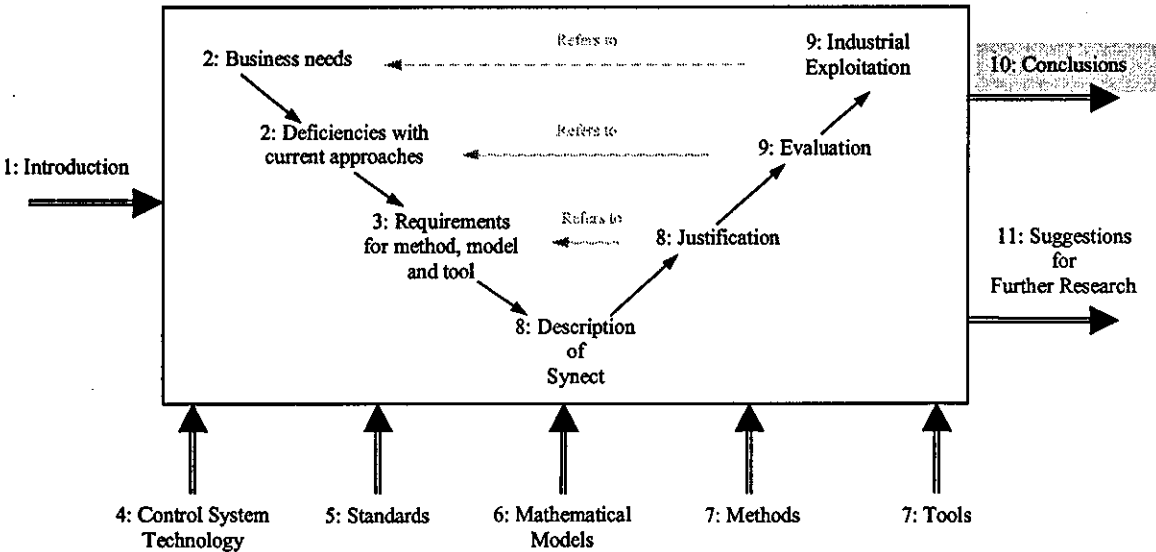


Figure 63 Relationship Between This Chapter And The Thesis Map

The key conclusions from these studies, which have been considered in detail within the main body of the thesis, are presented in the following sections along with the corresponding contributions to knowledge.

10.1 The Business Need

The shortening of product life-cycles is adding pressure to the need to be able to engineer industrial control systems faster, at lower cost and providing maximum flexibility to accommodate process changes to respond to market conditions. In application areas such as discrete manufacturing and batch automation, a significant proportion of the application effort is spent addressing the requirements for sequence logic.

Traditional approaches have been implementation-oriented rather than problem-oriented. Organisational relationships have tended to inhibit the necessary advances by focussing attention on a project by project basis rather than the wider business context. Whilst there is a growing awareness of the relationship of development to life-cycle costs, most projects are still evaluated on the basis of the elapsed time and cost to achieve beneficial operation of the plant.

10.2 The Requirements For A Method And Tool

Whilst there is a genuine business need to improve the approach to sequence logic in industrial applications in many phases of the life-cycle, the barriers to adoption must be addressed. Cost is often a factor, particularly when including training and development costs. However, more significant barriers are the capabilities required of the user, the conformance to widely accepted standards and the seamlessness of the analysis, design and implementation environments. A close relationship between the commissioning, support and maintenance environments and the design environment appears to be a less important criterion, although examination of costs per life-cycle phase would suggest this should be assigned a high priority.

In order to gain industrial acceptance, the perceived learning curve must be short. This implies the need to use familiar notations:

- For sequence logic, the most widespread notations are the state transition diagram (STD) and the sequential function chart (SFC). Although the SFC appears to have the advantage of explicitly modelling concurrency, this is of limited value when an object oriented paradigm is adopted because the concurrency is implicit in the federated behaviour of the objects. The SFC also has the disadvantage of fragmenting a coherent specification because the actions in a step are not specified on the SFC diagram. The author is unaware of an object-oriented method or structured method which uses SFCs in preference to STDs. For these reasons, the STD is the preferred notation.
- Object based (alternatively called component-based) approaches are gaining widespread acceptance, help to encourage the development of re-usable modules of software and are an intuitive modelling approach. The inter-object communication, however, can become overly complex unless a structure is imposed. One of the simplest and most intuitive structures is the strict hierarchy and this also guides the practitioner to consider the coordination required over otherwise disparate objects. Whilst the hierarchy of communication is preferred, the analyst should have the freedom to implement different structures if desired.

One of the main benefits which mathematical modelling can offer the developers of industrial sequence logic is by providing an executable model. Whilst property querying approaches are valuable through their ability to verify the presence of absence of particular behaviour, analysts lack confidence that all necessary queries could be

identified. Although paper-based specifications could be used to verify system behaviour, executable models enable the analyst to gain a deeper understanding and are a more powerful review medium across organisational boundaries. Audit trail functionality is easily supported, offering the ability to replay the history of external stimuli and state evolution.

The tool should support alternative visualisations of system behaviour. At a minimum the tool should be capable of animating the analyst's specification. However, different visualisations offer alternative perspectives on the system's behaviour and may be more appropriate for different audiences and help to prevent saturation with one viewpoint. For example, a 3D solid modeller might be the most appropriate view for a flexible manufacturing cell application whereas a 2D plant mimic diagram would be more appropriate for a process-oriented application. It is therefore important that the tool should be able to drive external software packages to exploit the considerable functionality offered by widely available, low cost software packages such as Wonderware's InTouch SCADA product [76].

The relationship between the notations with which the analyst expresses the design and the mathematical model is subject to several requirements:

- The analyst should be able to use the environment with minimal awareness of the mathematical model.
- For traceability, the mapping from user specification to mathematical model must be clearly visible.
- The mathematical model should be in a form which the analysts and implementers can understand. This appears to contradict the first bullet point above but reflects the healthy scepticism to which industrial innovations are subjected. The analyst needs a working knowledge of the model's mechanisms in order to be able to interpret the traceability information. Developers of implementation architectures benefit from an awareness of the model to understand the degrees of freedom regarding automatic code generation.

The tool should be an enabler for the evaluation of alternative formal methods and/or tools by being able to export the analyst's specification and the mathematical model to a neutral format.

Ultimately, developers of industrial control systems need to deliver a functioning system. Whilst the author would assert that insufficient attention is paid to the early analysis

activities, industrial perception is that generation of control code is of paramount importance. Consequently, there must either be a very simple and effective strategy for translating the design into code or seamless auto code generation is required.

10.3 Industrial Automation Technology

Industrial automation solutions are still dominated by proprietary technology but are converging. Development tools have primarily been a means of implementing the solution on the vendor's platform rather than supporting the full project life-cycle or, ideally, supporting entity life-cycles which span many projects. The tools have also tended to become feature rich but methodologically poor, typically supporting the lowest common denominators of techniques such as language editors and configuration templates.

Paradoxically, the dominance of one vendor, Microsoft, has significantly enhanced the ability to integrate other vendors' software packages by providing data exchange mechanisms such as DDE, OLE, etc..

As hardware capabilities have converged, the control system vendors have attempted to differentiate their offerings through the associated software and are now moving towards a focus on delivery of solutions as the software capabilities also converge.

The fieldbus initiatives offer the opportunity to reduce the burden on centralised controller hardware even further, distributing the required intelligence to the field devices.

10.4 Model-Based and Formal Methods Approaches

Although there is substantial evidence in the published literature of the potential benefits which mathematically oriented techniques may offer, through individual applications of model-based and formal methods approaches, their adoption by industrial users has been poor. Prevailing weaknesses which model-based and formal methods approaches could tackle include:

- A proposed solution cannot be evaluated until it has been implemented.

- Extensive testing is required to verify both the intended solution and the implementation of that solution.
- Significant commissioning effort will still be required before the plant can begin beneficial operation.

The author asserts that this disparity of potential benefit to the accepted norm is a function of:

- The lack of focus on project life-cycle phases unconnected with implementation.
- The user-unfriendliness of traditional mathematical methods to the target user community combined with perceived lack of tool support .
- The absence of a de-facto or international standard regarding the application of a mathematical method to a class of application. This would be less significant if there were widespread knowledge of case histories demonstrating the benefits.

10.5 Methods And Tools

It is difficult to completely decouple the potential benefits of methods and tools. Whilst it is theoretically possible to adopt a method without tool support and, likewise, multi-method CASE tools are available, anecdotal evidence suggests such approaches are rarely adopted. Whereas a method can be valuable in ensuring a consistent approach to projects and help to ensure an appropriate solution is delivered which is maintainable, tools which support the method may at least offer substantial productivity gains and may be a key to the successful application of the method.

Whilst there is evidence of the application of structured methods and object oriented methods to systems engineering projects in general, there appears to have been greater emphasis on either information processing systems or the large, complex and critical real-time applications, such as air-traffic control or applications in the nuclear industry. There is minimal evidence of their application to small to medium sized discrete or batch projects which probably form the majority of industrial automation projects.

There is also a discontinuity between the analysis and design activities supported by tools and the implementation environment. This increases the perceived risk associated with adopting an analysis or design method.

10.6 Contributions to Knowledge

The current state of the art consists of graphical methods supported by CASE tools and mathematical models supported by CASE tools. The major contribution to knowledge of this thesis is the justification for the combination of an appropriate graphical method, mathematical model and CASE tool to help satisfy business needs relating to sequence logic in small to medium sized applications in the discrete and batch process industries. These business drivers include the ability to support agile manufacturing, lower life-cycle costs and collapse project timescales. The evaluation of the Synect method and tool set has demonstrated how significant benefits can be realised.

The contributions to knowledge resulting from this research are:

- Clarification of the need with respect to industrial sequence logic and deficiencies with current approaches.
- Justification of the benefits of combining an expressive notation with mathematical modelling and supporting these with a CASE tool.
- An industrially usable solution to the requirements incorporating:
 - Methodology using graphical notations and mapping to Ordinary Petri net.
 - CASE tools supporting the methodology and mathematical analyses.
 - The development of proof of concept case study applications using the methodology and tool.
 - Evaluation against the criteria established in chapter three, Requirements for Method, Model and Tool, and comparison throughout this thesis with alternatives such as relay ladder logic solutions.
 - Evidence of its industrial relevance and acceptability from its use by independent developers.
- An approach to managing the state explosion problem inherent in Petri net reachability analysis through mechanisms to partition the system using the analyst's object hierarchy.
- Evidence of the benefits of integrating an evolving design with visualisation techniques in order to support prototyping and multi-disciplinary rapid application development.

- Code generation requirements, resulting in the design and implementation of code generators, for a variety of target control system platforms from a Petri net model:
 - With built-in diagnostics
 - Interrupt-driven and scan-based
 - Centralised and distributed
 - In various languages
- Evidence of the benefits of a seamless method and tool support offering a common environment from analysis and design, through implementation and commissioning and into operational usage with subsequent support and maintenance demands.

Chapter 11 Suggestions for Further Research

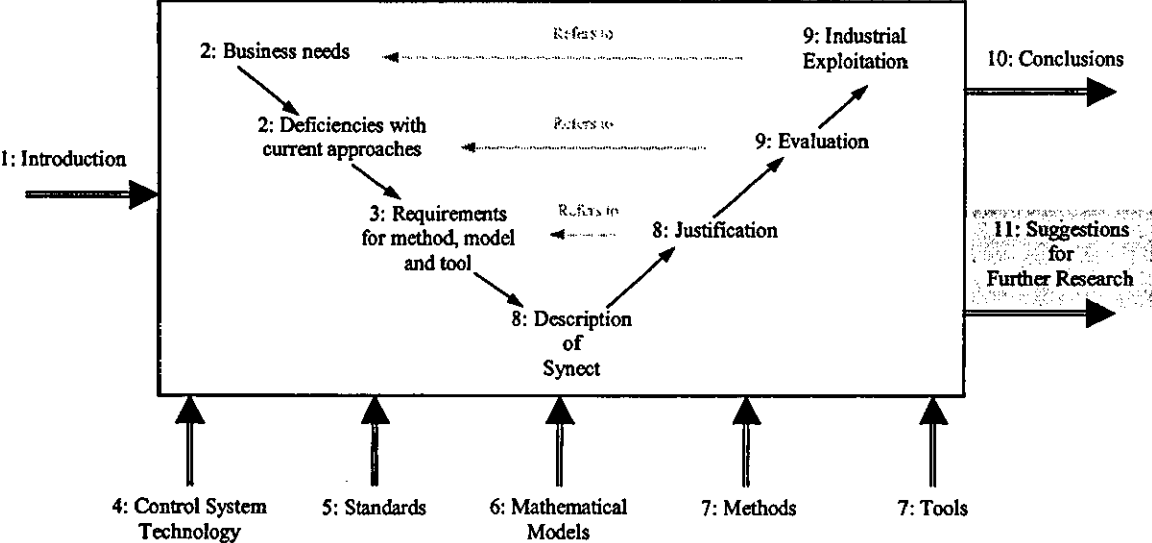


Figure 64 Relationship Between This Chapter And The Thesis Map

This chapter considers how the business needs could be more effectively met by extending the capabilities of the Synect solution and complementing it with alternative approaches. As shown in figure 64, this is expressed in terms of suggestions for further research.

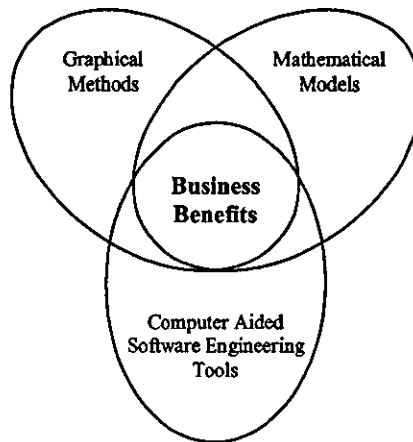


Figure 65 Core Theme of this Thesis

Figure 65 is a copy of figure 1 from chapter one, Introduction, showing the thesis assertion that business benefits can be gained from the combination of graphical methods, mathematical models and software tools.

Considering each of these three components, figure 66 represents the increased effectiveness sought in a future solution. The highest priority is considered to be enhancing the method, followed by CASE tool support and finally, complementary mathematical modelling capability.

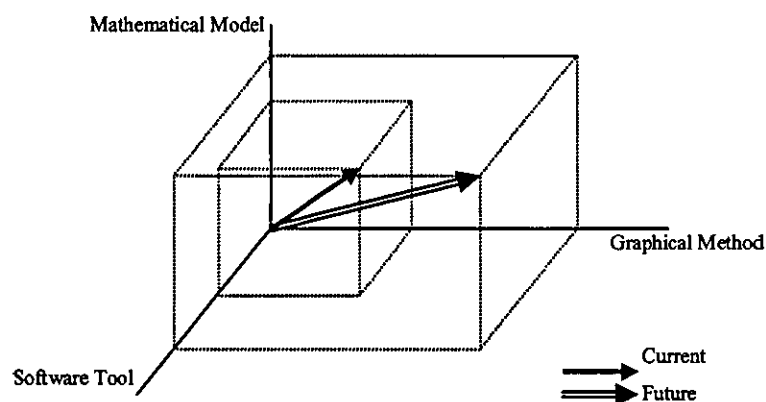


Figure 66 Effectiveness of Method, Model and Tool Support

11.1 Graphical Method

11.1.1 Human Factors

The business benefits are achieved through the support which Synect offers one or more human beings. Although many suggestions may be proposed for enhancing the method, further research should be based on improving the effectiveness of the overall methodology rather than adding “features” to the method. For example, the existing method, explicitly supported by the editor tool, minimises the risk of introducing syntax errors through diagramming conventions and the use of pick-lists. An idealised method would offer guidance on constructing well-behaved applications by conforming to pre-defined patterns of interaction.

Alternative visualisations help to overcome “analysis paralysis” by offering a fresh perspective on the control system behaviour. Although the Simulator tool supports interaction with external software which could show behaviour in alternative forms, (examples discussed in this thesis include 3D graphics, process-industry mimics and scenario of use diagrams), the relative effectiveness of these or other techniques in a variety of project scenarios has not been assessed. After identifying particularly valuable representations, these could be built into the method and explicitly supported by the tools.

The use of a method and tool such as Synect should be part of an overall project methodology. The methodology then prompts the analyst to undertake appropriate design reviews, both end-user and peer, to verify the correctness of the proposed solution. These reviews have been found to be critical to project success [30]. A research activity to determine the most effective methodology using technology such as Synect could be very valuable.

11.1.2 Domain Specificity

Whilst state diagrams and object-based representations are generally applicable to a wide range of domains, from corporate information systems to shop-floor control applications, a more domain-specific method might increase its acceptability and offer higher productivity. For example, an S88.01-specific variant for the batch process industry would explicitly support concepts such as phase logic with separate sequences for normal running, reaction to abnormal process condition, etc..

11.1.3 Object-Oriented and Component-Based Support

The support for class libraries, incorporating a hierarchy of type, could be beneficial but must be used with caution. End-user reviews can be highly effective when the control requirements are modularised. However, if the end-user needs to simultaneously reference many documents to understand the proposed behaviour of a component, the effectiveness is compromised.

As UML (Universal Modelling Language) would appear to have significant vendor support, suggesting it may become a de-facto standard, the use of UML notations may increase Synect's acceptability.

One of the major business benefits is to be realised through the re-use of proven components. The method should therefore support concepts such as object libraries, configuration and version control, etc.. The current message communication syntax ought to be relaxed to ensure that objects are more widely applicable, such as removing the need for an object's response to be "consumed" by another object.

11.1.4 Sequence Notations

The state transition diagram was adopted as a compromise between expressivity and simplicity. Additional support could be added for notations such as statecharts, which offer a richer set of notations useful for more complex applications, and IEC 61131-3 sequential function charts, which are more familiar to control system developers.

Explicit support for exception logic would increase expressivity and reduce diagram clutter, for example supporting a transition from an "any state" state instead of separate transitions from each state. It would also facilitate more meaningful analyses. For example, system reset logic ought to be ignored when searching for deadlocks.

11.2 Mathematical Model

Chapter six, Mathematical Models, made reference to replacing or complementing the Petri net model with alternative or complementary models.

11.2.1 Process Algebras

Although the Petri net is a highly effective executable model, process algebras such as CCS and CSP may offer better support for the querying of behavioural properties. The algorithm incorporated in the Prover tool [128] is particularly worthy of investigation.

11.2.2 Explicit Support for Time

Support for behavioural queries involving time, as offered by RTL [47] for example, would be valuable where the control system is required to conform to hard real-time constraints, such as synchronisation of drives.

11.2.3 Petri Net Variants

Alternative variants of Petri net could also be investigated, for example to determine whether state machine decomposable nets offer a more intuitive mapping from the object and state diagram specification.

11.3 Software Tool

11.3.1 Code Generators

Additional code generators are required to support alternative target platforms, such as DCSs and the emerging IEC 61131-3 PLC programming tools.

11.3.2 Support for Enhanced Method

Many of the suggestions relating to the method and wider project methodology in section 11.1, Graphical Methods above, would require enhancements to software tool support. For example, library management and configuration management would require additional software modules to be developed.

11.3.3 Wizards

A wizard refers to software functionality which asks questions and then uses the answers to partially replace interactive tool usage, thus increasing productivity and minimising style variations. For example, a Synect Wizard could offer pre-defined templates for cyclic STDs and produce a dialog asking the user for the number of states and the name of each state. From this information, the STD could be created, removing the need for the developer to add each state individually and then lay out the diagram as desired.

References

- 1 "ANSI/ISA-S88.01-1995 Batch Control Part 1: Models And Terminology", Instrument Society Of America, 1995, North Carolina 27709
- 2 Fisher, T.G., "Batch Control Systems: Design, Application, and Integration", ISA, 1990, p 2
- 3 Reeve, A., "Tablet Technology", Control And Instrumentation, February 1997, pp 45-47
- 4 Benveniste, A., "Synchronous Languages Provide Safety In Reactive Systems Design", Control Engineering, September 1994, pp 67-69
- 5 Harel, D. et al., "Statecharts: A Visual Formalism For Complex Systems", Science of Computer Programming, Vol. 8, 1987, pp 231-274
- 6 Shlaer, S. and Mellor, S.J., "Object Lifecycles. Modeling The World In States", Yourdon Press, 1992, ISBN 0-13-629940-7
- 7 McGrath, M.F., "Making Supply Chains Agile For Niche Products", APICS – The Educational Society For Resource Management, 1996 Conference Proceedings, pp 167-171
- 8 Reeve, A., "The Future For Process Control", Control And Instrumentation, May 1997, pp 105-106
- 9 Anon, "Success Will Follow From Working Smarter", Eureka, December 1997, p 3
- 10 "Strength where it matters", 1 April 1998, available from WWW site: uniqu.unilever.com/inside.view/introducing/strength.html
- 11 Miller Smith, C., "Satellite Presentation by ICI Chief Executive", March 5, 1996
- 12 Fisher, T.G., "Batch Control Systems: Design, Application, and Integration", ISA, 1990, pp 335-341
- 13 Mitchell R. and Roscoe, S., "PLC Validation During Project Implementation", Measurement And Control, Volume 31, February 1998, pp 10-13
- 14 Sonley, M., "Production Of A Batch Plant Control System", Analysis and Design of Event-Driven Operations in Process Systems, Imperial College, Centre for Process Systems Engineering, London, 10-11 April 1995
- 15 "Automation Strategies", Automation Research Corporation, April 1996
- 16 Plansky, P., "Time To Market- Two Weeks To Prototype, One Year To Production", VLSI Technology News, June 1990 (Citing McKinsey & Co. Report)
- 17 Hammer, M. and Champy, J., "Reengineering The Corporation: A Manifesto For Business Revolution", Brealey Publishing, 1993
- 18 Kappelhoff, R., "S88 Impact On Health And Beauty Care Operations", World Batch Forum, 1996
- 19 Strothman, J., "Where ERP Meets Process Control, S88 Can Help", InTech, July 1997, pp 49-50
- 20 Schumann, A., "SAP-R/3 In Process Industries: Expectations, Experiences And Outlooks", ISA Transactions, Vol. 36, No. 3, 1997, pp 161-166
- 21 Coleman, D. et al., "Object Oriented Development: The Fusion Method", Prentice-Hall, 1994, ISBN 0-13-338823-9

- 22 DeMarco, T., "Controlling Software Projects, Management, Measurement and Estimation", Yourdon Press, ISBN 0-13-171711-1
- 23 "Guidelines For The Development Of Programmable Logic Controller Application Software For Safety Related Applications", IEE, May 1996
- 24 Anon, "Out of Control", Health and Safety Executive, England
- 25 Brayford, N., "The Problem Scenario Applying Structured Methods To Projects", Control And Instrumentation, May 1997, pp 43-44
- 26 "Case And The Meta-Tools", ButlerBloor Ltd, p 167
- 27 Ericsson, G., "Functional Specification Of Industrial Control Systems", Proceedings Of The 1994 IEEE Conference On Control Applications Part 2 Of 3, 1994, pp 1347-1352
- 28 Rockwell Software, Kiln Farm, Milton Keynes, Bucks
- 29 Parkin, G.I. and Austin, S., "Formal Methods: A Survey", Division of Information Technology and Computing, National Physical Laboratory, Teddington, Middlesex, 31 March 1993
- 30 Hopkinson, P. et al., "Implementing S88 Batch Control Systems In The Pharmaceutical Industry", Measurement And Control, Vol. 31, February 1998, pp 20-24
- 31 Cooling, J.E., "Software Design For Real-Time Systems", International Thomson Computer Press, 1995
- 32 Lewis, R., "Programming Industrial Control Systems Using IEC 1131-3", ISBN 0-85296-827-2
- 33 Burns, G.L., "The Use Of Hierarchical Petri Nets For The Automatic Generation Of Ladder Logic Programs", International Programmable Controllers Conference Proceedings, 1994, pp 169-180
- 34 Harrison, R. et al., "Improving Manufacturing Automation By The Integration Of Machine Design And Control", 26th International Symposium on Industrial Robots, Singapore, Oct. 1995, pp 51-56, ISBN 1-86058-000-9
- 35 Reich, J.E., "Symbolic Simulation-Based Techniques For Debugging Discrete Control Programs", Department of Electrical and Computer Engineering, Carnegie Mellon University, May 1996
- 36 Anon, "Scaling Up: A Research Agenda For Software Engineering", Communications of the ACM, March 1990, Vol. 33, No. 3, pp 281-293
- 37 Womack, J.P., Jones, D.T. and Roos, D., "The Machine That Changed The World: Based On The Massachusetts Institute of Technology 5-Million Dollar 5-year Study On The Future Of The Automobile", Maxwell Macmillan, 1990
- 38 Davenport, T.H., "Process Innovation: Reengineering Work Through Information Technology", Harvard Business School Press, c1993
- 39 Brooks, F., "No Silver Bullet: Essence And Accidents Of Software Engineering", IEEE Computer, Vol. 20, No. 4, April 1987
- 40 Hatton, L., "Software Failures. Follies and Fallacies", IEE Review, March 1997, pp 49-52
- 41 Ven Der Biezen, H., "What's The Real Cost Of Ownership?",
- 42 Anon, "Power Market Thirsts For Automation Technology", Control Systems, May 1997, p 6
- 43 Anon, "Component Based Development", 1998 Report Series, Vol. 1, Butler Consulting Group Ltd, Hessle, East Yorkshire, England

- 44 Booch, G., "Object Oriented Design With Applications", Benjamin/Cummings, 1991, ISBN 0-8053-0091-0
- 45 McDonald, M.H.B., "Marketing Plans", Butterworth Heinemann, 1989, ISBN 0-7506-0107-8
- 46 Mallaband, S., "Specification Of Real Time Control Systems By Means Of Sequential Function Charts", IEE Conference Publication, 1991, No. 344, pp 57-62
- 47 Bucci, G. et al., "Tools For Specifying Real-Time Systems", Real-Time Systems, Vol. 8, 1995, pp 117-172
- 48 Dijkstra, E., "Programming Considered As A Human Activity", Classics In Software Engineering, New York NY, Yourdon Press
- 49 Boriani, D.V., "Object-Oriented Development Of Control Software", ISA Transactions, Vol. 36, No. 2, 1997, pp 131-138
- 50 Chang, C.K. et al., "Integral: Petri Net Approach To Distributed Software Development", Information And Software Technology, Vol. 31, No. 10, December 1989, pp 535-545
- 51 McGinnes, S., "CASE Support For Collaborative Modelling: Re-Engineering Conceptual Modelling Techniques To Exploit The Potential Of CASE Tools", Software Engineering Journal, July 1994, pp 183-189
- 52 Gaskell, C. and Phillips, R., "Executable Specifications And CASE", Software Engineering Journal, July 1994, pp 174-182
- 53 Anon, "Rockwell Automation Surges Past 50 Percent PLC Market Share In North America", Automation Research Corporation, 17 February 1998
- 54 Davidson, C. and McWhinnie, J., "Stepping Off The Ladder", IEE Review, September 1997, pp 210-212
- 55 Morihara, R.H., "State-Based Ladder Logic Programming",
- 56 Falcione, A. and Krogh, B.H., "Design Recovery For Relay Ladder Logic", 1st IEEE Conference on Control Applications, Dayton, Ohio, Vol. 1/2, Chapter 198, 1992, pp 648-653
- 57 Borchers, G., "Software Engineering Techniques For Ladder Logic", EDS Plant Automation Division Controls Engineering Group
- 58 Schelberg, C., "What?! Ladder Logic Dead!", Flavors Technology, Inc.
- 59 Stewart, I., "The Problems Of Mathematics", Oxford University Press, 1987, ISBN 0-19-219201-9
- 60 Anon, "More Than A Relay, Less Than A PLC And No Programming" Control And Instrumentation, November 1996, p 60
- 61 Pollard, J.R., "Open Architecture For Control?", Industrial Computing, June 1996, pp 16-18
- 62 Siemens, Bracknell, Berkshire, England
- 63 The Foxboro Company, 33 Commercial Street, Foxboro, MA 02035, USA
- 64 Peach, M., "Team Players Can Seem To Be United", Control And Instrumentation, May 1998, p5&9
- 65 Anon, "Softlogic Threatens The DCS, PLCs Fare Better", Control Systems, August 1997
- 66 Anon, "PC Set To Oust PLCs For Control By Year 2000", Control And Instrumentation Europe, April 1996, p 11

- 67 Gledhill, B., "Germany Biggest DCS Consumer", Control And Instrumentation, August 1998, p 15
- 68 Tinham, B., "Control Systems", Control And Instrumentation, May 1997, pp 59-60
- 69 Tinham, B., "Control Systems", Control And Instrumentation, December 1997, p 8
- 70 Anon, "Industry Ploughs £3 Billion Into IT", Eureka Transfers Technology, January 1997, p 8
- 71 Anon, "'Softlogic' Providers Foresee A Bonanza", Control Systems, February 1997, p 11
- 72 "Software Logic Into Simplicity", Control And Instrumentation, April 1997
- 73 Anon, "Advanced And Soft PLC To Bring New Choices", Control And Instrumentation, June 1997, p 7
- 74 Ballard, A., "Has The PLC Had Its Day?", Control Systems, June 1997, p 25
- 75 Sequencia Corporation, 2429 West Desert Cove Avenue, Phoenix, AZ 85029, USA
- 76 Wonderware, PO Box 30, Twickenham, Middlesex, England
- 77 Bond, A., "An Enabling Technology For Fundamental Change", Fieldbus Supplement, November 1996, pp S1-S3
- 78 Schickhuber, G. and McCarthy, O., "Distributed Fieldbus And Control Network Systems", Computing And Control Engineering Journal, February 1997, pp 21-32
- 79 Reeve, A., "Fieldbus '97 Progress Or Prognostication?", Control And Instrumentation, May 1997, pp 101-102
- 80 Anon, "PC Will Beat PLC 'Within a Decade'", Manufacturing Computer Solutions, March 1996, p 4
- 81 Saward, P., "Taking Fieldbus Into Hazardous Areas", Control And Instrumentation, February 1997, page 48-49
- 82 Lewis M., "Go With The Right Bus", Fieldbus Supplement, November 1996, pp S5-S6
- 83 Jones, J., "Why Wait For Fieldbus?", Control And Instrumentation, April 1992
- 84 "Programmable Controllers – Part 3: Programming Languages", IEC 61131-3 (1993-03), IEC, 3 rue de Varembe, PO Box 131, CH-1211 Geneva 20, Switzerland
- 85 Lewis, R. and Tinham, B., "Control Software Standard Emerging", Control And Instrumentation, September 1992, pp 51-53
- 86 Stroustrup, B., "What Is Object-Oriented Programming?", IEEE Software, Vol. 5, No. 3, May 1998, p 10
- 87 Neumann, P., "Function Block Technology Standard", Control And Instrumentation Europe, April 1996, pp 60-62
- 88 Anon, "Keeping Costs Under Control", Control Systems, May 1997, pp 33-34
- 89 Anon, "Software Complies With Latest Euro Standards", Eureka, May 1997, p 15
- 90 Anon, "PLC Stretches To Meet Tough Applications", Eureka, February 1997, p 51
- 91 ConCept, AEG Schneider Automation, North Andover, MA 01845, USA
- 92 Anon, "Better Standards Of Control Needed", Manufacturing Computer Solutions, October 1997, p 13
- 93 IsaGraf, CJ International, 86 rue de la Liberte, F-38180 Seyssins, France
- 94 CADEPA, Famic Ltd, Foxholes Business Park, Hertford, Hertfordshire, England

- 95 PLCOpen Organisation, Postbus 2015, 5300 CA Zaltbommel, The Netherlands
- 96 Anon, "Flexibility From Your PLC", Control And Instrumentation, April 1997, p 7
- 97 Juer, J. and Hughes, I.P., "IEC 65A Control Languages - A Practical View", Eurotherm International PLC, England
- 98 Lewis, R., "Design Of Distributed Control Systems In The Next Millennium", Computing And Control Engineering Journal, August 1997, pp 148-152
- 99 Carrick, K., "All At 'C'", Control Systems, April 1998, pp 27-28
- 100 Peshek, C.J. and Mellish, "Recent Developments And Future Trends in PLC Programming Languages And Programming Tools For Real-Time Control", IEEE Cement Industry Technical Conference, May 1993, Toronto Canada, pp 219-230
- 101 Anon, "STEPS' Specification", Version 3.1, Ford Motor Company Limited
- 102 Krause, Germany
- 103 Hopkinson, P. and Hancock, J., "A Case History Of The Implementation Of An S88-Aware Batch Control System", World Batch Forum, 1998
- 104 Haxthausen, N. and Hopkinson, P., "The Application Of The S88 Batch Control Standard In The Pharmaceutical Industry", Computer Systems For The New Millennium Conference, International Society Of Pharmaceutical Engineers, 4th - 5th March 1998, Amsterdam
- 105 Crowl, T.E. and Minnich, L.C., "Configuration Techniques For A Validated Plant", ISA Transactions, Vol. 36, No. 3, 1997, pp 209-218
- 106 Vaitsis, A., "CORBA – Or ActiveX", Control And Instrumentation, May 1997, pp 33-36
- 107 Taylor, D., "Object-Linking And Embedding - What Is OPC About?", Control And Instrumentation Europe, April 1996, pp 34-35
- 108 Tinham, B., "Why OPC For Systems?", Control And Instrumentation, April 1997, pp 24-26
- 109 OLE for process control, final release, version 1.0, OPC Task Force, 1996, available from WWW site: www.industry.net/OPC
- 110 Vaitsis, A., "CORBA, OLE And More", Control And Instrumentation, June 1997, pp 26-27
- 111 Anon, "CORBA 'Righteous, And Late'", Manufacturing Computer Solutions, May 1997, p 6
- 112 Anon, "JavaBeans", <http://www.webadvisor.com/javabeans.html>.
- 113 Lynch, G., "World Wide Web, What?", Control And Instrumentation, April 1997, p 29
- 114 Carson, M. and Tomasello, M., "Let's Get Surfing - Seriously!", Control Systems, March 1997, pp 29-30
- 115 Barroca, L.M. and McDermid, J.A., "Formal Methods: Use And Relevance For The Development Of Safety Critical Systems", University Of York, England
- 116 Brackett, J. W., "Formal Specification Languages: A Marketplace Failure", IEEE International Conference on Computer Languages, October 1988, Miami Beach, Florida, USA
- 117 Hoare, C.A.R., "Communication Sequential Processes", Prentice Hall, 1985, ISBN 0-13-153271-5
- 118 Milner, R., "Communication And Concurrency", Prentice Hall, ISBN 0-13-115007-3
- 119 Spivey, M., "The Z Notation: A Reference Manual", Prentice Hall International, Englewood Cliffs, NJ, 1990

- 120 Jones, C.B., "Systematic Software Development Using VDM", 2nd Edition, Prentice Hall International, Englewood Cliffs, NJ, 1990
- 121 Anon, "Estelle: A formal Description Technique Based on an Extended State Transition Model", ISO 9074, International Standards Organisation, 1988.
- 122 Diaz, M. and Vissers, C., "SEDOS: Designing Open Distributed Systems", IEEE Software, November 1989, pp 24-33
- 123 Diaz, M. and Vissers, C., "SEDOS: Designing Open Distributed Systems", IEEE Software, November 1989, pp 24-33
- 124 Durr, E., "VDM++ Language Reference Manual", AFRODITE Document afro/cg/ed/lrm/v9.1, May 1994
- 125 Duke, R. et al., "The Object-Z specification Language", Version 1, Technical Report, Software Verification Research Centre, University of Queensland, Queensland, Australia, May 1991
- 126 Lano, K. and Haughton, H., "The Z++ Manual Technical Report", Lloyds Register Of Shipping, 29 Wellesley Road Croydon, England, 1992
- 127 Probst, S.T., "Chemical Process Safety And Operability Analysis Using Symbolic Model Checking", PhD Thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, May 1996
- 128 Prover, National Physical Laboratory, Teddington, Middlesex
- 129 Saflund, M., "Modelling And Formally Verifying Systems And Software In Industrial Applications", National Physical Laboratory, Teddington, Middlesex
- 130 Stalmarck, G. and Saflund, M., "Modeling And Verifying Systems And Software In Propositional Logic", IFAC, SafeComp90, London, 1990, pp 31-36
- 131 Borah, A. and Agren, H., "Formal Verification Of Programmable Logic Controllers", Master's Theses in Computing Science 82, ISSN 1100-1836
- 132 Tinham, B., "Check Systems Automatically", Control And Instrumentation, July 1995, pp 41-42
- 133 Ramage, P.J.G. and Wonham, W.M., "The Control Of Discrete Event Systems", Proceedings of the IEEE, 77(1), 1989, pp 81-98
- 134 Rotstein, G.E. et al., "Synthesis Of Procedural Controllers And The Automatic Generation Of Sequential Control Code", Analysis and Design of Event-Driven Operations in Process Systems, Imperial College, Centre for Process Systems Engineering, London, 10-11 April 1995
- 135 Peterson, J.L., "Petri Net Theory And The Modeling Of Systems", Prentice-Hall, 1981, ISBN 0-13-661983-5
- 136 Bonney, M.C. et al., "UNISON - A Tool For Enterprise Integration", Department of Manufacturing Engineering & Operations Management, University of Nottingham, England
- 137 Silva, M. and Valette, R., "Petri Nets And Flexible Manufacturing", Lecture Notes in Computer Science 424, Springer Verlag, 1990, pp 374-417
- 138 Jafari, M.A. and Boucher, T.O., "The Design Of A Petri Net Controller From An IDEFO Specification", Factory Automation and Information Management, pp 804-815
- 139 Valette, R. et al., "A Petri Net Based Programmable Logic Controller", pp 103-116
- 140 D'Souza, K.A. and Khator, S.K., "A Survey Of Petri Net Applications In Modeling Controls For Automated Manufacturing Systems", Computers In Industry, Vol. 24, 1994, pp 5-16

- 141 Murata, T. et al., "A Petri Net-Based Controller For Flexible And Maintainable Sequence Control And Its Applications In Factory Automation", IEEE Transactions on Industrial Electronics, February 1986, Vol.IE-33, No.1, pp 1-8
- 142 Komoda, N., "An Autonomous, Decentralized Control System For Factory Automation", Computer, December 1984, pp 73-83
- 143 Menga, G. and Morisio, M., "Prototyping Discrete Part Manufacturing Systems", Information And Software Technology, Vol. 31, No. 8, October 1989, pp 429-437
- 144 D'Souza, K.A. and Khator, S.K., "A Petri Net Approach For Modelling Controls Of A Computer Integrated Assembly Cell", International Journal of Computer Integrated Manufacturing, 1993, Vol. 6, No. 5, pp 302-310
- 145 Narahari, Y. and Viswanadham, N., "A Petri Net Approach To The Modelling And Analysis Of Flexible Manufacturing Systems", Annals Of Operations Research 3, 1985, pp 449-472
- 146 Kamath, M. and Viswanadham, N., "Application Of Petri Net Based Models In The Modelling And Analysis Of Flexible Manufacturing Systems", 1986, pp 262-267
- 147 Valette, R., "Petri Nets For Control And Monitoring Specification, Verification, Implementation", Analysis and Design of Event-Driven Operations in Process Systems, Imperial College, Centre for Process Systems Engineering, London, 10-11 April 1995
- 148 Andreu, D. et al., "Interaction Of Discrete And Continuous Parts Of A Batch Process Control System", LAAS-CNRS
- 149 Zhou, M.C. and Twiss, E., "A Comparison Of Relay Ladder Logic Programming And Petri Net Approach For Sequential Industrial Control Systems", 4th IEEE Conference on Control Applications, 1995, Albany, pp 748-753
- 150 Cortadella, J. et al., "Synthesising Petri Nets From State-Based Models", Computer Aided Design International Conference, 13 November 1995, San Jose, pp 164-171
- 151 Murata, T., "Modeling And Analysis of Concurrent Systems", Handbook Of Software Engineering, 1983, pp 39-63
- 152 Saha, B. and Bandyopadhyay, S., "Representation And Analysis Of Petri Nets Via The Matrix State Equation Approach", International Journal of Electronics, Vol. 65, No.1, July 1988, pp 1-7
- 153 Jensen, K., "Coloured Petri Nets. A Way to Describe and Analyse Real World Systems Without Drowning in Unnecessary Details", Proceedings of the 5th International Conference on Systems Engineering, Dayton, 1987, New York: IEEE, pp 395-401
- 154 Jensen, K., "Coloured Petri Nets: A High Level Language For System Design And Analysis", Advances In Petri Nets, 1990, pp 342-416
- 155 Design/CPN, Meta Software Corporation, MA, USA
- 156 Harhalakis, G. et al., "Formal Representation, Verification and Implementation Of Rule-Based Information Systems For Integrated Manufacturing (INSIM)", Technical Report TR 91-19, Systems Research Center, University of Maryland, College Park, 1991
- 157 Barozzi, S. et al., "Petri Net Based Real Time Simulation Of Industrial Plants", IEEE Conference on Systems, Man And Cybernetics, October 1994, San Antonio, Vol. 2, pp 1983-1988
- 158 Caloini, A. et al., "A Technique For Designing Robotic Control Systems Based On Petri Nets", IEEE Transactions on Control Systems Technology, Vol. 6, No. 1, January 1998, pp 72-87

- 159 Hack, M., "Analysis Of Production Schemata By Petri Nets", Master's Thesis, Department Of Electrical Engineering, MIT, 1972
- 160 Ramamoorthy, C.V. and Ho, G.S., "Performance Evaluation Of Asynchronous Concurrent Systems Using Petri Nets", IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, September 1980, pp 440-449
- 161 Lin, J.T. and Lee, C.C., "A Modular Approach For The Modelling Of A Class Of Zone-Control Conveyor System Using Timed Petri Nets", Int. J. Computer Integrated Manufacturing, 1992, Vol. 5, Nos. 4 & 5, pp 277-289
- 162 Molloy M.K., "Performance Analysis Using Stochastic Petri Nets", IEEE Transactions on Computers, Vol. 31, No. 9, 1982, pp 913-917
- 163 Marsan, M.A. et al., "A Class Of Generalised Stochastic Petri Nets For The Performance Evaluation Of MultiProcessor Systems", ACM Transactions On Computer Systems, Vol. 2, No. 2, May 1984, pp 93-122
- 164 Greene, J., "Petri Net Design Methodology For Sequential Control", Measurement and Control, Vol. 22, December/January 1989/90, pp 288-291
- 165 Teng, S.H. and Black, J.T., "Manufacturing System Control With Petri Nets In Cellular Manufacturing Systems", Computers ind. Engineering, Vol. 19, Nos. 1-4, 1990, pp 150-154
- 166 Garnousset, H.E. et al., "Efficient Tools For Analysis And Implementation Of Manufacturing Systems Modelled By Petri Net With Objects: A Production Rules Compilation-Based Approach", IECON Proceedings (Industrial Electronics Conference), 1989, Vol.3, pp.543-549
- 167 Murata, T. et al., "A Petri Net Based Factory Automation Controller For Flexible And Maintainable Control Specifications", pp 362-366
- 168 Di Stefano, A. and Miabella, O., "A Fast Sequence Control Device Based On Enhanced Petri Nets", Microprocessors and Microsystems, Vol. 15, No. 4, May 1991, pp 179-186
- 169 Bruno, G. and Marchetto, G., "Process Translatable Petri Nets For The Rapid Prototyping Of Process Control Systems", IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, February 1986, pp 346-357
- 170 Notomi, M. and Murata, T., "Hierarchically Organised Petri Net State Space For Reachability And Deadlock Analysis", 6th International Conference on Parallel Processing, March 1992, Beverly Hills, pp 616-623
- 171 Yau, S.S. and Caglayan, M.U., "Distributed Software System Design Representation Using Modified Petri Nets", IEEE Transactions On Software Engineering, Vol. 9, No. 6, November 1983, pp 733-745
- 172 Alla, H., "Modelling And Simulation Of Event Driven Systems By Petri Net", Analysis and Design of Event-Driven Operations in Process Systems, Imperial College, Centre for Process Systems Engineering, London, 10-11 April 1995
- 173 Chang, C.K. et al., "INTEGRAL – An Integrated Framework For Distributed Software Validation And Verification", Proceedings of the Workshop on Future Trends Of Distributed Computing Systems In the 1990s, 1988, pp 301-310
- 174 Stotts, P.D. and Cai, Z.N., Hierarchical Graph Models Of Concurrent CIM Systems", IEEE Workshop on Lang for Autom Symbiotic and Intell Rob., 1988, No.1988, pp.100-105
- 175 Ramaswamy, S. et al., "A High Level Specification Mechanism For The Analysis And Design Of Manufacturing Systems", IEEE Conference on Systems, Man and Cybernetics, 1995, Vancouver, Vol. 1, pp 524-529

- 176 Lee-Kwang, H. et al., "Generalized Petri Net Reduction Method", IEEE Transactions on Systems Man and Cybernetics, Vol. SMC-17, No. 2, March/April 1987, pp 297-303
- 177 Janicki, R. and Koutny, M., "On Equivalent Execution Semantics Of Concurrent Systems", Advances In Petri Nets, pp 89-103
- 178 Janicki, R. and Koutny, M., "Optimal Simulations, Nets And Reachability Graphs", Advances In Petri Nets, 1991, pp 205-226
- 179 Ozsu, M.T., "Modeling And Analysis Of Distributed Database Concurrency Control Algorithms Using An Extended Petri Net Formalism", IEEE Transactions On Software Engineering, Vol. SE-11, No. 10, October 1985, pp 1225-1239
- 180 He, X. and Lee, J.A.N., "A Methodology For Constructing Predicate Transition Net Specifications", Software – Practice And Experience, Vol. 21, No. 8, August 1991, pp 845-875
- 181 Anon, "High Level Petri Nets – Concepts, Definitions and Graphical Notation", Committee Draft ISO/IEC 15909, 2 October 1997, Version 3.4
- 182 "Sequence Logic Right First Time", Case Study, Eutech Engineering Solutions, Belasis Hall Technology Park, Billingham, Cleveland
- 183 Yourdon, E., "Modern Structured Analysis", 1989, Prentice Hall, Englewood Cliffs, NJ, USA
- 184 Ward, P.T. and Mellor, S.J., "Structured Development For Real-Time Systems", Prentice-Hall, Englewood Cliffs, NJ, USA, 1985
- 185 Hatley, D.J. and Pirbhai, I.A., "Strategies For Real Time System Specification", Dorset House Publishing, New York, 1987
- 186 Arnold et al., "Evaluation Of Five Object-Oriented Development Methods", Journal Of Object Oriented Programming: Focus on Analysis and Design, SIGS Publication Inc., New York, 1991, pp 101-121
- 187 Cribbs et al., "An Evaluation Of Object Oriented Analysis And Design Methodologies", SIGS Publications Inc., New York, 1992
- 188 de Champeaux, D. and Faure, P., "A Comparative Study Of Object Oriented Analysis Methods", Journal Of Object Oriented Programming 5(1):21-33 March/April 1992
- 189 Sutcliffe, A.G., "Object Oriented Systems Development: Survey Of Structured Methods", Information And Software Technology, Vol. 33, No. 6, July/August 1991, pp 433-442
- 190 Fowler, M., "A Comparison Of Object-Oriented Analysis And Design Methods", Object World, July 1992
- 191 Mellor, S.J., "A Comparison Of The Booch Method And Shlaer-Mellor OOA/RD", Project Technology, 2 May 1993
- 192 Shlaer, S., "A Comparison Of OOA and OMT", Project Technology, 7 August 1992
- 193 Rumbaugh, J. et al., "Unified Modeling Language Reference Manual", ISBN 0-201-30998-X, Addison Wesley, 1997. Available from WWW site: www.awl.com/cp/uml/uml.html.
- 194 Anon, "Unified Modeling Language For Real-Time Systems Design", Ver. 2.0, 15 September 1998, available from WWW site: <http://www.rational.com>
- 195 Coleman, D. and Hayes, F., "Getting The Best From Objects: The Experience Of HP", 18 December 1990
- 196 Cook, S. and Daniels J., "Object-Oriented Methods And The Great Myth", Objects in Europe, Autumn 1994, pp13-18

- 197 Yourdon, E., "Application Development Strategies", available from WWW site:
<http://www.rational.com/support/techpapers/ads/rational.pdf>
- 198 Anon, "Promod-PLUS Analysis", <http://www.bergson.nl/tools/prman.html>
- 199 Anon, "OO Tool Developers to Integrate Products",
<http://www.tdtech.com/press/cayenne.html>
- 200 Anon, "Rational Rose 98", <http://www.rational.com/products/rose/>
- 201 Kennedy Carter, Thornton Road, London, SW19 4NB
- 202 ProtoSoft, 17629 Ei Camino Real 202, Houston, TX 77058
- 203 Visio Corporation, 520 Pike Street, Suite 1800, Seattle, WA 98101, USA
- 204 SPADE, Praxis Critical Systems, 20 Manvers Street, Bath, England
- 205 MALPAS, TA Consultancy Services Ltd, 'The Barbican' East Street, Farnham, Surrey, England
- 206 Harel, D. et al., "STATEMATE: A Working Environment For The Development Of Complex Reactive Systems", IEEE Transactions on Software Engineering, Vol. 16, No. 4, April 1990, pp 403-414
- 207 Stateflow, Cambridge Control, Cambridge, England
- 208 BetterState Pro, Integrated Systems Inc., 201 Moffett Park Drive, Sunnyvale, CA 94089
- 209 Harel, D. et al., "On The Formal Semantics Of Statecharts", Proceedings: 2nd IEEE Symposium on Logic in Computer Science, Ithaca, NY, 1987, pp 54-64
- 210 MacLeod, A., "Identifying Problems At Concept Stage", New Electronics, September 1993
- 211 Gurewich, N. and Gurewich, O., "Teach Yourself Visual Basic 5 in 21 Days", Paperback 4th edition (April 1997), Sams, ISBN: 0672309785
- 212 "Visual Engineer", Control And Instrumentation, April 1997
- 213 Anon, "HP VEE", Amplicon Liveline, 1997, pp 58-63
- 214 Anon, "LabVIEW Links To DT's API And Acquisition Boards", Control And Instrumentation, April 1997, p 30
- 215 MatLab, Cambridge Control, Cambridge, England
- 216 Workspace, Robot Simulations Ltd, 21 High Bridge, Newcastle upon Tyne, England
- 217 Witness, AT&T Istel, Redditch, England
- 218 AutoMod, AutoSimulations, 655 Medical Drive, Bountiful, Utah 84010, USA
- 219 Carey, J.M. and Currey, J.D., "The Prototyping Conundrum", Datamation, 1 June 1989, pp 29-33
- 220 Cooling, J.E. and Hughes, T.S., "The Emergence Of Rapid Prototyping As A Real-Time Software Development Tool", 2nd International Conference on Software Engineering, 1989, pp 60-64
- 221 Ince, D.C. and Hekmatpour, S., "Software Prototyping – Progress And Prospects", Information And Software Technology, Vol. 29, No. 1, January/February 1987, pp 8-13
- 222 Waterbury, R.C., "APT Cuts Programming Time And Costs", INTECH, May 1991, pp 30-32
- 223 Graham, D.R., "Computer Aided Software Testing: The CAST Report", Unicom Seminars Ltd, Brunel Science Park, Cleveland Road, Uxbridge, Middlesex, England, 1991

- 224 DirectLink, PANTEK, Stockport, Cheshire
- 225 SystemSpecs, AutoLogic Systems Ltd, Farnham, Surrey, England
- 226 Bates, I.D. et al., "A Case Study In The Automatic Programming Of A PLC Based Control System Using Statemate Statecharts", Newcastle EPSRC Engineering Design Centre, England
- 227 "Synect User Guides", Hopkinson Computing Limited, 29 Deepdale, Guisborough, Cleveland
- 228 Pezze, M. et al., "Graph Models For Reachability Analysis Of Concurrent Programs", ACM Transactions on Software Engineering And Methodology, Vol. 4, No. 2, April 1995, pp 171-213
- 229 Echelon, 4015 Miranda Avenue, Palo Alto, CA 94304, USA
- 230 Kordon, F. and El Kaim, W., "H-COSTAM: A Hierarchical Communicating State-Machine Model For Generic Prototyping", IEEE, 1995, pp 131-138
- 231 Silva, M. and Velilla, S., "Programmable Logic Controllers And Petri Nets: A Comparative Study", IFAC Software For Computer Control, Madrid, Spain, 1982, pp 83-88
- 232 Harrison, R. et al., "Interactive Visualisation Of Sequence Logic And Physical Machine Components Within An Integrated Design And Control Environment", 4th IFAC Workshop on International Manufacturing Systems, July 1997, Seoul, Korea
- 233 Harrison, R. and Charles, G.P., "Applying Lonworks To The Distributed Control Of Manufacturing Machines", Lonusers International Conference 24-25 October 1995, Frankfurt, Germany.
- 234 Langley, N., "Production Line-Up", Computing, 1 February 1996, p 21
- 235 Harrison, R. and Hopkinson P., "Synect: A Method and CASE Tool for Generating Distributed Sequence Logic", LUI International Conference, Santa Clara, California, 19-21 May 1996.

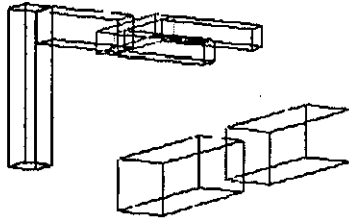
Appendix A Walkthrough of Synect Application Development

A.1 Introduction

This appendix shows how a simple application could be developed using Synect and describes business benefits to be gained compared with traditional approaches.

A.2 Description of the Plant Equipment

The diagram shows a flexible manufacturing cell consisting of a feed conveyor, robot arm, machine and exit conveyor (exit conveyor not shown in the diagram). The robot arm is shown positioned over the feed conveyor. The robot arm has a gripper which can open and close, elevation control and can move between the feed conveyor, the machine and the exit conveyor. The machine executes two phases sequentially to manufacture the part.



The I/O associated with this equipment (all discrete) is:

Feed conveyor

Inputs

- Proximity sensor detecting presence of a new raw part to be machined (1 = part present)

Outputs

- Conveyor motor control (1 = run motor)

Robot gripper

Inputs

- Limit switch on closed position (1 = gripper is closed)
- Limit switch on open position (1 = gripper is open)

Outputs

- Solenoid valve control (0 = open gripper, 1 = close gripper)

Robot arm elevation

Inputs

- Limit switch on raised position (1 = in raised position)
- Limit switch on lowered position (1 = in lowered position)

Outputs

- Solenoid valve control (0 = raise arm, 1 = lower arm)

Robot arm traversal

Inputs

- Limit switch when robot arm is above the feed conveyor (1 = in position)
- Limit switch when robot arm is above the machine (1 = in position)
- Limit switch when robot arm is above the exit conveyor (1 = in position)

Outputs (valid combinations are (0,0), (0,1), (1,0) i.e. (1,1) is invalid)

- Move towards feed conveyor (1 = move)
- Move towards exit conveyor (1 = move)

Machine

Inputs

- Finished first phase (1 = finished)
- Finished second phase (1 = finished)

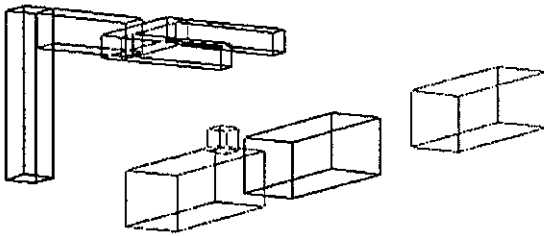
Outputs (valid combinations are (0,0), (0,1), (1,0) i.e. (1,1) is invalid)

- Run phase 1 (1 = run)
- Run phase 2 (1 = run)

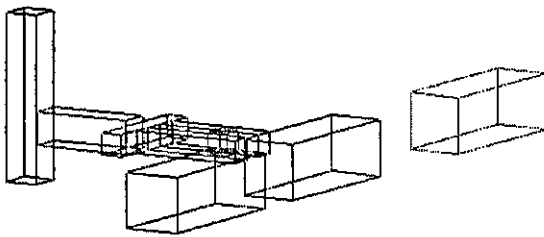
A.3 Description of the Process

The objective is to machine raw parts to transform them into manufactured (finished) parts. A raw part arrives on the feed conveyor and leaves via the exit conveyor.

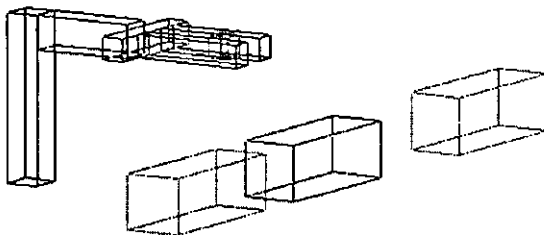
When a new raw part arrives, the robot arm is to pick it up and take it to the machine before returning to its "home" position at the feed conveyor. To avoid collisions with fixed plant equipment, the robot arm must be in the raised position when traversing between the feed conveyor, machine and exit conveyor. When the machine has finished its second phase, the robot arm should pick up the finished part and take it to the exit conveyor before returning home again.



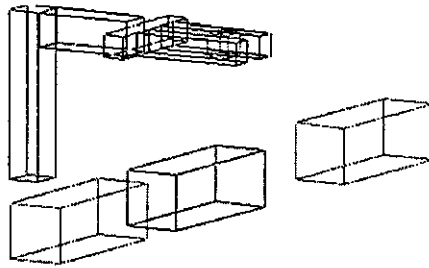
The new raw part arrives on the feed conveyor.



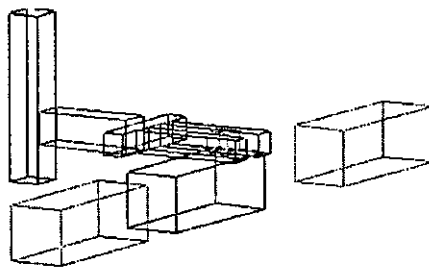
The robot picks up the raw part ...



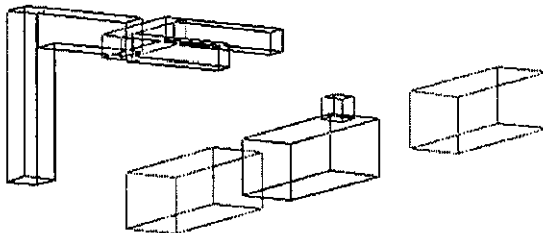
And takes it to the machine ...



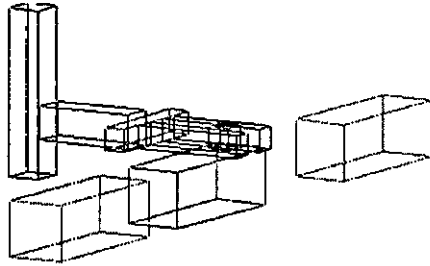
With the arm in the raised position in order to avoid collisions.



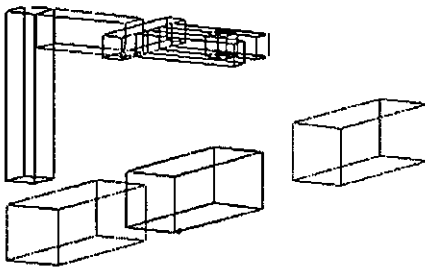
The part is passed to the machine and the machine started...



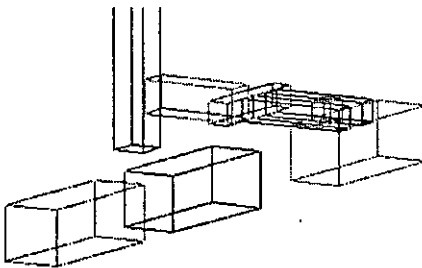
Whilst the robot arm returns to its home position, waiting for the machine to complete its operation.



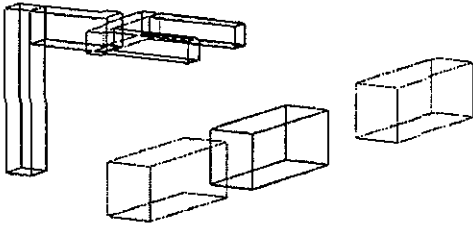
The robot then picks up the finished part from the machine ...



And takes it to the exit conveyor via the raised position again.



The finished part is passed to the exit conveyor.

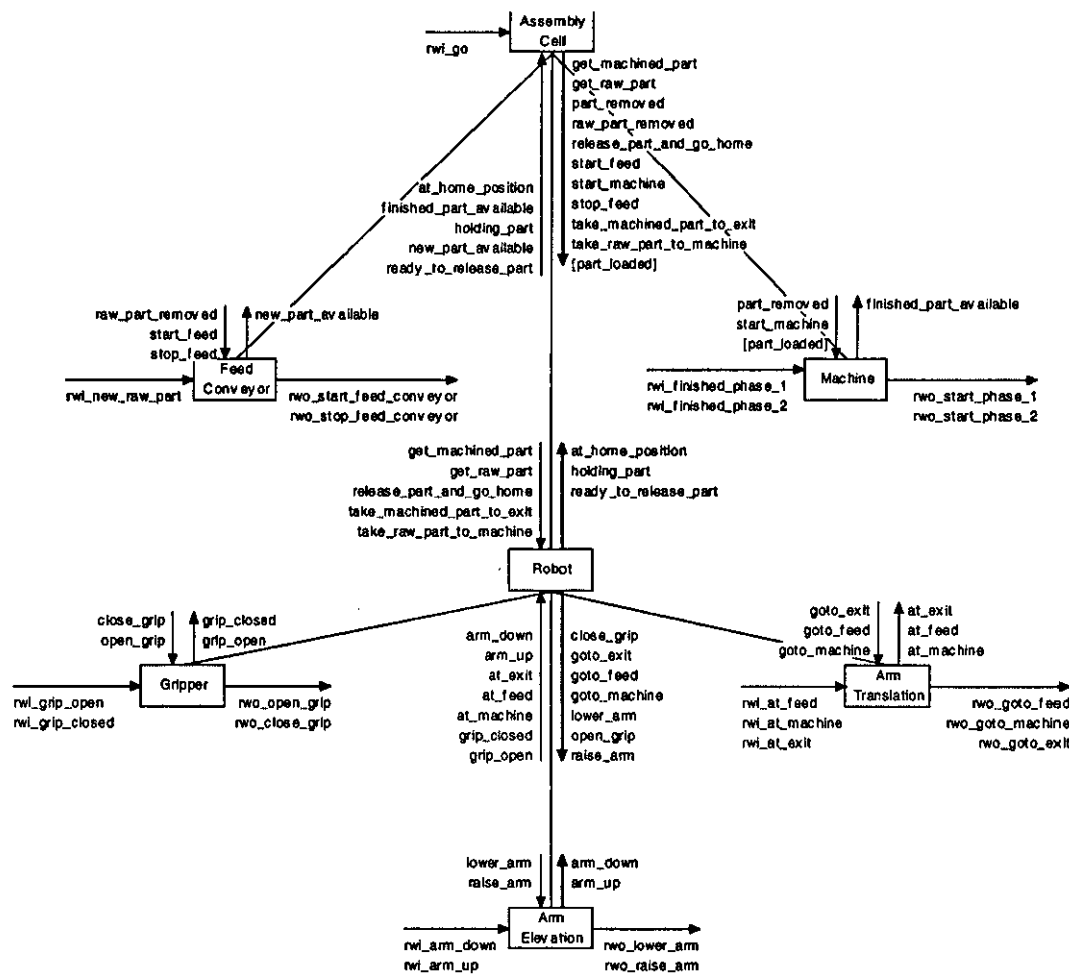


And the robot returns home. The cycle is complete.

A.4 A Solution Using Synect

The following pages describe a solution implemented using Synect, using screen-captures to show the Synect tools. For completeness, the entire solution is initially described.

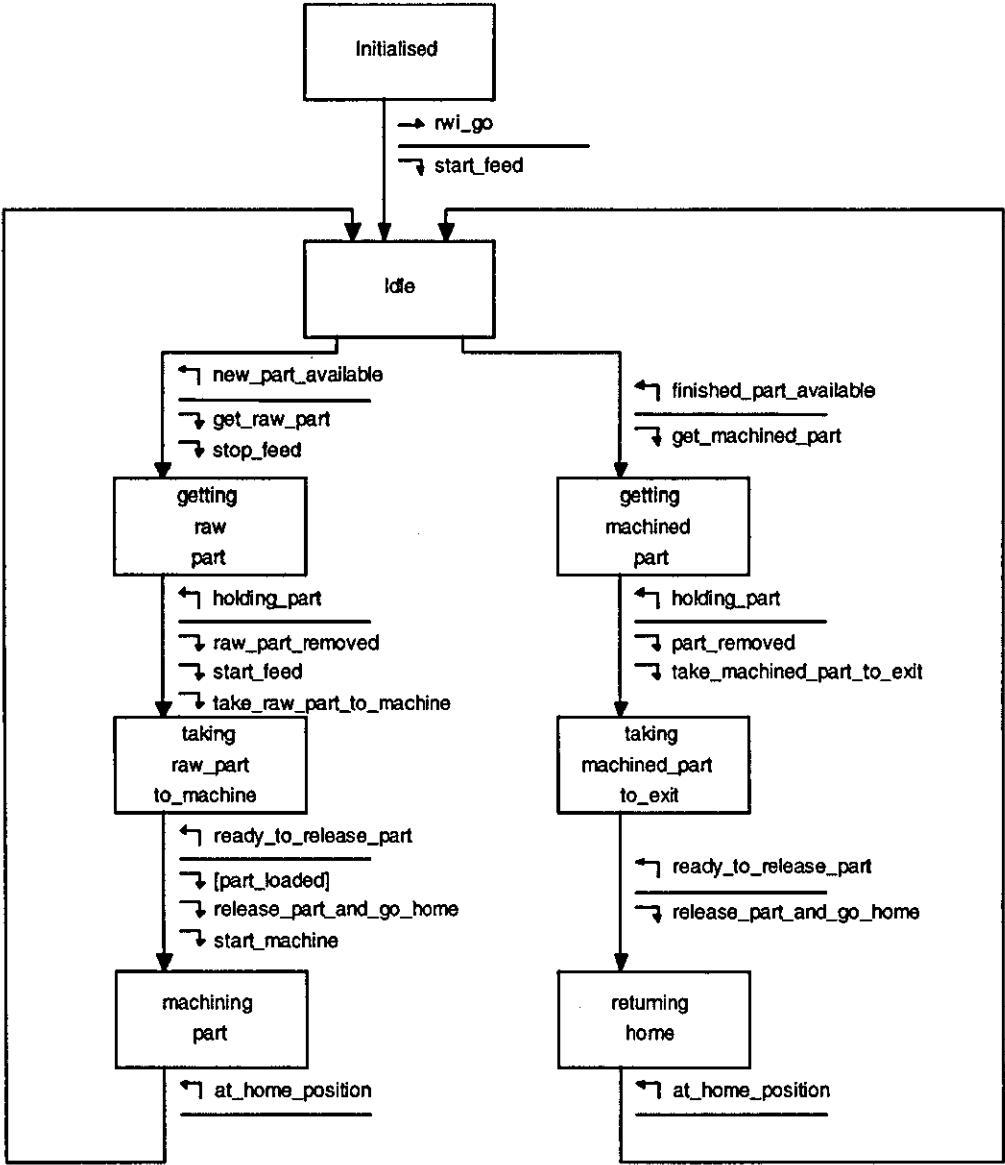
A.4.1 The Object Hierarchy



The object hierarchy shows the objects modelled in the solution, the communication between them and the interface with the plant's sensors and actuators (referred to as real-world inputs and outputs). At the highest level of abstraction, the application consists of an assembly cell. This is comprised of a feed conveyor, a machine and a robot (the exit conveyor is ignored because it is assumed to run continuously). The robot consists of a gripper, arm elevation control and arm translation control.

The vertical arrows indicate messages between objects (only adjacent layers i.e. the assembly cell cannot communicate directly with the gripper, for example). For example, the arrow from the bottom of the assembly cell object includes a reference to message “get_machined_part” indicating that it sends this message to one of its children. Correspondingly, the robot object also refers to this message beside the arrow into the top of its box.

A.4.2 Assembly Cell STD

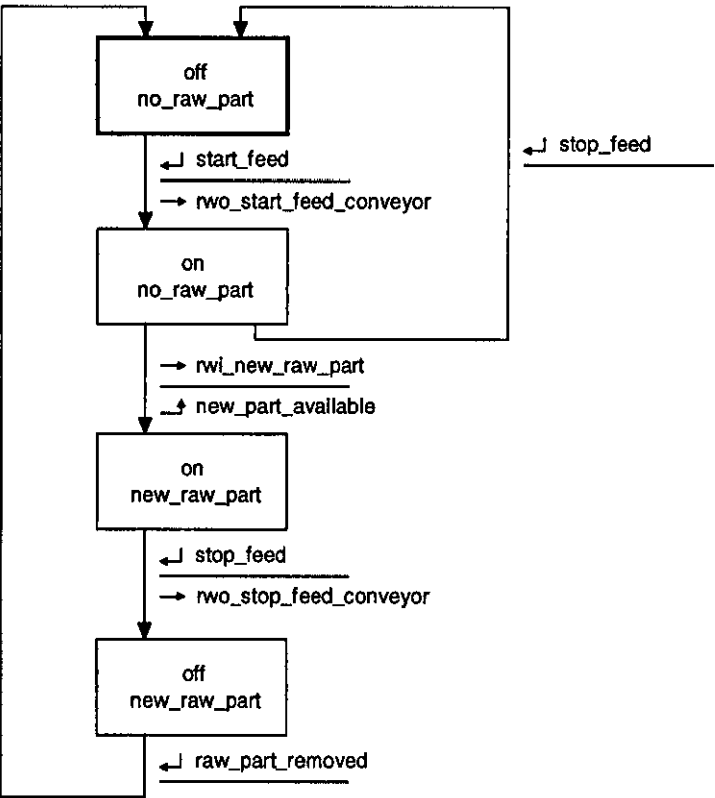


The diagram shows the state transition diagram (STD) corresponding to the “Assembly Cell” object. The initial state is “Initialised”. When condition “rwi_go” is true, the transition from state “Initialised” fires, invoking the action to send message “start_feed” to the feed

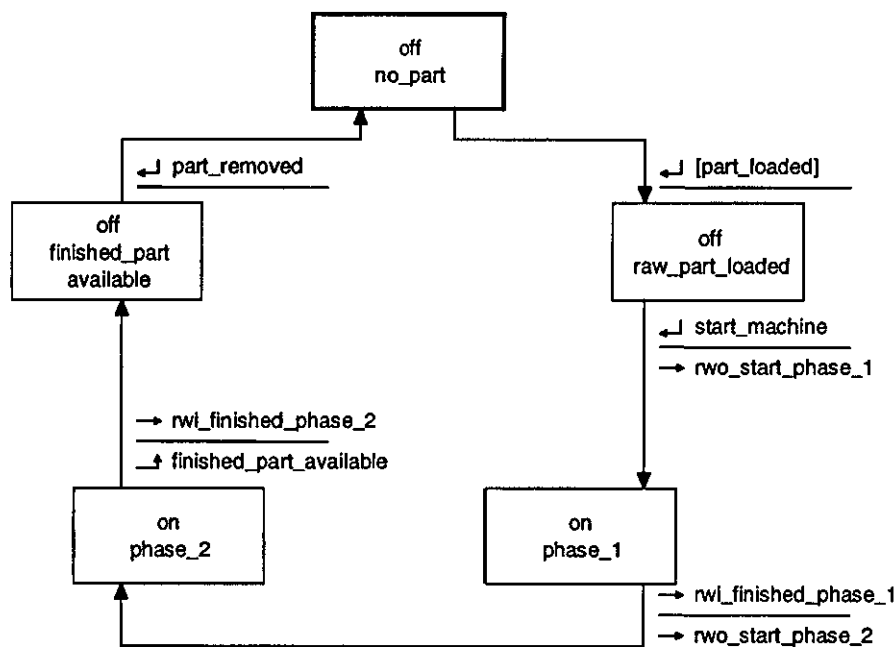
conveyor object and changing the current state to "Idle". From state "Idle", the STD awaits receipt of either the "new_part_available" message or the "finished_part_available" message.

The conditions and actions can be seen to correspond to the text shown around the "Assembly Cell" object.

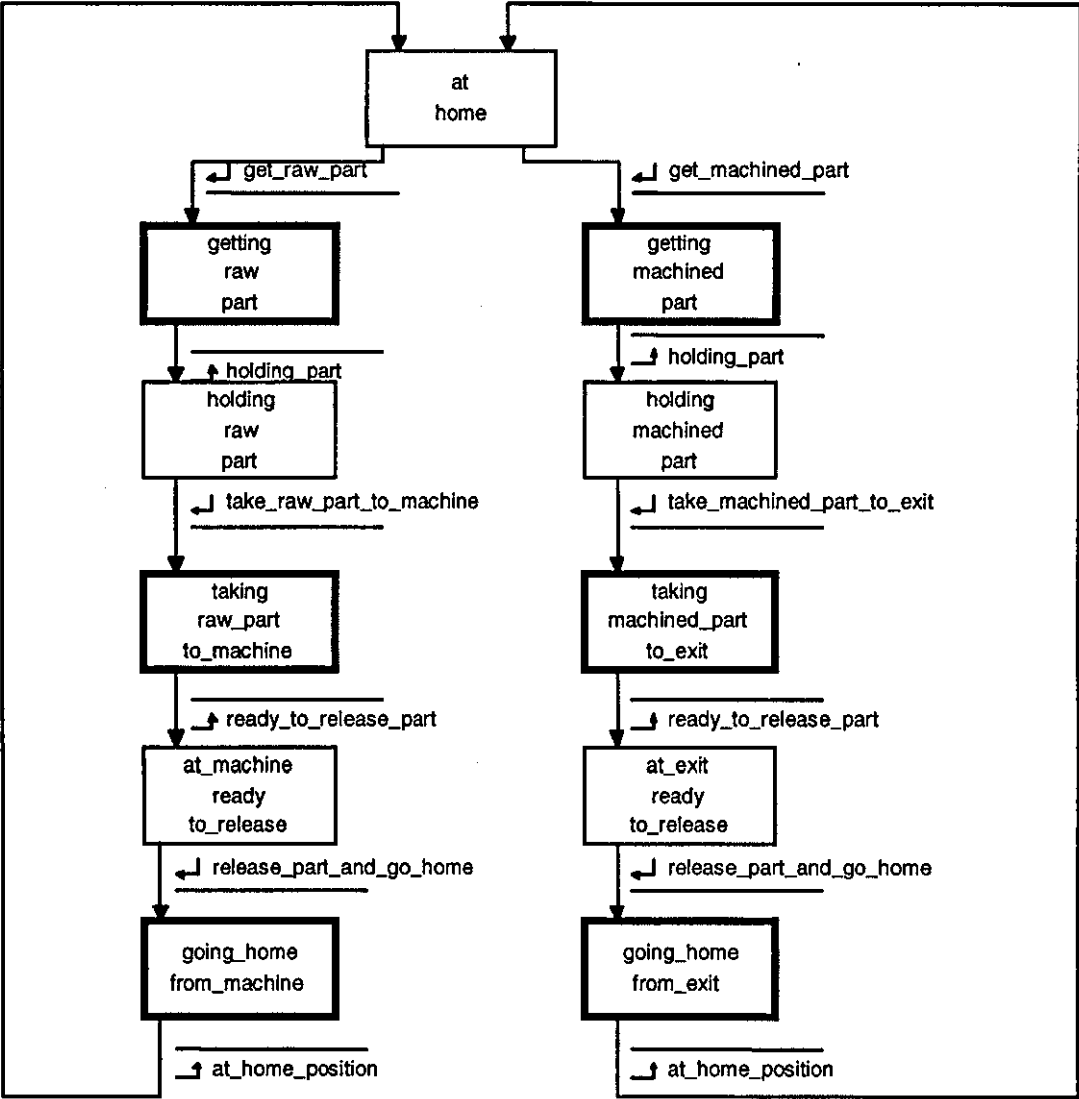
A.4.3 Feed Conveyor STD



A.4.4 Machine STD

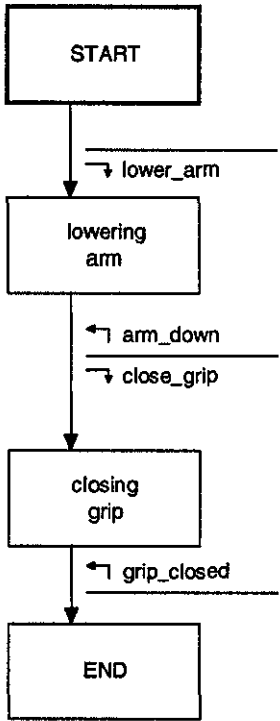


A.4.5 Robot STD

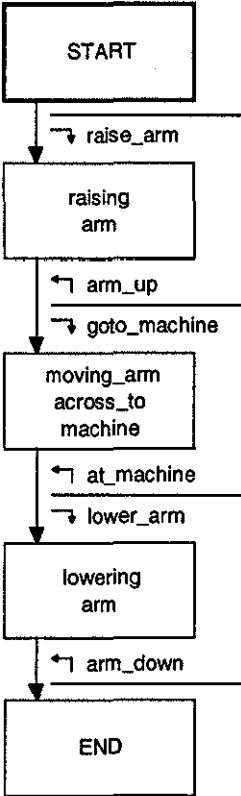


The robot STD shows examples of macro states (represented by very thick borders). Each macro state contains a sub-sequence STD, as shown in the STDs on the following page.

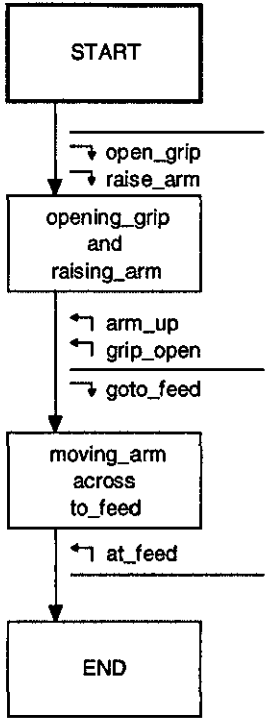
Getting Raw Part



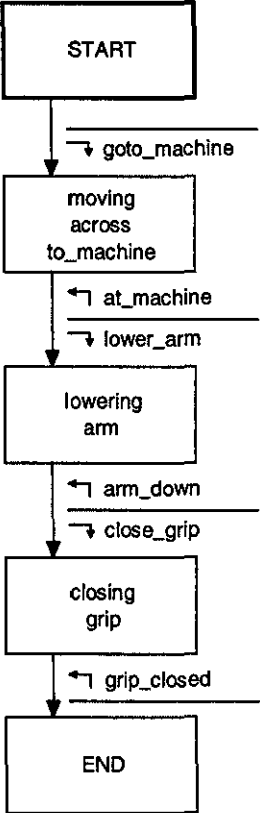
Taking Raw Part To Machine



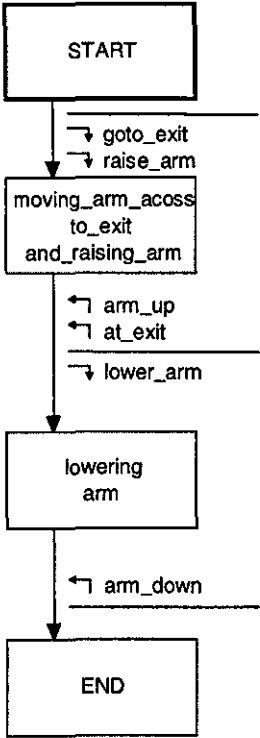
Going Home From Machine



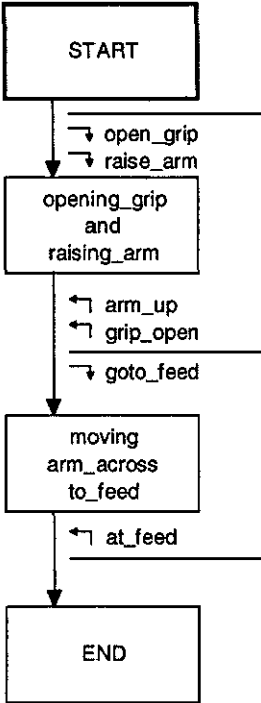
Getting Machined Part



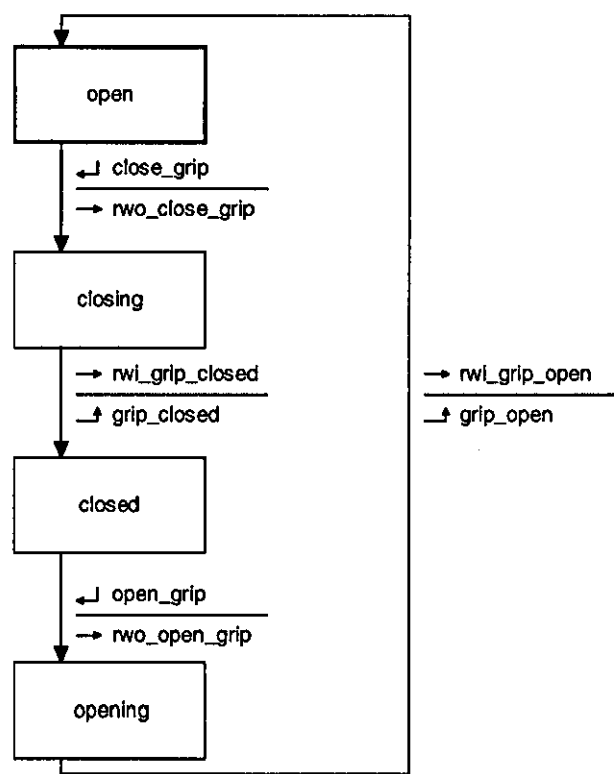
Taking Machined Part To Exit



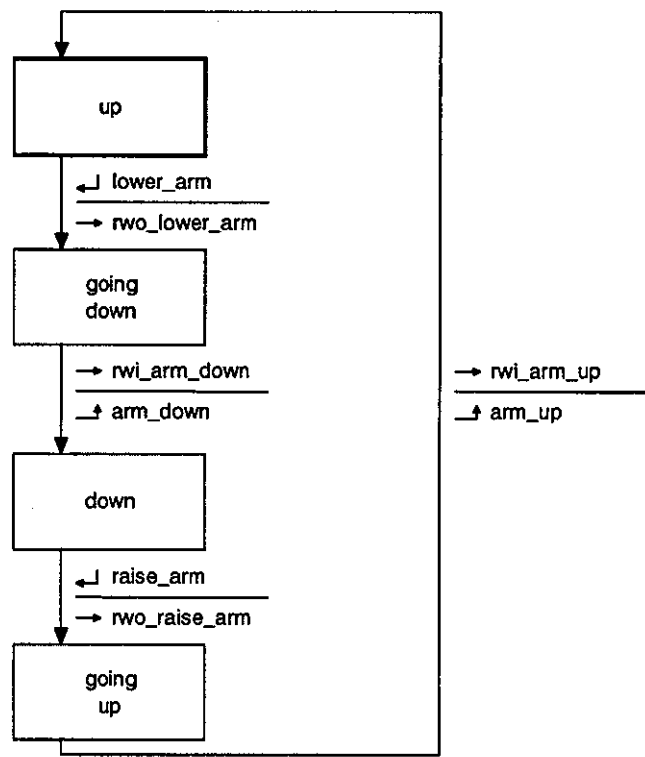
Going Home From Exit



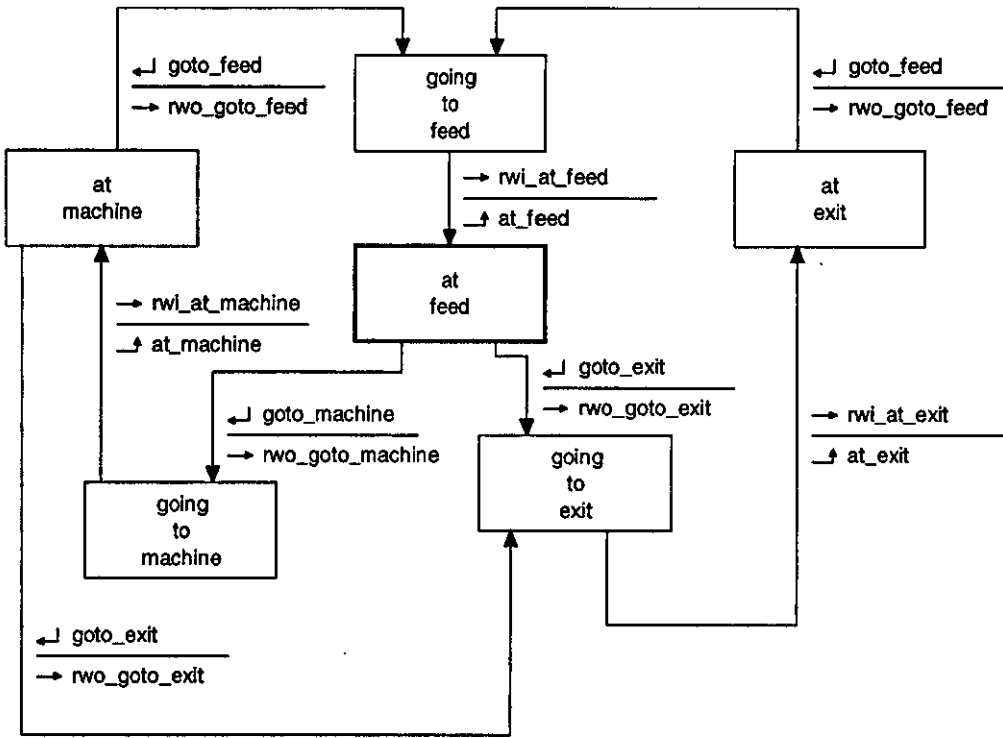
A.4.6 Gripper STD



A.4.7 Arm Elevation STD

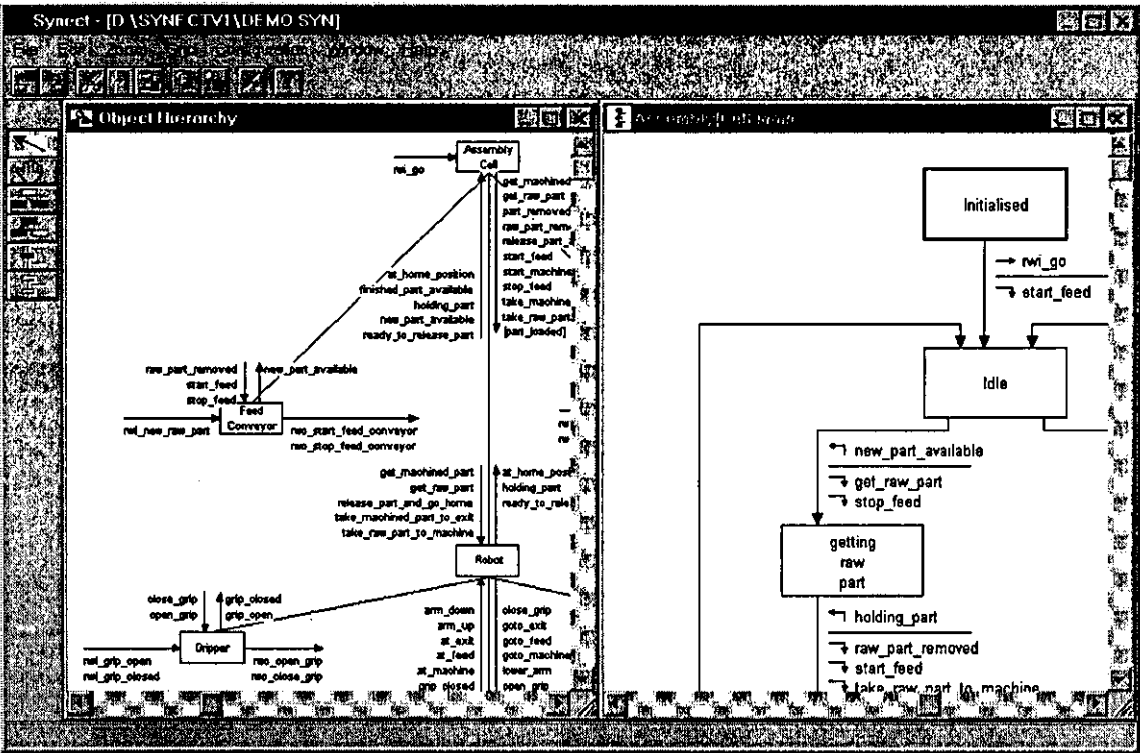


A.4.8 Arm Translation STD



A.4.9 Using The Synect Tools to Develop The Application

A.4.9.1 Specify

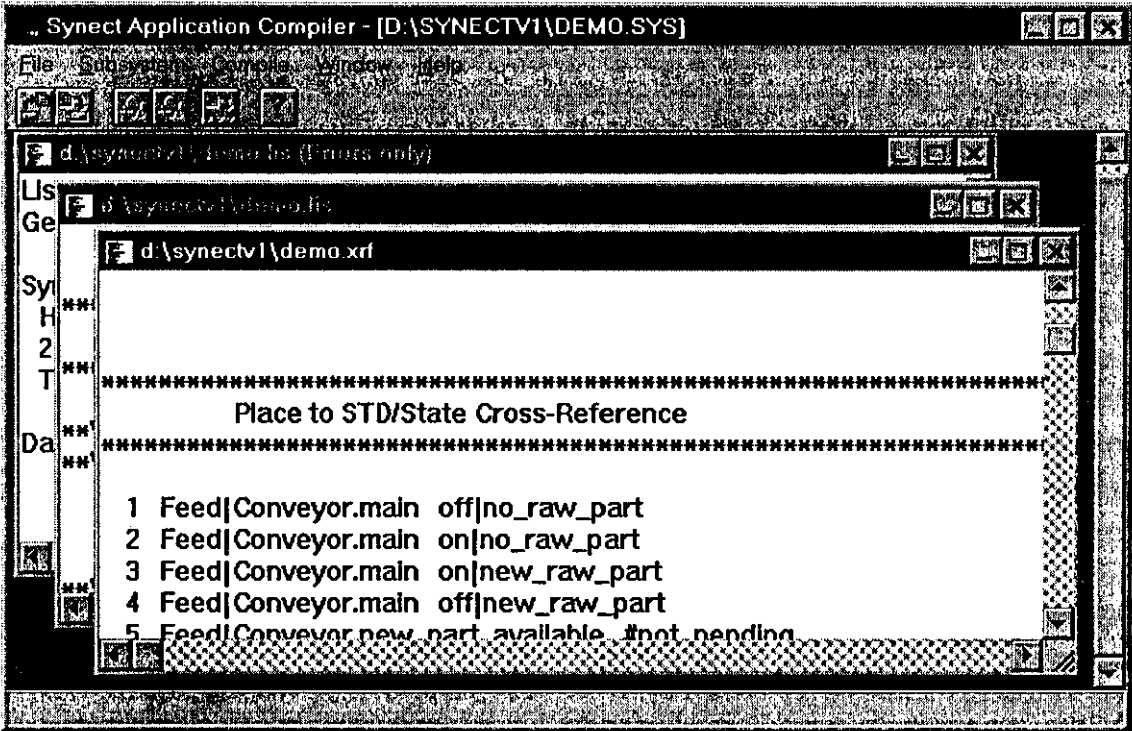


The analyst uses the Application Editor to specify the object hierarchy and the STDs. The Editor provides better support than a standard drawing package by being method-aware. For example, typing each name once and then using pick-lists assists in the early development of a syntactically correct specification.

Even at this stage, the diagrams could be printed out and used as the basis of a review with the end-user. The use of clear and simple notations in the object hierarchy and state transition diagrams enables the review team to focus on one aspect of the application at a time. Multi-disciplinary review teams can therefore be highly effective.

On subsequent projects, objects could be re-used to minimise development time and cost.

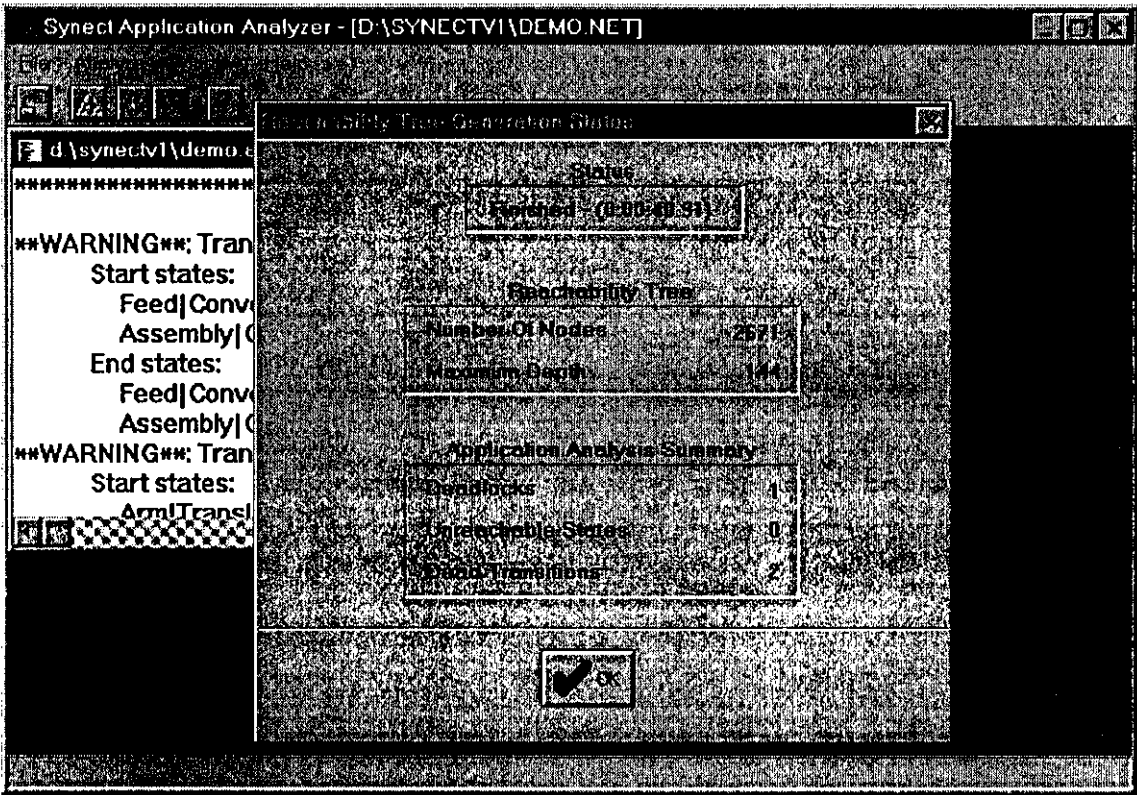
A.4.9.2 Compile



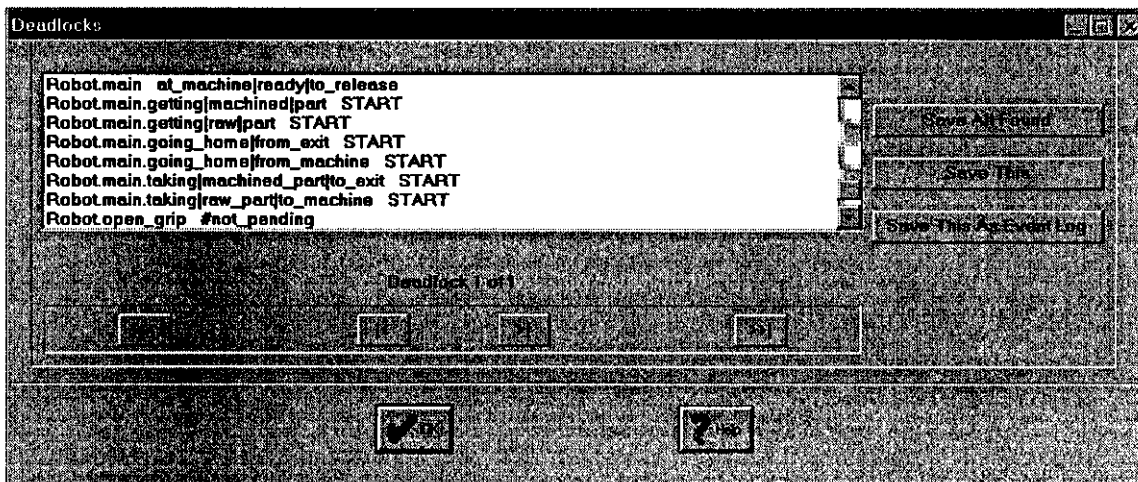
The Compiler verifies that the specification is syntactically correct and derives a corresponding Petri net, producing diagnostic information such as a cross-reference from Petri net place to STD state.

To gain additional confidence in the correctness of the logic, a part of the application may be extracted and written to a separate file for subsequent analysis and simulation. For example, the behaviour of the robot and its children could be investigated. Where large applications make the analysis of the whole system infeasible, this mechanism provides a mechanism for partitioning the system into intuitively meaningful sub-systems with limited scope.

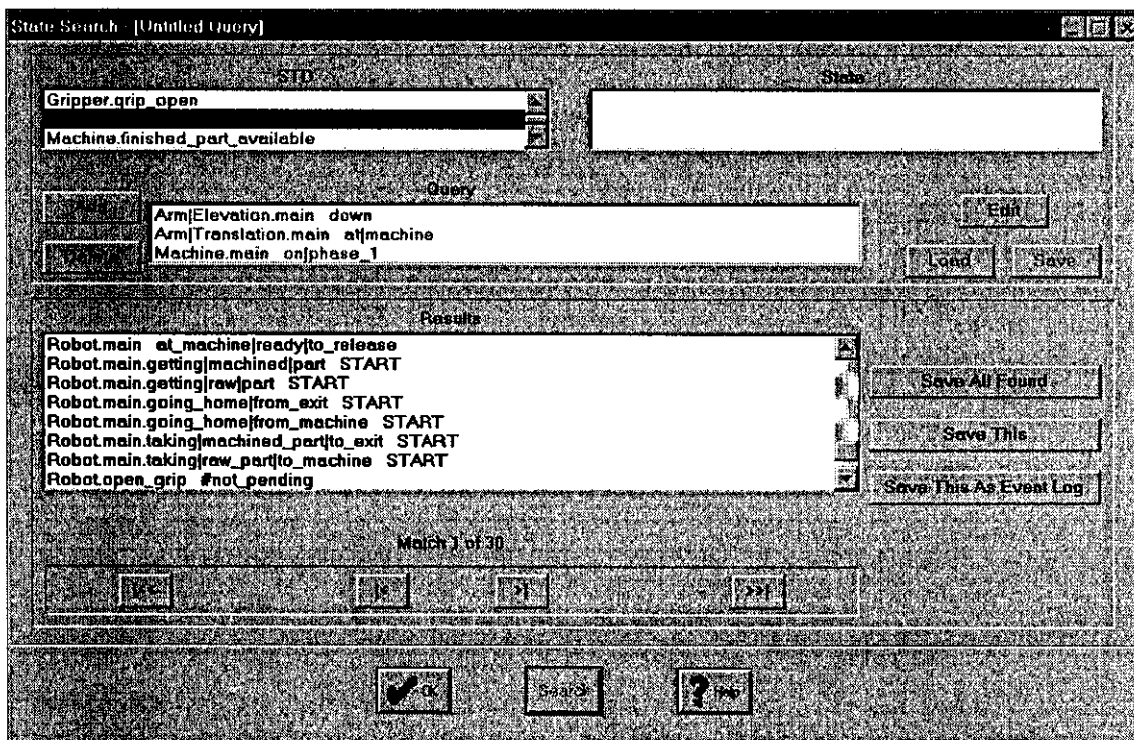
A.4.9.3 Analyse



The analyst checks the behavioural properties of the application. The Analyzer generates the reachability tree and summarises the number of deadlocks, unreachable states and dead transitions found. More detailed information is written to a list file.



The analyst invokes the Analyzer's deadlock query dialog to obtain more information about the deadlock and can generate a text file showing the state evolution from start to the deadlocked state and can also save this information as an event log for replay via the Simulator.

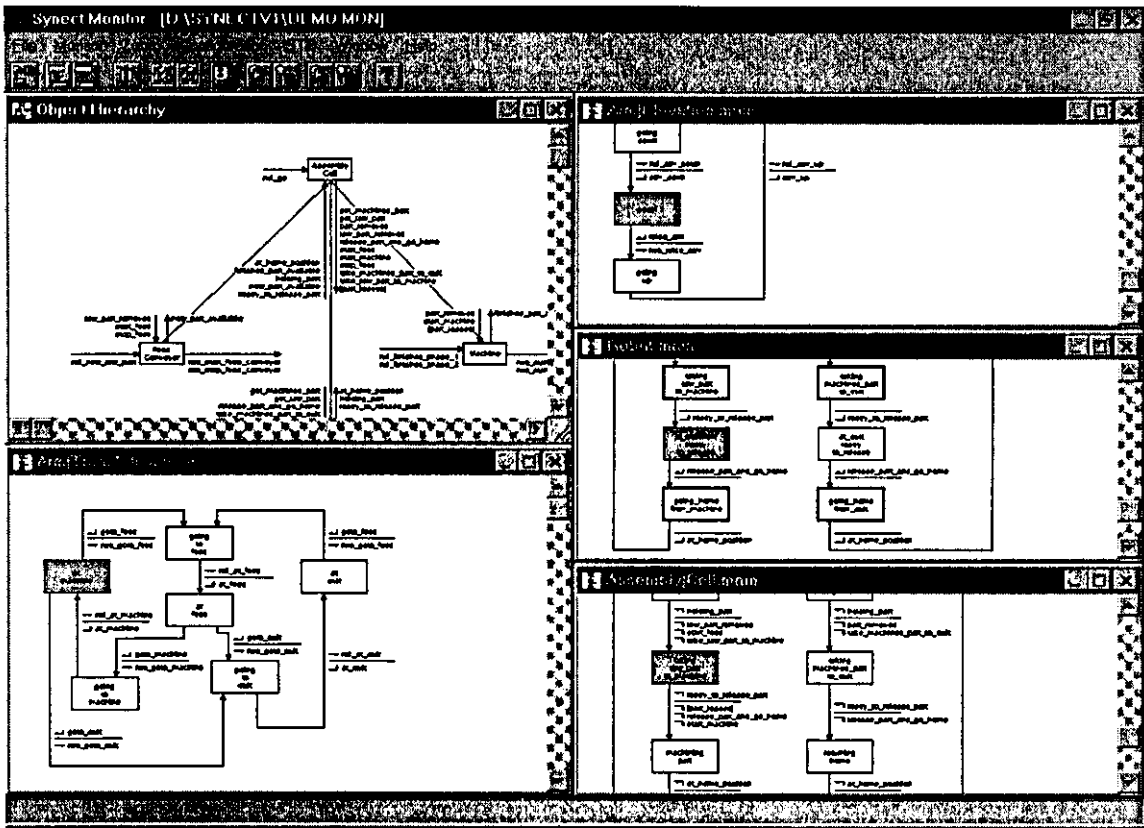


The analyst can use the state search query dialog to test for reachable states. In the example above, the query has revealed that the application can reach a system state where the arm elevation is down whilst at the machine with the machine running the first phase of its operation. As with the deadlock dialog, the path from the system's start state to the state found can be saved to a text file or as an event log for replay via the Simulator.

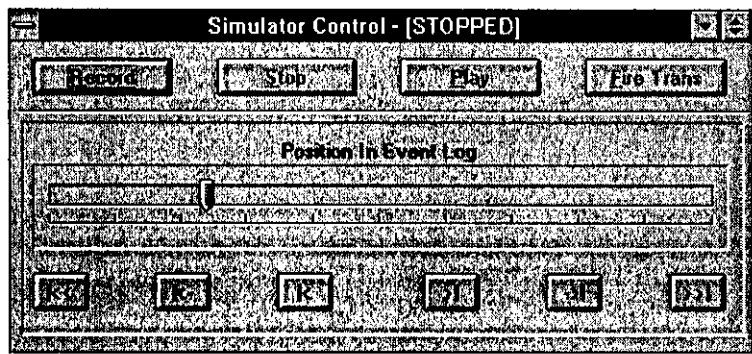
A library of queries can be saved to disk for subsequent application verification following a modification.

The Analyzer offers the analyst considerably greater confidence in the control logic. Simulation tools alone would not guarantee that the particular sequence of events resulting in the deadlock, for example, would be tested. In the example, the path to deadlock is initiated by a new raw part arriving at the feed conveyor whilst the machine is busy. The robot fetches the new raw part but cannot pass it to the machine, nor can it fetch the finished part from the machine. If the simulation testing always allowed the machine to finish before the next new raw part arrived, the fault would go undetected. After the system has entered beneficial operation, external factors such as a de-bottlenecking exercise on the upstream plant equipment, could cause the logic error to manifest itself, resulting in plant downtime and consequently lost production.

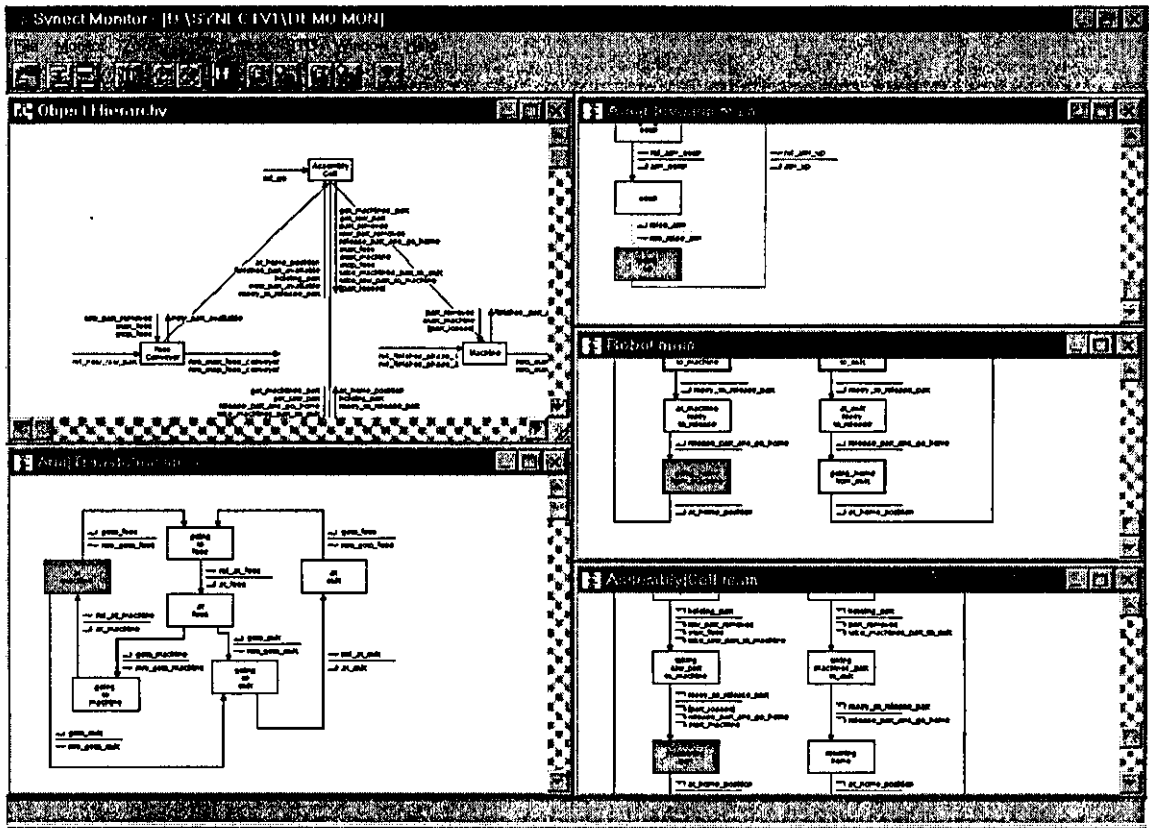
A.4.9.4 Simulate and Animate



The STD Monitor application shows the current state of each STD in grey. The application is loaded into the Simulator and one of the event logs written by the Analyzer is loaded (in the graphic above, the event log corresponding to the deadlock was loaded). The STD Monitor then shows the system state at the end of the event log (i.e. the system state at deadlock in the above example).



Using the Simulator's control panel, the analyst single-steps the application through the event log to determine why the deadlock occurred. After each step, the STD Monitor shows the current state of the STDs on display.



Real World Input Status

rwi_go

> rwi_new_raw_part

> rwi_grip_open

rwi_grip_closed

rwi_arm_down

> rwi_arm_up

rwi_at_feed

rwi_at_machine

rwi_at_exit

> rwi_finished_phase_1

rwi_finished_phase_2

Real World Outputs

21:55:55.94

rwo_start_feed_conveyor

21:56:02.37

rwo_stop_feed_conveyor

21:56:02.48

rwo_lower_arm

21:56:06.43

rwo_close_grip

21:56:12.97

rwo_start_feed_conveyor

21:56:13.08

rwo_raise_arm

21:56:17.86

rwo_goto_machine

21:56:19.94

rwo_lower_arm

21:56:23.18

rwo_start_phase_1

21:56:23.40

rwo_open_grip

21:56:23.51

rwo_raise_arm

Clear Listbox

The analyst can “drive” the application interactively via the Simulator. When instructed to execute the model (by clicking on the Record button on the Simulator control panel), the “Real World Input Status” dialog shows which real-world inputs are being examined via the “>” symbol. The analyst selects the entry to simulate the condition being satisfied. The status of these real-world inputs is reflected in the STD Monitor display on the Object Hierarchy and STD displays.

As the Simulator fires transitions, time-stamped references to the corresponding real-world outputs are appended to the Real-World Outputs dialog.

The analyst can stop the execution of the Petri net, scroll backwards and forwards through the event log and begin executing from a different system state. The event log can be saved to disk and reloaded for replay as a demonstration to the end-user of a scenario of the system’s reaction to a sequence of events.

Alternative visualisations of system behaviour, such as 3D modellers and process-industry mimics, can be obtained through the use of external software packages communicating with the Simulator to read and write real-world input statuses and be notified of real-world output invocation.

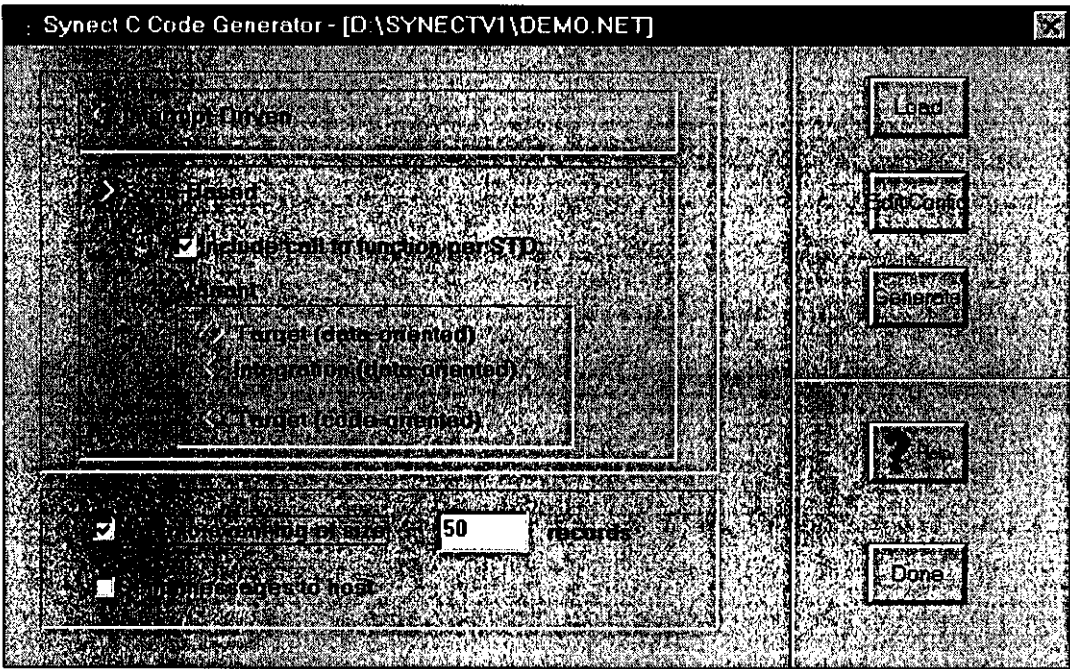
These external package can also be used as an alternative user interface for driving the simulator, useful for operator training, or to determine performance attributes. For example, a 3D software package could emulate the arm elevation equipment by waiting for the “two_raise_arm” real-world output, delaying by a pre-configured period and then setting the “rwi_arm_up” real-world input. An additional application could then communicate with the Simulator to determine performance characteristics, such as cycle time, machine utilisation statistics, etc..

As a consequence, the analyst is able to gain a much deeper understanding of the behaviour of the proposed control logic and has an effective means for reviewing this behaviour with a multi-disciplinary team.

Typical development is iterative, reflecting “round-trip gestalt design” [44], with the analyst correcting errors and verifying the behaviour via a review process with process, operations and control system personnel. For the purposes of this example, the specification is now assumed to be as required – an unambiguous and well-understood specification of the required control system behaviour.

A.4.9.5 Code Generation

The Petri net is an excellent model for the control logic for which code generators can easily be developed. Synect currently includes relay ladder logic, Echelon LonWorks Neuron C and ANSI C code generators. The use of the ANSI C code generator will be described.



The example above generates scan-based ANSI C code, with a call to a function corresponding to each STD and writing details of transitions fired to a circular event log. If the target control system exhibits anomalous behaviour, the event log can be copied to a remote computer and investigated via the Simulator and STD Monitor.

The “Send messages to host” checkbox provides a mechanism for the control system to report the firing of transitions to a remote application in real-time. Other code generators, such as the Neuron C Code Generator, require the remote application to poll the control system for its current state. In each case, the goal is to display the current state of the live control system using the STD Monitor to animate the analyst’s original specification.

These facilities ensure that the code produced is a faithful representation of the specified control logic, ensure consistency of implementation architecture and facilitate rapid maintenance through the use of built-in diagnostics.

The files produced by the code generator from the above configuration are as follows.

A.4.9.5.1 Demo.c

```
/*
C control program corresponding to C:\SYNECT\DEMO.SYN

Generated by Synect C Code Generator V1.8
Synect is a registered trademark belonging to:
Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough, Cleveland, TS14 8JY, England
Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk

Generated at: Mon Aug 24 23:35:25 1998

Program type : scan based
Code for integration purposes omitted from program
Call to function per STD inserted in program
Program writes to event log of size 50 records
Program does not send messages to host

Related files (compile as C source and link in, or replace with your own):
.RWI    Skeletal real world input functions
.RWO    Skeletal real world output functions
.ENV    Functions under the control of the target environment
.STD    Skeletal function per STD
*/

/* Application-independent constant definitions */
#define TRUE 1
#define FALSE 0
#define RWI_NOT_TESTED 0
#define RWI_DISABLED 1
#define RWI_ENABLED 2

/* Application-independent typedefs */
typedef int (*REAL_WORLD_INPUT_FUNCTION) ();
typedef void (*REAL_WORLD_OUTPUT_FUNCTION) ();
typedef void (*STD_FUNCTION) ();

/* Include the application-dependent defines and declarations */
#include "demo.h"

/* Application-independent user function declarations */
extern void initialise_environment (void) ;
extern void deadlock_detected (void) ;
extern void scan_complete (int) ;
extern struct el_date_time_struct get_current_date_time (void) ;
extern void event_log_error (int) ;
```

```

/* Function prototypes */
int main (void) ;
void initialise (void) ;
int is_trans_net_enabled (int) ;
int is_trans_io_enabled (int) ;
int select_trans_to_fire (void) ;
int fire_pn_trans (int) ;
void invoke_trans_io_real_world_outputs (int) ;
extern void Feed_Conveyor_main (int) ;
extern void Feed_Conveyor_new_part_available (int) ;
extern void Gripper_main (int) ;
extern void Gripper_grip_closed (int) ;
extern void Gripper_grip_open (int) ;
extern void Arm_Elevation_main (int) ;
extern void Arm_Elevation_arm_down (int) ;
extern void Arm_Elevation_arm_up (int) ;
extern void Arm_Translation_main (int) ;
extern void Arm_Translation_at_exit (int) ;
extern void Arm_Translation_at_feed (int) ;
extern void Arm_Translation_at_machine (int) ;
extern void Robot_main (int) ;
extern void Robot_main_getting_raw_part (int) ;
extern void Robot_main_taking_raw_part_to_machine (int) ;
extern void Robot_main_going_home_from_machine (int) ;
extern void Robot_main_getting_machined_part (int) ;
extern void Robot_main_taking_machined_part_to_exit (int) ;
extern void Robot_main_going_home_from_exit (int) ;
extern void Robot_at_home_position (int) ;
extern void Robot_holding_part (int) ;
extern void Robot_ready_to_release_part (int) ;
extern void Robot_close_grip (int) ;
extern void Robot_goto_exit (int) ;
extern void Robot_goto_feed (int) ;
extern void Robot_goto_machine (int) ;
extern void Robot_lower_arm (int) ;
extern void Robot_open_grip (int) ;
extern void Robot_raise_arm (int) ;
extern void Machine_main (int) ;
extern void Machine_finished_part_available (int) ;
extern void Assembly_Cell_main (int) ;
extern void Assembly_Cell_get_machined_part (int) ;
extern void Assembly_Cell_get_raw_part (int) ;
extern void Assembly_Cell_part_removed (int) ;
extern void Assembly_Cell_raw_part_removed (int) ;
extern void Assembly_Cell_release_part_and_go_home (int) ;
extern void Assembly_Cell_start_feed (int) ;
extern void Assembly_Cell_start_machine (int) ;
extern void Assembly_Cell_stop_feed (int) ;
extern void Assembly_Cell_take_machined_part_to_exit (int) ;
extern void Assembly_Cell_take_raw_part_to_machine (int) ;
void save_rwi_change (int, int) ;
void save_pn_trans_fired (int) ;
void write_to_event_log (int, int, int) ;

/* Application-dependent user function prototypes */
extern int rwi_new_raw_part (void) ;
extern int rwi_grip_closed (void) ;
extern int rwi_grip_open (void) ;
extern int rwi_arm_down (void) ;
extern int rwi_arm_up (void) ;
extern int rwi_at_machine (void) ;
extern int rwi_at_exit (void) ;
extern int rwi_at_feed (void) ;
extern int rwi_finished_phase_1 (void) ;

```

```

extern int rwi_finished_phase_2 (void) ;
extern int rwi_go (void) ;
extern void rwo_start_feed_conveyor (void) ;
extern void rwo_stop_feed_conveyor (void) ;
extern void rwo_close_grip (void) ;
extern void rwo_open_grip (void) ;
extern void rwo_lower_arm (void) ;
extern void rwo_raise_arm (void) ;
extern void rwo_goto_machine (void) ;
extern void rwo_goto_exit (void) ;
extern void rwo_goto_feed (void) ;
extern void rwo_start_phase_1 (void) ;
extern void rwo_start_phase_2 (void) ;

/* Structure declarations */
REAL_WORLD_INPUT_FUNCTION real_world_input_routine [] = {
    rwi_new_raw_part,
    rwi_grip_closed,
    rwi_grip_open,
    rwi_arm_down,
    rwi_arm_up,
    rwi_at_machine,
    rwi_at_exit,
    rwi_at_feed,
    rwi_finished_phase_1,
    rwi_finished_phase_2,
    rwi_go} ;

REAL_WORLD_OUTPUT_FUNCTION real_world_output_routine [] = {
    rwo_start_feed_conveyor,
    rwo_stop_feed_conveyor,
    rwo_close_grip,
    rwo_open_grip,
    rwo_lower_arm,
    rwo_raise_arm,
    rwo_goto_machine,
    rwo_goto_exit,
    rwo_goto_feed,
    rwo_start_phase_1,
    rwo_start_phase_2} ;

struct trans_io_struct {
    int num_real_world_input_routines ;
    int rwi_index [REAL_WORLD_INPUT_LIST_SIZE] ;
    int num_real_world_output_routines ;
    int rwo_index [REAL_WORLD_OUTPUT_LIST_SIZE] ;
} ;

STD_FUNCTION place_to_std [] = {
    Feed_Conveyor_main,
    Feed_Conveyor_main,
    Feed_Conveyor_main,
    Feed_Conveyor_main,
    Feed_Conveyor_new_part_available,
    Feed_Conveyor_new_part_available,
    Gripper_main,
    Gripper_main,
    Gripper_main,
    Gripper_main,
    Gripper_grip_closed,
    Gripper_grip_closed,
    Gripper_grip_open,
    Gripper_grip_open,
    Arm_Elevation_main,

```

Arm_Elevation_main,
 Arm_Elevation_main,
 Arm_Elevation_main,
 Arm_Elevation_arm_down,
 Arm_Elevation_arm_down,
 Arm_Elevation_arm_up,
 Arm_Elevation_arm_up,
 Arm_Translation_main,
 Arm_Translation_main,
 Arm_Translation_main,
 Arm_Translation_main,
 Arm_Translation_main,
 Arm_Translation_main,
 Arm_Translation_at_exit,
 Arm_Translation_at_exit,
 Arm_Translation_at_feed,
 Arm_Translation_at_feed,
 Arm_Translation_at_machine,
 Arm_Translation_at_machine,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main,
 Robot_main_getting_raw_part,
 Robot_main_getting_raw_part,
 Robot_main_getting_raw_part,
 Robot_main_taking_raw_part_to_machine,
 Robot_main_taking_raw_part_to_machine,
 Robot_main_taking_raw_part_to_machine,
 Robot_main_taking_raw_part_to_machine,
 Robot_main_going_home_from_machine,
 Robot_main_going_home_from_machine,
 Robot_main_going_home_from_machine,
 Robot_main_getting_machined_part,
 Robot_main_getting_machined_part,
 Robot_main_getting_machined_part,
 Robot_main_getting_machined_part,
 Robot_main_taking_machined_part_to_exit,
 Robot_main_taking_machined_part_to_exit,
 Robot_main_taking_machined_part_to_exit,
 Robot_main_going_home_from_exit,
 Robot_main_going_home_from_exit,
 Robot_main_going_home_from_exit,
 Robot_at_home_position,
 Robot_at_home_position,
 Robot_holding_part,
 Robot_holding_part,
 Robot_ready_to_release_part,
 Robot_ready_to_release_part,
 Robot_close_grip,
 Robot_close_grip,
 Robot_goto_exit,
 Robot_goto_exit,
 Robot_goto_feed,
 Robot_goto_feed,
 Robot_goto_machine,
 Robot_goto_machine,


```

Robot_lower_arm,
Robot_lower_arm,
Robot_open_grip,
Robot_open_grip,
Robot_raise_arm,
Robot_raise_arm,
Machine_main,
Machine_main,
Machine_main,
Machine_main,
Machine_main,
Machine_finished_part_available,
Machine_finished_part_available,
Assembly_Cell_main,
Assembly_Cell_main,
Assembly_Cell_main,
Assembly_Cell_main,
Assembly_Cell_main,
Assembly_Cell_main,
Assembly_Cell_main,
Assembly_Cell_main,
Assembly_Cell_get_machined_part,
Assembly_Cell_get_machined_part,
Assembly_Cell_get_raw_part,
Assembly_Cell_get_raw_part,
Assembly_Cell_part_removed,
Assembly_Cell_part_removed,
Assembly_Cell_raw_part_removed,
Assembly_Cell_raw_part_removed,
Assembly_Cell_release_part_and_go_home,
Assembly_Cell_release_part_and_go_home,
Assembly_Cell_start_feed,
Assembly_Cell_start_feed,
Assembly_Cell_start_machine,
Assembly_Cell_start_machine,
Assembly_Cell_stop_feed,
Assembly_Cell_stop_feed,
Assembly_Cell_take_machined_part_to_exit,
Assembly_Cell_take_machined_part_to_exit,
Assembly_Cell_take_raw_part_to_machine,
Assembly_Cell_take_raw_part_to_machine) ;

/* Global definitions */
int I [NUM_TRANS] [NUM_PLACES] ;
int O [NUM_TRANS] [NUM_PLACES] ;
int curr_marking [NUM_PLACES] ;
struct trans_io_struct trans_io [NUM_TRANS] ;
int trans_net_enabled [NUM_TRANS] ;
int trans_io_enabled [NUM_TRANS] ;
int rwi_status [NUM_RWI] ;
FILE *event_log_fp = NULL ;
long int start_index_pos = 0 ;
long int rec_l_pos = 0 ;
int el_start_marking [NUM_PLACES] ;
struct el_date_time_struct el_start_date_time ;
int el_start_index = 0 ;
int el_curr_index = 0 ;
int el_is_full = 0 ;

int main (void) {
    int i, num_net_enabled, num_io_enabled, pn_trans_to_fire ;

    initialise () ;

```

```

/* Forever */
while (TRUE) {

    /* Find which transitions are enabled wrt Petri Net and i/o */
    num_net_enabled = num_io_enabled = 0 ;
    for (i=0 ; i < NUM_TRANS ; i++) {
        if (trans_net_enabled [i] = is_trans_net_enabled (i)) {
            num_net_enabled++ ;
            if (trans_io_enabled [i] = is_trans_io_enabled (i))
                num_io_enabled++ ;
        }
        else
            trans_io_enabled [i] = FALSE ;
    }

    /* If there are no net transitions enabled, the system is deadlocked */
    if (num_net_enabled == 0)
        deadlock_detected () ;
    else {

        /* If one or more transitions are i/o enabled */
        if (num_io_enabled >= 1) {

            /* Select which transition to fire */
            pn_trans_to_fire = select_trans_to_fire () ;

            /* Change the net marking */
            if (fire_pn_trans (pn_trans_to_fire))

                /* And invoke the real_world_output routines to control actuators etc. */
                invoke_trans_io_real_world_outputs (pn_trans_to_fire) ;
        }
    }

    /* Call each STD's function, letting it know the current state */
    for (i=0 ; i < NUM_PLACES ; i++)
        if (curr_marking [i] > 0)
            (*place_to_std [i]) (i) ;

    /* Find out how many transitions are i/o enabled ready for calling
       the scan_complete function (it may decide to perform another iteration
       immediately if one or more transitions could fire) */
    num_net_enabled = num_io_enabled = 0 ;
    for (i=0 ; i < NUM_TRANS ; i++) {
        if (trans_net_enabled [i] = is_trans_net_enabled (i)) {
            num_net_enabled++ ;
            if (trans_io_enabled [i] = is_trans_io_enabled (i))
                num_io_enabled++ ;
        }
        else
            trans_io_enabled [i] = FALSE ;
    }

    /* Call the environment function to denote scan complete. Called function might
       want to invoke a delay or schedule a wake-up to control the iteration period to
       prevent this task from monopolising the processor */
    scan_complete (num_io_enabled) ;

} /* end forever */
return(0);
} /* end function main */

```

```

void initialise () {
    int i, j ;

    /* Initialise the environment (such as the controlled system) */
    initialise_environment () ;

    /* Initialise Petri Net to no arcs anywhere */
    for (i=0 ; i<NUM_TRANS ; i++)
        for (j=0 ; j<NUM_PLACES ; j++)
            I [i] [j] = O [i] [j] = 0 ;

    /* Initialise Petri Net arcs */
    I [ 0] [ 0] = 1 ;
    O [ 0] [ 1] = 1 ;
    O [ 0] [ 110] = 1 ;
    I [ 0] [ 111] = 1 ;
    I [ 1] [ 1] = 1 ;
    O [ 1] [ 2] = 1 ;
    I [ 1] [ 4] = 1 ;
    O [ 1] [ 5] = 1 ;
    I [ 2] [ 2] = 1 ;
    O [ 2] [ 3] = 1 ;
    O [ 2] [ 114] = 1 ;
    I [ 2] [ 115] = 1 ;
    O [ 3] [ 0] = 1 ;
    I [ 3] [ 3] = 1 ;
    O [ 3] [ 106] = 1 ;
    I [ 3] [ 107] = 1 ;
    O [ 4] [ 0] = 1 ;
    I [ 4] [ 1] = 1 ;
    O [ 4] [ 114] = 1 ;
    I [ 4] [ 115] = 1 ;
    O [ 5] [ 4] = 1 ;
    I [ 5] [ 5] = 1 ;
    I [ 5] [ 92] = 1 ;
    O [ 5] [ 93] = 1 ;
    I [ 5] [ 102] = 1 ;
    O [ 5] [ 103] = 1 ;
    I [ 5] [ 114] = 1 ;
    O [ 5] [ 115] = 1 ;
    I [ 6] [ 6] = 1 ;
    O [ 6] [ 7] = 1 ;
    O [ 6] [ 71] = 1 ;
    I [ 6] [ 72] = 1 ;
    I [ 7] [ 7] = 1 ;
    O [ 7] [ 8] = 1 ;
    I [ 7] [ 10] = 1 ;
    O [ 7] [ 11] = 1 ;
    I [ 8] [ 8] = 1 ;
    O [ 8] [ 9] = 1 ;
    O [ 8] [ 81] = 1 ;
    I [ 8] [ 82] = 1 ;
    O [ 9] [ 6] = 1 ;
    I [ 9] [ 9] = 1 ;
    I [ 9] [ 12] = 1 ;
    O [ 9] [ 13] = 1 ;
    O [ 10] [ 10] = 1 ;
    I [ 10] [ 11] = 1 ;
    I [ 10] [ 40] = 1 ;
    O [ 10] [ 41] = 1 ;
    O [ 10] [ 55] = 1 ;

```

```

I [ 10] [ 58] = 1 ;
I [ 10] [ 67] = 1 ;
O [ 10] [ 68] = 1 ;
O [ 11] [ 10] = 1 ;
I [ 11] [ 11] = 1 ;
I [ 11] [ 35] = 1 ;
O [ 11] [ 36] = 1 ;
O [ 11] [ 45] = 1 ;
I [ 11] [ 47] = 1 ;
I [ 11] [ 67] = 1 ;
O [ 11] [ 68] = 1 ;
O [ 12] [ 12] = 1 ;
I [ 12] [ 13] = 1 ;
O [ 12] [ 20] = 1 ;
I [ 12] [ 21] = 1 ;
I [ 12] [ 44] = 1 ;
O [ 12] [ 44] = 1 ;
I [ 12] [ 63] = 1 ;
O [ 12] [ 64] = 1 ;
I [ 12] [ 75] = 1 ;
O [ 12] [ 76] = 1 ;
O [ 13] [ 12] = 1 ;
I [ 13] [ 13] = 1 ;
O [ 13] [ 20] = 1 ;
I [ 13] [ 21] = 1 ;
I [ 13] [ 39] = 1 ;
O [ 13] [ 39] = 1 ;
I [ 13] [ 53] = 1 ;
O [ 13] [ 54] = 1 ;
I [ 13] [ 75] = 1 ;
O [ 13] [ 76] = 1 ;
I [ 14] [ 14] = 1 ;
O [ 14] [ 15] = 1 ;
O [ 14] [ 79] = 1 ;
I [ 14] [ 80] = 1 ;
I [ 15] [ 15] = 1 ;
O [ 15] [ 16] = 1 ;
I [ 15] [ 18] = 1 ;
O [ 15] [ 19] = 1 ;
I [ 16] [ 16] = 1 ;
O [ 16] [ 17] = 1 ;
O [ 16] [ 83] = 1 ;
I [ 16] [ 84] = 1 ;
O [ 17] [ 14] = 1 ;
I [ 17] [ 17] = 1 ;
I [ 17] [ 20] = 1 ;
O [ 17] [ 21] = 1 ;
O [ 18] [ 18] = 1 ;
I [ 18] [ 19] = 1 ;
I [ 18] [ 42] = 1 ;
O [ 18] [ 43] = 1 ;
O [ 18] [ 59] = 1 ;
I [ 18] [ 61] = 1 ;
I [ 18] [ 69] = 1 ;
O [ 18] [ 70] = 1 ;
O [ 19] [ 18] = 1 ;
I [ 19] [ 19] = 1 ;
I [ 19] [ 35] = 1 ;
O [ 19] [ 35] = 1 ;
I [ 19] [ 46] = 1 ;
O [ 19] [ 47] = 1 ;
I [ 19] [ 71] = 1 ;
O [ 19] [ 72] = 1 ;
O [ 20] [ 18] = 1 ;

```

```

I [ 20] [ 19] = 1 ;
I [ 20] [ 37] = 1 ;
O [ 20] [ 38] = 1 ;
O [ 20] [ 48] = 1 ;
I [ 20] [ 51] = 1 ;
I [ 20] [ 69] = 1 ;
O [ 20] [ 70] = 1 ;
O [ 21] [ 18] = 1 ;
I [ 21] [ 19] = 1 ;
I [ 21] [ 40] = 1 ;
O [ 21] [ 40] = 1 ;
I [ 21] [ 57] = 1 ;
O [ 21] [ 58] = 1 ;
I [ 21] [ 71] = 1 ;
O [ 21] [ 72] = 1 ;
O [ 22] [ 20] = 1 ;
I [ 22] [ 21] = 1 ;
I [ 22] [ 37] = 1 ;
O [ 22] [ 37] = 1 ;
I [ 22] [ 49] = 1 ;
O [ 22] [ 50] = 1 ;
I [ 22] [ 77] = 1 ;
O [ 22] [ 78] = 1 ;
O [ 23] [ 20] = 1 ;
I [ 23] [ 21] = 1 ;
O [ 23] [ 28] = 1 ;
I [ 23] [ 29] = 1 ;
I [ 23] [ 42] = 1 ;
O [ 23] [ 42] = 1 ;
I [ 23] [ 60] = 1 ;
O [ 23] [ 61] = 1 ;
I [ 23] [ 79] = 1 ;
O [ 23] [ 80] = 1 ;
I [ 24] [ 22] = 1 ;
O [ 24] [ 23] = 1 ;
O [ 24] [ 77] = 1 ;
I [ 24] [ 78] = 1 ;
I [ 25] [ 22] = 1 ;
O [ 25] [ 26] = 1 ;
O [ 25] [ 73] = 1 ;
I [ 25] [ 74] = 1 ;
I [ 26] [ 23] = 1 ;
O [ 26] [ 24] = 1 ;
I [ 26] [ 32] = 1 ;
O [ 26] [ 33] = 1 ;
I [ 27] [ 24] = 1 ;
O [ 27] [ 25] = 1 ;
O [ 27] [ 75] = 1 ;
I [ 27] [ 76] = 1 ;
I [ 28] [ 26] = 1 ;
O [ 28] [ 27] = 1 ;
I [ 28] [ 28] = 1 ;
O [ 28] [ 29] = 1 ;
O [ 29] [ 25] = 1 ;
I [ 29] [ 27] = 1 ;
O [ 29] [ 75] = 1 ;
I [ 29] [ 76] = 1 ;
I [ 30] [ 24] = 1 ;
O [ 30] [ 26] = 1 ;
O [ 30] [ 73] = 1 ;
I [ 30] [ 74] = 1 ;
O [ 31] [ 22] = 1 ;
I [ 31] [ 25] = 1 ;
I [ 31] [ 30] = 1 ;

```

```

O [ 31] [ 31] = 1 ;
O [ 32] [ 30] = 1 ;
I [ 32] [ 31] = 1 ;
O [ 32] [ 34] = 1 ;
I [ 32] [ 44] = 1 ;
O [ 32] [ 62] = 1 ;
I [ 32] [ 64] = 1 ;
I [ 32] [ 65] = 1 ;
O [ 32] [ 66] = 1 ;
O [ 33] [ 30] = 1 ;
I [ 33] [ 31] = 1 ;
O [ 33] [ 34] = 1 ;
I [ 33] [ 39] = 1 ;
O [ 33] [ 52] = 1 ;
I [ 33] [ 54] = 1 ;
I [ 33] [ 65] = 1 ;
O [ 33] [ 66] = 1 ;
O [ 34] [ 32] = 1 ;
I [ 34] [ 33] = 1 ;
I [ 34] [ 40] = 1 ;
O [ 34] [ 40] = 1 ;
I [ 34] [ 56] = 1 ;
O [ 34] [ 57] = 1 ;
I [ 34] [ 79] = 1 ;
O [ 34] [ 80] = 1 ;
O [ 35] [ 32] = 1 ;
I [ 35] [ 33] = 1 ;
I [ 35] [ 37] = 1 ;
O [ 35] [ 37] = 1 ;
I [ 35] [ 50] = 1 ;
O [ 35] [ 51] = 1 ;
I [ 35] [ 79] = 1 ;
O [ 35] [ 80] = 1 ;
I [ 36] [ 34] = 1 ;
O [ 36] [ 35] = 1 ;
I [ 36] [ 45] = 1 ;
O [ 36] [ 46] = 1 ;
I [ 36] [ 79] = 1 ;
O [ 36] [ 80] = 1 ;
O [ 36] [ 102] = 1 ;
I [ 36] [ 103] = 1 ;
I [ 37] [ 36] = 1 ;
O [ 37] [ 37] = 1 ;
I [ 37] [ 48] = 1 ;
O [ 37] [ 49] = 1 ;
I [ 37] [ 83] = 1 ;
O [ 37] [ 84] = 1 ;
O [ 37] [ 118] = 1 ;
I [ 37] [ 119] = 1 ;
I [ 38] [ 38] = 1 ;
O [ 38] [ 39] = 1 ;
I [ 38] [ 52] = 1 ;
O [ 38] [ 53] = 1 ;
I [ 38] [ 81] = 1 ;
O [ 38] [ 82] = 1 ;
I [ 38] [ 83] = 1 ;
O [ 38] [ 84] = 1 ;
O [ 38] [ 108] = 1 ;
I [ 38] [ 109] = 1 ;
I [ 39] [ 34] = 1 ;
O [ 39] [ 40] = 1 ;
I [ 39] [ 55] = 1 ;
O [ 39] [ 56] = 1 ;
I [ 39] [ 77] = 1 ;

```

```

O [ 39] [ 78] = 1 ;
O [ 39] [ 100] = 1 ;
I [ 39] [ 101] = 1 ;
I [ 40] [ 41] = 1 ;
O [ 40] [ 42] = 1 ;
I [ 40] [ 59] = 1 ;
O [ 40] [ 60] = 1 ;
I [ 40] [ 73] = 1 ;
O [ 40] [ 74] = 1 ;
I [ 40] [ 83] = 1 ;
O [ 40] [ 84] = 1 ;
O [ 40] [ 116] = 1 ;
I [ 40] [ 117] = 1 ;
I [ 41] [ 43] = 1 ;
O [ 41] [ 44] = 1 ;
I [ 41] [ 62] = 1 ;
O [ 41] [ 63] = 1 ;
I [ 41] [ 81] = 1 ;
O [ 41] [ 82] = 1 ;
I [ 41] [ 83] = 1 ;
O [ 41] [ 84] = 1 ;
O [ 41] [ 108] = 1 ;
I [ 41] [ 109] = 1 ;
O [ 42] [ 65] = 1 ;
I [ 42] [ 66] = 1 ;
O [ 42] [ 92] = 1 ;
I [ 42] [ 98] = 1 ;
O [ 43] [ 65] = 1 ;
I [ 43] [ 66] = 1 ;
O [ 43] [ 92] = 1 ;
I [ 43] [ 95] = 1 ;
O [ 44] [ 67] = 1 ;
I [ 44] [ 68] = 1 ;
I [ 44] [ 96] = 1 ;
O [ 44] [ 97] = 1 ;
I [ 44] [ 104] = 1 ;
O [ 44] [ 105] = 1 ;
I [ 44] [ 116] = 1 ;
O [ 44] [ 117] = 1 ;
O [ 45] [ 67] = 1 ;
I [ 45] [ 68] = 1 ;
I [ 45] [ 93] = 1 ;
O [ 45] [ 94] = 1 ;
I [ 45] [ 106] = 1 ;
O [ 45] [ 107] = 1 ;
I [ 45] [ 110] = 1 ;
O [ 45] [ 111] = 1 ;
I [ 45] [ 118] = 1 ;
O [ 45] [ 119] = 1 ;
O [ 46] [ 69] = 1 ;
I [ 46] [ 70] = 1 ;
I [ 46] [ 97] = 1 ;
O [ 46] [ 98] = 1 ;
I [ 46] [ 108] = 1 ;
O [ 46] [ 109] = 1 ;
O [ 47] [ 69] = 1 ;
I [ 47] [ 70] = 1 ;
I [ 47] [ 85] = 1 ;
O [ 47] [ 86] = 1 ;
I [ 47] [ 94] = 1 ;
O [ 47] [ 95] = 1 ;
I [ 47] [ 108] = 1 ;
O [ 47] [ 109] = 1 ;
I [ 47] [ 112] = 1 ;

```

```

O [ 47] [ 113] = 1 ;
I [ 48] [ 86] = 1 ;
O [ 48] [ 87] = 1 ;
O [ 48] [ 112] = 1 ;
I [ 48] [ 113] = 1 ;
I [ 49] [ 87] = 1 ;
O [ 49] [ 88] = 1 ;
I [ 50] [ 88] = 1 ;
O [ 50] [ 89] = 1 ;
I [ 50] [ 90] = 1 ;
O [ 50] [ 91] = 1 ;
O [ 51] [ 85] = 1 ;
I [ 51] [ 89] = 1 ;
O [ 51] [ 104] = 1 ;
I [ 51] [ 105] = 1 ;
O [ 52] [ 90] = 1 ;
I [ 52] [ 91] = 1 ;
I [ 52] [ 92] = 1 ;
O [ 52] [ 96] = 1 ;
I [ 52] [ 100] = 1 ;
O [ 52] [ 101] = 1 ;
O [ 53] [ 92] = 1 ;
I [ 53] [ 99] = 1 ;
I [ 53] [ 110] = 1 ;
O [ 53] [ 111] = 1 ;

/* Set up current net marking to be the initial marking */
for (i=0 ; i<NUM_PLACES ; i++)
    curr_marking [i] = 0 ;

curr_marking [ 0] = 1 ;
curr_marking [ 4] = 1 ;
curr_marking [ 6] = 1 ;
curr_marking [ 10] = 1 ;
curr_marking [ 12] = 1 ;
curr_marking [ 14] = 1 ;
curr_marking [ 18] = 1 ;
curr_marking [ 20] = 1 ;
curr_marking [ 22] = 1 ;
curr_marking [ 28] = 1 ;
curr_marking [ 30] = 1 ;
curr_marking [ 32] = 1 ;
curr_marking [ 34] = 1 ;
curr_marking [ 45] = 1 ;
curr_marking [ 48] = 1 ;
curr_marking [ 52] = 1 ;
curr_marking [ 55] = 1 ;
curr_marking [ 59] = 1 ;
curr_marking [ 62] = 1 ;
curr_marking [ 65] = 1 ;
curr_marking [ 67] = 1 ;
curr_marking [ 69] = 1 ;
curr_marking [ 71] = 1 ;
curr_marking [ 73] = 1 ;
curr_marking [ 75] = 1 ;
curr_marking [ 77] = 1 ;
curr_marking [ 79] = 1 ;
curr_marking [ 81] = 1 ;
curr_marking [ 83] = 1 ;
curr_marking [ 85] = 1 ;
curr_marking [ 90] = 1 ;
curr_marking [ 99] = 1 ;
curr_marking [ 100] = 1 ;
curr_marking [ 102] = 1 ;

```



```

curr_marking [ 104] = 1 ;
curr_marking [ 106] = 1 ;
curr_marking [ 108] = 1 ;
curr_marking [ 110] = 1 ;
curr_marking [ 112] = 1 ;
curr_marking [ 114] = 1 ;
curr_marking [ 116] = 1 ;
curr_marking [ 118] = 1 ;

/* Initialise lists of io routines */
trans_io [ 0].num_real_world_input_routines = 0 ;
trans_io [ 0].num_real_world_output_routines = 1 ;
trans_io [ 0].rwo_index [ 0] = 0 ;
trans_io [ 1].num_real_world_input_routines = 1 ;
trans_io [ 1].rwi_index [ 0] = 0 ;
trans_io [ 1].num_real_world_output_routines = 0 ;
trans_io [ 2].num_real_world_input_routines = 0 ;
trans_io [ 2].num_real_world_output_routines = 1 ;
trans_io [ 2].rwo_index [ 0] = 1 ;
trans_io [ 3].num_real_world_input_routines = 0 ;
trans_io [ 3].num_real_world_output_routines = 0 ;
trans_io [ 4].num_real_world_input_routines = 0 ;
trans_io [ 4].num_real_world_output_routines = 0 ;
trans_io [ 5].num_real_world_input_routines = 0 ;
trans_io [ 5].num_real_world_output_routines = 0 ;
trans_io [ 6].num_real_world_input_routines = 0 ;
trans_io [ 6].num_real_world_output_routines = 1 ;
trans_io [ 6].rwo_index [ 0] = 2 ;
trans_io [ 7].num_real_world_input_routines = 1 ;
trans_io [ 7].rwi_index [ 0] = 1 ;
trans_io [ 7].num_real_world_output_routines = 0 ;
trans_io [ 8].num_real_world_input_routines = 0 ;
trans_io [ 8].num_real_world_output_routines = 1 ;
trans_io [ 8].rwo_index [ 0] = 3 ;
trans_io [ 9].num_real_world_input_routines = 1 ;
trans_io [ 9].rwi_index [ 0] = 2 ;
trans_io [ 9].num_real_world_output_routines = 0 ;
trans_io [ 10].num_real_world_input_routines = 0 ;
trans_io [ 10].num_real_world_output_routines = 0 ;
trans_io [ 11].num_real_world_input_routines = 0 ;
trans_io [ 11].num_real_world_output_routines = 0 ;
trans_io [ 12].num_real_world_input_routines = 0 ;
trans_io [ 12].num_real_world_output_routines = 0 ;
trans_io [ 13].num_real_world_input_routines = 0 ;
trans_io [ 13].num_real_world_output_routines = 0 ;
trans_io [ 14].num_real_world_input_routines = 0 ;
trans_io [ 14].num_real_world_output_routines = 1 ;
trans_io [ 14].rwo_index [ 0] = 4 ;
trans_io [ 15].num_real_world_input_routines = 1 ;
trans_io [ 15].rwi_index [ 0] = 3 ;
trans_io [ 15].num_real_world_output_routines = 0 ;
trans_io [ 16].num_real_world_input_routines = 0 ;
trans_io [ 16].num_real_world_output_routines = 1 ;
trans_io [ 16].rwo_index [ 0] = 5 ;
trans_io [ 17].num_real_world_input_routines = 1 ;
trans_io [ 17].rwi_index [ 0] = 4 ;
trans_io [ 17].num_real_world_output_routines = 0 ;
trans_io [ 18].num_real_world_input_routines = 0 ;
trans_io [ 18].num_real_world_output_routines = 0 ;
trans_io [ 19].num_real_world_input_routines = 0 ;
trans_io [ 19].num_real_world_output_routines = 0 ;
trans_io [ 20].num_real_world_input_routines = 0 ;
trans_io [ 20].num_real_world_output_routines = 0 ;
trans_io [ 21].num_real_world_input_routines = 0 ;

```

[illegible]

```

trans_io [ 49].num_real_world_input_routines = 1 ;
trans_io [ 49].rwi_index [ 0] = 8 ;
trans_io [ 49].num_real_world_output_routines = 1 ;
trans_io [ 49].rwo_index [ 0] = 10 ;
trans_io [ 50].num_real_world_input_routines = 1 ;
trans_io [ 50].rwi_index [ 0] = 9 ;
trans_io [ 50].num_real_world_output_routines = 0 ;
trans_io [ 51].num_real_world_input_routines = 0 ;
trans_io [ 51].num_real_world_output_routines = 0 ;
trans_io [ 52].num_real_world_input_routines = 0 ;
trans_io [ 52].num_real_world_output_routines = 0 ;
trans_io [ 53].num_real_world_input_routines = 1 ;
trans_io [ 53].rwi_index [ 0] = 10 ;
trans_io [ 53].num_real_world_output_routines = 0 ;

for (i=0 ; i < NUM_RWI ; i++)
    rwi_status [i] = RWI_NOT_TESTED ;

event_log_fp = fopen ("synect.cel", "wt+" ) ;
fprintf (event_log_fp, "%4.4x\n%4.4x\n%4.4x\n%4.4x\n",
        NUM_PLACES, NUM_TRANS, NUM_RWI, EVENT_LOG_CAPACITY) ;

start_index_pos = ftell (event_log_fp) ;

/* Start index, curr index, is full */
fprintf (event_log_fp, "%4.4x\n%4.4x\n%4.4x\n", 0, 0, 0) ;

for (i=0 ; i < NUM_PLACES ; i++) {
    el_start_marking [i] = curr_marking [i] ;
    fprintf (event_log_fp, "%1.1d ", el_start_marking [i]) ;
}
fprintf (event_log_fp, "\n") ;

el_start_date_time = get_current_date_time () ;
fprintf (event_log_fp, "%4.4d\n%2.2d\n%2.2d\n%2.2d\n%2.2d\n%2.2d\n",
        el_start_date_time.year,
        el_start_date_time.day,
        el_start_date_time.month,
        el_start_date_time.min,
        el_start_date_time.hour,
        el_start_date_time.hund,
        el_start_date_time.secs) ;

rec_1_pos = ftell (event_log_fp) ;
} /* end function initialise */

int is_trans_net_enabled (int pn_trans_no) {
    int enabled = TRUE ;
    int place_no ;

    for (place_no=0 ; ((place_no<NUM_PLACES) && (enabled)) ; place_no++)
        if ((I [pn_trans_no] [place_no] != 0) && (curr_marking [place_no] < I
[pn_trans_no] [place_no]))
            enabled = FALSE ;

    return (enabled) ;
} /* end function is_trans_net_enabled */

int is_trans_io_enabled (int pn_trans_no) {
    int enabled = TRUE ;

```

```

int i ;

for (i=0 ; ((i<trans_io [pn_trans_no].num_real_world_input_routines) && (enabled)) ;
i++) {
    int index = trans_io [pn_trans_no].rwi_index [i] ;
    int now_enabled = (*real_world_input_routine [index]) () ;
    int this_rwi_status = (now_enabled) ? RWI_ENABLED : RWI_DISABLED ;

    if (!now_enabled)
        enabled = FALSE ;

    if (rwi_status [index] != this_rwi_status) {
        rwi_status [index] = this_rwi_status ;
        save_rwi_change (index, now_enabled) ;
    }

}

return (enabled) ;
} /* end function is_trans_io_enabled */


int select_trans_to_fire () {
    static int prev_trans_fired = -1 ;
    int i, start_trans ;

    /* In future, this may select transition based on least recently fired,
       least frequently fired or randomly. This release simply chooses the
       first one it finds after the previous transition fired */
    start_trans = prev_trans_fired + 1 ;
    for (i=start_trans ; i<NUM_TRANS ; i++)
        if (trans_io_enabled [i])
            return (prev_trans_fired = i) ;

    for (i=0 ; i<start_trans ; i++)
        if (trans_io_enabled [i])
            return (prev_trans_fired = i) ;

    /* If control reaches this point, there is a major error in this software.
       In a future release, some exception mechanism might be invoked, but for now,
       just return an index which will cause (hopefully) a memory access violation. */
    return (-1) ;
} /* end function select_trans_to_fire */


int fire_pn_trans (int pn_trans_no) {
    int place_no ;
    int i ;

    for (place_no=0 ; place_no<NUM_PLACES ; place_no++)
        curr_marking [place_no] = curr_marking [place_no] - I [pn_trans_no] [place_no] + O
        [pn_trans_no] [place_no] ;

    save_pn_trans_fired (pn_trans_no) ;

    /* Reset rwi statuses to unknown */
    for (i=0 ; i < NUM_RWI ; i++)
        rwi_status [i] = RWI_NOT_TESTED ;

    return (TRUE); /* The transition has been fired */
} /* end function fire_pn_trans */

```

```

void invoke_trans_io_real_world_outputs (int pn_trans_no) {
    int i ;

    for (i=0 ; i<trans_io [pn_trans_no].num_real_world_output_routines ; i++) {
        int index = trans_io [pn_trans_no].rwo_index [i] ;
        (*real_world_output_routine [index]) () ;
    }
} /* end function invoke_trans_io_real_world_outputs */


void save_rwi_change (int index, int status) {
    write_to_event_log (0, index, status) ;
}

void save_pn_trans_fired (int pn_trans_no) {
    write_to_event_log (1, pn_trans_no, 0) ;
}

void write_to_event_log (int whether_trans_fired_rec, int value1, int value2)
{
    int i ;
    struct el_date_time_struct curr_date_time ;
    long int curr_pos = 0 ;
    int num_items_read = 0 ;
    if (el_curr_index < EVENT_LOG_CAPACITY)
        el_curr_index ++ ;
    else
    {
        el_curr_index = 1 ;
        el_is_full = TRUE ;
        /* Move the file pointer to the start of the first record */
        fseek (event_log_fp, rec_1_pos, SEEK_SET) ;
    }
    /* If this is the first record ever written */
    if (el_start_index == 0)
        el_start_index = 1 ;
    /* Remember where we're to start writing */
    curr_pos = ftell (event_log_fp) ;
    /* If the event log is full such that we're overwriting existing records, adjust index
of start record */
    if (el_is_full)
    {
        el_start_index = (el_start_index < EVENT_LOG_CAPACITY) ? el_start_index+1 : 1 ;
        /* Get the date and time and marking from the record which we're going to
overwrite */
        num_items_read = fscanf (event_log_fp, "%d\n%d\n%d\n%d\n%d\n%d\n%d\n",
                                &el_start_date_time.year,
                                &el_start_date_time.day,
                                &el_start_date_time.month,
                                &el_start_date_time.min,
                                &el_start_date_time.hour,
                                &el_start_date_time.hund,
                                &el_start_date_time.secs) ;
        if (num_items_read != 7)
            event_log_error (1) ;
    }
}

```

```

        fscanf (event_log_fp, "%d\n%x\n%d\n");          /* Type of info in record */
        for (i=0 ; i < NUM_PLACES ; i++)
        {
            num_items_read = fscanf (event_log_fp, "%d ", &el_start_marking [i]) ;
            if (num_items_read != 1)
                event_log_error (2) ;
        }
    }
    /* Now update the index info in the event log */
    fseek (event_log_fp, start_index_pos, SEEK_SET) ;
    /* Start index, curr index, is full */
    fprintf (event_log_fp, "%4.4x\n%4.4x\n%4.4x\n",    el_start_index,    el_curr_index,
    el_is_full) ;
    for (i=0 ; i < NUM_PLACES ; i++)
        fprintf (event_log_fp, "%1.1d ", el_start_marking [i]) ;
    fprintf (event_log_fp, "\n") ;
    fprintf (event_log_fp, "%4.4d\n%2.2d\n%2.2d\n%2.2d\n%2.2d\n%2.2d\n",
        el_start_date_time.year,
        el_start_date_time.day,
        el_start_date_time.month,
        el_start_date_time.min,
        el_start_date_time.hour,
        el_start_date_time.hund,
        el_start_date_time.secs) ;
    /* Set the file pointer back to where we're to start writing */
    fseek (event_log_fp, curr_pos, SEEK_SET) ;
    /* Get the current date and time */
    curr_date_time = get_current_date_time () ;
    fprintf (event_log_fp, "%4.4d\n%2.2d\n%2.2d\n%2.2d\n%2.2d\n%2.2d\n",
        curr_date_time.year,
        curr_date_time.day,
        curr_date_time.month,
        curr_date_time.min,
        curr_date_time.hour,
        curr_date_time.hund,
        curr_date_time.secs) ;
    fprintf (event_log_fp, "%1.1d\n%4.4x\n%1.1d\n",    whether_trans_fired_rec,    value1,
    value2) ;
    for (i=0 ; i < NUM_PLACES ; i++)
        fprintf (event_log_fp, "%1.1d ", curr_marking [i]) ;
    fprintf (event_log_fp, "\n") ;
    /* Very crudely, force a flush */
    curr_pos = ftell (event_log_fp) ;
    fclose (event_log_fp) ;
    event_log_fp = fopen ("synect.cel", "rt+") ;
    fseek (event_log_fp, curr_pos, SEEK_SET) ;
}

/* End of control program generated by Synect Code Generator V1.8 */

```

A.4.9.5.2 Demo.env

```
/*
Environment-dependent C source file corresponding to C:\SYNECT\DEMO.SYN
Generated by Synect C Code Generator V1.8
Synect is a registered trademark belonging to:
    Hopkinson Computing Limited
    29 Deepdale, Pine Hills, Guisborough, Cleveland, TS14 8JY, England
    Tel/Fax: +44 (0) 1287 638606
    email: synect@hopkinsn.demon.co.uk

Generated at: Mon Aug 24 23:35:25 1998

*/

#include <dos.h>

/* Include the application-dependent defines and declarations */
#include "demo.h"
/* Skeletal function called on completion of scan */
void scan_complete (int num_io_enabled) {
} /* end function scan_complete */

/* Skeletal function to handle identification of deadlock*/
void deadlock_detected () {
} /* end deadlock real_world_input function */

/* Skeletal function to get date and time (this works for DOS!) */
struct el_date_time_struct get_current_date_time () {

    struct el_date_time_struct dt ;
    struct date cd ;
    struct time ct ;
    getdate (&cd) ;
    gettime (&ct) ;

    dt.year = cd.da_year ;
    dt.day = cd.da_day ;
    dt.month = cd.da_mon ;
    dt.min = ct.ti_min ;
    dt.hour = ct.ti_hour ;
    dt.hund = ct.ti_hund ;
    dt.secs = ct.ti_sec ;

    return (dt) ;
}

/* Skeletal function called on detection of an error when reading from event log */
void event_log_error (int error_type) {

    if (error_type == 1)
        ; /* Error reading date and time */
    else if (error_type == 2)
        ; /* Error reading marking */
}

/* End of environment-dependent (C source) file generated by Synect Code Generator V1.8
*/
```

A.4.9.5.3 Demo.h

```
/*
C include file corresponding to C:\SYNECT\DEMO.SYN

Generated by Synect C Code Generator V1.8
Synect is a registered trademark belonging to:
Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough, Cleveland, TS14 8JY, England
Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk

Generated at: Mon Aug 24 23:35:25 1998

*/

/* Application-dependent constant definitions */
#define NUM_TRANS 54
#define NUM_PLACES 120
#define REAL_WORLD_INPUT_LIST_SIZE 1
#define REAL_WORLD_OUTPUT_LIST_SIZE 1
#define NUM_RWI 11
#define EVENT_LOG_CAPACITY 50

/* Application-dependent real_world_input function declarations */
extern int rwi_new_raw_part (void) ;
extern int rwi_grip_closed (void) ;
extern int rwi_grip_open (void) ;
extern int rwi_arm_down (void) ;
extern int rwi_arm_up (void) ;
extern int rwi_at_machine (void) ;
extern int rwi_at_exit (void) ;
extern int rwi_at_feed (void) ;
extern int rwi_finished_phase_1 (void) ;
extern int rwi_finished_phase_2 (void) ;
extern int rwi_go (void) ;

/* Application-dependent real_world_output function declarations */
extern void rwo_start_feed_conveyor (void) ;
extern void rwo_stop_feed_conveyor (void) ;
extern void rwo_close_grip (void) ;
extern void rwo_open_grip (void) ;
extern void rwo_lower_arm (void) ;
extern void rwo_raise_arm (void) ;
extern void rwo_goto_machine (void) ;
extern void rwo_goto_exit (void) ;
extern void rwo_goto_feed (void) ;
extern void rwo_start_phase_1 (void) ;
extern void rwo_start_phase_2 (void) ;

/* Application-independent structure declarations */
struct el_date_time_struct {
    int year ;
    int day ;
    int month ;
    int min ;
    int hour ;
    int hund ;
    int secs ;
} ;

/* End of include file generated by Synect Code Generator V1.8 */
```


A.4.9.5.4 Demo.rwi

```
/*
Real World Inputs (C source) file corresponding to C:\SYNECT\DEMO.SYN
Generated by Synect C Code Generator V1.8
Synect is a registered trademark belonging to:
Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough, Cleveland, TS14 8JY, England
Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk

Generated at: Mon Aug 24 23:35:25 1998

*/

/* Application-independent constant definitions */
#define TRUE 1
#define FALSE 0

/* Skeletal real_world_input functions */
int rwi_new_raw_part () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_grip_closed () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_grip_open () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_arm_down () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_arm_up () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_at_machine () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_at_exit () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_at_feed () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_finished_phase_1 () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_finished_phase_2 () {
    return (TRUE) ;
} /* end real_world_input function */

int rwi_go () {
    return (TRUE) ;
} /* end real_world_input function */

/* End of real world inputs (C source) file generated by Synect Code Generator V1.8 */
```

A.4.9.5.5 Demo.rwo

```
/*
Real World Outputs (C source) file corresponding to C:\SYNECT\DEMO.SYN
Generated by Synect C Code Generator V1.8
Synect is a registered trademark belonging to:
Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough, Cleveland, TS14 8JY, England
Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk

Generated at: Mon Aug 24 23:35:25 1998

*/

/* Skeletal function to initialise the environment e.g. to reset the
controlled system to a known state */
void initialise_environment () {
} /* end function initialise_environment */

/* Skeletal real_world_output functions */
void rwo_start_feed_conveyor () {
} /* end real_world_output function */

void rwo_stop_feed_conveyor () {
} /* end real_world_output function */

void rwo_close_grip () {
} /* end real_world_output function */

void rwo_open_grip () {
} /* end real_world_output function */

void rwo_lower_arm () {
} /* end real_world_output function */

void rwo_raise_arm () {
} /* end real_world_output function */

void rwo_goto_machine () {
} /* end real_world_output function */

void rwo_goto_exit () {
} /* end real_world_output function */

void rwo_goto_feed () {
} /* end real_world_output function */

void rwo_start_phase_1 () {
} /* end real_world_output function */

void rwo_start_phase_2 () {
} /* end real_world_output function */

/* End of real world outputs (C source) file generated by Synect Code Generator V1.8 */
```

A.4.9.5.6 Demo.std

```
/*
Function per STD (C source) file corresponding to C:\SYNECT\DEMO.SYN

Generated by Synect C Code Generator V1.8
Synect is a registered trademark belonging to:
Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough, Cleveland, TS14 8JY, England
Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk

Generated at: Mon Aug 24 23:35:25 1998

*/
```

```
void Feed_Conveyor_main (int place_no) {
    switch (place_no) {
        case 0: /* off|no_raw_part */
            break ;

        case 1: /* on|no_raw_part */
            break ;

        case 2: /* on|new_raw_part */
            break ;

        case 3: /* off|new_raw_part */
            break ;

        default:
            break ;
    }
}

void Feed_Conveyor_new_part_available (int place_no) {
    switch (place_no) {
        case 4: /* #not_pending */
            break ;

        case 5: /* #pending */
            break ;

        default:
            break ;
    }
}

void Gripper_main (int place_no) {
    switch (place_no) {
        case 6: /* open */
            break ;

        case 7: /* closing */
            break ;

        case 8: /* closed */
            break ;

        case 9: /* opening */
            break ;
    }
}
```

```

        default:
            break ;
    }
}

void Gripper_grip_closed (int place_no) {
    switch (place_no) {
        case 10: /* #not_pending */
            break ;

        case 11: /* #pending */
            break ;

        default:
            break ;
    }
}

void Gripper_grip_open (int place_no) {
    switch (place_no) {
        case 12: /* #not_pending */
            break ;

        case 13: /* #pending */
            break ;

        default:
            break ;
    }
}

void Arm_Elevation_main (int place_no) {
    switch (place_no) {
        case 14: /* up */
            break ;

        case 15: /* going|down */
            break ;

        case 16: /* down */
            break ;

        case 17: /* going|up */
            break ;

        default:
            break ;
    }
}

void Arm_Elevation_arm_down (int place_no) {
    switch (place_no) {
        case 18: /* #not_pending */
            break ;

        case 19: /* #pending */
            break ;

        default:
            break ;
    }
}

void Arm_Elevation_arm_up (int place_no) {

```

```

switch (place_no) {
    case 20: /* #not_pending */
        break ;

    case 21: /* #pending */
        break ;

    default:
        break ;
}

}

void Arm_Translation_main (int place_no) {
    switch (place_no) {
        case 22: /* at|feed */
            break ;

        case 23: /* going|to|machine */
            break ;

        case 24: /* at|machine */
            break ;

        case 25: /* going|to|feed */
            break ;

        case 26: /* going|to|exit */
            break ;

        case 27: /* at|exit */
            break ;

        default:
            break ;
    }
}

void Arm_Translation_at_exit (int place_no) {
    switch (place_no) {
        case 28: /* #not_pending */
            break ;

        case 29: /* #pending */
            break ;

        default:
            break ;
    }
}

void Arm_Translation_at_feed (int place_no) {
    switch (place_no) {
        case 30: /* #not_pending */
            break ;

        case 31: /* #pending */
            break ;

        default:
            break ;
    }
}

void Arm_Translation_at_machine (int place_no) {

```

```

switch (place_no) {
    case 32: /* #not_pending */
        break ;

    case 33: /* #pending */
        break ;

    default:
        break ;
}
}

void Robot_main (int place_no) {
    switch (place_no) {
        case 34: /* at|home */
            break ;

        case 35: /* getting|raw|part */
            break ;

        case 36: /* holding|raw|part */
            break ;

        case 37: /* taking|raw_part|to_machine */
            break ;

        case 38: /* at_machine|ready|to_release */
            break ;

        case 39: /* going_home|from_machine */
            break ;

        case 40: /* getting|machined|part */
            break ;

        case 41: /* holding|machined|part */
            break ;

        case 42: /* taking|machined_part|to_exit */
            break ;

        case 43: /* at_exit|ready|to_release */
            break ;

        case 44: /* going_home|from_exit */
            break ;

        default:
            break ;
    }
}

void Robot_main_getting_raw_part (int place_no) {
    switch (place_no) {
        case 45: /* START */
            break ;

        case 46: /* lowering|arm */
            break ;

        case 47: /* closing|grip */
            break ;

        default:

```

```

        break ;
    }
}

void Robot_main_taking_raw_part_to_machine (int place_no) {
    switch (place_no) {
        case 48: /* START */
            break ;

        case 49: /* raising|arm */
            break ;

        case 50: /* moving_arm|across_to|machine */
            break ;

        case 51: /* lowering|arm */
            break ;

        default:
            break ;
    }
}

void Robot_main_going_home_from_machine (int place_no) {
    switch (place_no) {
        case 52: /* START */
            break ;

        case 53: /* opening_grip|and|raising_arm */
            break ;

        case 54: /* moving_arm|across|to_feed */
            break ;

        default:
            break ;
    }
}

void Robot_main_getting_machined_part (int place_no) {
    switch (place_no) {
        case 55: /* START */
            break ;

        case 56: /* moving|across|to_machine */
            break ;

        case 57: /* lowering|arm */
            break ;

        case 58: /* closing|grip */
            break ;

        default:
            break ;
    }
}

void Robot_main_taking_machined_part_to_exit (int place_no) {
    switch (place_no) {
        case 59: /* START */
            break ;

        case 60: /* moving_arm_acoss|to_exit|and_raising_arm */

```

```

        break ;

    case 61: /* lowering|arm */
        break ;

    default:
        break ;
}
}

void Robot_main_going_home_from_exit (int place_no) {
    switch (place_no) {
        case 62: /* START */
            break ;

        case 63: /* opening_grip|and|raising_arm */
            break ;

        case 64: /* moving|arm_across|to_feed */
            break ;

        default:
            break ;
    }
}

void Robot_at_home_position (int place_no) {
    switch (place_no) {
        case 65: /* #not_pending */
            break ;

        case 66: /* #pending */
            break ;

        default:
            break ;
    }
}

void Robot_holding_part (int place_no) {
    switch (place_no) {
        case 67: /* #not_pending */
            break ;

        case 68: /* #pending */
            break ;

        default:
            break ;
    }
}

void Robot_ready_to_release_part (int place_no) {
    switch (place_no) {
        case 69: /* #not_pending */
            break ;

        case 70: /* #pending */
            break ;

        default:
            break ;
    }
}

```



```

void Robot_close_grip (int place_no) {
    switch (place_no) {
        case 71: /* #not_pending */
            break ;

        case 72: /* #pending */
            break ;

        default:
            break ;
    }
}

void Robot_goto_exit (int place_no) {
    switch (place_no) {
        case 73: /* #not_pending */
            break ;

        case 74: /* #pending */
            break ;

        default:
            break ;
    }
}

void Robot_goto_feed (int place_no) {
    switch (place_no) {
        case 75: /* #not_pending */
            break ;

        case 76: /* #pending */
            break ;

        default:
            break ;
    }
}

void Robot_goto_machine (int place_no) {
    switch (place_no) {
        case 77: /* #not_pending */
            break ;

        case 78: /* #pending */
            break ;

        default:
            break ;
    }
}

void Robot_lower_arm (int place_no) {
    switch (place_no) {
        case 79: /* #not_pending */
            break ;

        case 80: /* #pending */
            break ;

        default:
            break ;
    }
}

```

```

}

void Robot_open_grip (int place_no) {
    switch (place_no) {
        case 81: /* #not_pending */
            break ;

        case 82: /* #pending */
            break ;

        default:
            break ;
    }
}

void Robot_raise_arm (int place_no) {
    switch (place_no) {
        case 83: /* #not_pending */
            break ;

        case 84: /* #pending */
            break ;

        default:
            break ;
    }
}

void Machine_main (int place_no) {
    switch (place_no) {
        case 85: /* off|no_part */
            break ;

        case 86: /* off|raw_part_loaded */
            break ;

        case 87: /* on|phase_1 */
            break ;

        case 88: /* on|phase_2 */
            break ;

        case 89: /* off|finished_part|available */
            break ;

        default:
            break ;
    }
}

void Machine_finished_part_available (int place_no) {
    switch (place_no) {
        case 90: /* #not_pending */
            break ;

        case 91: /* #pending */
            break ;

        default:
            break ;
    }
}

void Assembly_Cell_main (int place_no) {

```

```

switch (place_no) {
    case 92: /* Idle */
        break ;

    case 93: /* getting|raw|part */
        break ;

    case 94: /* taking|raw_part|to_machine */
        break ;

    case 95: /* machining|part */
        break ;

    case 96: /* getting|machined|part */
        break ;

    case 97: /* taking|machined_part|to_exit */
        break ;

    case 98: /* returning|home */
        break ;

    case 99: /* Initialised */
        break ;

    default:
        break ;
}
}

void Assembly_Cell_get_machined_part (int place_no) {
    switch (place_no) {
        case 100: /* #not_pending */
            break ;

        case 101: /* #pending */
            break ;

        default:
            break ;
    }
}

void Assembly_Cell_get_raw_part (int place_no) {
    switch (place_no) {
        case 102: /* #not_pending */
            break ;

        case 103: /* #pending */
            break ;

        default:
            break ;
    }
}

void Assembly_Cell_part_removed (int place_no) {
    switch (place_no) {
        case 104: /* #not_pending */
            break ;

        case 105: /* #pending */
            break ;
    }
}

```

```

        default:
            break ;
    }
}

void Assembly_Cell_raw_part_removed (int place_no) {
    switch (place_no) {
        case 106: /* #not_pending */
            break ;

        case 107: /* #pending */
            break ;

        default:
            break ;
    }
}

void Assembly_Cell_release_part_and_go_home (int place_no) {
    switch (place_no) {
        case 108: /* #not_pending */
            break ;

        case 109: /* #pending */
            break ;

        default:
            break ;
    }
}

void Assembly_Cell_start_feed (int place_no) {
    switch (place_no) {
        case 110: /* #not_pending */
            break ;

        case 111: /* #pending */
            break ;

        default:
            break ;
    }
}

void Assembly_Cell_start_machine (int place_no) {
    switch (place_no) {
        case 112: /* #not_pending */
            break ;

        case 113: /* #pending */
            break ;

        default:
            break ;
    }
}

void Assembly_Cell_stop_feed (int place_no) {
    switch (place_no) {
        case 114: /* #not_pending */
            break ;

        case 115: /* #pending */
            break ;
    }
}

```

```

        default:
            break ;
    }
}

void Assembly_Cell_take_machined_part_to_exit (int place_no) {
    switch (place_no) {
        case 116: /* #not_pending */
            break ;

        case 117: /* #pending */
            break ;

        default:
            break ;
    }
}

void Assembly_Cell_take_raw_part_to_machine (int place_no) {
    switch (place_no) {
        case 118: /* #not_pending */
            break ;

        case 119: /* #pending */
            break ;

        default:
            break ;
    }
}

```

/* End of function per STD (C source) file generated by Synect Code Generator V1.8 */

Appendix B Copies of Published Papers

This appendix contains copies of the following papers:

Harrison, R. et al., "Improving Manufacturing Automation By The Integration Of Machine Design And Control", 26th International Symposium on Industrial Robots, Singapore, Oct. 1995, pp 51-56, ISBN 1-86058-000-9

Harrison, R. et al., "Interactive Visualisation Of Sequence Logic And Physical Machine Components Within An Integrated Design And Control Environment", 4th IFAC Workshop on International Manufacturing Systems, July 1997, Seoul, Korea

Hopkinson, P. et al., "Implementing S88 Batch Control Systems In The Pharmaceutical Industry", Measurement And Control, Vol. 31, February 1998, pp 20-24

Hopkinson, P. and Hancock, J., "A Case History Of The Implementation Of An S88-Aware Batch Control System", World Batch Forum, 1998

Haxthausen, N. and Hopkinson, P., "The Application Of The S88 Batch Control Standard In The Pharmaceutical Industry", Computer Systems For The New Millennium Conference, International Society Of Pharmaceutical Engineers, 4th - 5th March 1998, Amsterdam

IMPROVING MANUFACTURING AUTOMATION BY THE INTEGRATION OF MACHINE DESIGN AND CONTROL

R. Harrison*, C. D. Wright* and P. Hopkinson**

*MSI Research Institute, Loughborough University of Technology,
Loughborough, LE11 3TU, UK

**Hopkinson Computing Ltd, 29 Deepdale, Pine Hills, Guisborough,
Cleveland, TS14 8JY, UK

R.Harrison@lut.ac.uk, C.D.Wright@lut.ac.uk & P.Hopkinson@lut.ac.uk

Abstract

The use of computer controlled machines for manufacturing automation is now commonplace. Current approaches to the implementation of these machines are often characterised by poor verification of customer requirements, limited confidence in proposed designs, minimal software re-use and both time consuming and costly system maintenance/enhancement. In many cases, machines are so difficult and costly to modify that complete machine rebuild is necessary to accommodate relatively minor product changes. This paper outlines a new approach to machine lifecycle support, an "Integrated Machine Design and Control (IMDC)" environment, aimed at overcoming these problems. The IMDC environment has the potential to radically improve the effectiveness of machine and associated control system design/build and to enable efficient modification as requirements change. It also allows the integration of design and control system elements from a wide range of vendors. IMDC is a three year EPSRC (Engineering and Physical Sciences Research Council) funded collaborative project in the UK, between the Manufacturing Systems Integration (MSI) Research Institute and a consortium of industrial control system suppliers and machine builders.

The paper provides an overview of the concepts behind IMDC and its implementation. From a user's perspective, the two main elements of the IMDC environment are: 1) a software toolset and 2) a run-time control architecture. The software toolset covers the life cycle of manufacturing machines and supports the creation of application software for the target control architecture. Underlying these elements is the IMDC system software which integrates and manages the user toolset and links it to the run-time environment. Physically the IMDC environment utilises a network of one or more workstations or personal computers coupled to an embedded real-time control architecture which resides on each target machine.

The extendible toolset is composed of machine/control system design, configuration and management tools

which can be utilised at various phases of the machine life-cycle. These include application logic description and analysis, machine modelling, automatic code generation and run-time control. IMDC thus seeks to provide a highly integrated environment for system builders, providing much needed support for rapid prototyping, "what-if" analyses and enables machines to be incrementally enhanced. The paper considers the use of example design tools within the toolset and contrasts this new approach with the use of traditional methods.

The IMDC run-time architecture is based on the UMC (Universal Machine Control) methodology and software implementation which has been developed at the MSI Research Institute over the last eight years. UMC provides the basis for an open, structured, device independent method for building machine control systems which is now seeing industrial exploitation. The IMDC project is extending the UMC concept and software to support a physically distributed runtime environment. This enables control systems to be composed of intelligent devices, physically located at the locations in the machine where the control functionality is needed. Profibus has been selected as the main real-time control network although future implementations could be based on alternative fieldbuses or control networks. Other MSI projects are currently utilising CAN, FIP and Lonworks.

Introduction

Limitations of machine control systems

In manufacturing industry, shortening product life cycles combined with the need to offer increased product variety have made it necessary for machines to provide greater functionality, flexibility and reconfigurability. These factors have caused a steady growth in the use of computer controlled machinery over the last twenty years. They have also led to increasing awareness of the deficiencies in current approaches to the implementation of machine control systems. Much more is now demanded from control systems and as computer control systems have grown in complexity they have become

ch more difficult to design, build, operate, maintain
l modify.

lustrial automation spans a huge spectrum of com-
xity in terms of both the physical structures of ma-
nes and the tasks that they perform. This has led to
equally wide range of control system hardware and
ftware building blocks. The general lack of stan-
rdisation between different control system components
akes industrial control systems difficult to maintain,
odify and integrate. This has encouraged users to go to
single vendor for all their machine control needs in
der to minimise such problems. The dominance of
used, vendor specific solutions have generally resulted
stagnation rather than innovation and improvement
control systems.

ne machines themselves consist of suitable combina-
ons of mechatronics (mechanical and control system)
ements, with application specific code determining
ow a particular machine behaves. The inability of cur-
nt methods to efficiently cater for the visualisation of
ese mechatronic elements and the design of associated
pplication specific software, particularly as systems
crease in complexity, is seen in spiralling applications
osts as outlined by [1]. As summarised in figure 1,
onventional practise typically involves the use of a
umber of incompatible, vendor specific tools, each ca-
ring for a limited aspect of machine or control system
esign. As a result the designer is unable to adequately
isualise the application as a whole and often design
laws go undetected until a machine is commissioned.
The adoption of better integrated design approaches is
currently severely handicapped by a lack of sufficiently
open machine control environments.

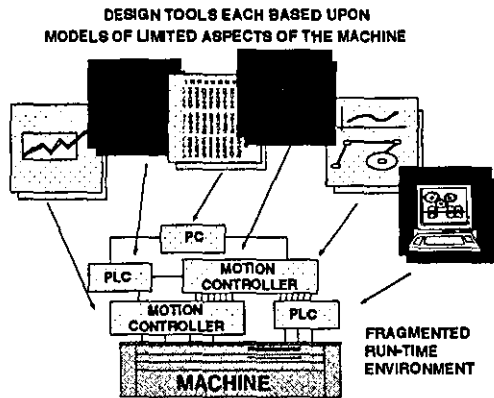


Fig. 1. Conventional Practise

The Benefits of Open Systems

Recently there has been increasing interest by both end-users and system builders in open modular control systems that, if adopted, promise greatly reduced costs over a machine's life-cycle [5][6]. Open, modular control systems, once established in the market could lead to the following benefits:

- Reduction in development costs and in particular development time for highly automated manufac-

turing applications through a major change to the adoption of configurable, compatible building block style automation components. These components being selected on a price/performance basis rather than vendor dependency.

- System design tools and configuration tools which are widely applicable and not tied to vendor specific target hardware. These being selected on a price/performance basis rather than vendor dependency.
- Reduction in the cost of eliminating faults (fast identification and replacement of defective building blocks) and ease of service and maintenance of the building block system components. Service and maintenance personnel being trained to support standard system components.
- Flexibility and adaptability of the system to changes in business direction during the life of the machine.
- Adaptations and alterations can be performed by the end user personnel themselves on the basis of an open modular system structure, without having to resort to an expensive system supplier.
- Ease of integration with current factory information and scheduling systems.

UMC/MDC

Introduction

The MSI Research Institute at Loughborough University of Technology has carried out research into new approaches to machine control for over eight years. This work has been mainly funded by the EPSRC. It has resulted in UMC (Universal Machine Control) which is an approach to creating open control systems for a diverse range of applications [5][11]. As shown in figure 2, UMC consists of a runtime control system based on a non application specific reference architecture together with an associated set of tools and libraries.

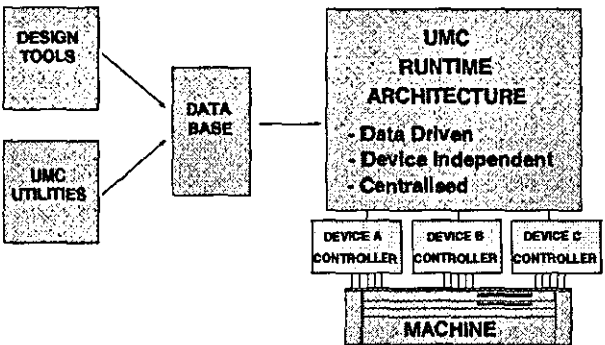


Fig. 2. Existing UMC System

UMC provides an open control approach to meet the needs of the special purpose modular machine builder who might typically produce particular machines with a

ge number of model variants and makes frequent angles to the control system configuration.

major objective of UMC was to provide facilities to sily configure, modify and upgrade the control funcns of an application. This is achieved by storing MC machine configurations conforming to the UMC ference architecture in a database. UMC machine nfigurations can be defined and revised via a series of F-line design tools. A set of UMC utilities are then ed to configure and test the UMC machine at runne. New components can be progressively defined or d ones selected for reuse as new UMC machine degn evolve.

he UMC project highlighted the need for an integrated pproach to the creation of computer controlled machines and their lifecycle support. The current IMDC ntegrated Machine Design and Control) project at MSI addressing this goal and will closely integrate the al-time control system with machine design (and other ff-line lifecycle activities).

Run-Time Architecture

The UMC reference architecture was not designed to rescriptively impose a particular application architecture. The emphasis has been placed on specification of capabilities and services to build an interconnecting structure between interoperating UMC components. The UMC reference architecture simply segments the control functions into three hierarchical layers: handler, task and machine, as shown in figure 3.

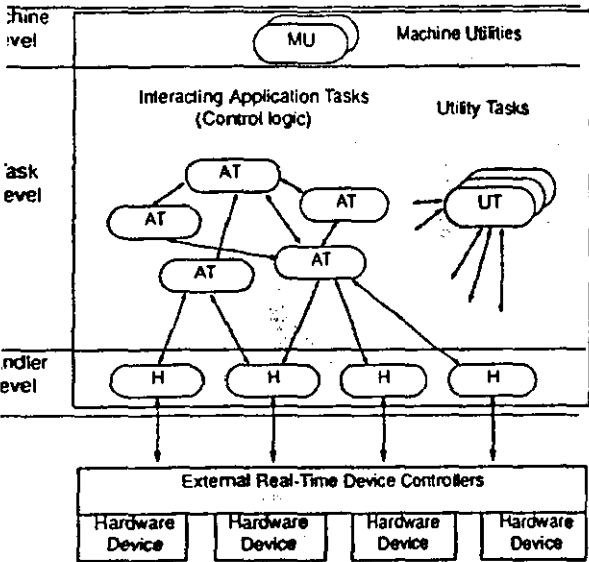


Fig. 3. UMC Run-Time Architecture

Handlers provides a consistent interface between external devices and UMC tasks: A given handler encapsulates the behaviour of a specific external device. Pre-written handlers are held within the UMC database and provide the virtual device library for UMC. The various types of handler include single axis, multiple axis and

I/O. Each provides an interface to a specific device from a particular manufacturer as shown in figure 4. Alternatively a handler supporting a software emulation of the device may be used: typically driving an appropriate component of a solid model as illustrated in figure 8.

The current UMC architecture provides the basis for an open, structured, device independent method for building machine control systems which is now experiencing industrial exploitation. The IMDC project is extending the UMC concept to support a physically distributed runtime environment [12][13]. Detailed discussion of the facilities provided by the run-time environment are beyond the scope of this paper. They can be summarised as the provision of:

- external device interfaces (for both fieldbus and non-fieldbus devices),
- support for inter-process communication, (between both local and distributed processes) and
- support services for the creation, monitoring and reconfiguration of the run-time control system elements (in centralised or distributed form).

Distributed runtime architectures can offer advantages particularly in physically large systems or where a high degree of user reconfigurability is required. Important benefits include:

- control systems can be composed of intelligent devices, physically located at the locations in the machine where the control functionality is needed;
- control system installation is simplified by replacing a complex wiring harness with a control network to reduce weight, assembly cost and processing demands on the central controller;
- inherent scalability: the freedom to vary the number and type of control nodes on a particular machine in order to modify its functionality;
- the ability to individually configure and test segments of the machine before they are combined.

Overview of the IMDC Environment

The overriding aim of IMDC is to provide a highly productive environment for machine life cycle support. Conceptually IMDC provides the "machine builder" with design and run-time tools available in the order he/she needs them on a single "workbench". Physically IMDC utilises a network of one or more user workstations coupled to an embedded real-time control architecture which resides on the target machine. As shown in figure 4, the two main user components of the IMDC environment are an off-line environment and a run-time environment.

IMDC Off-line Environment

The off-line environment is composed of the software tools set seen by the system designer. This is underpinned by the integration platform and associated IMDC database where all machine configuration information is stored. The off-line environment deals primarily with the issues involved in designing and creating a machine control system. Typical tools would be for requirements specification, application logic description, auto-code generation, simulation and run-time system configuration. The run-time control system is invoked, monitored and reconfigured with many of these tools. Hence the off-line and run-time environments must be closely coupled.

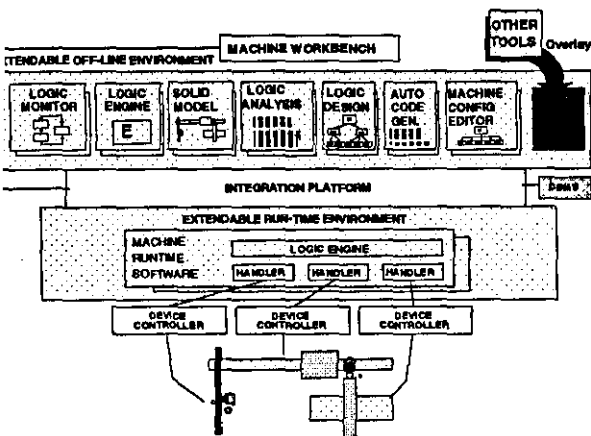


Fig. 4. Concept of an IMDC environment

IMDC Run-Time Environment

The run-time environment provides the run-time control functionality to support IMDC control system configurations previously defined by the off-line environment (software toolset). It is made up of application specific processes, device interfaces and standard utilities. Once invoked, the run-time environment may function without further intervention of the off-line system. However many off-line tools interact with the components of the run-time system for application debugging, monitoring, reconfiguration.

Integration Platform

The integration platform provides a set of generic integration services which provide the mechanisms for flexible configuration and management of distributed objects. Objects in the IMDC context are the components of the off-line or runtime environments (software tools, application logic and handler processes). A distributed object management system (DOMS) is being used to provide the necessary integration services. The DOMS provides a way of "pulling" the potentially diverse components together and managing their communications [7].

Software Toolset

The IMDC integration platform provides the means for integrating a suite of diverse machine design and control

tools into a single environment. The toolset can be progressively changed, enhanced and extended via a set of generic interfaces. These tools are co-ordinated by a "Machine Workbench" and operate on globally shared data. They provide integrated support over the whole machine design and control lifecycle by addressing application logic description and analysis, machine modelling and visualisation, hardware and software topology, information storage, code generation and run-time support. Integration of the tools facilitates rapid prototyping such that potential solutions (and modifications to existing solutions) can be quickly verified. IMDC promotes iterative analysis and revision of these potential solutions - rapid support for "what-if" analyses enabling fundamental design errors to be quickly identified.

Toolset Example: Application Logic

Introduction

The application logic must be created using an appropriate design tool. Application logic is best described using a tool which utilises an internal model which is appropriate to the problem. For example, many machine control applications in manufacturing involve predominantly sequential logic: assembly machines, transfer lines. Synect, an application design tool developed by Hopkinson Computing and also IEC 1131-3 conformant tools (which offer sequential function chart and relay ladder logic capabilities) are well suited to the description of sequential logic problems [2] [3] [4] [8]. For applications that require considerable data modelling other tools are more appropriate, for example Estelle [9]. All these tools can be integrated into the IMDC platform. The Synect approach to application description is considered below.

Synect Methodology

Functional approaches to application design have the benefit of supporting the concept of sequences, typically in the form of state diagrams. The functional approach helps the designer to consider how different components need to be co-ordinated to carry out a particular operation, such as picking a part from a feeding device. Object-oriented approaches encapsulate the sequence logic into objects which model the real-world components' allowable behaviour. Synect therefore provides a methodology for combining the co-ordination of the functional approach with the encapsulation of object orientation.

Applications are described in terms of a hierarchy of communicating objects, as shown in figure 5. The lines of communication are shown by the lines connecting the objects i.e. the Gripper can only communicate with the Robot - it cannot directly communicate with the Assembly Cell or the Arm Elevation objects for example. It follows, therefore, that the Robot is responsible for co-

linating the Gripper, Arm Elevation and Arm Translation objects.

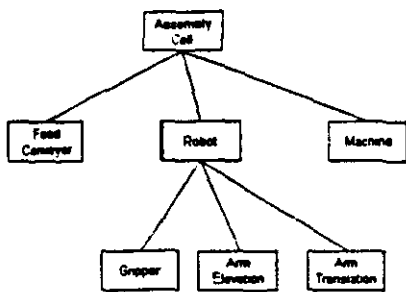


Fig. 5. Object Hierarchy

Objects may interact in either a synchronous or asynchronous manner. The Gripper object's external interface is shown in figure 6. The vertical arrows depict messages between objects in the control system and the horizontal arrows depict points of contact with the system being controlled. (The "rwi" (real-world input) and "rwo" (real-world output) prefixes are used to ensure the uniqueness of names and to aid clarity). The internal logic for each object is expressed in the form of one or more state transitions diagrams (STDs). Figure 6 also shows the Gripper's internal logic.

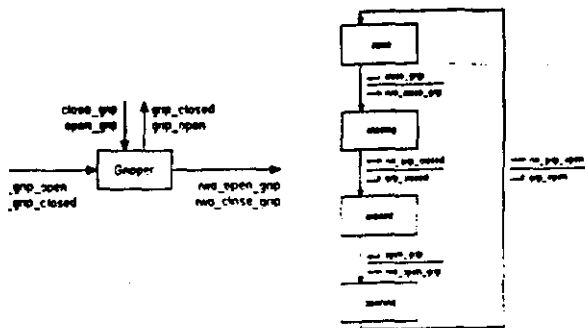


Fig. 6. External Interface and Internal STD

Synect Tools

Synect provides a set of software tools (application editor, compiler, logic engine, logic monitor and code generator) which are being integrated into the IMDC platform [8]. Applications are described by means of a graphical editor. Synect is based on the generation of a Petri-Net model from the application description [10]. Once generated by means of the compiler the model can be checked for errors such as deadlocks and unwanted state combinations. The automatic code generation provides C source code which is then compiled with appropriate libraries for the UMC run-time architecture.

IMDC Lifecycle Support

Integrated Development

Processes executing in the IMDC environment can communicate using the underlying integration services. This capability spans both the off-line and run-time en-

vironments. The use of design tools like Synect thus extends throughout the lifecycle of the machine: remote diagnostics, analysis of event logs, evaluating the effect of changes in the control logic and/or machine configuration. It should also be noted that "design" tools no longer need to be used in isolation but can be used in combination.

For example, as illustrated in figure 9, the executing control system logic may be linked to the Synect application logic STD monitor. This enables the application's current state, enabled transitions and the status of real-world inputs to be presented by animating the designers specification. Similarly the solid modeller can be driven from the target controllers for machine visualisation. If, as shown in figure 8, emulation device handlers are used then the application logic can be initially tested on the target control system without the need for physical I/O devices.

During normal operation a circular event log can be maintained on the target machine controllers. If problems occur the logged information can then be replayed on the modeller and/or the Synect STD monitor for fault finding purposes.

Rapid Prototype Machine Build

This section lists a typical set of activities which a machine builder would undertake using the IMDC environment. It is not intended to be a exhaustive list, but serves to illustrate the iterative and highly interactive manner in which combinations of tools can be used.

- 1 Select machine components - Use of IMDC machine modeller for initial machine visualisation and construction. Rapid prototyping of the machine mechanical configuration: overall design concept, configuration, sizing and selection of actuators, sensors, fixtures, tooling etc.
- 2 Application logic design - The application logic is graphically specified using an appropriate design tool. (E.g. Synect Editor.)
- 3 Application logic analysis - The logic is compiled into a Petri-Net model and then analysed for deadlocks and unreachable states. A logic simulation engine is then created. (This phase involves the use of the Synect Compiler, Analyser and Simulator.)
- 4 Simulation - As shown in figure 7, the logic simulation engine can now be used to interactively drive the modelled machine elements and the logic monitor may also be driven to animate the application logic specification. (Note that unlike conventional robot modelling packages the IMDC Modeller is not being used to specify the application control logic and does not contain the logic execution engine.)
- 5 Auto-code generation - Compilable task C code is automatically generated for the UMC centralised (and in future distributed) runtime control system. This C code is compiled with appropriate UMC libraries. The Synect

code generator generates an ANSI standard C program from the Petri-Net based logic model. The application logic is thus exactly equivalent to the behaviour the logic engine used to drive the modeller during the various simulation phase.

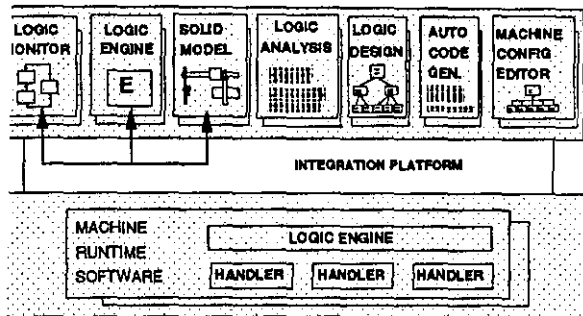


Fig. 7. Simulation Phase

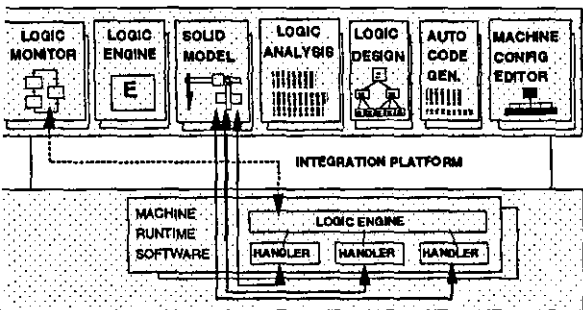


Fig. 8. Use of Emulation Handlers

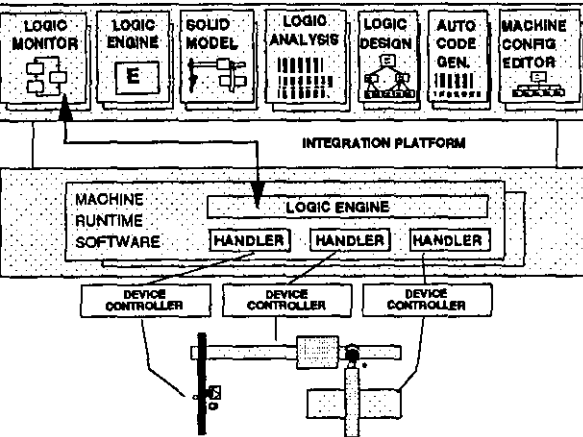


Fig. 9. Machine Execution via Device Handlers

6 Loading and running the prototype machine - Figure 8 shows a prototype machine, configured and executing with requests/responses routed between the off-line tools and the run-time target environment. Initially the safest approach is to utilise emulation handlers and drive elements of the solid model. The modeller performs the same functions as in the simulation phase (4) but it is interacting with the actual target control system rather than the off-line logic engine. The modelled elements can then be successively replaced with real devices as shown in figure 9.

Conclusions

This paper has presented an overview of the integrated machine design and control environment (IMDC) which

provides the machine builder with an integrated solution for co-ordinating tools throughout the complete machine life cycle. The commercial benefits of IMDC relate to both its impact through improved manufacturing efficiency for the end user and its potential to provide a stimulus to machine and control system vendors by:

- reducing the development cost and time of highly automated applications, and encouraging reuse of software/hardware building blocks.
- reducing the cost of eliminating design faults (fast identification) and ease of service and maintenance.
- allowing more effective adaptation and alteration of control systems, without resorting to the expensive services of a specific system supplier.

The proof of concept and effectiveness of the IMDC approach is being evaluated using representative industrial problems in packaging, assembly and transfer-line applications. The IMDC approach is currently being compared with existing design and control methods: typically either with PLC based control systems or custom built real-time computer systems.

IMDC has the potential to provide a highly effective environment for integrating design and control system elements from a diverse range of manufacturers. The approach gives SMEs the opportunity to participate far more effectively in a vast and growing automation market which is still dominated by large vendors of "closed" systems.

References

1. Muir K., "Stating the CASE for Industrial Automation, Drives and Controls", June 1990, pp 22-24.
2. "ISaGRAF Industrial Software Architecture", CJ Int., 86, Rue de la Liberte, 38180 Seyssins, France.
3. "APT Application Productivity Tool", sales catalogue, Texas Instruments, 1992.
4. Bekkum, J., "The Coming Of Open Programmable Controller Software", Control Engineering, Oct. 1993.
5. Weston R.H., Harrison R., Booth A.H. and Moore P.R., "A New Approach To Machine Control", Journal of Computer-Aided Engineering, 1989, pp 27-32.
6. "MOSAIC, Modular Open System Architecture for Industrial Motion Control", ESPRIT II project 5292.
7. Object Management Group., "The Common Object Request Broker: Architecture and Specification, OMG Document No 91.12.1, 1991.
8. "Synect User Guide", Hopkinson Computing Ltd, 29 Deepdale, Guisborough, UK, 1995.
9. ISO 9074. "Estelle: A formal Description Technique Based on an Extended State Transition Model", 1988.
10. Peterson J.L., "Petri Net Theory and the Modelling of Systems", Prentice Hall, 1981.
11. Harrison R., "A Generalised Approach to Machine Control", PhD Thesis, Loughborough Univ., Jan. 1991.
12. Reeve A., "Plots and Pressure Focus on Fieldbus", Intech, 40 No. 7, 1993, pp 21-23.
13. "Fieldbus Update: ISP WorldFIP, SP50", Control Engineering, 1993, pp 40 No. 9, 23.

INTERACTIVE VISUALISATION OF SEQUENCE LOGIC AND PHYSICAL MACHINE COMPONENTS WITHIN AN INTEGRATED DESIGN AND CONTROL ENVIRONMENT

R. Harrison, A.A. West, P. Hopkinson and C.D. Wright

*Manufacturing Systems Integration (MSI) Research Institute, Loughborough University,
Loughborough, United Kingdom, LE11 3TU.*

Abstract: An Integrated Machine Design and Control (IMDC) environment for the visual representation and integration of the physical machine components and control logic is discussed in this paper. The approach taken is unique in that (a) the control logic and physical models of the elements can be investigated individually for correctness and completeness, (b) the control logic can be easily integrated with the solid models to animate the model of the physical machine and (c) reconfiguration enables the same control logic to be applied to real world physical machine elements. At any stage during the machine design and implementation process, the user of the environment can pause and question the validity of certain operations and control system parameters.

Keywords: Machine, Control, Logic, Design, Distributed, Objects, Modelling, Petri-nets.

1. INTRODUCTION

The design and implementation of manufacturing machines is under increasing time and financial pressures as customers demand increased product variety and quality at reduced product cost (Young, 1995). Increased competition and governmental pressure to focus on environmental issues has forced modern machine builders and users to consider the requirements for the next generation of machines that allow the reconfiguration of both the control software and physical hardware (Rahkonen, 1995). Machines will be required to be developed in the minimum amount of time and comprise (a) vendor independent hardware components, (b) sophisticated control algorithms, (c) intelligent sensors and actuators and (d) user friendly interfaces. In addition, open systems issues concerning the ease of integration, interoperability of the software and hardware components and available standards must be addressed (Crowcroft, 1995) to ensure that reuse and reconfiguration can be achieved. The inherent complexity inevitably necessitates the increased application of machine modelling software toolkits (i.e. for control logic and physical machine element design and analysis) and ad-

vanced computer technology throughout the machine life cycle from requirements definition, through the design and build stages to maintenance and reconfiguration.

End users, machine designers and machine builders require technical and operational knowledge from disparate disciplines and specialised domain experts at various stages throughout the machine requirements, design, build, installation, set-up, maintenance and reconfiguration life cycle (Carrott, *et al.*, 1997). Common frames of reference are vital to improve the communications between the above stakeholders in the machine design and build process. In particular it is important that discussions are focused around both the control system software and physical hardware to ensure that the required logical operation and physical functionality is achieved.

Visual representation of machine control software is currently limited. The majority of machine control system software is developed for implementation on programmable logic controllers (PLC's) (Michel, 1990). The basic representation of the logical mapping of sensor inputs to actuator outputs is inherently simple, but soon becomes complex as more function-

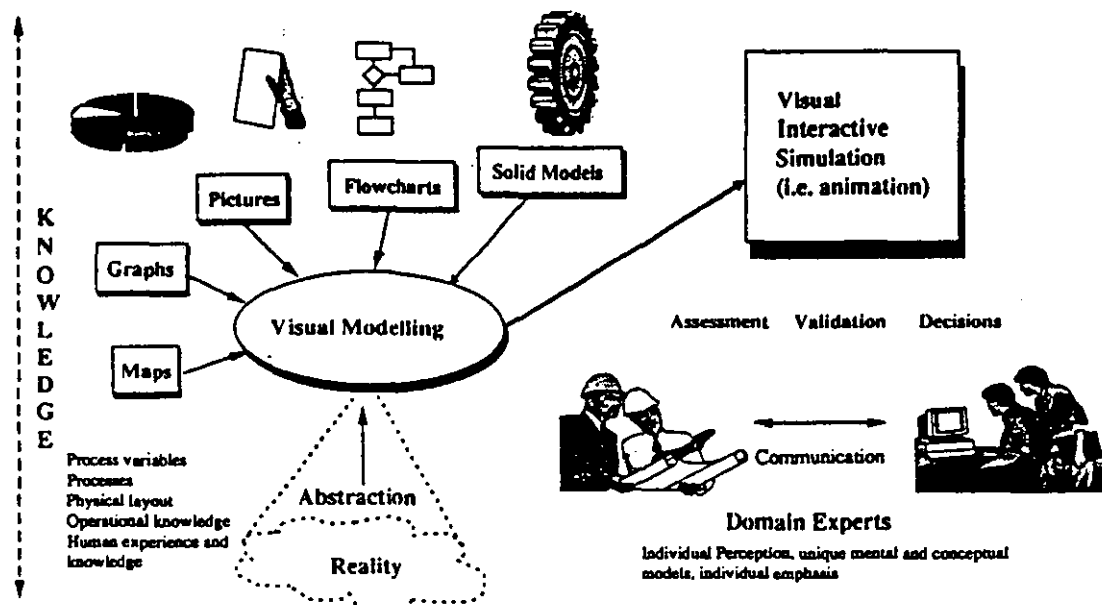


Fig. 1. Illustration of the Visual Interactive Simulation Process.

ality is included. The development of complex manufacturing logic for PLC's is a specialised activity, and it is difficult for a non expert to appreciate the operation of the complete system (Venkatesh, *et al.* 1994). Ladder logic diagrams and sequential function charts (David and Alla, 1992) provide graphical representation of sequential operation but the design of flexible, reusable and maintainable software is nevertheless difficult to achieve.

There has been widespread use of visual representation of manufacturing products in terms of surface and solid models (Hoffmann, 1997). Computer aided design package usage e.g. AutoCAD and Unigraphics (Liang *et al.*, 1996) enables design engineers to visualise physical components and layouts prior to commissioning. In certain cases, operational logic has been included into the solid model to enable a dynamic animation of the modelled system elements to be observed and optimised (Hoffmann, 1997). A major limitation with this approach has been the fact that in order to transfer the results of the modelling exercise to a real system, the logic must be reimplemented outside of the solid model and the opportunity for errors and sub optimal implementation behaviour proliferate (Wright and Case, 1995). There is a requirement for an environment in which (a) the control logic and physical models of the elements can be investigated individually for correctness and completeness, (b) the control logic can be easily integrated with the solid models to animate the machine solid model and (c) reconfiguration enables the same control logic to be applied to real world physical control elements.

A major research theme at the Manufacturing Systems Integration (MSI) Research Institute at Loughborough University, UK is the realisation of the next generation of machine systems (Carrott, 1996). An Integrated Machine Design and Control (IMDC) en-

vironment (an integrated software environment and toolset comprising third party and "in house" tools) has been developed that seeks to support the work of control system engineers and mechanical designers throughout the design and development life cycle of manufacturing machines. Visibility of the physical machine (using the IMDC Machine Modeller developed around the ACIS solid modelling kernel (Murry and Yue, 1993) and control logic software (using the Synect modelling tool produced by Hopkinson Computing Ltd. (Anon, 1995)) is a core requirement in the IMDC system and is discussed in this paper.

The environment is based around a distributed object oriented representation of manufacturing machines as aggregations of basic components (Joannis and Krieger, 1992): single axes, multi-axes, digital and analogue input / output and dumb and intelligent sensors (e.g. vision systems and robots). Distributed object technology (DOT) (Orfali, *et al.*, 1996) provides the core framework within which the tools and real system intercommunicate.

2. VISUAL INTERACTIVE SIMULATION.

The visual interactive simulation paradigm (also termed visual interactive modelling and visual interactive problem solving was originally applied to the discrete event simulation of job shop scheduling problems in manufacturing (Hurion, 1980) and has mainly been utilised in Operational Research (Bell, 1995). Visual interactive simulation is particularly appropriate in the manufacturing machine domain (Sadashir, *et al.*, 1989) and can provide a common frame of reference to facilitate human communication of ideas. In the manufacturing machine domain it is vital that both the physical machine and control logic are available for scrutiny. Fig. 1 illustrates the process. A visual model (i.e. a model based upon non textural and non verbal elements to communicate the

tate of a system) is developed that provides an abstraction of reality.

The application of visual interactive simulation to industrial machine design and control projects using the IMDC environment has resulted in a number of observations and benefits:

- It is important to ensure general interaction and early involvement by developing an animated picture as soon as possible.
- Interaction allows the end users and machine builders to make complex decisions with increased confidence due to their increased understanding of the machine operation and interaction.
- The visual image is widely accepted and unexpected situations can be envisaged via what if? scenarios.
- Of vital importance is the integration of the control logic with the visual simulation in the IMDC environment. This enables the verification (by the developer) and validation (by the user) of the physical, functional and logical performance of the machine. In addition, realistic scenarios and results can be replayed to managers to ensure their participation and ownership of the project.

3. VISUALISATION OF SEQUENCE LOGIC

Manufacturing and process industry control system applications invariably include sequence logic. The complexity of the application logic typically involves the need to manage several concurrent activities, co-ordinating their behaviour to achieve the desired application goal.

In addition to the standard software engineering problems of defining how the proposed system is to work, and expressing the design in a form which is readily understood, control systems designers must ensure that the system's dynamics do not contain design errors, such as deadlock, livelock and undesirable modes of operation. Traditional approaches have tended to be ad-hoc or have inadequately addressed the designer's needs, leaving such errors to be identified late in the life-cycle, resulting in costly modifications.

Synect™ provides a set of software tools (application editor, compiler, logic engine, logic monitor and code generator) which are integrated into the IMDC platform (Anon, 1995). Applications are described by means of a graphical editor. Synect is based on the generation of a Petri-Net model from the application description (Peterson, 1981). The evolving design can then be examined analytically for structural and behavioural deficiencies e.g. checking for errors such as deadlocks and unwanted state combinations. An executable logic model is created which can then drive a visual solid model of the physical machine

(see section 4). The automatic code generation tool produces code which is then compiled with IMDC runtime libraries enabling the application logic to communicate with the distributed machine components.

3.1 Logic Visualisation / Representation

The designer of a sequential control application can visualise the problem and present the solution in a number of ways. Three possible approaches are:

- consider the application as consisting of a set of interlocks. If the target environment is hard-wired logic or, as in the majority of current industrial automation projects, a PLC programmed in ladder logic, the design solution maps easily on to the implementation (Michel, 1990).
- take a functional view. One of the more popular structured methods used for real-time applications has been the Ward-Mellor variant of the Yourdon method (Ward and Mellor, 1985). Control transforms are triggered by events and then react by sending signals to other transforms.
- take an object oriented view. Object orientation has grown from a modelling paradigm and, it is claimed, leads to more intuitive solutions which are more maintainable and more amenable to re-use.

The combinational logic approach is potentially the preferred option if the application logic is very simple or there is a strong need for the solution to require the minimum of memory or execute as fast as possible. Otherwise, the solution can be more difficult to verify, is not conducive to diagnosing operational mis-behaviour and is more difficult to modify without causing unwanted side-effects.

Functional approaches have the benefit of supporting the concept of sequences, typically by a form of state diagram. The functional approach helps the designer to consider how the different components need to be co-ordinated to carry out a particular operation.

Object-oriented approaches encapsulate the sequence logic into the objects which model the allowable behaviour of real-world components. The overall system logic is however now fragmented across different objects and this can lead to unanticipated modes of operation.

Synect provides a methodology which combines the co-ordination of the functional approach with the encapsulation of object orientation.

IMDC integrates the Synect simulator with the 3D solid modeller (see fig. 2). As described in section 4, the modeller incorporates concepts such as timing information and sensor emulation into the solid modelling software so that the solid model shows a real-

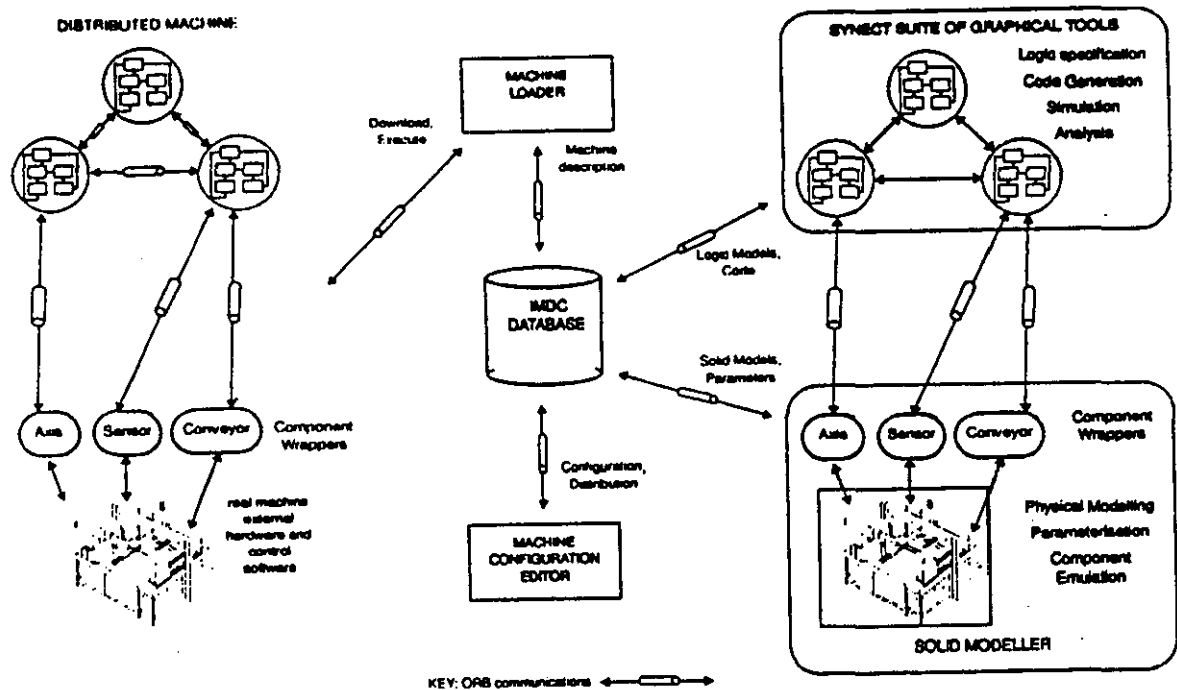


Fig. 2. Illustration of IMDC Interaction between Synect, the Solid Modeller and the Distributed Real Machine.

time emulation of how the implemented system will perform.

4. VISUALISATION OF MACHINE ELEMENTS

A machine designer can utilise solid models in order to visualise the physical machine elements and their interaction (Hoffmann, 1997). The conventional approach to the solid modelling of machines involves:

1. initial static visualisation of the machine elements and
2. determination of the system dynamics and machine element interaction by embedding operational logic within the modelling tool (Yong *et al.*, 1985).

The limitation of this approach is that the coding for the real system operation is normally undertaken after the simulation phase and the software frequently bears little relation to the mechanisms used to drive the simulation. In addition this approach to the modelling of the physical machine elements cannot be used throughout the life cycle and cannot be used interactively with other design tools.

The approach adopted within the IMDC environment is fundamentally different and offers a means to overcome these difficulties. Graphical support is provided for design evaluation which enables designers to rapidly investigate what if scenarios, using the same control logic throughout the life cycle.

Models of physical machines can be graphically constructed within the machine modeller from component building blocks such as actuators, sensors, conveyors, alarms and structural elements. The model components incorporate both geometric and behav-

ioural models of the external systems being represented. The tool can be used to specify component states, operational parameters, motion parameters and locations. Components, parameters and complete sub-assemblies can be stored in the IMDC database.

The interface to components of each specific type is exported via a wrapper object using distributed object technology. External objects residing in remote processes can connect to components using the wrapper and either drive them (e.g. actuators, conveyors) or be notified as events occur inside them (e.g. sensors). Interaction via a GUI displaying the 3D model workspace allows the user to move around the model and examine elements in detail as the machine models are being driven.

5. ENVIRONMENT TO SUPPORT VISUAL INTERACTIVE SIMULATION: IMDC

The IMDC environment comprises four distinct areas of functionality:

- user tools to enable, for example, logic simulation, motion design, machine modelling and tools associated with the control and monitoring of the runtime system. The user tools can be progressively changed or extended via a set of generic interfaces.
- systems tools to enable system administration, access to information and security. Typical system configuration information includes the logical and physical system layout (machines and networks), user names, passwords, access rights, project and user environment information.
- the IMDC object oriented database (POET a product from the POET Software Corporation (Anon, 1994) which provides persistent storage for the outputs from life cycle activities, system

configuration, traceability and version control and holds the generated elements of specific target control solutions

- the distributed runtime machine composed of communicating software components, a subset of which provide interfaces to external devices and third party control software for the monitoring and control of the physical machine.

These functional elements all inter-communicate via the underlying Object Request Broker (ORB) architecture as described in the following section.

5.1 IMDC System Architecture

The adoption of an object-oriented approach, particularly the use of distributed object technology, has been the key to providing flexibility in the choice of implementation technologies. Fig. 3 illustrates the principle of object distribution across heterogeneous host and network architectures. Identical client software located on different host platforms 1 and 3 communicate with the server object on host 2. Hosts 1 and 3 may be physically linked to host 2 by different network types.

A multi-schema architecture has been implemented to separate client (typically software tools) and server (such as data repository) applications via an underlying integration infrastructure. The integration infrastructure has been built using distributed object technology based upon the Common Object Request Broker Architecture (CORBA) specification from the OMG (Anon, 1991). The infrastructure acts as a system-wide broker for object services and provides abstraction from low level device specific problems. Object services are dynamically registered and de-registered with the infrastructure by processes which implement the services (object servers). Client processes query the infrastructure for available services and are given the necessary connection information to access services.

6. INTERACTIVE USE OF PHYSICAL AND CONTROL LOGIC MODELLING TOOLS

Application logic is not embedded in the modelling tool but is generated by use of the Synect logic tool-set. By functioning as a server, the modeller can be controlled by remote processes for example, (a) the Synect logic simulator tool, or (b) task control software running in the target control system. This highlights an important feature of the run-time architecture, namely that the IMDC defined interfaces to the controlled elements of these solid models are identical to the IMDC interfaces to the real control system. Hence, the control logic processes can drive either the modelled hardware elements (thus animating the model), the real world hardware elements, or a mixture of the two. This permits incremental proving of the control logic in a hardware independent manner

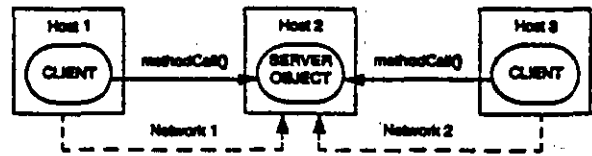


Fig. 3. The Principle of Object Distribution Across Heterogeneous Host and Network Architectures.

and also allows hardware to be included into the system as and when it becomes available.

6.1 Application Example

The proof of concept and effectiveness of the IMDC approach is being evaluated using representative industrial problems in packaging, assembly, transfer-line and PCB handling applications.

Within Synect, applications are described in terms of a hierarchy of communicating objects. Part of the object hierarchy for an example PCB handling machine is illustrated in the right hand window of fig. 4. The internal logic for each object is expressed in the form of state transitions diagrams (STDs). A fragment of a typical STD can be seen in the left hand window of fig. 4.

The PCB handling machine consists of a framework which supports a board carriage system, a movable gantry and a robot arm which are used to populate PCBs with components. To enable visual interactive simulation the solid model of each of these component can be associated with real world input and output contact points from the control logic (see fig. 2).

7. CONCLUSIONS

The novelty of the IMDC approach is that the control system software and physical machine components can be individually conceptualised and interactively tested. Furthermore IMDC enables modelled components to be progressively replaced with real system components until the final solution is attained. A particularly attractive feature is that the same sequence logic is used to control both the modelled and real world components.

The commercial benefits of IMDC relate to both its impact through improved manufacturing efficiency for the end user and its potential to provide a stimulus to machine and control system vendors by:

- reducing the development cost and time of highly automated applications, and encouraging reuse of software/hardware building blocks.
- reducing the cost of eliminating design faults (fast identification) and ease of service and maintenance.
- allowing more effective adaptation and alteration of control systems, without resorting to the expensive services of a specific system supplier.

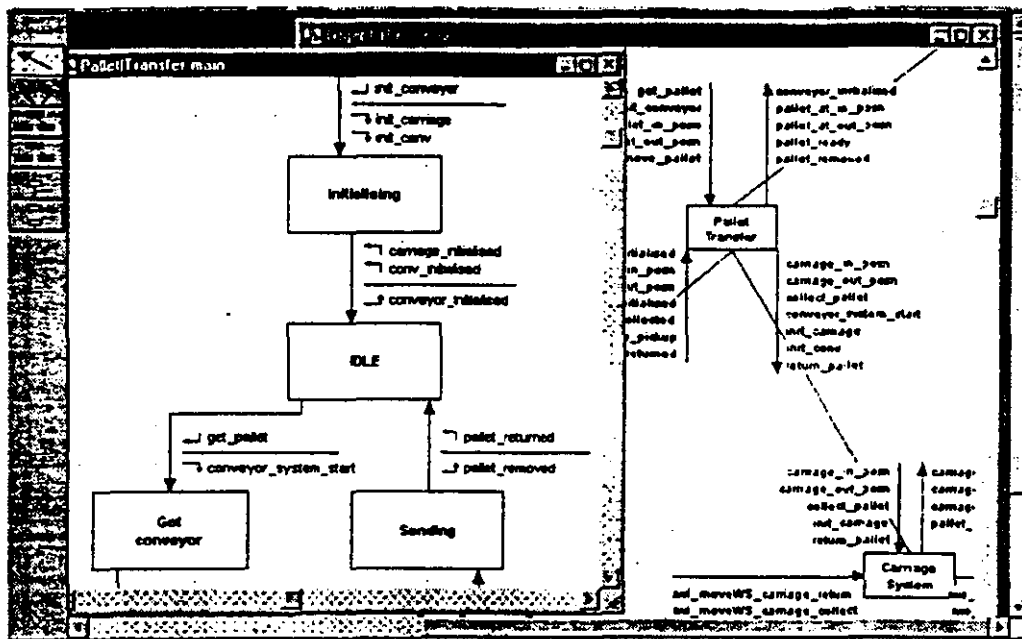


Fig. 4. Typical User Interface Screen for the Synect Editor showing an Object Hierarchy and STD.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the EPSRC for the provision of research funding and Hopkinson Computing Ltd. for their collaborative input.

REFERENCES

- Anon, (1991). *The Common Object Request Broker: Architecture and Specification*, Object Management Group (OMG) Document, No. 91.12.1.
- Anon, (1994). *POET Reference Manual, Version 2.1*, POET Software Corporation.
- Anon, *Synect User Guide*, (1995). Hopkinson Computing, Ltd., 29 Deepdale, Guisborough, UK.
- Bell, P.C. and O'Keefe, R. (1995), Visual Interactive Simulation - History, Recent Developments and Major Issues, *Simulation*, 49, No 3, pp. 109-116.
- Carrott, A.J., Moore, P.R., Weston, R.H. and Harrison, R., (1996). The UMC Software Environment for Machine Control System Integration, Configuration and Programming, *IEEE Trans. on Industrial Electronics*, 43, No 1, pp. 88-97.
- Carrott, A.J., Wright, C.D., West, A.A., Harrison, R. and Weston, R.H. (1997). A Toolset for Distributed Real Time Machine Control, *SPIE Photonics East Procs.*, Ma. USA, 2913, pp.2-12.
- Crowcroft, J.(1995). *Open Distributed Systems*, Boston; London, Artech House.
- David, R. and Alla, H. (1992). *Petri Nets and Grafcet*, Englewood Cliffs, NJ, Prentice Hall.
- Hoffmann C, Rossignac J (1997). Special issue: Solid modelling, *Comp.-Aided Des.*, 29, No.2, p.87.
- Hurion, R.D., (1980). An Interactive Visual Simulation System for industrial Management, *European journal of Operational Res.*, 5, pp. 86-93.
- Joannis, R. and Krieger M., (1992). Object Oriented Approach to the Specification of Manufacturing Systems, *Computer Integrated Manufacturing Systems*, 5, No.2, pp. 133-145.
- Liang M., Ahamed S. and vandenBerg B. (1996). A STEP Based Tool Path Generation System for Rough Machining of Planar Surfaces, *Computers in Industry*, 1996, 32, No.2, pp.219-231.
- Michel, G. (1990). *Programmable Logic Controllers: Architectures and Applications*, Wiley, UK.
- Murray J.L. and Yue Y. (1993). Automatic Machining of 2.5D Components with the ACIS Modeller, *Int. Journal of Computer Integrated Manufacturing*, 6, No.1-2, pp.94-104.
- Orfali, R., Harkey, D. and Edwards, J., (1996). *The Essential Distributed Objects Survival Guide*, Wiley and Sons, Chichester.
- Peterson J.L., (1981). *Petri Net Theory and Modeling of Systems*, Prentice Hall.
- Rahkonen, T.(1995). Distributed Industrial Control Systems - A Critical Review Regarding Openness, *Ctrl. Eng. Pract.*, 3, No. 8, pp.1155-1162.
- Sadashir, A., (1989) Software Modelling of Manufacturing Systems: The Case for an Object Oriented Programming Approach, *Annals of Operational Research*, 17, pp. 363-378.
- Venkatesh, K., Zhou, M., and Caudill, R.J. (1994). Comparing Ladder Logic Diagrams and Petri Nets for Sequence Controller Design Through a Discrete Manufacturing System, *IEEE Trans. on Ind. Electronics*, 1994, 41, No 6, pp. 611-619.
- Ward, T.W. and Mellor, S.J. (1995). *Structured Development for Real-Time Systems*, Vol. 1, Yourdon Press, Prentice-Hall.
- Wright, C.D. and Case K. (1996) Emulation of Modular Manufacturing Machines using CAD Modelling, *Mechatronics*, 4, No. 7, pp. 713-735.
- Young, S.L.(1995). Technology: The Enabler for Tomorrow's Agile Enterprise, *ISA Trans.*, No 4, pp. 335-341.
- Yong, Y.F. et al (1985). *Off-Line Programming of Robots, Handbook of Industrial Robotics*, John Wiley, New York, pp 366-386.

Implementing S88 batch control systems in the pharmaceutical industry

by Peter Hopkinson, Mike Sonley and Guy Wingate
Autotech Engineering Solutions Limited

Introduction

Many pharmaceutical processes are batch-oriented. That is, the process leads to the production of finite quantities of material (batches) by subjecting quantities of input materials to a defined order of processing actions¹. Other classifications are continuous and discrete processing. The goal of a continuous process is to produce a steady stream of product using mainly continuous control behaviour, such as 3-term control loops. A discrete process exhibits predominantly sequence behaviour. Batch processes often contain continuous and sequence behaviour.

Automation of the process, by a batch control system, can help to maximise the plant throughput and yield and quality product. The control system automatically sequences plant equipment to control the batch and consequently avoids unnecessary delays. Because each batch is subjected to the same control, consistency is also improved. However it is often inappropriate to fully automate a plant e.g. dosing and sampling checkpoints may be candidates for manual intervention.

Timescales

Pressure to collapse project timescales is becoming ever more intense. Rapidly changing business environments and market opportunities, in addition to financial incentives associated with the project itself, all serve to shorten the period between project sanction and beneficial operation. As the available timescale shortens, the impact of an undetected error during development becomes more severe. Clearly, the required focus of a project methodology is on risk management and the development of an excellent technical solution.

Standards

Many batch control systems have been implemented from systems entirely in a PLC to Distributed Control Systems (DCS). Some, particularly the PLC solutions, have been entirely bespoke solutions whereas others have been based on proprietary software platforms.

Consequently there are now many batch control solutions which have limited consistency in terms of terminology or structure. This applies to solutions using different vendors, platforms and implementations using the same platform.

Good manufacturing practice

The pharmaceutical industry, under Good Manufacturing Practice (GMP) regulations, is subject to stringent inspections regarding both the manufactured product and equipment used, including the control system which could have a significant influence on product quality². The control system itself must offer sufficient evidence of its capability to consistently behave as required, and must typically generate evidence of the processing associated with each batch of product. Indeed, without the batch record, the manufactured product is effectively worthless.

In order to deliver an excellent solution which tackles these issues, a project methodology specifically designed for batch control systems in the pharmaceutical industry is required. Elegance and fit for purpose are the concepts underpinning both the project methodology and the control system solution. The project methodology must exploit the spirit and detail of the S88 standard for batch control in

order to satisfy the business drivers. It should aid communication between all parties involved in a batch control system project by providing consistent models and common terminology. The separate modelling of equipment control from product recipes will enable the control system to be more responsive to recipe and plant equipment changes.

Validation methodology

The UK GAMP Forum Supplier Guide³ is structured around a formal management system to help suppliers of automated systems to the pharmaceutical industry ensure Good Manufacturing Practice. A validation lifecycle for new and replacement automation systems is shown in Fig 1.⁴

Validation planning

Validation planning addresses how the GMP requirements of the batch control system are to be satisfied. The validation plan organises the validation activities including supplier audits, roles and responsibilities, procedures and required documentation relating to the other lifecycle phases, and ongoing support issues such as change control and operating procedures. The user requirements are also specified at this stage.

System specification

The supplier responds to the user requirements by specifying the capabilities of the proposed system in the system specification phase. This specification describes the functionality of the system and areas of non-compliance. Acceptance testing, particularly the Operational Qualification, will be derived from the functional specification so it is im-

portant that it contains statements which are verifiable. It must also provide the definition necessary for the design activities to follow.

S88

S88.01 is the standard defining common models and terminology for batch control (see appendix 1). The system specification should describe the proposed solution using S88 terminology, separating the capabilities of the plant equipment from its use in product recipes. The grouping of control modules (such as valves and pumps) into equipment modules and the definition of the corresponding phase control logic is particularly important. The system specification must also state whether an S88-aware software platform is to be used as this will also significantly affect the solution's capabilities and limitations.

The information available at the project start will be different depending on whether the plant is a greenfield site or a brownfield site. Documentation is likely to include the process descriptions for the products and a set of Piping and Instrumentation Diagrams (P&IDs) or Engineering Line Diagrams (ELDs) corresponding to the plant. In a replacement project, there may also be documentation about the implementation of the original control system. The original requirements may not be available nor a specification of the functionality the control system. The implementation documentation, such as sequence specifications and database configurations, may also be incomplete. S88 requires that the capabilities of the plant are decoupled from the product recipes in order to maximise flexibility. An existing non-S88 control system implementation is therefore likely to be a subset of the desired control capability. An exercise to understand the initial control intent from the existing implementation is complicated by its 'accidental' complexity¹ and constraints imposed by the original development environment. The existing implementation will reflect the original control system's intended functionality whereas there are likely to be additional requirements for the new system. Knowledge of the plant's process capability must be added to produce the specification for the new system.

The S88 analysis is critical to the control system design. There may be

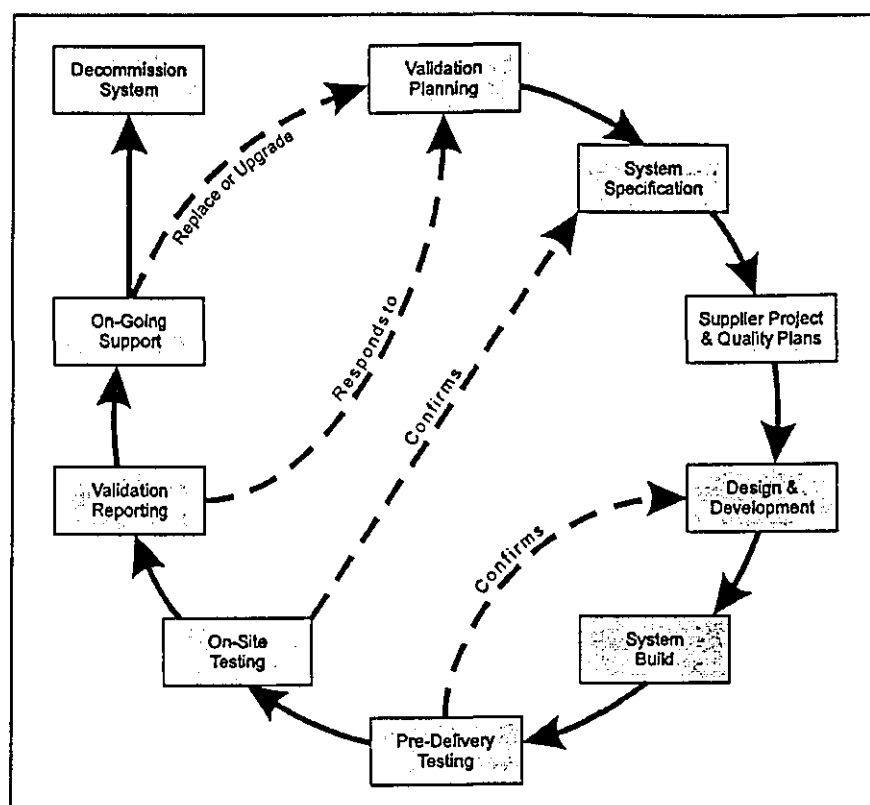


Fig 1 A Validation life cycle

several different groupings of control modules into equipment modules (see appendix 1), potentially offering different granularity of phase logic. Choosing fewer phases limits the flexibility of the control system but there are disadvantages with partitioning the plant into too many equipment modules, such as:

- Unnecessary coupling between the phase logic, bringing support and maintenance implications.
- Additional responsibility on the recipe designer to ensure dependent phases are appropriately configured
- Potentially more complex failure mode analyses are required, particularly where the co-ordination may span control devices and communications networks.
- Unnecessary batch software platform licence costs.

A method is therefore required for the S88 analysis. The method must consider top-down approaches, which examine the products, and bottom-up approaches, which focus on the equipment and its capabilities. This is an iterative exercise, requiring contribution from process, operations and control system experts.

The original implementation may only address the normal operation of

the plant, with abnormal occurrences being recognised and dealt with by plant operational staff. But significant benefits to product quality and yield may be available if abnormal behaviour is considered at the control system design stage and the necessary mechanisms for bringing the plant to a controlled quiescent state are specified, along with the necessary processing to re-establish the process.

Design reviews

Design reviews are key to the success of the project. The intended control strategy is walked through with the operations and control systems personnel in a Control and Operability review. A checklist approach similar to the use of guide words in Hazard and Operability (HAZOP) studies ensures that the control system behaviour is studied in depth. A diagrammatic method, using notations like state transition diagrams or sequential function charts, is preferred because it presents a precise definition in a simple form. Whereas this design review verifies the control system behaviour, additional design reviews later in the lifecycle verify the mechanisms and configuration used to

plement the system.

Cutover philosophy

For a replacement project, one of the major design decisions to be made early in the project's lifecycle is the change over philosophy from the existing control system to the new control system. The two main options are big bang and a phased cutover.

The big bang approach uses a major plant shutdown to entirely replace the old control system. This helps to focus the effort of the project team to the defined shutdown window, can help to avoid project timescale drift and may be the only option if the control scheme is complex and highly integrated. In some cases there is physically not room for both old and new control systems. Among the disadvantages are the need to utilise and manage significant numbers of people during the changeover and the possibility of commissioning overruns if a major problem is encountered. A feature of the big bang approach is that following the major shutdown, the operator is required to control the whole plant using the new system. Substantial training is therefore needed prior to start-up.

One approach to managing the phased cutover is to divide the plant into relatively stand-alone areas and then progressively cut them over from the old control system to the new. In some cases, it may only be possible to cutover a proportion of the IO in a phased manner with the remainder to be cutover during a shutdown. Phased cutover provides the opportunity to prove the engineering principles and processes on relatively small plant areas and offers the operators a less intensive learning curve. The disadvantages are that it prolongs the changeover period which can lead to difficulties in retaining the key personnel, requires both old and new control systems simultaneously and may be difficult to implement due to the structure of the old system and the difference in philosophy between systems.

Supplier project and quality plans

Supplier quality and project plans define how the supplier is going to execute the project in terms of quality procedures to be followed and

deliverables. Ideally, the supplier will hold ISO 9000 accreditation. Two factors which must be considered at this stage are construction and safety.

Construction

The UK Health and Safety at Work Act 1974 placed safety responsibility on designers of any article for use at work. The Construction (Design and Management) Regulations 1994 (CDM) extend the duties to construction work, particularly with regard to the erection of the designed items and the subsequent maintenance and demolition. Batch control system projects in the UK are typically notifiable under the provisions of the CDM regulations and require the supplier to provide to the project an organisation chart which will ensure that the requirements of the above legislation are met. Ideally established procedures and guidelines for compliance with the statutory requirements of CDM will be referenced by the supplier.

Safety

A manufacturing plant is typically protected with a high integrity independent trip system. The control system attempts to control the plant to remain within its boundary of acceptable operation but a malfunction does not compromise safety due to the trip system. The control system can be considered to impose a demand on the safety system. So if the control system is replaced, the safety system must be reviewed. IEC 1508 is a forthcoming standard aiming to ensure that the safety related systems which protect and control equipment and plant are specified, engineered and operated to standards appropriate to the risks involved.

Design and development

In a typical software development project, software architecture design is a major activity; the use of SCADA and batch control system platforms reduces this to a minimum. The design now specifies the structure of the platform configurations.

The S88 analysis will have produced a precise definition of the required control logic. Because an implementation architecture is implicit in the use of the S88-aware platform, the design effort concen-

trates on defining implementation rules for transforming the control logic into software code. A consistent implementation can then be coded.

Prototyping

Poor performance can affect the ability of the control system to meet the user requirements and is also an emotive subject when the operator interface is affected. The functional specification must therefore define performance using measurable criteria. But predicting performance analytically can be difficult. Consequently, early performance prototyping is used to confirm design decisions providing time to correct any problems.

System build

System build includes software implementation, configuration of software platforms and hardware manufacture. Site-based construction and electrical activity may also be included.

There are typically at least two scenarios regarding the configuration of the software platform:

- Bulk configuration as the control system is developed
- Amendments to the configuration, often in the operational phase of the control system's lifecycle.

The platform configuration tools are often oriented at the latter usage. They are often graphical or tabular in nature, designed for the infrequent user to progress step by step through the configuration change, providing maximum feedback and confirmation warnings as necessary. These tools tend to be very inefficient for the bulk entry of configuration data because they do not enable existing data to be applied automatically or patterns in the required configuration to be exploited. When the starting point for the replacement control system project is several thousand pages of flowchart definition and I/O points in the order of hundreds or thousands, it is clear that a more intelligent approach is required.

The use of rapid configuration tools and a project database can assist considerably. Fortunately, most platforms can export and import significant sections of the configuration to an external format, such as the comma separated variable (CSV) format for manipulation by spread-

sheets. The project database stores, in a neutral format, the information for generating configuration data for the PLC/DCS database, SCADA database and the database in the batch control system platform. The rapid configuration tools can automatically populate some of the project database.

Pre-delivery testing

The batch control system will typically be staged at the suppliers premises although in the case of a physically large system, it may be assembled on-site. The first part of pre-delivery testing is the Installation Qualification (IQ) to confirm physical properties of the system, such as the manufacturer, model and software versions. The next part is the Operational Qualification (OQ) to confirm that the system satisfies the capabilities in the functional specification.

De-Bottlenecking

In a fast track project, bottlenecks must be eliminated wherever possible. One candidate is the acceptance testing phase although in some projects this contributes to operator training. A simplistic approach would require that the client witnesses in-depth testing of all aspects of the control system. For a large batch control system this requires substantial effort and elapsed time.

One solution is to ensure that all software is independently inspected and tested and that this process is monitored by an impartial quality engineer with particular knowledge of the requirements of the pharmaceutical industry. The software module implemented by one member of the development team is passed to another member of the team for independent testing. The benefits of this approach can include:

- Documented evidence of source code review.
- Consistent application of development standards.
- Sharing of knowledge regarding implementation strategies which can help in the continuous improvement of implementation standards
- Motivation for individuals to stress-test other developers' software. This increases further the software standards as the developer strives to write defect-free code.
- Ability to increase the team size be-

cause learning is built into the development process.

- Test evidence which supports user OQ
- This process ensures that the software accurately implements the design. The accuracy of the design against the original control system and the required additional functionality will have been verified by means of walkthroughs with the client. The OQ can therefore be confidently addressed by progressing a set of test cases through the batch control system, substantially reducing the necessary involvement of the client.

Test environment

An appropriate test environment, using validated tools, is of value for software testing. Simulation tools are available which can be programmed to mimic plant behaviour in correct and fault conditions. Ideally a test tool would be available to capture the raw data generated during the test and also be able to automatically repeat a set of tests. Simulation tools can also be of benefit in terms of gaining operator acceptance and operator training whilst the control system is being developed.

Confidence checkpoint

The methodology can be considered to consist of several small feedback loops, feedback being an integral part of each activity, with several confirmation phases towards the end of the lifecycle. This could place significant risk of an unforeseen problem being propagated through to pre-delivery testing, when there is no time left to rectify it. Consequently, our project methodology requires that the target control system is assembled at the earliest opportunity and an evolutionary approach to integration adopted, using good configuration control mechanisms, to identify any problems as soon as possible. Furthermore, a confidence checkpoint is incorporated into the project process, whereby the client is invited to a formal test of the skeletal control system early in its development.

On-Site testing

After delivery, a more complete IQ and OQ may be undertaken. A layered approach to testing, from electrical checks, through phase testing and on

to full recipe management, ensures that testing remains focused on a clearly defined area of functionality. Having verified that the appropriate hardware and software is present (via the IQ), the batch control system behaves as specified (via the OQ), the capability of the combined plant equipment and control system to consistently produce manufactured product within specification must be confirmed via a Performance Qualification (PQ).

Validation reporting

Validation reporting summarises the results of the validation effort during the project and relates directly to the validation planning phase. The validation report signals the completion of the development project.

Ongoing support and operational compliance

During the operational phase of the lifecycle, ongoing support and maintenance are required to ensure that malfunctions are identified and rectified. This is often managed under a service level agreement which provides a contractual basis for the agreed level of support. Factors helping these activities include:

- Minimising bespoke software.
- Good design in terms of the software engineering principles of high cohesion and low coupling.
- Rapid configuration tools and project database. These reduce the configuration effort and elapsed time and also help to enforce a pattern or structure on the configuration which eases the learning curve for support staff.
- Implementation rules. One of the goals of the implementation rules is to minimise style variations between software developers and even between the same developer on different occasions. The use of autocode generators can be of additional assistance.

Our project methodology satisfies the needs of prospective validation. Careful planning of the validation strategy can also bring significant savings when recipe changes and plant modifications are implemented during the system's operational phase of its lifecycle.

Decommission system

All systems must eventually be de-

ommissioned. The age of the control system hardware and the difficulty of obtaining spares and maintenance over can lead to poor availability and the risk of complete failure. This may justify an upgrade to the control system or replacement with an entirely new system, starting another iteration of the validation lifecycle.

Key points

We have described three major issues involved in pharmaceutical batch control system projects:

Timescales.

Standards.

Good Manufacturing Practice (GMP).

To satisfy these challenges, a fit for purpose project methodology and excellent project management are required to ensure that the right solution is delivered at the right time. Of particular concern are:

Validation lifecycle: the project must be managed according to a lifecycle which satisfies GMP, ensuring that documentary evidence of specification, test, change control, etc. is gathered.

Design reviews: key to ensuring that the right solution is delivered, the proposed design is subjected to walk-through with the operations and control system staff. This can be a time consuming exercise and must be carefully managed to maintain focus; the alternative would be to risk development of a flawed control system.

Cutover philosophy: in a project to replace an existing batch control system, the switch from the existing to the new system must be considered early in the project. The choice between big bang and a phased cutover has major impact on project timings and the high level design of the replacement system.

S88 analysis: whilst the S88 standard defines consistent terminology and models which promote flexibility, a method for its application is required. The analysis must combine equipment and product oriented viewpoints to produce a solution which satisfies current and future process needs.

Appropriate technology: the use of an S88-aware batch control system platform can significantly benefit both timescales and technical elegance by lifting implementation effort from development of framework functionality to configuration. Rapid configuration tools and a project database speed up the configuration activities.

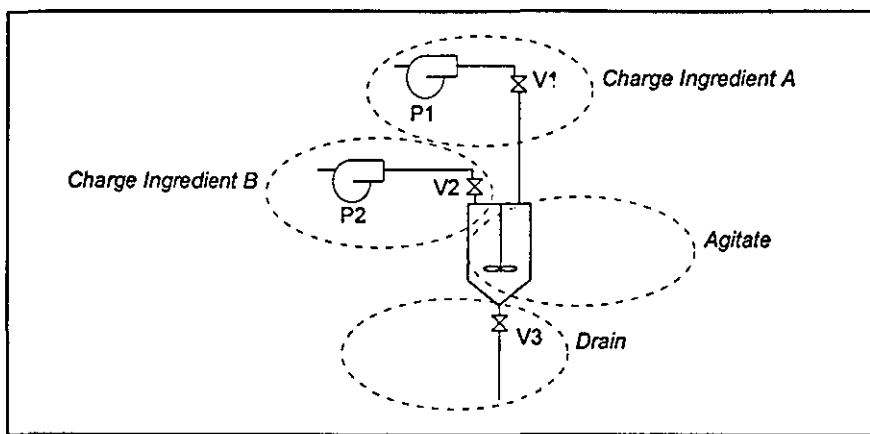


Fig 2 possible partitionings of control modules

Conclusions

This paper has outlined some of the issues which must be addressed during a pharmaceutical batch control system project in a minimum timescale. Key elements of a project methodology to manage these issues have been described by relating them to the phases of a validation lifecycle. The methodology has been applied to a number of projects, often with minimum timescales and short plant shut-down windows for control system changeover.

Appendix One

S88 - What Is It?

In 1989 the ISA established committee number 88 (SP88) to produce a set of terms and models for batch control which would be applicable from the most complex to the simplest batch process, whether fully automated or entirely manually operated.

S88.01 is a relatively new standard offering a common terminology and models for batch control. One benefit of the standard is to focus analysis on the production of a highly flexible automation system. This ensures that capabilities of the plant can be fully utilised to meet market demands for variations of existing products and for the manufacture of additional products. The S88 analysis separates the modelling of the plant equipment from the definition of recipes.

However the standard does not define how this analysis should be applied. In particular, many alternative partitionings of control modules (valves, motors, etc.) into equipment modules may be viable but

the standard does not specify how to select the most appropriate partitioning. For example, in fig 2, the control modules have been grouped into equipment modules as shown by the dotted lines. This results in phases ChargeIngredientA, ChargeIngredientB, Agitate and Drain being available to the process chemist when configuring recipes. An alternative partitioning could have grouped P1, P2, V1 and V2 into an equipment module, resulting in phase ChargeIngredients.

References

- ¹ Brooks Jr, F.P. 1987. 'No Silver Bullet. Essence and Accidents of Software Engineering', *Computer*, 20 (4), 10-19.
- ² European Union Guide to Directive 91/356/EEC 1991. *European Commission Directive Laying Down the Principles of Good Manufacturing Practice for Medicinal Products for Human Use*.
- ³ Instrument Society of America 1995. *ANSI/ISA-S88.01-1995 Batch Control, Part 1: Models and Terminology*. North Carolina 27709.
- ⁴ UK GAMP Forum 1996. *Supplier Guide for Validation of Automation Systems in Pharmaceutical Manufacture, Version 2*. International Society of Pharmaceutical Engineers. The Hague, Netherlands.
- ⁵ US Code of Federal Regulations Title 21: Part 211, *Current Good Manufacturing Practice for Finished Pharmaceuticals*.
- ⁶ Wingate, G.A.S. 1997. *Validating Automated Manufacturing and Laboratory Applications. Putting Principles into Practice*. Interpharm Press, Illinois 60089, pp 31-32.

A Case History of the Implementation of an S88-Aware Batch Control System

presented at the
World Batch Forum

Authors Peter Hopkinson and Joe Hancock
Company Eutech
Address Belasis Hall Technology Park
Billingham
Cleveland
TS23 4YS
England
Telephone +44 (0) 1642 372000

KEY WORDS

Batch control, S88, SP88, Analysis, Design, Implementation

Abstract

This paper is a case history of a recent project to install an ISA S88 aware control system on to a new batch plant at an ICI Films (now DuPont Polyester) site in Scotland. A key element of the project was the essential use of S88 standard models and terminology to deliver a highly flexible control system enabling variations to the plants product portfolio to be easily accommodated.

The paper describes the user requirements and corresponding control system solution, focusing on a subset of the plant equipment and process, and describing the project constraints. Alternative hardware and software control system solutions were considered and the justification for the selection of an S88-aware batch control system platform is given. A particularly interesting aspect of the S88 analysis concerning a shared two-position four-port valve and the associated implementation is described. The subject of phase coupling is also considered, with particular reference to the maintainability of the control system.

The use of design reviews with the process, control system and operations experts are considered to be essential in ensuring that the true needs are reflected in the control system solution. The impact of S88 on this activity is considered.

The anticipated benefits are reviewed in the light of several months experience of operating the plant.

Introduction

As part of a major project to build a new film production plant at ICI Films' (now DuPont Polyester) Dumfries site, a requirement was identified for a batch plant to manufacture a feedstock additive slurry. The chemical process was reasonably well understood and Eutech was appointed as consultant, initially to propose the most appropriate control system solution and subsequently to deliver the solution. The control system project was initially highly timescale critical, starting in August 1996 with production trials scheduled for January 1997, although external factors delayed completion until April 1997.

The Plant and Process

Figure 1 shows a simplified section of the plant.

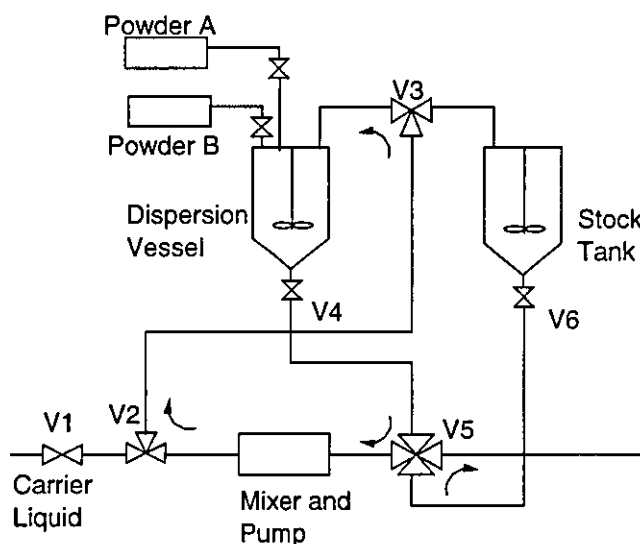


Figure 1 - Simplified Section of Masterbatch Plant

Figure 1 is a simplified section of the 110 I/O plant, showing the dispersion vessel unit, the stock tank unit and some of their associated equipment. Valves V2 and V3 are three port routing valves. Valve V2 determines the source of the feed - either slurry from the Mixer and Pump equipment or Carrier Liquid from valve V1. Valve V3 directs the liquid/slurry to either the Dispersion Vessel or the Stock Tank. All other valves are block valves, such as drain valves V4 and V6. Valve V5 is a four port, two position routing valve which can be in one of the following positions:

- Connecting dispersion vessel to mixer/pump
And connecting stock tank to a mill unit (off the right edge of the diagram).
- Connecting dispersion vessel to mill unit
And connecting stock tank to mixer/pump.

A typical use of this equipment would be as follows:

- A quantity of carrier liquid is charged into the dispersion vessel via valves V1, V2 and V3.
- A recirculation loop is established through the mixer and pump via valves V4, V5, V2 and V3.
- One of the two powders, Powder A or Powder B, is added at a controlled rate.
- The recirculation via the mixer and pump continues until the powder has been suitably dispersed in the carrier liquid to form a slurry.
- The contents of the dispersion vessel are milled via valves V4 and V5 (and other equipment not shown) to the Stock Tank.

Choice of Control System Platform

To simplify the control system requirements, the process designer offered the following restrictions on essential functionality:

- Only one batch would be in the plant at any one time.
- Small number of recipes, although some scope for the development of experimental recipes was required.

A number of alternative control system platforms were considered. Factors such as the I/O count, plant personnel familiarity with the technology and the need for cost effectiveness guided the solution towards the use of a PLC for the plant control with a SCADA (supervisory control and data acquisition) package for the operator interface. The recent availability of S88-aware batch software platforms introduced an alternative to the development of custom software in the PLC and SCADA to support concepts such as ingredients and recipes. We decided that the additional cost of the batch software platform was justified by the following:

- The short timescale of the project. Use of an additional software package to define the structure and framework would save development effort.
- The elegance of the solution. The leading SCADA packages directly support concepts such as alarms and continuous data logging and typically offer scripting facilities for application-specific functionality. But batch control software packages explicitly support the concepts we were manipulating, such as recipes, so the solution would be easier to support and maintain.
- Support for S88 models and terminology. ANSI/ISA-S88.01-1995 is a standard for batch control models and terminology. The batch software platform directly supports the spirit and detail of the S88.01 standard.

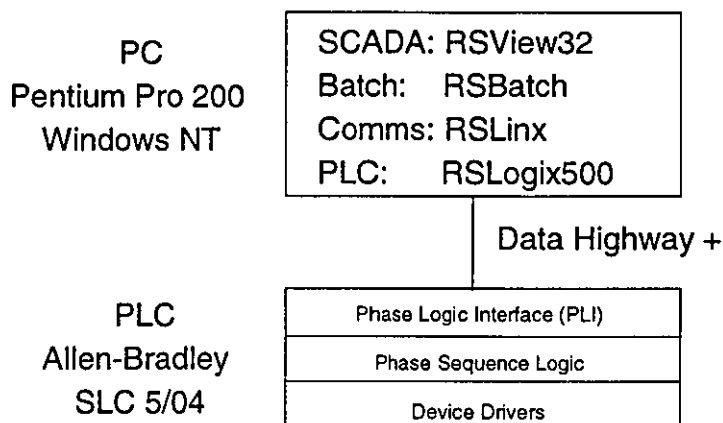


Figure 2 - Hardware and Software Architecture

Figure 2 shows the PLC and PC hardware platforms adopted and shows the Data Highway Plus network over which they communicate.

RSView32 running on the PC provides the operator interface to the control system and RSBatch is the S88-aware batch control software. RSBatch offers its own operator interface displays but, in this application, we chose to use RSView32 displays for batch control in addition to the usual SCADA functionality in order to keep the operator interface as simple and consistent as possible.

S88 Analysis

One of the benefits of the S88 standard is the emphasis on separating what the plant equipment is capable of doing from how it is controlled to make a batch of a particular product. This helps to ensure that a flexible batch control system is developed.

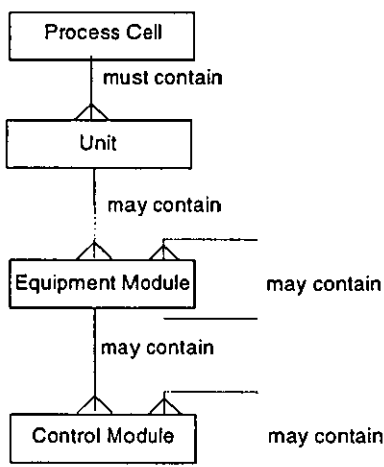


Figure 3 - S88 Physical Model

The physical model in figure 3 shows control modules as the lowest level in the hierarchy followed by equipment modules. S88 defines these as follows:

- | | |
|------------------|---|
| control module | the lowest level grouping of equipment in the physical model that can carry out basic control. |
| equipment module | a functional group of equipment that can carry out a finite number of specific minor processing activities. |

Referring to the plant schematic shown in figure 1, each valve and each motor should be considered to be control modules. The powder A addition system, consisting of its conveyor system and valve, satisfy the definition for an equipment module.

But how should the other control modules be partitioned into equipment modules? The S88 standard appears to offer minimal guidance in this respect so we used our experience with object-oriented software development, considering S88 equipment module partitioning to be analogous to the identification of object boundaries. A simplistic approach might consider that any control modules which are physically connected ought to form part of the same equipment module. Taking this approach, valves V1 to V6 would form part of the same equipment module. This would compromise the goal of producing an intuitive model of the plant because it would obscure the ability of the dispersion vessel and the stock tank to feed the mixer and the mill concurrently via valve V5. An alternative approach would be to define several equipment modules but this has the disadvantage of introducing dependencies (or "coupling" to use a software engineering term) between the control of the equipment modules. Our solution was to consider that a control module belongs to an existing equipment module unless it can undertake a process-oriented task independently of the equipment module. This ensures maximum flexibility with minimum coupling.

Although the plant schematic in figure 1 appears at first sight to be very simple, in practice there exist several real and difficult issues, not least of which was the control module partitioning mentioned above, to be resolved during the S88 analysis.

A poor partitioning could easily result in a solution which:

- Is difficult to support and enhance.
- Fails to exploit the inherent flexibility of the plant.
- Requires the control system developers to assist the process specialists when developing new recipes.

Each equipment module has an associated phase which defines the control of the equipment in terms of:

- Sequence logic.
S88 specifies that the phase can be in one of a number of predefined states. Separate sequence logic is therefore specified for states: running, holding, restarting, stopping and aborting.
- Parameters which are downloaded from the recipe to the phase, such as a target quantity to charge.

- Reports uploaded from the phase, such as the actual quantity charged.

Referring to the diagram of the hardware and software architecture in figure 2, the PLC is seen to have three levels of software, of which the phase sequence logic is the middle level. The phase logic interface (PLI) is a layer of software supplied by Rockwell which provides the interface to the S88 batch software running on the PC. The device drivers encapsulate the control of the control modules. A typical flow of command would therefore be as follows:

- The batch software in the PC instructs a phase to run by sending a command to the PLI.
- The PLI communicates with the phase sequence logic, such that the phase's running logic starts.
- The phase sequence logic communicates with device drivers.

Implementation

It is most important that the design representation facilitates effective review and provides the basis for support and maintenance. The device drivers and associated permissive logic were expressed using the easily understood graphical representation of boolean logic gates. In keeping with the S88.01, the sequence logic was expressed in the form of sequential function charts. The partitioning into phase "building blocks" enables meaningful discussion to take place regarding the scope of recipe flexibility. The well-defined software phase boundaries and graphical definitions of sequence logic help to ensure that all parties to the review have a common understanding. The use of sequential function charts ensures that the sequence logic specifications are simple to understand but rigorous in definition.

A set of rules was defined for translating the design into ladder logic code. The rules help to minimise style variations between individuals and even between different modules coded by the same individual. The benefit is felt in terms of ease of review and maintainability.

Project Process

The requirements of the sequence control were initially expressed in the form of a text document describing the overall processing required. This was supplied by the process engineer at the start of the project. The project process included defined review points so that the evolving design could be evaluated against the original specification. Early reviews with process, control system and operations personnel ensured a common understanding of the plant operating constraints and process requirements. Subsequent reviews confirmed that the design satisfied the requirements. The implementation team could then develop the code and configure the software packages, using reviews such as code walkthroughs to verify that the implementation corresponded to the design. System acceptance test procedures closed the loop by defining the tests to demonstrate that the delivered system provided the functionality agreed earlier in the project.

Conclusions

S88 encourages control system flexibility matching the plant equipment flexibility in order to prevent unnecessary operating constraints imposed by the control system alone. This flexibility was utilised

shortly after the plant began beneficial operation:

- A requirement arose to exploit the concurrency of operation available in the plant, violating the initially offered process requirement restricting the plant to serial batches. To increase throughput, a dispersed slurry would be transferred to the stock tank such that the milling could be done from the stock tank whilst the next batch was being prepared in the dispersion vessel. As a result of adopting S88, new recipes were easily developed to satisfy this requirement, with no modifications necessary to PLC software.
- The initial requirement defined a small number of recipes. But having installed the system, a number of alternative cleaning recipes were designed, by the client, to satisfy different needs. Once again, the flexibility inherent in the S88 analysis, along with the standard configuration facilities offered by the batch control system platform, ensured that the new recipes were rapidly and correctly configured.
- The process expert has edited existing recipes and designed new recipes without reference to control system development personnel.
- As anticipated, the software platform functionality over and above the user's initial requirements provided opportunities for rapid and minimum cost system enhancement. An example of this is the batch logging capability, providing a batch history for post-batch analyses.

Consequently, on balance, it has been shown that the desired benefits arising from the application of S88 and the use of an S88-aware platform have largely been met. Although reservations were expressed regarding the adoption of a new approach and platform software, the project ran to time and budget and significant learning has been achieved relating to an S88 methodology and its implementation on current platforms. This learning is being employed in subsequent batch process control projects.

However whilst it is clearly much too early to have evidence on the long-term supportability of the control system, the current indications are very positive.

Our ongoing experience with batch control applications suggests that, with appropriate interpretation, S88.01 and S88.01-aware control system platforms can offer significant benefits for some types of application but may be marginal for others.

Bibliography

Peter Hopkinson is a process automation consultant with Eutech and can be contacted at Eutech, Belasis Hall Technology Park, Billingham, Cleveland, TS23 4YS England, telephone +44 (0) 1642 372000, fax +44 (0) 1642 372166, email Peter.Hopkinson@eutech.com.

Joe Hancock is a business manager with Eutech and can be contacted at Eutech, Brunner House, Winnington, Cheshire, CW8 4FN England, telephone +44 (0) 1606 708888, fax +44 (0) 1606 704733, email Joe.Hancock@eutech.com.

The Application of the S88 Batch Control Standard in the Pharmaceutical Industry

Niels Haxthausen, Novo Nordisk Engineering

and

Peter Hopkinson, Eutech Engineering

Wednesday 4th March 1998

Introduction

Batch Automation

Most pharmaceutical production processes are batch processes. This is due partly to the nature of the processes - and partly to the particular pharmaceutical requirements for traceability and containment.

Batch processes involve a number of challenges:

- You have to manage the sequential nature of batch processes
- You frequently have to manage several simultaneous batches - with the necessary coordination of processes and separation of material.
- Batch processes have the potential of producing a number of different products in the same facility - in which case you have to manage the different product specific behaviour

These challenges have posed a bottleneck - or complication in automating batch production. To date we have seen many different approaches - and many different levels of automation - in batch plants. Automation has the potential of maximising the plant throughput and yield and quality of the product. The control system can automatically sequence the plant equipment to exert the necessary control over the batch and consequently avoid unnecessary delays which might otherwise be introduced. Because each batch is subjected to the same control, batch to batch consistency is also improved. Batch production can be documented electronically by the control system. However it is often inappropriate to fully automate a plant. For example, dosing and sampling checkpoints in the process may be candidates for manual intervention. But even in these cases we see an increasing use of electronic execution support and documentation (Manufacturing Execution Systems, Electronic Batch Record Systems).

An international standard for batch control: S88.01 (IEC61512)

Over the last couple of years the S88.01 standard has been adopted as the dominating international standard covering batch control has been developed. S88.01 has been developed as an American standard - with international influence. By beginning of this year the standard has been adopted as an official international standard by the International Electrotechnical Committee, as "IEC 61512".

The S88 standard has been very widely accepted - practically all batch control vendors have aligned their offerings with the standard - and it is becoming the predominant platform for descriptions and implementations of batch control. Some of the key features of the standard are:

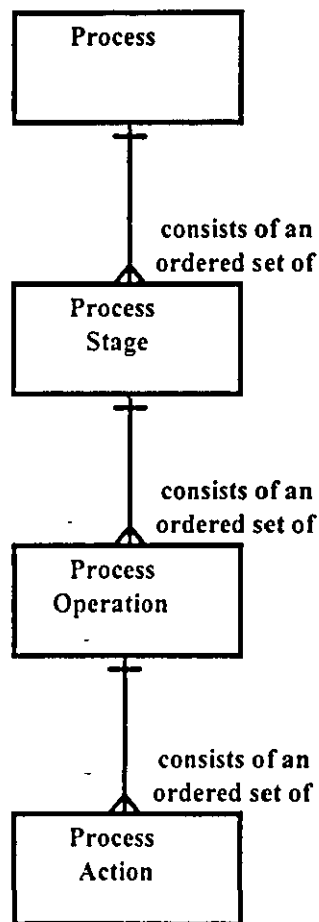
- A standard terminology in the field of batch control
- Modular structure of process and equipment
- Product independent plant design
- Equipment independent recipes
- A structured way of operating batch plants

S88.01 introduction

Basic models: process and equipment

The S88 standard recognizes 2 distinct dimensions when describing batch production: The process that the batch - understood as material - undergoes - and the physical plant in which it is produced. The 2 dimensions are represented by models, where the process and the plant is decomposed hierarchically into ever smaller parts.

The process model



A batch process leads to the production of finite quantities of material (a batch) by subjecting quantities of input materials to a defined order of processing actions using one or more pieces of equipment. The subdivisions of a batch process can be organized in a hierarchical fashion as shown in Figure 1.

Process stages

The process consists of one or more process stages which are organized as an ordered set, which can be serial, parallel, or both. A process stage is a part of a process that usually operates independently from other process stages.

Process operations

Each process stage consists of an ordered set of one or more process operations. Process operations represent major processing activities. A process operation usually results in a chemical or physical change in the material being processed.

Process actions

Each process operation can be subdivided into an ordered set of one or more process actions that carry out the processing required by the process operation. Process actions describe minor processing activities that are combined to make up a process operation.

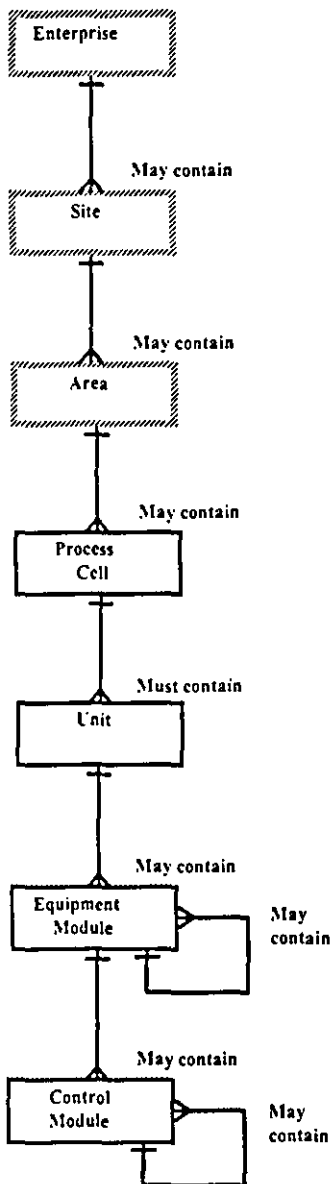
Physical model

The equipment we use to produce a batch is the central object of the S88.01 model. Equipment in this context means the vessels, pipes, valves etc. which physically contain or manipulate the batch - including the control that enables this to happen.

The equipment is the tool used to manipulate the batch - physically and chemically. The actions of the equipment therefore is the essence of batch. The modeling of equipment in the control system is therefore the key to any decent design of a batch control system.

The model has seven levels, starting at the top with an enterprise, a site, and an area. These three levels are frequently defined by business considerations and are not modeled further in this document.

The lower four levels of this model refer to specific equipment entities. An equipment entity is a collection of physical processing and control equipment grouped together for a specific purpose. The lower levels in the model are specific to technically defined and bounded groupings of equipment. The four lower equipment levels (process cells, units, equipment modules, and control modules) are defined by engineering activities. During these engineering activities, the equipment at lower levels is grouped together to form a new higher level equipment grouping. This is done to simplify operation of that equipment by treating it as a single larger piece of equipment. Once created, the equipment cannot be split up except by re-engineering the equipment in that level.



Process cell level

A process cell is a logical grouping of equipment that includes the equipment required for production of one or more batches. The equipment actually used or expected to be used by a batch is called the path.

Unit level

The unit level is critical to the S88 concepts. A unit typically combines all necessary physical processing and control equipment required to perform a processing stage. A unit does not operate on more than one batch at the same time. Physically, it includes or can acquire the services of all logically related equipment necessary to complete the major processing task(s) required of it. Units operate relatively independently of each other. A unit frequently contains or operates on a complete batch of material at some point in the processing sequence of that batch. However, in other circumstances it may contain or operate on only a portion of a batch.

Equipment module level

An equipment module can carry out a finite number of specific minor processing activities such as dosing and weighing. It combines all necessary physical processing and control equipment required to perform those activities. An equipment module may be part of a unit or a stand-alone equipment grouping within a process cell. If engineered as a stand-alone equipment grouping, it can be an exclusive-use resource or a shared-use resource.

Control module level

A control module is typically a collection of sensors, actuators, other control modules, and associated processing equipment that, from the point of view of control, is operated as a single entity. A control module can also be made up of other control modules. For example, a header control module could be defined as a combination of several on/off automatic block valve control modules.

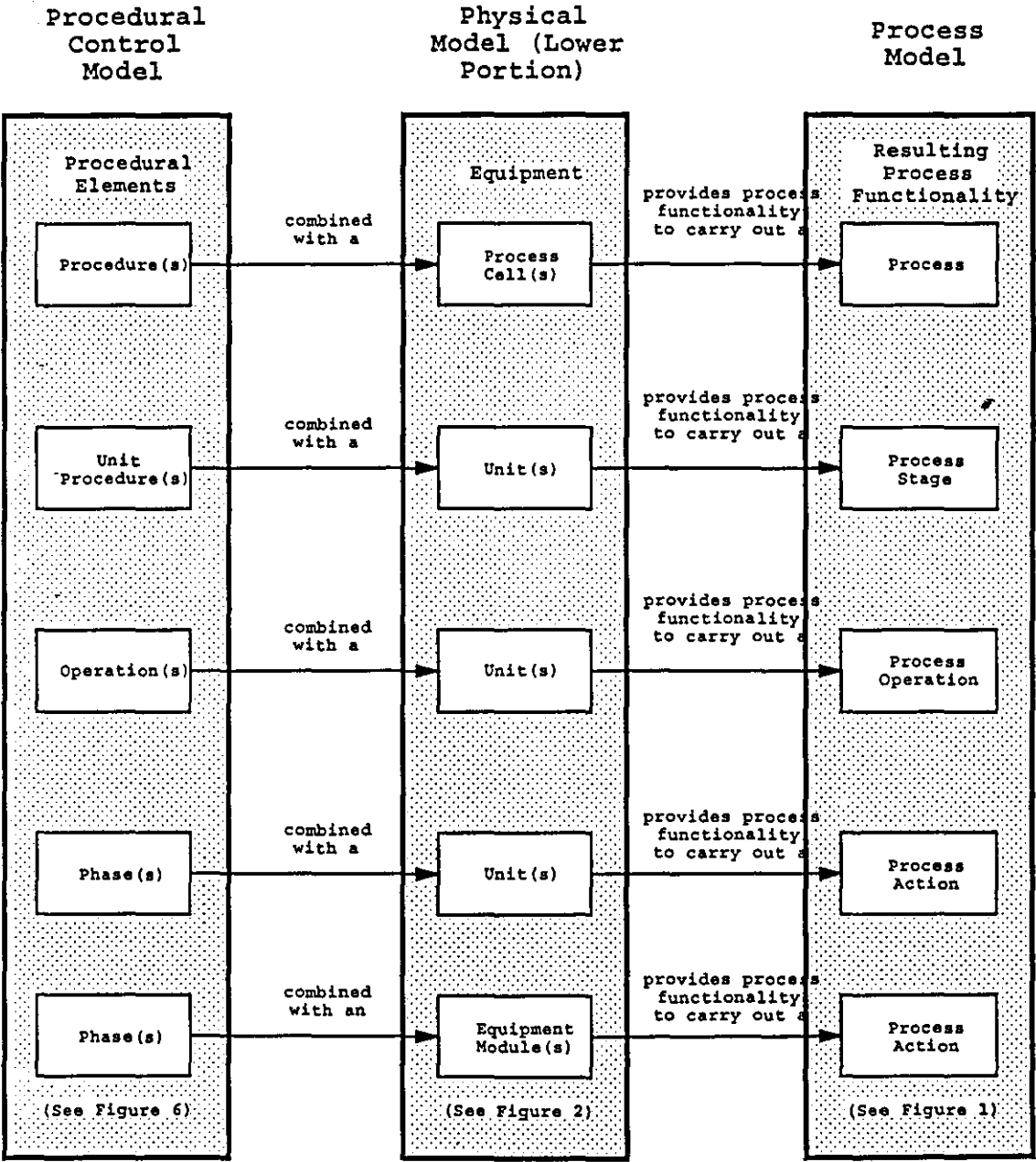
Both the process model and the physical model may be collapsed or expanded - i.e. levels may be excluded in particular applications or new levels may be introduced.

Procedural control

The main purpose of the S88 standard is to describe the interaction of the 2 dimensions - i.e how equipment is controlled in order to make the processes happen in a well structured way that optimally caters with product demands - including the demands for product variations and equipment utilization.

In order to make batch processes happen in physical equipment you have to execute procedural control. Procedural control is the kind of control that involves a sequence of commands to the equipment making it perform specific tasks - e.g. batch processes.

Procedural control is modeled hierarchically in a way very similar to the process model as procedural elements that are hierarchically decomposed. The figure below illustrates the linking of the process model and the physical model through exercising procedural control.



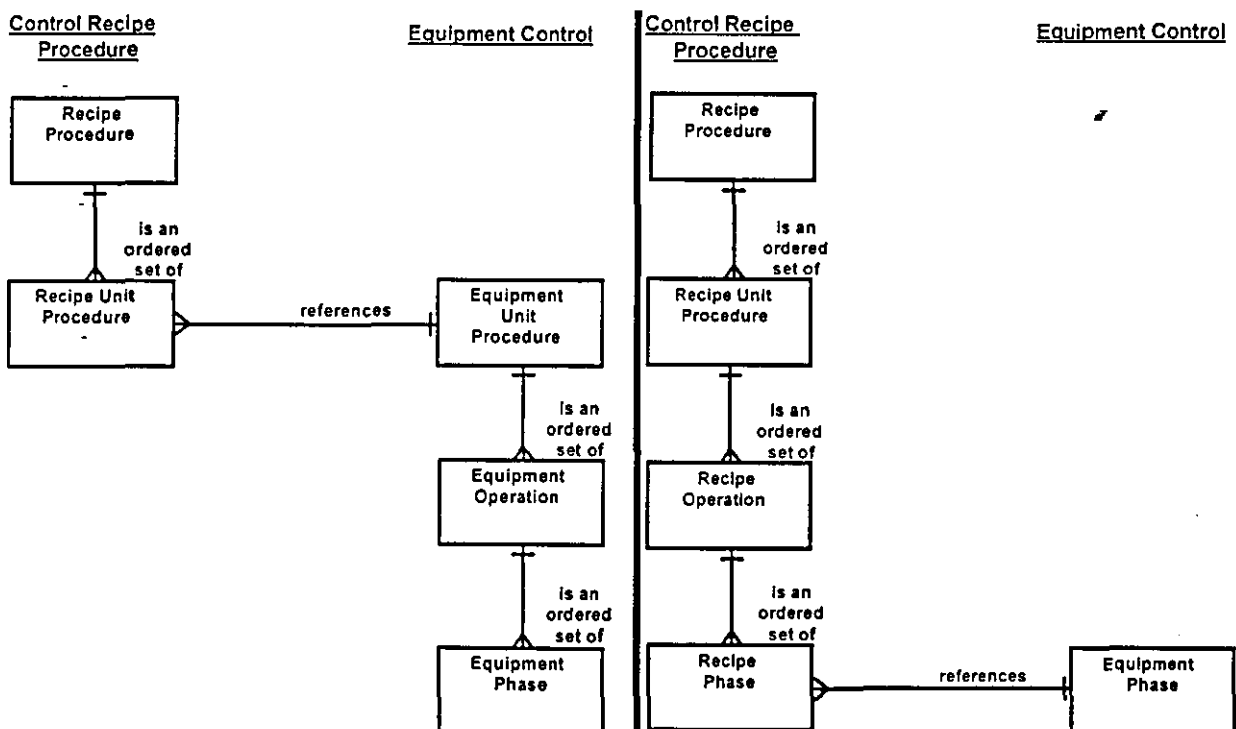
Recipes

The procedural control may be "hard coded" with the equipment - typically if we are dealing with a single purpose plant. It may also be defined in recipes - if it varies from product to product. S88.01 defines recipes as "The necessary set of information that uniquely defines the production requirements for a specific product". The recipe attached to a specific batch is called a control recipe. A control recipe is derived from a master recipe, that includes the specification of how to produce a specific product in a process cell.

How much information is included in the recipe depends on the extent of product variations - and the intelligence of the equipment. If the equipment is preconfigured to support specific products the recipe may be reduced to a simple selector. If the equipment on the other hand is very generic - e.g. a collection of multi purpose equipment with basically any transfers possible - the recipe will have to specify the procedure to a great detail.

A recipe will include some level of procedural specification - and it will include associated parameters (a formula). When executed the recipe procedural element will activate equivalent equipment procedural elements in the chosen units - and hand over the relevant parameters.

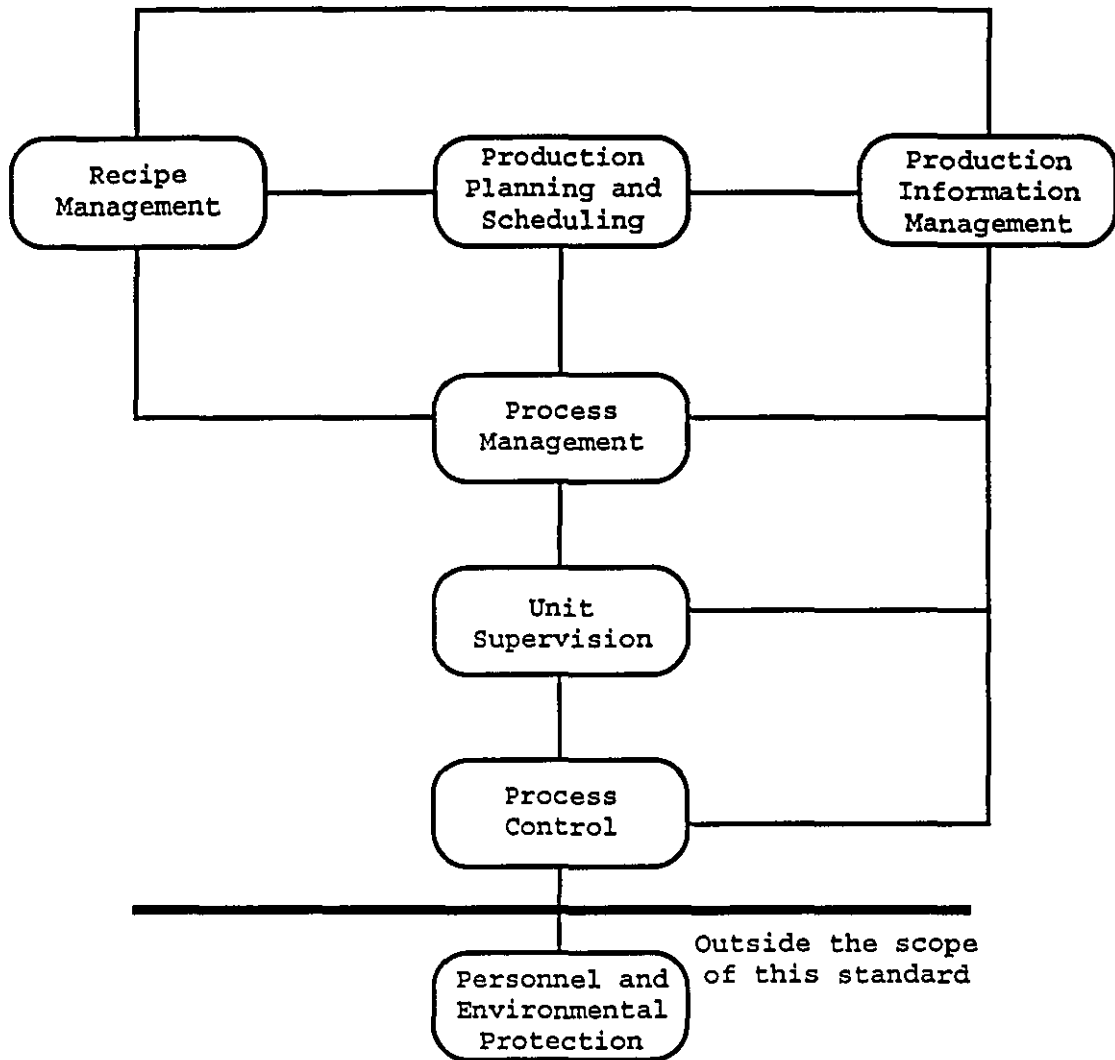
The 2 figures below show different ways the referencing can be done. In the first case the recipe basically just specifies the sequence of unit procedures - leaving the execution order of the phases up to the equipment. In the second case the recipe includes specification and parameters of the individual phases.



The concept of a product specific recipe that sequences equipment specific procedural logic opens up for a lot of flexibility:

- Different recipes (/products) can use the same equipment and procedural logic in different combinations.
- The highest level procedure - the sequence of unit procedures - allows for selection of different paths through a plant

- The same recipe may be executed on different equipment - as long as the equipment can match the recipe procedural requirements (e.g. the same function and parameters).
- S88 control activity model**



The control activities shown relate to real needs in a batch manufacturing environment. The need to have control functions that can manage general, site, and master recipes implies a need for the Recipe Management control activity. Production of batches must occur within a time domain that is planned and subsequently carried out. Production Planning and Scheduling is the control activity where these control functions are discussed. Various types of production information must be available, and the collection and storage of batch history is a necessity. The Production Information Management control activity in the model covers these control functions.

Control recipes must be generated, batches must be initiated and supervised, unit activities require coordination, and logs and reports must be generated. These control functions fall under the Process Management control activity in the model. There are many control functions needed at the Unit Supervision control activity level. For example, there is a need to allocate resources, to supervise the execution of procedural elements, and to coordinate activities

taking place at the Process Control level. In Process Control, control functions are discussed that deal directly with equipment actions such as the need to implement control functions using regulating devices and/or state-oriented devices.

S88 and Good Engineering practice

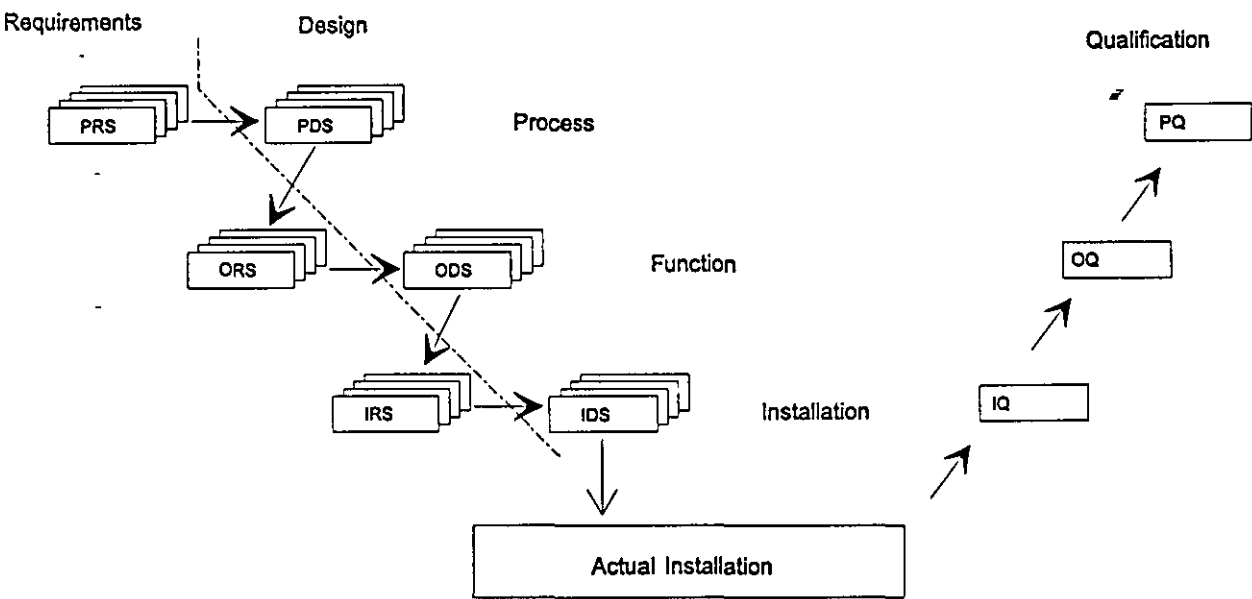
The potential for failure in a project to implement a batch control system in the pharmaceutical industry is high. The business drivers of project timing, regulatory compliance, cost and demand for more product variants and improved product quality require critical management of an appropriate project methodology including a good structuring that combines robustness and flexibility. The S88 provides such structuring - both due to the use of S88 concepts in the engineering and due to vendor provided tools that support that.

The S88 batch control standard provides an excellent platform for structuring this problem - not only from a control perspective, but from the perspective of the multi-disciplinary, complete specification and test of a plant. The following life-cycle models and terminology are therefore in no way control-specific, but cover the complete installation - including mechanical equipment and installations and manual functions.

Specification and Qualification

In the following the relationship between SP88 and good engineering practice - in particular requirements and design at the process and function levels - will be expanded.

The engineering of pharmaceutical plants is illustrated in many different life-cycle models. One is the "V" model - which shows the increasing levels of detail and the timeflow:



Down the left leg we have the "waterfall" situation: process requirements are met by a process design - from which functional requirements to the individual units or parts of the plant are derived. These functional requirements are met by a functional design - including PI-D's and control strategies - from which the detailed requirement to the individual components, the piping, the electrical and the SW modules are derived.

By systematically reviewing the design against the requirements, and the installation

against the design (by quality control during ordering, at the vendors, at reception and at installation) one can establish an alternative track to verify that the requirements are met - a track that follows the engineering process concurrently and thus has better access to the relevant information than what can be obtained in the final installation.

The end result is improved quality control and a faster track for the project. **But** it requires a very distinct set of requirement documents, and a good match between requirements and design - so that the review of the design against requirements may be done in a controlled fashion. Here the SP88-structures become critical - in particular at the functional level.

Process Requirements

The process requirements may be expressed in the terms of the SP88 process model - a hierarchical breakdown of processes into stages, operations and actions. This model is in no way revolutionary - many other similar approaches have been used in process industries for years. By applying the SP88 model it is however ensured, that the process requirements may be expressed in a modular fashion that fits with the implementation of units and of procedural logic. This is the first requirement for a traceability - that design and requirements are mappable - so that one avoids having to check any design element (or even worse - modification) against the complete set of requirements.

Process design

S88 plant structure - product independence

In the process design you select the key processing equipment and dimension the plant. Furthermore you determine the logistics of the plant. Do you want a single line - do you want a lot of cross coupling - or do you want free movement of material. Should the individual parts or lines be single purpose or multipurpose? Should it run 24 hours - automatically or manually?

S88.01 is a control standard - but the models introduced provide a very good foundation or the overall process design. The concepts of equipment entities - in particular units - can be used to structure the plant. And the recipe concept enables the process designer to create a generic plant and let the recipe control the specific equipment selection.

The following criteria should be taken into consideration in the segmentation of the process cell into units:

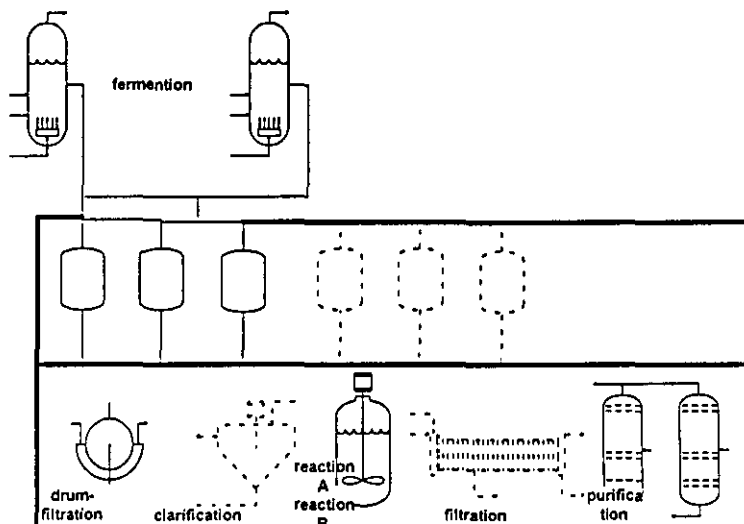
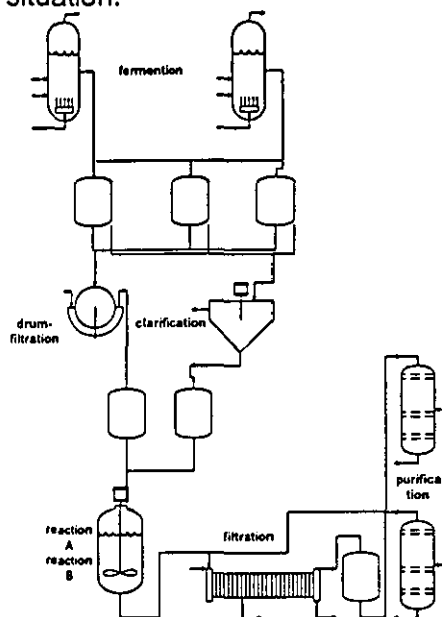
- a good fit with the segmentation of the process description (units that match process stages)
- good adaptation to the flexibility and timing requirements
- avoiding bottlenecks or complex interlocks associated to transfers
- a good functional batch separation facilitating the association of information with the individual batches.

Example

The process cell shown is capable of executing the 2 processes described above. Utility and raw material supply equipment and the volumes and capacities should be added - resulting in a flow-diagram with at least all material flows. Further the overall control strategy and level of automation should be described, as well as cleaning processes, critical material specifications etc.

The process cell design shown in the first figure takes into account the fact that the fermentation process takes up more time than the reaction. The fermentation and reactor units may be used for both products - buffer vessels ensure batch separation and remove scheduling constraints.

The second figure illustrates how the process cell could be made more generic - allowing for other (yet unknown) combinations of unit procedures - and for expansion. The point of good S88 design is - that if you make the units sufficiently independent and generic - and base your control on an S88 aware recipe concept - it is not more complicated to manage the more flexible situation.



Functional Requirements

The functional requirements specify what the equipment should be able to do.

Functional requirement specifications would typically be organized per unit. In addition functional requirements on the process cell level may be specified (e.g. requirements to the process management functions), and general requirements that apply to all units (e.g. unit modes and states, requirements for manual operation of devices etc.) should be specified.

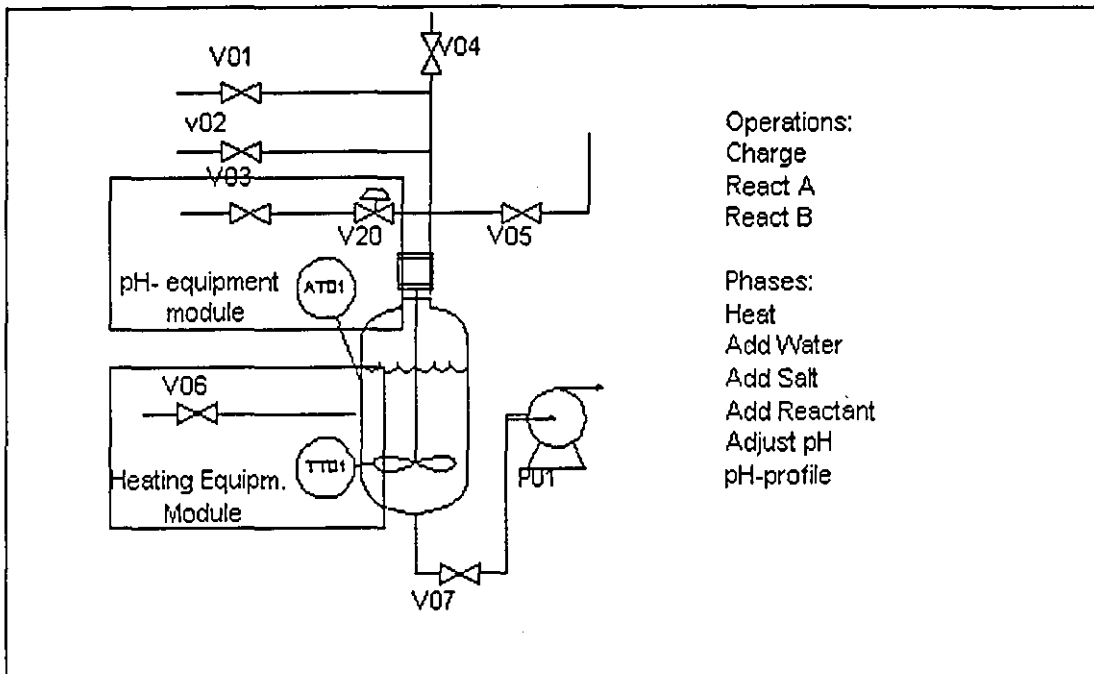
The functional requirements should be expressed in S88 terms: unit procedures, operations and phases, which the unit should be able to execute. (This matches nicely to the concept of recipes - what recipes should the unit be able to execute). In this way you get a clear structure to the requirement - and what's more - a structure that maps well to the S88 based design of the control system.

The functional requirement specification should include:

- the names of the procedural elements, a brief description of their required functionality with specific measurable criteria (sequencing, time requirements etc.) and the formula parameters with allowable ranges of formula values
- materials to be handled (volumes, flows, specifications)
- constraints that should be imposed - in terms of batch separation and cleaning requirements, allowable recipes, allowable manual intervention, process

- interlocks
- critical alarms with associated alarm limits
- measurement and data collection requirements

The recipes themselves are not considered part of the plant - and are therefore not described as part of the design process. Changes in recipes - or creation of new recipes - should ideally involve only process qualification activities - not revalidation of SW and equipment.



Product independent Functional Design

The functional design is typically made up of a PI-D, some overall specifications of lay-out and materials and associated descriptions of the control and measurement functions implemented.

In the functional design we have to describe the actual components that provide the unit with the capability of performing procedural functions.

The functional design of the unit should follow the S88 structures - using equipment modules as tools for structuring the more complex units. The required procedural functions should be implemented as equipment phases, operations and unit procedures - preferably 1:1. Note that some of the phases may be implemented through equipment modules - where as others may be associated with the unit as a whole.

Using a strict S88 structure makes it simple to relate the individual elements of the functional requirements directly to the application SW building blocks - thus making the test easier - and paving the way for a manageable change control, where the ripple-effect on modifications can be contained. When technology

matures it will even become possible to make an electronic association between a phase "object", the requirement and design specification documents covering it, the test status and results, and its change history. This will significantly reduce the cost of change management.

Design recommendations

The system specification should describe the proposed solution using S88 terminology, separating the capabilities of the plant equipment from its use in product recipes. The grouping of control modules (such as valves and pumps) into equipment modules and the definition of the corresponding phase control logic is particularly important. The system specification must also state whether an S88-aware software platform is to be used as this will also significantly affect the solution's capabilities and limitations. In a project to replace an existing control system, the S88 analysis to be performed must take account of deficiencies in the documentation likely to be available and the outputs required from the analysis.

The S88 analysis is critical to the design of the control system. There may be several different groupings of control modules into equipment modules to be considered, potentially offering different granularity of phase logic. Choosing fewer phases limits the flexibility of the control system but there are disadvantages with partitioning the plant into too many equipment modules, such as:

- Unnecessary coupling between the phase logic, bringing support and maintenance implications.
- Additional responsibility will be placed on the recipe designer to ensure that dependent phases are appropriately configured in the recipe. Potentially more complex failure mode analyses required, particularly where the co-ordination may span control devices and communications networks.
- Unnecessary batch software platform licence costs.

Pre-S88 implementations may have typically addressed the normal operation of the plant, with abnormal occurrences being recognised and dealt with by plant operational staff. But significant benefits to product quality and yield may be available if abnormal behaviour is considered at the control system design stage and the necessary mechanisms for bringing the plant to a controlled quiescent state are specified, along with the necessary processing to re-establish the process.

Design Reviews

A fit for purpose project methodology is essential to ensure that risks are identified and addressed. Of particular importance is the need to ensure that the functional requirements and the proposed solution are fully understood.

Design reviews are the key to this aspect of the project. The intended control strategy is walked through with the operations and control systems personnel in a Control and Operability review. The use of common S88 terminology and models helps to promote a common understanding. Adopting S88 helps to focus attention on functional modules of limited scope, making the review process more manageable. A diagrammatic method for specifying phase logic sequences, using notations like state transition diagrams or sequential function charts, is preferred because it presents a precise definition in a form which is simple to understand and review.

S88 based control systems in pharmaceutical applications

Automation of the process, by means of a batch control system, can help to maximise the plant throughput and yield and quality of the product. The control system automatically sequences the plant equipment to exert the necessary control over the batch and consequently avoids unnecessary delays which might otherwise be introduced. Because each batch is subjected to the same control, batch to batch consistency is also improved.

However it is often inappropriate to fully automate a plant. For example, dosing and sampling checkpoints in the process may be candidates for manual intervention. But even in these cases S88 aware systems may be in place to guide, monitor and log the manual execution.

S88 aware batch execution systems

Practically all control systems targeted for the batch market provide some S88 aware batch execution functions. Over the last couple of years the market for batch execution systems has consolidated to a relatively small number of standard systems - working as independent packages on top of SCADA + PLC's - or embedded in DCS systems.

The S88 batch execution offerings typically provide a relatively loyal interpretation of the S88 concepts - in a prepackaged and relatively easy to use fashion.

However they frequently only offer one or a few of the options catered for in S88. For example the linking at different levels - and the collapsibility and expandability called for by the standard - is not provided by most standard solutions. This may result in relatively rigid solutions that might have a poor fit to the application requirements.

MES/EBRS systems

The manufacturing execution systems - or electronic batch recording systems - that are in use in an increasing number of pharmaceutical plants - are S88 aware only to a limited extent. This poses some problems - in part because the systems are missing some critical concepts - e.g. the relation between equipment procedural logic and recipes - and in part because of the resulting integration problems with lower level control functions. One is often faced with having to choose either an MES solution - which takes good care of the manual operations - or a batch execution system - that organises batch execution in processing equipment very well. However a merge of these worlds is underway - with MES's becoming more S88 aware and batch execution systems becoming better at material management and handling of manual operations.

Data exchange

S88 describes a framework for data-exchange with the surroundings - easing the use of integration tools, electronic signature etc. This is paramount to reducing the huge overhead of paper-related costs associated with pharmaceutical production - in particular in the area of batch documentation. A part 2 of the standard(S88.02) is under way that provides standardised frameworks and mechanisms for integration of different systems involved in batch control and batch production in general.

Experiences from S88 analysis, design and implementation

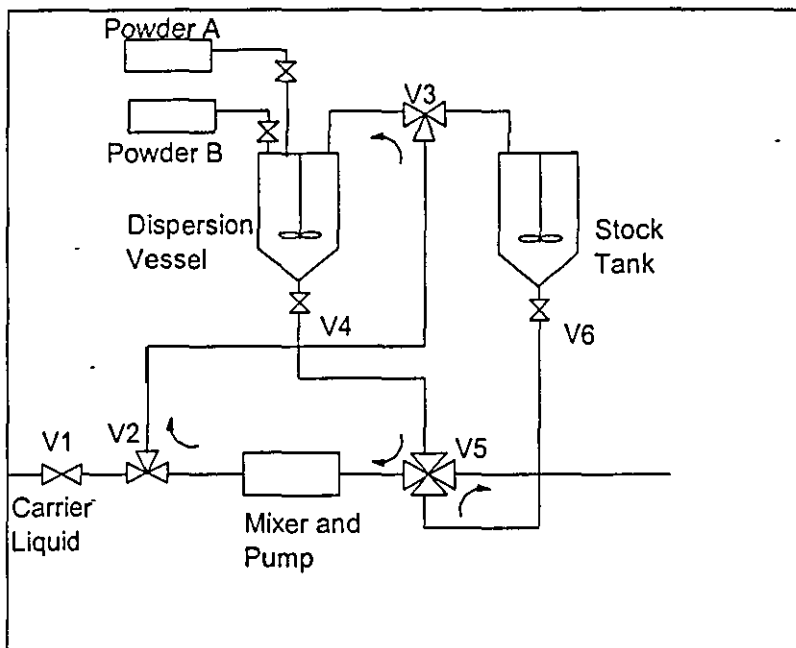
Challenges

The potential for failure in a project to implement a batch control system in the pharmaceutical industry is high. The business drivers of project timing, regulatory compliance, cost and demand for more product variants and improved product quality require critical management of an appropriate project methodology.

A Case History

This section describes a case history of a project to supply a batch control system for a new fine chemicals batch plant. The project decided to adopt the S88 standard to promote a common understanding and to encourage the development of a flexible control system which modeled the flexibility inherent in the plant design.

The Plant and Process



Simplified Section of Plant

Figure 1 shows a simplified section of a part of the plant containing two units - a dispersion vessel and a stock tank. There are three different types of valve:

- Valves V2 and V3 have three ports and are used to route product. Valve V3 determines whether product is routed to the dispersion vessel or the stock tank. Valve V2 sources either carrier liquid from valve V1 or slurry from the mixer and pump.
- Valve V5 is a four port, two position routing valve which can be in one of the following positions:
 - Connecting dispersion vessel to mixer/pump and connecting stock tank to a mill unit (off the right edge of the diagram).
 - Connecting dispersion vessel to mill unit and connecting stock tank to

mixer/pump.

- All other valves are block valves, such as drain valves V4 and V6.

A typical use of this equipment would be as follows:

- A quantity of carrier liquid is charged into the dispersion vessel via valves V1, V2 and V3.
- A recirculation loop is established through the mixer and pump via valves V4, V5, V2 and V3.
- One of the two powders, Powder A or Powder B, is added at a controlled rate.
- The recirculation via the mixer and pump continues until the powder has been suitably dispersed in the carrier liquid to form a slurry.
- The contents of the dispersion vessel are milled via valves V4 and V5 (and other equipment not shown) to the Stock Tank.

S88 Analysis

Referring to the plant schematic shown above, each valve and each motor should be considered to be control modules. The powder A addition system, consisting of its conveyor system and valve, satisfy the definition for an equipment module.

The product routing equipment is more difficult to partition into equipment modules. A simplistic approach might consider that any control modules which are physically connected ought to form part of the same equipment module. Taking this approach, valves V1 to V6 would form part of the same equipment module. This would compromise the goal of producing an intuitive model of the plant because it would obscure the ability of the dispersion vessel and the stock tank to feed the mixer and the mill concurrently via valve V5. An alternative approach would be to define several equipment modules but this has the disadvantage of introducing dependencies (or "coupling" to use a software engineering term) between the control of the equipment modules. The approach adopted was to consider that a control module belongs to an existing equipment module unless it can undertake a process-oriented task independently of the equipment module. This ensures maximum flexibility with minimum coupling.

Conclusions

The use of S88 models and terminology explicitly supported by an S88-aware software product helped to ensure that the control system did not impose unnecessary constraints on the flexibility of the plant. This flexibility was utilised shortly after the plant began beneficial operation:

- A requirement arose to exploit the concurrency of operation available in the plant, violating the initially offered process requirement restricting the plant to serial batches. To increase throughput, a dispersed slurry would be transferred to the stock tank such that the milling could be done from the stock tank whilst the next batch was being prepared in the dispersion vessel. As a result of adopting an S88-aware solution, new recipes were easily developed to satisfy this requirement, with no modifications necessary to phase sequence logic.
- The initial requirement defined a small number of recipes. But having installed the system, a number of alternative cleaning recipes were designed, by the client, to satisfy different needs. Once again, the flexibility inherent in the S88 analysis, along with the standard configuration facilities offered by the batch control system platform,

ensured that the new recipes were rapidly and correctly configured.

- The process expert has edited existing recipes and designed new recipes without reference to control system development personnel.
- As anticipated, the software platform functionality over and above the user's initial requirements provided opportunities for rapid and minimum cost system enhancement. An example of this is the batch logging capability, providing a batch history for post-batch analyses.

Additional Conclusions From Other Projects

Experience from batch control system projects in the pharmaceutical and other sectors suggests that with appropriate interpretation, S88.01 can offer significant benefits. Depending on the type of application, S88.01-aware control system platforms support the development of an elegant solution in minimum timescales.

References

ISA-S88.01, Batch Control, Part 1: Models and Terminology

ISBN: 1-55617-562-0

ISA

67 Alexander Drive

P. O. Box 12277

Research Triangle Park, NC

IEC 61512-1, Batch Control, Part 1: Models and Terminology

International Electrotechnical Committee, 1997

SP88 - The painkiller in validation

Niels Haxthausen

ISA transactions 34 (1995) 369-378

Bottlenecks in the batch integration - can standards help removing them?

Niels Haxthausen

World Batch Forum, 1998

A Case History of the Implementation of an S88-aware Batch Control System

Peter Hopkinson and Joe Hancock

World Batch Forum, 1998

Implementing S88 Batch Control Systems in the Pharmaceutical Industry

Peter Hopkinson, Mike Sonley, Guy Wingate

The Transactions of the Institute of Measurement and Control Special Issue on Validation Technologies

February 1998

The Authors:

Niels Haxthausen is a director of Process Automation with Novo Nordisk Engineering, Krogshøjvej 55, DK 2880 Bagsvaerd, Denmark

Telephone: +45 44422725, fax +45 44443777

Email: hax@novo.dk

Peter Hopkinson is a process automation consultant with Eutech,

Belasis Hall Technology Park, Billingham, Cleveland, TS23 4YS England,

Telephone +44 (0) 1642 372000, fax +44 (0) 1642 372166,

Email Peter.Hopkinson@eutech.com.

Appendix C Synect User Guides

This appendix contains the following Synect User Guides:

- Application Editor
- Compiler
- Analyzer
- STD Monitor
- Simulator
- ANSI C Code Generator
- Distributed Neuron C Generator
- Ladder Logic Generator

Synect

Application Editor User Guide

Version 1.4

**Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough
Cleveland, TS14 8JY
England**

**Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk**

© Copyright 1994, 1995, 1996, 1997 Hopkinson Computing Limited. All rights reserved.

Synect is a registered trademark of Hopkinson Computing Limited

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Information in this User Guide is subject to change without notice and does not represent a commitment on the part of Hopkinson Computing Limited.

The software described in this User Guide is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. No part of this User Guide may be reproduced or transmitted in any form or by any means, electronic or otherwise, including photocopying and recording, for any purpose, without the express written permission of Hopkinson Computing Limited.

Document History

10 June 1996

Re-issue reflecting Application Editor V1.1 - new cover sheet, contents, chapters 1, 3 & 8 and Appendix B.

28 October 1996

Re-issue reflecting Application Editor V1.2 - changes to cover sheet, contents, chapters 4, 5, 6 & 8.

30 April 1997

Re-issue reflecting Application Editor V1.3 - cover sheet only

13 May 1997

Re-issue reflecting Application Editor V1.4 - cover sheet only

Contents

Chapter 1	Introducing Synect	1
	User Interface	1
	The Method	1
	Synect Documentation	1
	The Tools	2
	Document Conventions	2
Chapter 2	Getting Started	3
	System Requirements	3
	Installation	3
	Starting the Application Editor	4
	On-Line Help	5
Chapter 3	Basic Concepts	7
	The Object Hierarchy	7
	Object interaction	7
	Messaging	8
	Interface with the controlled system	8
	Internal Events	9
	State Transition Diagram	10
	State	10
	Transition	11
Chapter 4	Create, Save and Open An Application	13
	Creating A New Application	13
	Opening An Existing Application	13
	Recover Mode	13
	Saving An Application	14
Chapter 5	Editing The Object Hierarchy	15
	Adding A New Root	15
	Adding A New Child	15
	Using Cut And Paste	15
	Zooming	16
	Renaming An Object	16
	Moving An Object	17
	Changing Font	17
Chapter 6	Editing An Object's External Interface	19

	Using Cut And Paste	20
	Moving The External Interface Text Strings	21
	Changing Font	21
	Specifying An Object's External Interface	21
	Parent Messages	25
	Real World Inputs and Outputs	25
	Child Messages	26
	Editing The List Of Names	26
	Rename vs. Delete + Add Functionality	27
	Changing Synchronous To/From Asynchronous	27
Chapter 7	Editing An Object's Internal Events	29
	Editing An Object's Internal Commands	29
	Editing An Object's Variables	29
	Adding a Variable	29
	Changing the Min, Max or Initial Value	30
	Deleting a Variable	30
	Renaming a Variable	30
	Editing a Variable's Tests and Operations	30
	Adding a Variable Test	30
	Deleting a Variable Test	31
	Adding a Variable Operation	31
	Deleting a Variable Operation	31
Chapter 8	Editing An Object's STDs	33
	Editing The Set Of Object STDs	33
	Adding an STD	33
	Renaming an STD	33
	Deleting an STD	33
	Editing an STD	33
	Editing an STD's States	34
	Adding a New State	34
	Renaming a State	34
	Toggling Between Macro and Primitive State	34
	Deleting a State	35
	Moving a State	35
	Changing the State Name Font	35
	Specifying the STD's Start State	35

	Using Cut And Paste	35
	Editing an STD's Transitions	36
	Adding a New Transition	36
	Defining External Conditions and Actions (Messages)	36
	Adding a New Condition	37
	Removing a Condition	37
	Adding a New Action	37
	Removing an Action	37
	Defining External Conditions and Actions (STDs)	37
	Defining Internal Conditions and Actions	38
	Adding a New Condition	38
	Removing a Condition	38
	Adding a New Action	39
	Removing an Action	39
	Editing an Existing Transition's Conditions and Actions	39
	Moving a Segment	40
	Splitting a Segment	40
	Moving the Transition's Delimiter	41
	Using Cut and Paste	42
	Ordering of Conditions and Actions	42
	Cutting And Pasting An STD	43
	Editing A Sub-Sequence STD	43
	Zooming	44
Chapter 9	Configuration	45
	Changing The Fonts Used On The Diagrams	45
	Grid	45
	Zoom Factor	45
Chapter 10	Printing	47
	Printing The Contents Of The Active Window	47
	Printing An Object's Details	47
Chapter 11	Loading Available RWI/RWO From File	49
	Loading The Available RWIs	49
	Loading The Available RWOs	49
Appendix A	Rules	51
	External - Object Related	51
	External - Interface Related	51

	Internal - Commands	51
	Internal - Variables	52
	Internal - Object STD	52
	Internal - Sub-Sequence STD	52
	Internal - STD State	52
	Internal - STD Transition	53
Appendix B	Formal Definition Of An Application	55
Appendix C	Menus	57
Appendix D	Toolbar Buttons	59
	Operation Toolbar	59
	Mode Toolbar	60

1 Introducing Synect

Synect is a set of software tools which helps the designer of a control system to produce a specification which is clear, precise and free of errors. Synect combines the ease of use of a graphical user interface, with a widely used diagrammatic notation and the power of mathematical modelling.

User Interface

The Synect™ tools run on the Microsoft® Windows™ operating system. As such, you need to know how to use Windows before you use Synect. In particular, you will need to know how to use the mouse to click, double-click and drag. You will also need to know how to interact with menus, dialogs and how to move, resize and close windows. For information, refer to the Microsoft Windows User's Guide.

The Method

Synect uses an object-based method to enable you to model the system you want to control. It is useful to have some knowledge of object-based/object-oriented analysis and design techniques before attempting to use Synect in earnest.

Synect Documentation

Each of the Synect tools has an associated User Guide. This User Guide explains how to use each of the functions available in the Application Editor. It also explains why you might want to use the function.

Each application has context-sensitive on-line help. The Application Editor on-line help contains a "How Do I?" section, including a "How Do I Get Started?" sub-section for first-time users.

A Tutorial is also provided which offers a worked example and shows how each of the tools is used with the example application.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the Application Editor.
- how to start the Application Editor.
- the Application Editor window.

System Requirements

The Synect Application Editor requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

Other Synect tools require that you also have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press ENTER.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

The Tools

The tools which make up the Synect toolset are:

Application Editor	graphical means of defining the application.
Compiler	check the specification for consistency and possible warnings and generate a mathematical model of the application.
Analyzer	ability to check for design errors such as deadlock (where the system "hangs") and unwanted state combinations.
Simulator	provides the ability to interactively "drive" the application or replay past behaviour of the live control system.
STD Monitor	animates the specification (used in conjunction with the Simulator or the live control system).
C Code Generator	generate ANSI-standard C code to implement the application.
Neuron C Generator	generate Neuron C to run on one or more nodes on an Echelon LonWorks network to implement a distributed control solution.
Ladder Logic Generator	generate relay ladder logic to run on a programmable controller.

Document Conventions

The User Guide adopts the following conventions:

<code>application name</code>	text that you type or that you see on the screen.
<code>KEY NAME</code>	keyboard keys, such as ENTER, CTRL or DEL.
<code>Menu Choice</code>	a menu option, such as File Exit denoting choose the Exit command from the File menu.
<i>description</i>	description of a term with a specific meaning.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the Application Editor.
- how to start the Application Editor.
- the Application Editor window.

System Requirements

The Synect Application Editor requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

Other Synect tools require that you also have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

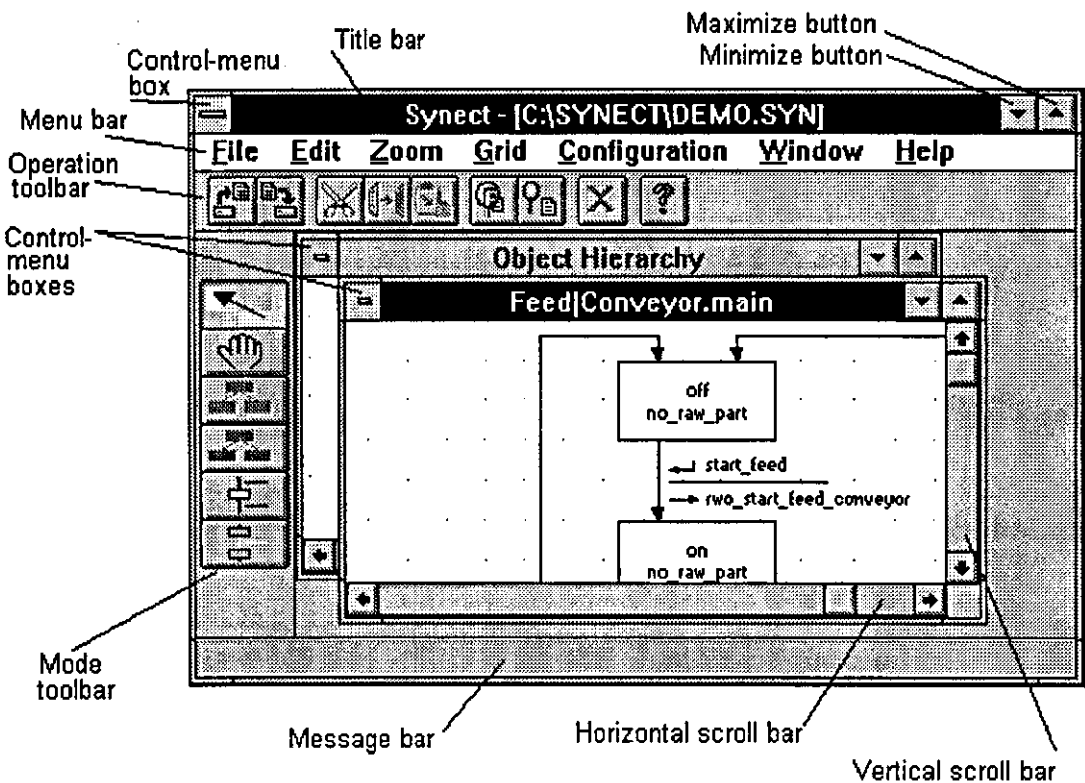
- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press ENTER.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

- 8 Read the installed file `readme.txt` for details of any changes to the product or documentation since the documentation was printed.

Starting the Application Editor

Ensure that the Program Manager window is on display. If the Synect window isn't visible, use the Window menu to open it. Double-click on the Application Editor icon or use the keyboard arrow key to select the icon and then press ENTER.

When using the Application Editor, the window will typically look like the following:



Title bar

The outer window is the Synect Application Editor window. The title bar therefore shows the product title *Synect* and the name of the application which is being edited (or [Untitled] if it's a new application which hasn't yet been saved to disk).

The inner windows show either the Object Hierarchy or a state transition diagram. The title bar of an inner window will therefore be either *Object Hierarchy* or the name of the state transition diagram.

Control-menu boxes

Allows you to restore, move, size, minimize, maximize, close (except for the Object Hierarchy window) the window. Also allows you to make another window the active window or open the control panel (Synect window only).

Menu bar

Lists the available menus.

Minimize box

Allows you to shrink the Synect window to an icon at the bottom of the screen.

Maximize box

Allows you to enlarge the Synect window to fill the entire screen.

Operation toolbar

Contains the momentary buttons which can be used as menu shortcuts.

Mode toolbar

Contains the radio buttons which you use to control which mode you want the software to operate in (for example, whether you are adding new states).

Message bar

When the cursor is moved over an Operation Toolbar button or Mode Toolbar button, or when a menu option is highlighted, the message bar will show a brief description of the function of the button/menu option.

Horizontal and vertical scroll bars

Allow you to pan around the diagram to change the portion which is displayed in the window.

On-Line Help



The on-line help is context-sensitive. So if you click on a help button in a dialog, you will automatically be shown the help information associated with that dialog. If no dialog is being displayed, you can choose [Help|Contents](#) to take you to the help contents page. The help information is shown in a separate window.

This page left intentionally blank

3 Basic Concepts

Synect enables you to build a model of your application. The role of the Application Editor is to let you specify this model. Other Synect tools allow you to check it for consistency and to verify that the model behaves as required. This chapter describes how the model is organised. Later chapters explain how to use the Application Editor to create a model.

The Object Hierarchy

An application is considered to consist of a strict hierarchy of objects. The topmost object is referred to as the root object and is the most abstract view of the application. This root object will typically consist of other objects, referred to as child objects (the root object is the parent of these children).

An object typically models an item in the system being controlled. For example, a manufacturing application might have separate objects for a machine, a robot and a conveyor system. An object which has no children is called a *primitive* object. An object with children is called a *composite* object.

An object bounds the functionality of the item of interest by defining:

- the messages which it can be sent and which it will return
- the interface with the controlled system (real world inputs and real world outputs)
- the sequential logic within the object (using state transition diagrams)

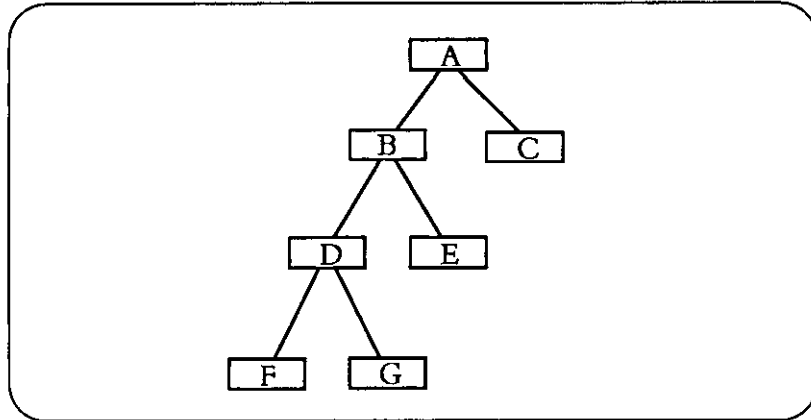
Object interaction

Objects interact in one of two ways. The first is by sending and receiving messages from other objects. This is described in more detail in the next section. The other type of interaction allows an object's STD to use the state of another STD as a condition of a transition. For example, if a light bulb is controlled by a switch, the first method would require that the switch object sends the light bulb object a message when the switch changes state from "off" to "on". The second method would require that the light bulb object monitors the state of the switch object and illuminates when it detects that the light switch is in state "on".

If you will be generating centralised control code, you can use either method. If, however, you intend to distribute the logic across multiple processing nodes, such as when using the Synect Distributed Neuron C Code Generator for use with Echelon's LonWorks technology, you will need to use the state-reference method. Synect requires that **two objects which are to be assigned to different nodes must not use message-based communication between them**. Objects which are to be assigned to the same node can use either or both of the message-based and state-reference methods.

Messaging

As stated in the previous section, parent and child objects can communicate with each other by sending *messages*. A message from a parent to a child is referred to as a *command*. A message from a child to its parent is referred to as a *response*. An object manages its children on behalf of the rest of the application – in the diagram, for example, object A can only communicate with objects D and E via object B. Similarly, objects F and G can only communicate with object B via object D.



A message can be either *synchronous* or *asynchronous*. A synchronous message is one which causes the sending transition and the receiving transition to fire as one. It therefore synchronises the two transitions. If the receiving STD is not in a position to receive the message, the transition in the sending STD cannot fire. A synchronous message is identified by the square brackets around the message name, for example:

```
[start_motor]
```

An asynchronous message is one which can be sent irrespective of whether the receiver is yet in a position to act on it. The message is placed in a buffer and is then said to be *pending*. If the message is already pending (i.e. the buffer is full), the transition in the sending STD cannot fire. Use of asynchronous messages can cause a substantial increase in the number of combinations of state that the application can reach, possibly to the extent of preventing the use of the Analyzer. It can also make the behaviour of the application more difficult to follow.

Interface with the controlled system

Real world inputs enable an object to read the value of sensors in the system being controlled (such as whether a switch is closed). *Real world outputs* enable an object to instruct the system being controlled to take some action (such as starting a motor).

A real world input is an input into the application from the controlled system. A real world input can be thought of as a boolean function which the application calls. For example, the following real world inputs may be available for a motor:

- motor_stopped

- motor_running
- motor_running_full_speed

Synect is independent of the target control system hardware. As such, the means by which a real world input function determines the boolean state to be returned is undefined.

A real world output is an output from the application to the controlled system. A real world output can be thought of as a function which the application calls. For example, the following real world outputs may be available for a motor:

- stop_motor
- start_motor
- run_motor_at_full_speed

Synect is independent of the target control system hardware. As such, the means by which a real world output function causes the controlled system to take the required action is undefined.

Internal Events

An object may also have *internal events*. An internal event is either an internal command, a variable test or a variable operation.

Warning

Synect variables appear to be less useful than originally envisaged. It is therefore recommended that you ignore the use of variables until familiar with the other capabilities of Synect. In particular, the Application Editor will let you configure a variable with a large range (maximum - minimum). However, when the Application Analyzer attempts to explore all of the possible states which the application can reach, it will probably run out of memory. If variables are to be used, their ranges should therefore be kept as small as possible.

Internal command

An object may contain several STDs to define the required sequential logic. These STDs may communicate with each other just as objects do - by sending messages. These internal messages are called internal commands. Internal commands may be synchronous or asynchronous.

Variable

An object may contain variables. A variable has the following configuration attributes:

- name
- minimum value
- maximum value
- initial value

The values are subject to the following constraints:

- minimum value \leq maximum value
- minimum value \leq initial value \leq maximum value

The variable has an integer value which can be changed by a variable operation and tested against by a variable test.

Variable test

A set of tests may be defined which are applicable to each variable, where each test is one of $<$, \leq , $=$, \geq or $>$. A variable test can then be used as a condition on a transition. A transition cannot contain more than one condition referring to any particular variable.

Variable operation

A set of operations may be defined which are applicable to each variable, where each operation is one of:

INCR	increment the value by 1 provided that the variable value is less than its maximum.
DECR	decrement the value by 1 provided that the variable value is greater than its minimum.
RESET	reset the variable to its initial value

A variable operation can then be used as an action on a transition. A transition cannot contain more than one action referring to any particular variable.

State Transition Diagram

The logic within each object is defined by one or more *state transition diagrams (STDs)*. Each STD typically consists of several states and several transitions.

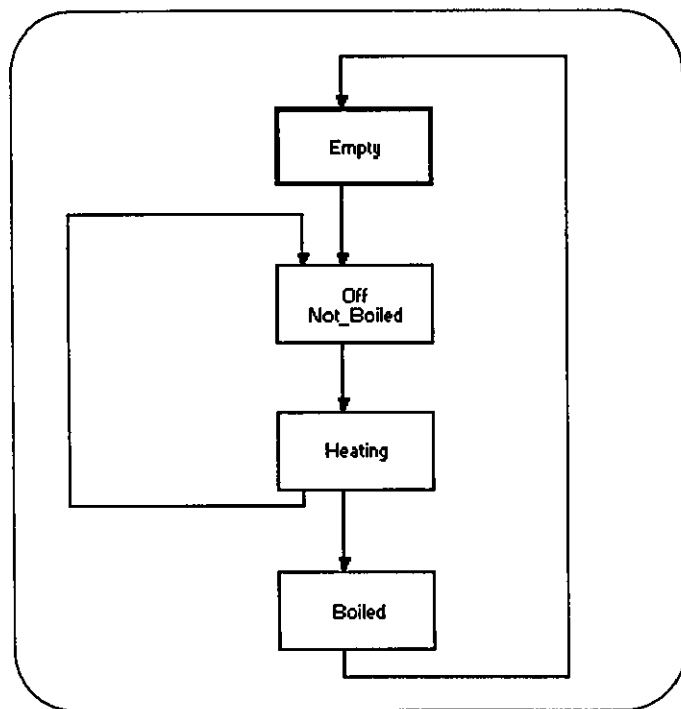
State

Each *state* is represented by a unique name which is displayed in the STD window in a rectangle. Each state represents some identifiable mode of operation of the item being modelled. For example, a gripper could have states denoting "open", "closing", "closed" and "opening". One of the states is designated as the *initial state* and is drawn with a thicker border in the STD window. The STD's current state will change to another state if a transition fires which starts at the current state and ends at a different state. An STD is said to be in a particular state – for example, the gripper STD is in state "closing".

A state may be either a *primitive state* or a *macro state*. A macro state is drawn with a very thick border and is further decomposed into a *sub-sequence STD*.

Transition

Transitions define which other states are reachable from each state. A transition is represented by an arrowed line where the arrow shows the direction from start state to end state. In the following example of a kettle control system, the STD starts in state "Empty". The only other state reachable from state "Empty" is "Off Not Boiled" (presumably when the kettle has been filled with water). From state "Heating", the STD may return to state "Off Not Boiled" (probably because the user has decided to abort the sequence) or state "Boiled" (boiling water detected).



Associated with each transition is a (possibly empty) set of conditions. If these conditions evaluate to true (an empty set always evaluates to true), a transition which has the current state as its start state is said to be enabled and can fire.

Also associated with a transition is a (possibly empty) set of actions. When a transition fires, the current state changes to the transition's end state and the actions are invoked. The transition is considered to fire instantaneously such that the STD is always in one of the defined states.

See Appendix B for a formal definition of the composition of an application.

This page left intentionally blank

4 Create, Save and Open An Application

When you first start the Synect Application Editor, very few of the menu items or buttons are enabled. This is because you must first indicate whether you want to create a new application or edit an existing one. During your editing session, you'll want to regularly save the application to disk. This will allow you to recover in the event of a power fail or if you want to discard a set of edits and revert to the application before the edits were made. **Remember that your application is NOT automatically written to disk at regular intervals - if you don't save your application to disk and a power/hardware/software failure occurs you will lose all of the changes since the start of the session or since the last save/load.**

Creating A New Application



To create a new application, choose **File|New**. The Synect window will show the title of the application as [Untitled] and an empty Object Hierarchy window will be displayed. The Add Root Object button will be enabled in the Mode Toolbar.

Opening An Existing Application



To open an existing application choose **File|Open Application**. The standard file open dialog will then be started, allowing you to specify the name of the file from which the application is to be loaded.

Recover Mode

There have been infrequent reports of the Object Hierarchy window not being visible following re-load of an application (.syn file). This may be accompanied by an "ObjectWindows Exception" message.

The workaround is to instruct the editor to ignore the Microsoft Windows-related data in the .syn file. To do this, make sure that the Program Manager is visible with the Synect Application Editor icon selected. Choose the **File|Properties** menu option to start the Program Item Properties dialog. The Command Line editbox will contain a line of the form: "c:\synectv1\synect.exe". Append to this a space and then the string "-r" so that the command line is of the form: "c:\synectv1\synect.exe -r". When you next start the Application Editor, it will be in "recover" mode and will ignore any Microsoft Windows-related data in the .syn file. Remember to remove this modification when the application has been recovered.

Saving An Application



To save the application to a file on disk, choose **File|Save Application**. If you are editing a named application, the corresponding file on disk will be updated. If the application is as yet untitled, the standard file save dialog will be started for you to specify the name and location of the file in which the application is to be saved.

To save the application to a different file on disk, choose **File|Save Application As** menu option. The standard file save dialog will be started for you to specify the name and location of the file into which the application is to be saved.

When an application is saved to disk, backup copies are automatically made. In the event of an error during the save operation, it is therefore possible to recover from the previous version. The latest version of your application is saved in the file with extension ".syn". The previous version is saved in the file with extension ".sb1" and the version before that is saved in the file with extension ".sb2". For example, assuming you have called your application "test.syn", the file "test.sb1" will contain the previous version and file "test.sb2" will contain the version before that.

5 Editing The Object Hierarchy

As described in chapter 3, an application is modelled by defining a hierarchy of communicating objects and then defining the logic within each of the objects. This chapter describes how to use the Application Editor to specify the hierarchy of objects. Chapter 6 covers the specification of the messaging between objects.

Adding A New Root



Ensure that the Object Hierarchy window is the active window. Click on the Add Root Object button in the Mode Toolbar. Move the mouse so that the cursor is in the Object Hierarchy window. Press and hold down the left mouse button. A rectangle is drawn in the Object Hierarchy window denoting the new root object. Drag the mouse and observe that the rectangle denoting the location of the new root moves correspondingly. When the rectangle is in the desired location, release the left mouse button and the Add Object dialog will be automatically started. Type a unique name into the edit box or click on one of the existing names in the list box and then edit the contents of the edit box. Click on "OK". The new root object will be shown in the Object Hierarchy window with specified name and at the chosen location.

Adding A New Child



Ensure that the Object Hierarchy window is the active window. Click on the Add Child Object button in the Mode Toolbar. Move the mouse so that the cursor is in the Object Hierarchy window inside the object which is to be the parent of the new child. Press the left mouse button and hold it down. Drag the mouse and observe that the rectangle denoting the location of the new child moves correspondingly. When the rectangle is in the desired location, release the left mouse button and the Add Object dialog will be automatically started. Type a unique name into the edit box or click on one of the existing names in the list box and then edit the contents of the edit box. Click on "OK". The new child object will be shown in the Object Hierarchy window with specified name and at the chosen location.

Using Cut And Paste



Ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Select the object you want to copy by clicking on it and then choose **Edit|Cut**. This will copy the selected object and all of the Object Hierarchy below it into the paste buffer before deleting it from

the Object Hierarchy. Alternatively, to perform the copy to paste buffer operation without the subsequent deletion from the existing Object Hierarchy, choose Edit|Copy.



When pasting the object, you can make it the new root object or a child of an existing object:



- To paste a new root object, choose Edit|Paste. Press and hold down the left mouse button in the Object Hierarchy window with the cursor outside existing objects. Drag the cursor so that the new root object is in the desired location and release the left mouse button.
- To paste a new child object, choose Edit|Paste. Press and hold down the left mouse button with the cursor inside the object which is to be the new object's parent. Drag the cursor so that the new child object is in the desired location and release the left mouse button.



Having chosen Edit|Paste, most of the menu options and buttons are disabled until the paste operation is complete. You can cancel the paste operation by choosing Edit|Cancel.

If the object name isn't unique, the *name dash character* "?" will be prefixed to the object name.

When an object is copied, all of the object's details are also copied i.e. its external interface, internal events and STDs.

Zooming



To reduce the size of a diagram in a window, choose Zoom|Out.



To increase the size of the diagram, choose Zoom|In.

Before using the zoom facilities, you should ensure that scalable fonts are being used. See chapter 9, Configuration, for an explanation of how to change the fonts.

When printing a diagram, the Application Editor takes into account the current magnification of the window. You may therefore want choose Zoom|Reset before printing to ensure consistency.

Renaming An Object



Ensure that the Object Hierarchy is displayed in the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is inside the object you want to rename. Double-click the left mouse

button to start the Edit Object dialog. Click on the Rename button to start the Rename Object dialog. Renaming the object may affect files which are used by other Synect tools (such as the Simulator and Neuron C Code Generator). If any of these files are found, the Application Editor will start the Apply Rename To Files dialog for you to specify which files it is to automatically edit.

Moving An Object



Ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the object you want to move. Press and hold down the left mouse button and drag the object to the desired location. Release the left mouse button and the object will be redrawn in its new location.

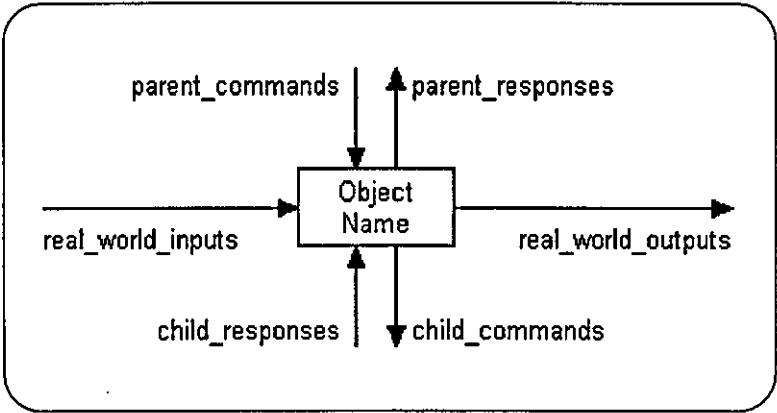
Changing Font

You can change the font used for writing object names by choosing Configuration|Object Name Font. This starts the standard Choose Font dialog.

This page left intentionally blank

6 Editing An Object's External Interface

An object's external interface defines the messages with which an object communicates with its parent and children and the interface with the system being controlled. Each of these is named and shown in the Object Hierarchy window with an arrow as follows:



parent commands	instructions which the object expects to receive from its parent.
parent responses	messages sent to the parent object.
child commands	instructions to child objects.
child responses	messages which the object expects to receive from its children.
real world inputs	inputs into the application from the controlled system. A real world input can be thought of as a boolean function which the application calls (e.g. to read a sensor).
real world outputs	outputs from the application to the controlled system (e.g. to take some action such as switching on a pump).

The text strings corresponding to each external interface are listed in alphabetical order with synchronous messages displayed after asynchronous messages. Strings which are prefixed with the *name dash character* "?" (as a result of a cut and paste operation) are listed first.

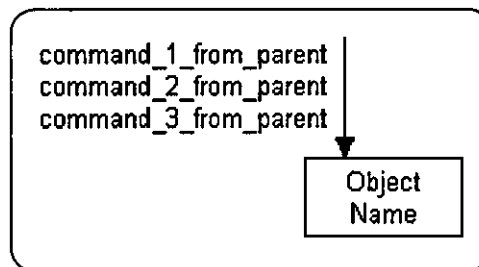
Using Cut And Paste



Ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Select the set of messages you want to copy by clicking on the text and then choose **Edit|Cut**. This will copy the selected set of messages into the paste buffer before deleting the messages from the object. Alternatively, to perform the copy to paste buffer operation without the subsequent deletion, choose **Edit|Copy**.



For example, if you want to copy the list of messages which an object receives from its parent, select the set of messages by clicking on the list of text strings associated with the arrow which enters the top of the object's box (as shown in the following diagram). Then choose **Edit|Copy**.



To paste the messages, choose **Edit|Paste**. Press and hold down the left mouse button in the Object Hierarchy window. As you move the cursor over sets of messages and over objects, you'll see the text being inverted. You can either paste the messages into an existing set by releasing the mouse button when that message set is shown inverted. Alternatively, you can release the mouse button with the object box shown inverted. This will start a dialog which you use to specify which message set the paste buffer messages are to be pasted into.

If pasting a message would violate the uniqueness criteria (see Appendix A, Rules), the name of the message will be prefixed with the *name-clash character* "?". For example, if you attempted to paste a message named `example_message` into an object's set of parent commands where the object already had a message named `example_message` defined as a parent response, the new parent command's name would be `?example_message`.



Having chosen **Edit|Paste**, most of the menu options and buttons are disabled until the paste operation is complete. You can cancel the paste operation by choosing **Edit|Cancel**.

Moving The External Interface Text Strings



To move the text strings associated with an object's messages, ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the corresponding message set. Press and hold down the left mouse button and drag the outline rectangle to the desired location.

Changing Font

You can change the font used for writing the external interface message names, real world inputs and real world outputs by choosing Configuration|Object Interface Font. This starts the standard Choose Font dialog.

Specifying An Object's External Interface



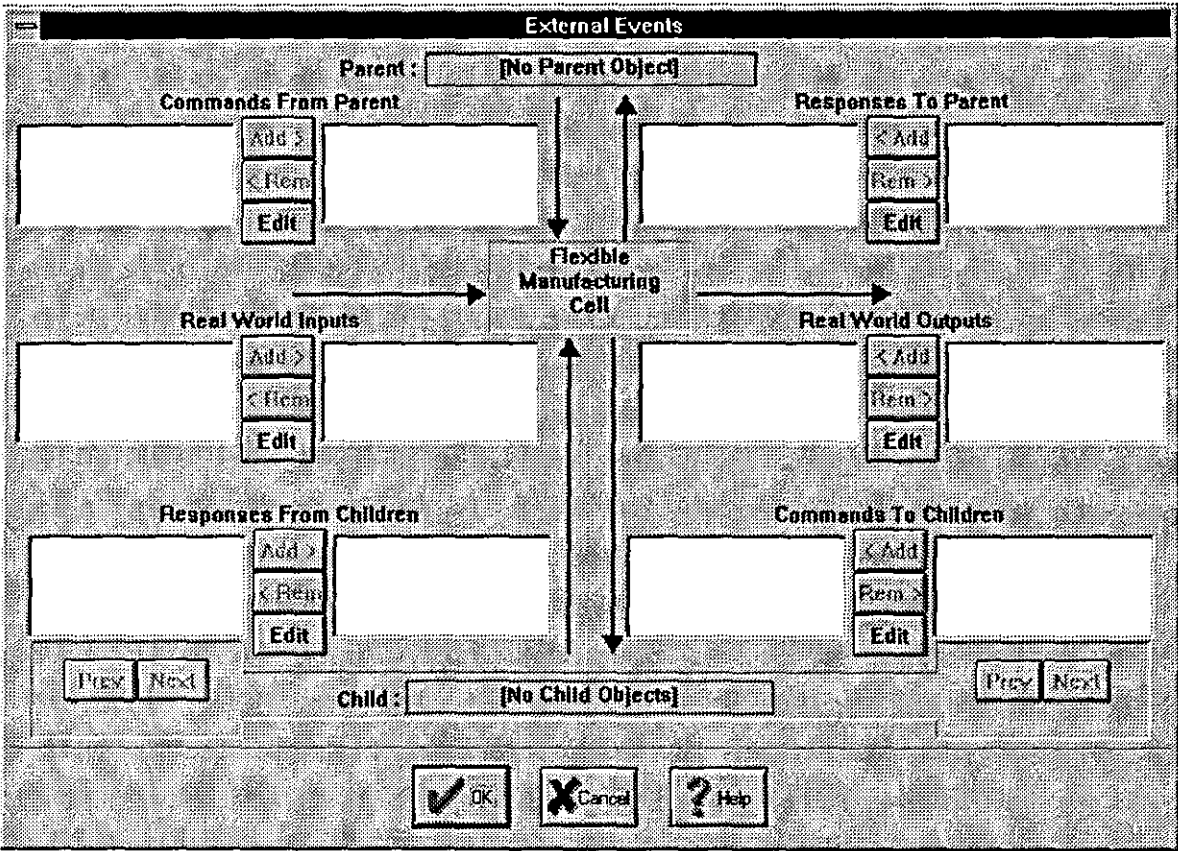
The primary means of specifying an object's external interface is via the External Events dialog (the other means is via the cut and paste functionality). To start this dialog, ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the object whose external interface you want to edit. Double-click with the left mouse button to start the Edit Object dialog. Click on the "External Interface" button to start the External Events dialog.

The example on the following page shows the dialog having been started to edit the external interface associated with object `Flexible Assembly Cell`. The dialog lists the messages which it knows other objects send to this object and expect from this object (there are none defined in the example). If a message has already been defined, you can simply select it rather than having to re-type the message name.

Just above the centre of the dialog is a static box showing the object's name (`Flexible Assembly Cell` in the example). At the top of the dialog is the name of the parent object (if this is the root object, the text `[No Parent Object]` is displayed). Towards the bottom of the dialog, the name of one of the child objects is displayed (if this object has no children, the text `[No Child Objects]` is displayed). If this object has several children, the "Prev" and "Next" buttons are enabled allowing you to change which child object is displayed.

Arrows are drawn to represent the external interface just as they appear on the Object Hierarchy and associated with each interface is a set of two list boxes. The list box closest to the arrow shows the list which is currently declared for this object. The other list box shows the list of available names.

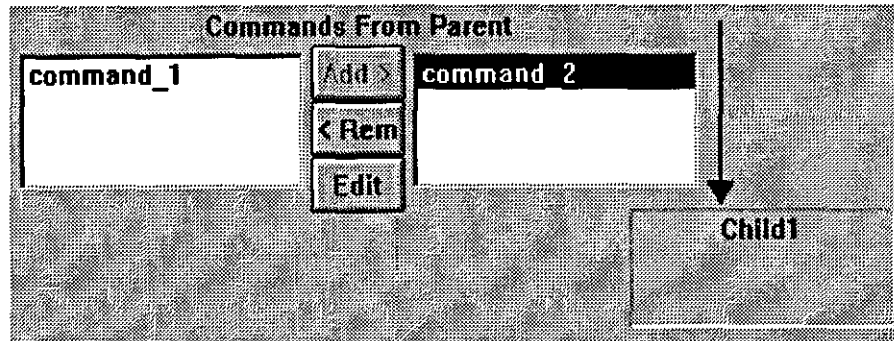
This page left intentionally blank



This page left intentionally blank

Parent Messages

Using a different example, consider the commands from `Child1`'s parent object to `Child1`:



Assume that the parent object sends commands `command_1` and `command_2`. The dialog shows that object `Child1` expects to receive the command `command_2` from its parent object but doesn't expect to receive command `command_1`. Any commands which the parent sends and which are declared for this object (in this example, `command_2`) are not listed in the left-hand listbox.

If you decide that object `Child1` should expect to receive command `command_1`, click on `command_1` in the left-hand listbox and then click on the "Add>" button. The string `command_1` will be removed from the left-hand listbox and added to the right-hand listbox.

If you decide that object `Child1` should not expect to receive command `command_2`, click on it in the right listbox and then click on the "<Rem" button. The string `command_2` will be removed from the right-hand listbox and added to the left-hand listbox. Any references to `command_2` in transition conditions in STDs belonging to this object will also be deleted.

If a parent command had been declared for this object which the parent object did not send, the name would be listed in the right-hand listbox as before. However, selecting it and then clicking on "<Rem" would not add it to the left-hand listbox.

The same procedure is used for responses to the parent object.

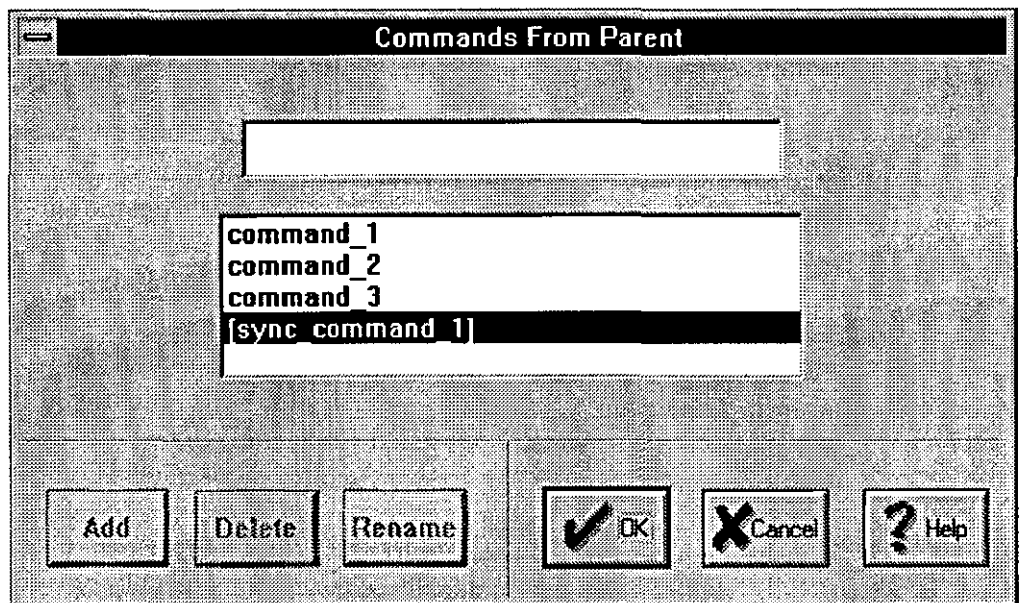
The same procedure is used for real world inputs and real world outputs but the set of available names is determined differently. The set of available real world inputs (displayed in the leftmost listbox) is populated from the list loaded via the File|Load RWI menu option. The set of available real world outputs (displayed in the rightmost listbox) is populated from the list loaded via the File|Load RWO menu option.

**Child
Messages**

A similar procedure is used for dealing with messages to/from child objects except that it is only possible to display the information about what one child is expecting to send/receive at a time. The "Prev" and "Next" buttons allow you to change which child's commands and responses are displayed in the outermost listboxes. When removing a command/response from the inner listbox, the dialog automatically switches to display the child to which the command/response belongs (if any).

**Editing The
List Of
Names**

Each external interface has an associated "Edit" button which allows you to start the corresponding dialog to edit the commands/responses/real world inputs/real world outputs declared for this object. These dialogs follow the same form. The only difference is that messages can be synchronous or asynchronous whereas real world inputs and real world outputs are always asynchronous. The following example shows the dialog being used to edit the set of commands which this object expects to receive from its parent.



The functions available are:

Add

Either type a name into the blank edit box or click on one of the existing names in the list box in which case the name is copied into the edit box to be used as a basis for the new name. Click on the "Add" button.

Delete

Click on the name you want to delete in the list box. Click on the "Delete" button. Any references in this object to this name by transitions conditions or actions will also be automatically deleted.

Rename

Click on the name you want to rename in the list box. Click on the "Rename" button to start the appropriate rename dialog. Type the new name and click on "OK". Any references in this object to this name by any transition's conditions or actions will also be automatically updated. Renaming a real world input/output may affect files which are used by other Synect tools (such as the Simulator and Neuron C Code Generator). If any of these files are found, the Application Editor will start the Apply Rename To Files dialog for you to specify which files it is to automatically edit.

Rename vs. Delete + Add Functionality

Whilst a rename operation might appear to be the same as a delete operation followed by an add operation, the delete will cause all references to that name to be removed from transitions in STDs belonging to the object being edited. The rename function, however, will also apply the rename operation to corresponding conditions or actions.

Changing Synchronous To/From Asynchronous

To change a message from being synchronous to asynchronous, you must rename the message such that it no longer has the enclosing square brackets. For example, [test_message] to test_message. However, because the uniqueness checks ignore the square brackets, it is first necessary to rename the message such that it is completely unique (e.g. xxxx) and then rename it again to be as required. For example:

- Rename operation 1 change the name from [test_message] to xxxx.
- Rename operation 2 change the name from xxxx to test_message.

As explained in section "Rename vs. Delete + Add Functionality" (above), do not delete the original name and then add the asynchronous name instead of performing the 2 rename operations. This would have the effect of deleting all references to the message from transitions within STDs belonging to the object being edited.

This page left intentionally blank

7 Editing An Object's Internal Events

In addition to an object's external events, the object may also have internal events defined. These may be internal commands or a test or operation on a variable.

Editing An Object's Internal Commands

An object may contain several STDs to define the required sequential logic. These STDs may communicate with each other just as objects do - by sending messages. These internal messages are called internal commands. Internal commands may be synchronous or asynchronous.



To edit an object's internal commands, ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the object whose internal commands you want to edit. Double-click with the left mouse button to start the Edit Object dialog. Click on the "Internal Commands" button to start the Edit Internal Commands dialog. The use of this dialog is as described in chapter 6, Editing An Object's External Interface, section Editing The List Of Names.

Editing an Object's Variables

Each object may have zero or more variables. See chapter 3, Basic Concepts, section Internal Events for more information about variables.

Variables can be used as counters (although the range of values must be kept very small if the Compiler or Analyzer are to be used) or flags.



To edit an object's variables, ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the object whose variables you want to edit. Double-click with the left mouse button to start the Edit Object dialog. Click on the "Variable Tests and Operations" button to start the Define Variable Tests and Operations dialog and, from this dialog, click on the "Edit" button to start the Define/Edit Variables dialog.

Adding a Variable

You may either type a name into the blank edit box or click on one of the existing names in the list box in which case the name is copied into the edit box to be used as a basis for the new name.

Click in the "Min" edit box and type the variable's minimum value. Repeat similarly for the maximum and the variable's initial value. Keep the range as small as possible to avoid an explosion of the number of combinations of system state when using the Application Analyzer

Click on the "Add" button.

Changing the Min, Max or Initial Value

Click on the variable you want to edit in the list box. Click in the "Min", "Max" or "Initial" edit boxes and type the amended value. Click on the "Edit" button.

Deleting a Variable

Click on the variable you want to delete in the list box. Click on the "Delete" button. Any variable tests or operations defined for this variable will be automatically deleted. Any references to these variable tests and operations by transitions conditions or actions will also be automatically deleted.

Renaming a Variable

Click on the variable you want to rename in the list box. Click on the "Rename" button to start the Rename Variable dialog. Type the new variable name and click on "OK". Any variable tests or operations defined for this variable will be automatically renamed. Any references to these variable tests and operations by transitions conditions or actions will also be automatically updated.

Editing a Variable's Tests and Operations

Before a transition's condition can refer to a test of a variables value, the variable test must be configured. Similarly for a transition's action referring to a variable operation. Configure the tests and operations to be available by using the Define Variable Tests and Operations dialog.



To start this dialog, ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the object whose variables you want to edit. Double-click with the left mouse button to start the Edit Object dialog and then click on the "Variable Tests and Operations" button.

Adding a Variable Test

Click on the desired variable in the "Variables" list box toward the top of the dialog (if no variables are listed, you'll need to define them first by clicking on the "Edit" button).

Click on one of the "Test Type" radio buttons (<, <=, =, >=, >) and then type a value into the "Value" edit box (where the value is within the range specified by the min and max defined for the variable).

Click on the "Add" button.

**Deleting a
Variable Test**

Click on the variable test in the "Tests" list box you want to delete and then click on the "Delete" button. Any transitions which referred to this variable test will be automatically updated to have the condition removed.

**Adding a
Variable
Operation**

Click on the desired variable in the "Variables" list box toward the top of the dialog (if no variables are listed, you'll need to define them first by clicking on the "Edit" button).

Click on one of the "Operation Type" radio buttons (DECR, INCR, RESET).

Click on the "Add" button.

**Deleting a
Variable
Operation**

Click on the variable operation in the "Operations" list box you want to delete and then click on the "Delete" button. Any transitions which referred to this variable operation will be automatically updated to have the action removed.

This page left intentionally blank

8 Editing An Object's STDs

Each object can have 1 or more state transition diagrams (STDs) defining its sequential logic. Each STD must have a unique name. With an STD on display in a window, you can modify it by, for example, adding and deleting states, adding and deleting transitions, etc.. The first step, however, is to define the set of STDs which will specify the object's behaviour.

Editing The Set Of Object STDs



To edit an object's STDs, ensure that the Object Hierarchy window is the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the object to which you want to new STD to belong. Double-click with the left mouse button to start the Edit Object dialog. Click on the "STDs" button to start the Edit Object's STDs dialog.

Adding an STD

You may either type a name into the blank edit box or click on one of the existing names in the list box in which case the name is copied into the edit box to be used as a basis for the new name. Click on the "Add" button.

Renaming an STD

Select the name in the listbox which you want to modify by clicking on it. Click on the "Rename" button to start the Rename STD dialog. Renaming the STD may affect files which are used by other Synect tools (such as the Simulator and Neuron C Code Generator). If any of these files are found, the Application Editor will start the Apply Rename To Files dialog for you to specify which files it is to automatically edit.

Deleting an STD

Nominate the STD you want to delete by clicking on the name in the listbox to select it. Click on the "Delete" button to delete it from the list. Clicking on "OK" to close the dialog will then cause the STD itself (i.e. the states and transitions) to be deleted. If the STD was being displayed in a window, the window will be closed.

Editing an STD

Either double click on the name of the STD in the listbox or select the name by single-clicking and then clicking on the "Edit" button. If the STD is not currently being displayed in a window, a new window will be created in which it will be displayed. Otherwise, the window displaying the STD will be made the active window.

Editing an STD's States

The following descriptions assume that the STD containing the state to be manipulated or to which the new state is to be added is being displayed in the active window.

Adding a New State



Click on the Add State button in the Mode Toolbar. Move the mouse so that the cursor is in the STD window. Press and hold down the left mouse button. A rectangle is drawn in the STD window denoting the new state. Drag the mouse and observe that the rectangle denoting the location of the new state moves correspondingly. When the rectangle is in the desired location, release the left mouse button and the Add State dialog will be automatically started. Type a unique name into the edit box or click on one of the existing names in the list box and then edit the contents of the edit box. Click on "OK". The new state will be shown in the STD window with specified name and at the chosen location. The Application Editor remains in add state mode so you can add another new state by repeating the above instructions from the point where the left mouse button is pressed and held down to cause the outline rectangle to be drawn.

Renaming a State



To rename a state, ensure that the Application Editor is in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is inside the state you want to rename. Double-click the left mouse button to start the Edit State dialog. Click on the "Rename" button to start the Rename State dialog. Renaming the state may affect files which are used by other Synect tools (such as the Simulator and Neuron C Code Generator). If any of these files are found, the Application Editor will start the Apply Rename To Files dialog for you to specify which files it is to automatically edit.

Toggling Between Macro and Primitive State



A macro state is one which is further decomposed into a sub-sequence STD. This is used to further partition the sequential logic to make each diagram more comprehensible. To convert a primitive state to a composite state, ensure that the Application Editor is in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is inside the state you want to convert. Double-click the left mouse button to start the Edit State dialog. Click on the "Add" button to convert the state to a macro state and to automatically add the sub-sequence STD and create a window in which it is displayed. If the state is already a macro state, the STD may be edited by clicking on the "Edit" button. To convert a state from a macro state to a primitive state, click on the "Delete" button. This will also delete the sub-sequence STD itself and, if the STD is being displayed in a window, the window will be closed.

If the state is the STD's start state, the "Add" button will be disabled because a state cannot be both the start state and a macro state.

Deleting a State

To delete a state, ensure that the Application Editor is in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is inside the state you want to delete. Select the state by clicking with the left mouse button and then hit the delete key. Any transitions which started or ended in the deleted state will also be deleted.

Moving a State



To move a state, ensure that the Application Editor is in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is inside the state you want to move. Press and hold down the left mouse button and drag the state to the desired location. Release the left mouse button to cause the state to be redrawn in the new location.

Changing the State Name Font

See chapter 9, Configuration, for details of how to change the typeface and character size used for displaying state names.

Specifying the STD's Start State



Each STD must have a start state. To nominate a state as the start state, ensure that the Application Editor is in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is inside the desired state and double-click the left mouse button to start the Edit State dialog. Click on the "Start State" checkbox to make this state the STD's start state. The same mechanism may be used to specify that a state is not to be the start state. If the state is a macro state, the "Start State" checkbox will be disabled because a state cannot be both a macro state and the STD's start state.

Using Cut And Paste



Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Select the state you want to copy by clicking on it. The selected state will now be shown in a different colour to the other states. Choose Edit|Copy to copy the state to the paste buffer. To have the state automatically deleted following the copy to the paste buffer, choose Edit|Cut instead.



To paste the state copy into an STD, ensure that the STD is displayed in the active window and that the Application Editor is in edit mode as before. Choose Edit|Paste. Most of the menu options and Operation Toolbar buttons will now be disabled. The Edit|Cancel menu option is enabled. To cancel the operation, choose Edit|Cancel. Otherwise, move the mouse so that the cursor is in the STD window. Press and hold down the left mouse button and drag the state rectangle to the desired location before releasing the left mouse button. If the state name isn't unique, the name clash character "?" will be prefixed to the state name.

If the pasted state is a macro state, its sub-sequence STD will also have been copied.

Editing an STD's Transitions

The following descriptions assume that the STD containing the transition to be manipulated or to which the new transition is to be added is being displayed in the active window.

Adding a New Transition



Click on the Add Transition button in the Mode Toolbar. Move the mouse so that the cursor is in the STD window over the transition's start state. Press and hold down the left mouse button. Drag the mouse and observe the first segments of the transition are drawn (vertical then horizontal) ending at the current cursor position.

If the transition's end state is different from its start state, you can drag the mouse so that the cursor is inside the end state and then release the left mouse button. The External Transition Conditions and Actions (STDs) dialog will be automatically started.

If the transition starts and ends in the same state, or if you want to have more control over the positions and lengths of the transition's segments, release the left mouse button with the cursor not inside a state. The first two segments of the transition are drawn ending at this intermediate point. Press and hold the left mouse button down again and drag the mouse. The next two segments are drawn starting at the intermediate point and ending at the current cursor position. This set of operations may be repeated as often as required. To complete the transition, release the left mouse button when the cursor is inside a state. The External Transition Conditions and Actions (STDs) dialog will be automatically started.

The External Transition Conditions and Actions (STDs) dialog is similar to the External Transition Conditions and Actions (Messages) dialog which will be described first for ease of explanation.

Defining External Conditions and Actions (Messages)

The purpose of the External Transition Conditions and Actions (Messages) dialog is to enable you to specify which conditions must be satisfied before this transition can fire and which actions are invoked when it does fire. This dialog allows you to deal with conditions and actions which relate to the external interface (messages to/from other objects and real world inputs/real world outputs). To deal with conditions and actions which relate to activities internal to the object, click the "Show Internal Events" button to start the Internal Transition Conditions and Actions dialog. To deal with conditions and actions which relate to the object's real world inputs, real world-outputs and state references, click the "Show STDs" button from the Internal Transition Conditions and Actions dialog to start the External Transition Conditions and Actions (STDs) dialog.

additional items to be added to the external interface, the "Edit" button can be used to start the External Events dialog.

The right hand side of the dialog refers to the transition being edited. Towards the top of the dialog, the transition's start and end states are shown. The topmost listbox shows the conditions currently tested by this transition which relate to the external interface. Next is a listbox showing the conditions associated with this transition which are derived from internal events (this is shown for information only in this dialog). The next listbox shows the actions relating to the external interface which are invoked when this transition is fired. The bottom-most listbox shows the actions relating to activities which are internal to the object (this is shown for information only in this dialog). Below this is an editbox into which a comment can be typed which is associated with this transition.

Adding a New Condition

Click on the real world input, command from parent or response from child which you want to be a condition of this transition. Click on the "Add>" button to the left of the External Conditions listbox. The string is removed from the listbox relating to the external events and is added to the External Conditions listbox.

Removing a Condition

Click on the condition you want to remove in the External Conditions listbox. Click on the "<Remove" button next to the External Conditions listbox. The string is removed from the External Conditions listbox and added to the appropriate external events listbox.

Adding a New Action

Click on the real world output, response to parent or command to child which you want to be an action of this transition. Click on the "Add>" button to the left of the External Actions listbox. The string is removed from the listbox relating to the external events and is added to the External Actions listbox.

Removing an Action

Click on the action you want to remove in the External Actions listbox. Click on the "<Remove" button next to the External Actions listbox. The string is removed from the External Actions listbox and added to the appropriate external events listbox.

Defining External Conditions and Actions (STDs)

The External Transition Conditions and Actions (STDs) dialog is very similar to the External Transition Conditions and Actions (Messages) dialog. It does not show the messages to/from parent or children but instead contains two listboxes to display STD information.

The leftmost listbox shows a list of all of the STDs which are currently defined. Click on an entry in this listbox and the other listbox will be populated with the states which belong to that STD. Click on a state name in the listbox and the

"Add>" button to the left of the External Conditions listbox will be enabled. Click on the "Add>" button to add the state reference as a condition.

To remove a state reference from the list of conditions for the current transition, click on it in the External Conditions listbox and then click on the "<Remove" button next to the External Conditions listbox.

**Defining
Internal
Conditions
and Actions**

The purpose of the Internal Conditions and Actions dialog is to enable you to specify which conditions must be satisfied before this transition can fire and which actions are invoked when it does fire. This dialog allows you to deal with conditions and actions which are internal to the object (internal commands and variable tests and operations). To deal with conditions and actions which relate to the object's real world inputs, real world-outputs and state references, click the "Show STDs" button to start the External Transition Conditions and Actions (STDs) dialog. To deal with conditions and actions which relate to messages to/from the object, click the "Show External Events" button from the External Transition Conditions and Actions (STDs) dialog to start the External Transition Conditions and Actions (Messages) dialog.

The transition being edited is part of an STD which, in turn, belongs to an object. On the left of the dialog, a representation is shown of the object's internal events and actions. If you decide that there are additional items to be added to any of these lists, the appropriate "Edit" button can be used which will start either the Edit Internal Commands dialog or the Define Variable Tests and Operations dialog.

The right hand side of the dialog refers to the transition being edited. Towards the top of the dialog, the transition's start and end states are shown. The topmost listbox shows the conditions currently tested by this transition which relate to the external interface (this is shown for information only in this dialog). Next is a listbox showing the conditions associated with this transition which are derived from internal events. The next listbox shows the actions relating to the external interface which are invoked when this transition is fired (this is shown for information only in this dialog). The bottom-most listbox shows the actions relating to activities which are internal to the object. Below this is an editbox into which a comment can be typed which is associated with this transition.

**Adding a
New
Condition**

Click on the "Tests of Variable Values" listbox item or "Internal Commands" listbox item which you want to be a condition of this transition. Click on the "Add>" button to the left of the "Internal Conditions" listbox. The string is removed from the listbox relating to the internal events and is added to the "Internal Conditions" listbox.

**Removing a
Condition**

Click on the condition you want to remove in the "Internal Conditions" listbox. Click on the "<Remove" button next to the "Internal Conditions"

Removing a Condition

Click on the condition you want to remove in the "Internal Conditions" listbox. Click on the "<Remove" button next to the "Internal Conditions" listbox. The string is removed from the "Internal Conditions" listbox and added to the appropriate internal events listbox.

Adding a New Action

Click on the "Operations on Variables" listbox item or "Internal Commands" listbox item which you want to be an action of this transition. Click on the "Add>" button to the left of the "Internal Actions" listbox. The string is removed from the listbox relating to the internal activities and is added to the "Internal Actions" listbox.

Removing an Action

Click on the action you want to remove in the "Internal Actions" listbox. Click on the "<Remove" button next to the "Internal Actions" listbox. The string is removed from the "Internal Actions" listbox and added to the appropriate internal activities listbox.

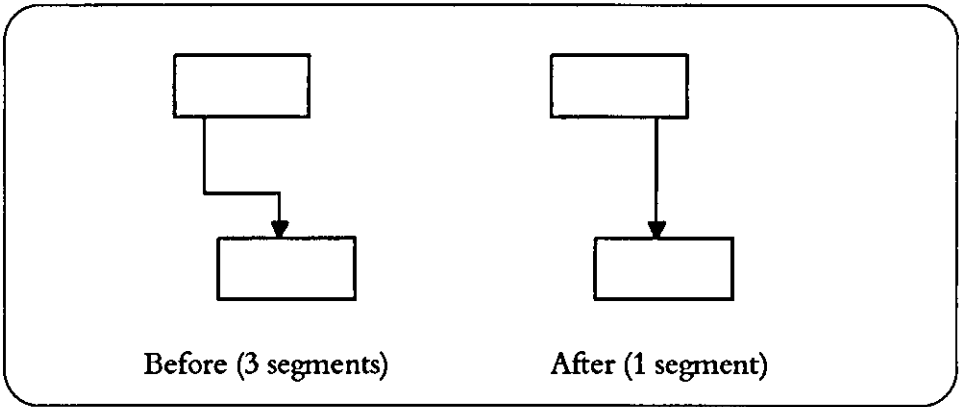
Editing an Existing Transition's Conditions and Actions

Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Invoke the External Transition Conditions and Actions (STDs) dialog by double-clicking on the horizontal delimiter between the transition's conditions and actions. The External Transition Conditions and Actions (Messages) dialog is reachable from this dialog and in turn the Internal Transition Conditions and Actions dialog is then reachable.

Moving a Segment



Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the segment you want to move. Press and hold down the left mouse button and drag the segment to the desired location. Release the left mouse button and the transition segment will be drawn in the new location. Following the move, any segments which are of zero length will be deleted. In the following example, there are initially 3 segments and finally 1 segment after the move:



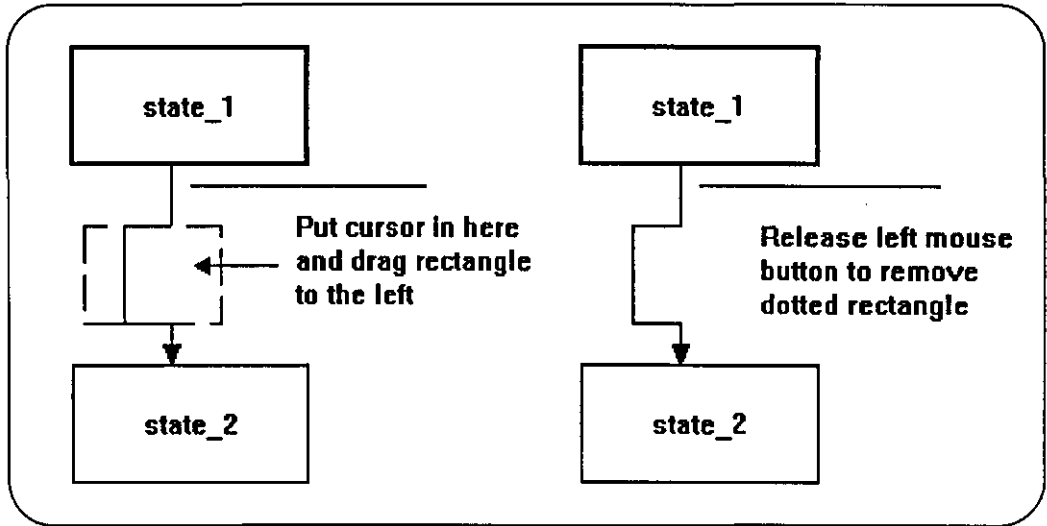
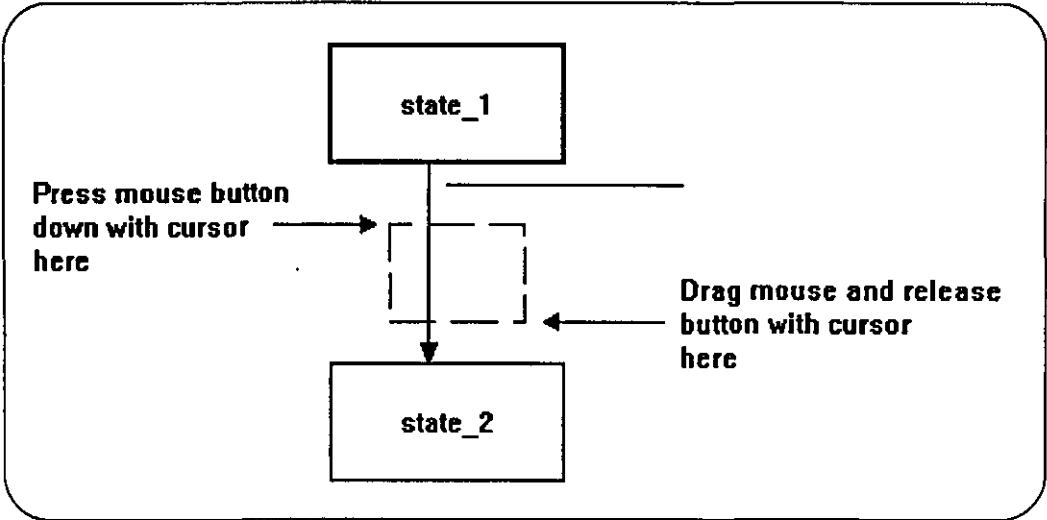
Splitting a Segment



Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. This example assumes that a vertical segment is to be split into 5 segments.

Press and hold down the left mouse button with the cursor to the left of the transition. Drag the mouse and observe that an elastic rectangle is drawn with one corner at the location where the drag operation was started and opposite corner at the current cursor position. When the elastic rectangle crosses the transition, release the left mouse button. A dotted rectangle will be drawn corresponding to the elastic rectangle. Move the cursor so that the mouse pointer is inside the dotted rectangle. Press and hold down the left mouse button and drag the dotted rectangle such that new transition segments are shown. Release the left mouse button and the dotted rectangle is removed,

leaving the new transition segments as placed.



Moving the Transition's Delimiter

A horizontal line delimits the transition's conditions (shown above the delimiter) from its actions (shown below the delimiter). The delimiter is initially positioned halfway along the first segment but can be dragged around the transition's segments.



Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Move the mouse so that the cursor is over the horizontal delimiter between the transition's conditions and actions. Press and hold down the left mouse button. By using the drag operation, the conditions and actions can be moved around the contour of the transition. Release the left mouse button and the conditions and actions will be redrawn in their new location.

Using Cut and Paste

Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary. Select the transition's conditions and actions by clicking on the horizontal delimiter between the conditions and actions. The conditions and actions will now be shown in a different colour to show that they are selected. Choose **Edit|Copy** to copy them to the paste buffer. To have the conditions and actions automatically deleted following the copy to the paste buffer, choose **Edit|Cut** instead.

To paste the conditions and actions into an STD, ensure that the STD is displayed in the active window and that the Application Editor is in edit mode as before. Choose **Edit|Paste**. Most of the menu options and Operation Toolbar buttons will now be disabled. The **Edit|Cancel** menu option is enabled. To cancel the operation, choose **Edit|Cancel**. Otherwise, move the mouse so that the cursor is in the STD window. Press and hold down the left mouse button and drag the rectangle denoting the conditions and actions such that it is over the horizontal delimiter of the transition into which you want to paste. Release the left mouse button. If any of the conditions or actions clash with existing names, they will be prefixed with the *name clash character* "?".

After pasting conditions and actions, the Application Editor will add entries to the object's events as necessary to make the application consistent. For example, if real world input `is_light_on` is pasted into a transition and this real world input is not defined for the object to which this STD belongs, the `is_light_on` real world input will be added to the list defined for the object and will therefore be apparent when the External Events dialog is used.

If a condition or action refers to a variable which is undefined in the new object, the variable will be added with range 0 to -1 (i.e. an illegal range).

Ordering of Conditions and Actions

The conditions are shown on an STD in the following order:

1. commands from the parent object
2. responses from the child objects
3. real world inputs
4. tests against variables
5. internal commands

The actions are shown in the following order:

1. responses to the parent object
2. commands to child objects
3. real world outputs
4. operations on variables
5. internal commands

Within the above categories, names are listed in alphanumeric order with names beginning with the *name clash character* "?" being listed before the others. Within a list of messages, synchronous messages are listed after asynchronous messages.

Cutting And Pasting An STD

Strictly speaking, there is no facility for cutting and pasting STDs. The facility does exist, however, for copying a set of states and transitions which can be used to copy the partial or entire contents of an STD.

Ensure that the STD which you want to copy is displayed in the active window. Ensure that the Application Editor is currently in edit mode by clicking on the Edit Mode button in the Mode Toolbar if necessary.



Press and hold down the left mouse button with the cursor in the STD window and to the top left of the STD. Drag the mouse and observe that an elastic rectangle is drawn with one corner at the location where the drag operation was started and opposite corner at the current cursor position. When the elastic rectangle encloses the whole STD, release the left mouse button. Choose the Edit|Copy menu option.



To paste the STD, ensure that the STD into which the states and transitions are to be pasted is displayed in the active window and that the Application Editor is in edit mode as before. Choose the Edit|Paste menu option. Most of the menu options and Operation Toolbar buttons will now be disabled. The Edit|Cancel menu option is enabled. To cancel the operation, choose Edit|Cancel. Otherwise, move the mouse so that the cursor is in the STD window. Press and hold down the left mouse button and drag the outline rectangle to the desired location before releasing the left mouse button. The Application Editor will then take the same steps as it would for pasting each state and each condition and action in terms of checking for uniqueness and prefixing with the *name dash character* "?" if necessary.



If any of the pasted states is a macro state, its sub-sequence STD will also have been copied.

Editing A Sub-Sequence STD

A sub-sequence STD defines the logic within a macro state. The STD containing the macro state is referred to as the sub-sequence STD's *parent STD*. A sub-sequence STD is similar to an object STD but always starts in state START and ends in state END. When the macro state becomes the current state, the sub-sequence STD changes state from START to one of the other states (depending on the transitions defined for the sub-sequence STD). When the sub-sequence STD changes state to END, a transition in its parent STD which starts at the macro state will fire.

Due to the relationship between the transitions in the parent STD which start or end in the macro state, and the transitions out of the START state and into the END state in the sub-sequence STD, there are additional constraints on the content of a sub-sequence STD (see Appendix A, Rules, for further details).

For example:

- a sub-sequence STD must have at least 3 states.
- you cannot define a transition which starts at the START state and ends in the END state.

Zooming



To reduce the size of a diagram in a window, choose **Zoom|Out**.



To increase the size of the diagram, choose **Zoom|In**.

Before using the zoom facilities, you should ensure that scalable fonts are being used. See chapter 9, Configuration, for an explanation of how to change the fonts.

When printing a diagram, the Application Editor takes into account the current magnification of the window. You may therefore want to choose **Zoom|Reset** before printing to ensure consistency.

9 Configuration

This chapter describes the changes you can make to the Application Editor environment. These changes are saved whenever you save the application to disk.

Changing The Fonts Used On The Diagrams

The Application Editor enables you to specify the font you want to use for the display of text in the diagrams. The options are available from the Configuration menu:

Object Name Font	object names (in the Object Hierarchy window)
Object Interface Font	external interface names (in the Object Hierarchy window)
State Name Font	state names (in an STD window)
Transition Name Font	conditions and actions associated with transitions (in an STD window)

TrueType fonts should be chosen because they are scalable. These are displayed correctly when the zooming functionality is used and when the diagram is printed.

Grid

When positioning items in a diagram, the item always snaps to the grid regardless of whether the grid is on display. Use the Grid|Show Grid menu option to toggle whether the grid is displayed. Use the Configuration|Grid Size menu option to change the grid spacing.

Zoom Factor



Use the Configuration|Zoom Factor menu option to change the amount by which the diagram is shrunk/enlarged when using the Zoom|Out / Zoom|In menu options.

This page left intentionally blank

10 Printing

The Application Editor enables you to obtain printouts of the diagrams as displayed on your screen and also to print textual information about the contents of an object, such as its internal commands, variables etc., which are not displayed on any of the diagrams.

Printing The Contents Of The Active Window

Use the `File|Print` menu option to start the Select Information To Be Printed dialog. To print the diagram being displayed in the active window, ensure that the "Active Window" radio button is checked. The printout will use the same magnification as is currently being used by the active window. So if you want to print out an enlarged version of the diagram which is tiled across several sheets of paper, use the `Zoom|In` menu option prior to invoking the `File|Print` menu option.



If a text string spans 2 or more pages, you'll need to set up your printer to print TrueType as graphics to prevent the text string being clipped at a character boundary. Use the `File|Printer Setup` menu option to start the Print Setup dialog and then click on the "Options" button to start the Options dialog. Ensure that the "Print TrueType as Graphics" checkbox is checked.

Printing An Object's Details

Use the `File|Print` menu option to start the Select Information To Be Printed dialog. To print details of one of the objects, check the "Object Details" radio button and then click on the object name in the listbox.

This page left intentionally blank

11 Loading Available RWI/RWO From File

The Application Editor gives you the ability to import the list of real world inputs (RWIs) and real world outputs (RWOs) which are available to the application. These are then displayed on the External Events dialog so that you can declare RWIs and RWOs for an object in the same way that messages can be declared - by selecting an entry in the outer listbox and clicking on the "Add" button.

Loading The Available RWIs

Use a text editor which can produce a standard ASCII file (such as Notepad) and type the list of names, 1 name per line, into the file. Lines starting with "!" are treated as comments and ignored. Use the File|Load RWI menu option to load the contents of the file. The default file extension is ".ril"

For example, the following file contents would load 3 real world inputs:

```
! This is an example file containing a list of real world inputs
!
! Real world inputs relating to the gripper
rwi_grip_closed
rwi_grip_open
!
! Real world inputs relating to the feed system
rwi_new_part_detected
```

Loading The Available RWOs

Real world outputs are treated as per real world inputs except the File|Load RWO menu option is used and the default file extension is ".rol". In the following example, the file contents would load 4 real world outputs:

```
! This is an example file containing a list of real world outputs
!
! Real world outputs relating to the gripper
rwo_close_grip
rwo_open_grip
!
! Real world outputs relating to the feed system
rwo_start_feed
rwo_stop_feed
```

This page left intentionally blank

Appendix A Rules

The Application Compiler will enforce rules to ensure that the application definition is valid. Many of these rules will be automatically satisfied because the Application Editor will not allow you to violate them. In some cases, however, the Application Editor would have to impose constraints on its functionality which would be overly restrictive, such as when using the cut and paste facilities.

External - Object Related

1. Object name must be legal and valid.
2. Objects must form a strict hierarchy and there must be a root object defined.
3. The root object cannot have any parent commands or parent responses defined.

External - Interface Related

1. Each name must be legal and valid.
2. Each parent command must be defined in the parent's child commands.
3. Each parent response must be defined in the parent's child responses.
4. Each child command must be defined in 1 and only 1 child's parent commands.
5. Each child response must be defined in 1 and only 1 child's parent responses.
6. Each message, real world input and real world output must be referenced by a transition condition/action in at least one of the STDs or sub-sequence STDs belonging to this object.

Internal - Commands

1. Each name must be legal and valid.
2. Each internal command must be sent by at least 1 STD and received by at least 1 STD.
3. Sending and receiving STDs must be different and must ultimately belong to different object STDs (so an object STD cannot send an internal command to one of its sub-sequence STDs for example).

Internal - Variables

1. Each name must be legal and valid.
2. Max must be \geq min (warning issued if they are identical).
3. Initial value must satisfy: $\text{min} \leq \text{initial} \leq \text{max}$.
4. Variable should be referenced by a transition condition/action (warning).
5. Variable value should be changed in at least one transition action (warning).

Internal - Object STD

1. Each name must be legal and valid.
2. There must be an initial state defined for the STD.
3. STD must contain at least 1 state.
4. STD should contain at least 1 transition (warning).

Internal - Sub-Sequence STD

1. Must have 1 and only 1 state named START.
2. Must have 1 and only 1 state named END.
3. There must not be a transition starting at START and ending at END.
4. There must not be a transition starting at END.
5. There must not be a transition ending at START.
6. There must be 1 and only 1 transition starting at START.
7. There must be 1 and only 1 transition ending at END.
8. The transition starting at START must have no conditions.
9. The transition ending at END must have no actions.
10. There must be at least 1 other state in addition to START and END.

Internal - STD State

1. Each name must be legal and valid.
2. If the STD is an object STD or the state is not named START, there should be a transition ending in this state, otherwise this is an unreachable state (warning).

3. If the STD is an object STD or the state is not named END, there should be a transition starting in this state, otherwise this is a terminal state (warning).
4. If the state is a macro state, there must be at least 1 transition ending in this state. There should also be 1 transition starting at this state (error if no transitions start at this state, warning if more than 1 transition because it cannot have any conditions i.e. undefined which transition will be fired).

Internal - STD Transition

1. A transition ending at a macro state must have no actions.
2. A transition starting at a macro state must have no conditions.
3. A transition's conditions cannot have 2 or more variable tests referring to the same variable (where synchronous internal commands are used, this rule applies to the combination of the sending and receiving transition).
4. A transition's actions cannot have 2 or more variable operations referring to the same variable (where synchronous internal commands are used, this rule applies to the combination of the sending and receiving transition).
5. The value in a variable test must satisfy: $\text{min} \leq \text{value} \leq \text{max}$.
6. A variable referenced by a variable test or variable operation must be declared for the object to which the STD belongs.
7. A message, real world input or real world output referenced by a condition or action must be declared for the object to which the STD belongs.
8. For transitions other than those starting or ending at a macro state and transitions starting at a sub-sequence STD's START state or ending at a sub-sequence STD's END state, the transition should have at least 1 condition and at least 1 action (warning).
9. If the value in a variable test is equal to the variable's minimum value, the test type cannot be "<".
10. If the value in a variable test is equal to the variable's minimum + 1 and the test type is "<", there must not be an action in this transition which decrements the variable's value (where synchronous internal commands are used, this rule applies to the combination of the sending and receiving transition).
11. There should be no variable test of the form "var <= min" (warning).
12. If the variable test is of the form "var <= min", there must be not be an action in this transition which decrements the variable's value (where synchronous internal commands are used, this rule applies to the combination of the sending and receiving transition)..

13. If the value in a variable test is equal to the variable's maximum value, the test type cannot be ">".
14. If the value in a variable test is equal to the variable's maximum - 1 and the test type is ">", there must not be an action in this transition which increments the variable's value (where synchronous internal commands are used, this rule applies to the combination of the sending and receiving transition).
15. There should be no variable test of the form "var >= max" (warning).
16. If the variable test is of the form "var >= max", there must be not be an action in this transition which increments the variable's value (where synchronous internal commands are used, this rule applies to the combination of the sending and receiving transition).
17. There must not be 2 or more transitions which contain a reference to any given variable and a synchronous message. The scope of this rule is the whole of the object i.e. the object STDs and sub-sequence STDs.

Appendix B Formal Definition Of An Application

This chapter provides a more precise definition of the components of an application. The notation is as follows:

=	consists of
+	and
	or
[]	optional
{ }	set of zero or more
'abc'	the literal character string <i>abc</i>

application = hierarchy of objects

hierarchy of objects = root object [+ sub-hierarchy of objects]

sub-hierarchy of objects = object [+ sub-hierarchy of objects]

root object = object name + child messages + {real world input}
 + {real world output}
 + internal events + {variable}
 + list of object stds

object = root object + parent messages

object name = name

name = letter + {letter | digit | underscore}

child messages = {child command} + {child response}

child command = message name

child response = message name

message name = synchronous message name | asynchronous message name

synchronous message name = name

asynchronous message name = '[' + name + '']

parent messages = {parent command} + {parent response}

parent command = message name

parent response = message name

real world input = name

real world output = name

internal events = {internal command} + {variable test} + {variable operation}

internal commands = message name

variable test = variable name + test type + test value

variable name = name

test type = < | <= | = | >= | >

test value = integer value subject to:

variable minimum <= test value <= variable maximum

variable operation = variable name + operation type

operation type = INCR | DECR | RESET

variable = variable name + variable minimum + variable maximum
+ variable initial value

variable minimum = integer value

variable maximum = integer value

variable initial value = integer value

subject to: -999 <= integer value <= 9999

variable minimum <= variable maximum

variable minimum <= variable initial value <= variable maximum

list of object stds = std + {std}

std = name + initial state + state + {state} + {transition}

state = name [+ sub-sequence STD]

transition = start state + end state + {condition} + {action}

start state = state

end state = state

sub-sequence STD = sub-sequence start state + sub-sequence end state

+ state + {state}

+ sub-sequence transition from start

+ sub-sequence transition to end

+ {transition}

sub-sequence start state = "START"

sub-sequence end state = "END"

sub-sequence transition from start = "START" + end state + {condition}
+ {action}

sub-sequence transition to end = start state + "END" + {condition} + {action}

condition = parent command | child response | real world input
| state reference | internal command | variable test

action = parent response | child command | real world output
| internal command | variable operation

Appendix C Menus

File	
New Application	Create a new application
Open Application	Open an existing application
Save Application	Save this application
Save Application As	Save this application with a new name
Load RWI	Load the available set of real world inputs
Load RWO	Load the available set of real world outputs
Print	Print the contents of the active window or object details
Printer Setup	Set up the print characteristics for this application
Exit	Finish running the Application Editor
Edit	
Cut	Deletes the selection after copying it into the paste buffer
Copy	Copies the selection into the paste buffer
Paste	Inserts the contents of the paste buffer
Delete	Deletes the selection
Cancel	Cancels the paste operation
Zoom	
In	Zoom In
Out	Zoom Out
Reset	Reset magnification to original value
Grid	
Show Grid	Show or hide the grid
Configuration	
Grid Size	Size of grid (in pixels)
Object Name Font	Font to use for object names
Object Interface Font	Font to use for object's external interface on the Object Hierarchy
State Name Font	Font to use for state names
Transition Name Font	Font to use for transition conditions and actions
Zoom Factor	Amount by which to zoom in/out
Window	
Cascade	Cascade open windows
Tile	Tile open windows
Arrange Icons	Arrange iconic windows along bottom
Close All STD Windows	Close all STD windows
Help	
Contents	Help table of contents
Using help	Help on using online Help
About	Information about Synect Application Editor










This page left intentionally blank

Appendix D Toolbar Buttons

The Application Editor uses 2 toolbars – the Operation Toolbar is displayed horizontally across the top of the screen and the Mode Toolbar is displayed vertically down the left side of the screen.

Operation Toolbar







The Operation Toolbar contains buttons which can be used as shortcuts instead of pulling down the corresponding menu and selecting the relevant item. The buttons will be greyed-out if the corresponding function is unavailable at that time.

	<u>Menu Equivalent</u>	<u>Usage</u>
	File Open Application	Load an application from a file on disk
	File Save Application	Save the application being edited to file on disk
	Edit Cut	Delete the selection after copying it into the paste buffer
	Edit Copy	Copy the selection into the paste buffer
	Edit Paste	Insert the contents of the paste buffer
	Zoom In	Make the diagram bigger
	Zoom Out	Make the diagram smaller
	Edit Cancel	Cancel the current paste operation
	Help Contents	Display help information

Mode Toolbar

- The Mode Toolbar allows you to change the mode in which the Application Editor is being used. The buttons can be divided into 3 groups:
- Buttons applicable to the Object Hierarchy window and STD windows
 - Buttons applicable only to the Object Hierarchy window
 - Buttons applicable only to STD windows

The buttons will be greyed-out if the corresponding function is unavailable at that time.

	<u>Applicable To</u>	<u>Usage</u>
	Object Hierarchy and STDs	Edit mode. Allows items to be selected and moved on the diagram
	Object Hierarchy and STDs	Move mode. Allows the whole diagram to be moved
	Object Hierarchy only	Add a new root object
	Object Hierarchy only	Add a new child object
	STD only	Add a new state
	STD only	Add a new transition

Synect

Compiler User Guide

Version 1.2

**Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough
Cleveland, TS14 8JY
England**

**Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk**

© Copyright 1994, 1995, 1996 Hopkinson Computing Limited. All rights reserved.

Synect is a registered trademark of Hopkinson Computing Limited

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Information in this User Guide is subject to change without notice and does not represent a commitment on the part of Hopkinson Computing Limited.

The software described in this User Guide is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. No part of this User Guide may be reproduced or transmitted in any form or by any means, electronic or otherwise, including photocopying and recording, for any purpose, without the express written permission of Hopkinson Computing Limited.

Document History

10 June 1996

Re-issue reflecting Compiler V1.1 - new cover sheet and chapter 1.

28 October 1996

Re-issue reflecting Compiler V1.2 - changes to cover sheet and chapters 2 & 5.

Contents

Chapter 1	Introducing Synect	1
	User Interface	1
	The Method	1
	Synect Documentation	1
	The Tools	2
	Document Conventions	2
Chapter 2	Getting Started	3
	System Requirements	3
	Installation	3
	Starting the Compiler	4
	On-Line Help	5
Chapter 3	Basic Concepts	7
	Subsystem Analyses	7
	Subsystem Only	7
	Subsystem Except	8
	Understanding The Compiler Output	8
	Dealing With Line Breaks	8
	Synchronous Messages	8
	Asynchronous Messages	9
	Variables	10
Chapter 4	Open And Save An Application	11
	Opening An Application	11
	Saving An Application	11
Chapter 5	Compiling The Application	13
	Starting The Compilation	13
	List File	14
	Analysis of User Specification	14
	Analysis of Compiled Application	14
	Cross-Reference File	15
	State Reference Notation	15
	Transition Reference Notation	15
	Model Dimensions	15
	Initial States	15
	Place to State Cross-Reference	16

	Transition to STD Transition Cross-Reference	16
Chapter 6	Using Subsystem Analyses	17
	Subsystem Only	17
	Subsystem Except	17
Appendix A	Menus	19
Appendix B	ToolBar Buttons	21

1 Introducing Synect

Synect is a set of software tools which helps the designer of a control system to produce a specification which is clear, precise and free of errors. Synect combines the ease of use of a graphical user interface, with a widely used diagrammatic notation and the power of mathematical modelling.

User Interface

The Synect™ tools run on the Microsoft® Windows™ operating system. As such, you need to know how to use Windows before you use Synect. In particular, you will need to know how to use the mouse to click, double-click and drag. You will also need to know how to interact with menus, dialogs and how to move, resize and close windows. For information, refer to the Microsoft Windows User's Guide.

The Method

Synect uses an object-based method to enable you to model the system you want to control. It is useful to have some knowledge of object-based/object-oriented analysis and design techniques before attempting to use Synect in earnest.

Synect Documentation

Each of the Synect tools has an associated User Guide. This User Guide explains how to use each of the functions available in the Compiler. It also explains why you might want to use the function.

Each application has context-sensitive on-line help. The Compiler on-line help contains a "How Do I?" section, including a "How Do I Get Started?" sub-section for first-time users.

A Tutorial is also provided which offers a worked example and shows how each of the tools is used with the example application.

The Tools

The tools which make up the Synect toolset are:

Application Editor	graphical means of defining the application.
Compiler	check the specification for consistency and possible warnings and generate a mathematical model of the application.
Analyzer	ability to check for design errors such as deadlock (where the system "hangs") and unwanted state combinations.
Simulator	provides the ability to interactively "drive" the application or replay past behaviour of the live control system.
STD Monitor	animates the specification (used in conjunction with the Simulator or the live control system).
C Code Generator	generate ANSI-standard C code to implement the application.
Neuron C Generator	generate Neuron C to run on one or more nodes on an Echelon LonWorks network to implement a distributed control solution.
Ladder Logic Generator	generate relay ladder logic to run on a programmable controller.

Document Conventions

The User Guide adopts the following conventions:

<code>application name</code>	text that you type or that you see on the screen.
<code>KEY NAME</code>	keyboard keys, such as ENTER, CTRL or DEL.
<code>Menu Choice</code>	a menu option, such as File Exit denoting choose the Exit command from the File menu.
<i>description</i>	description of a term with a specific meaning.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the Compiler.
- how to start the Compiler.
- the Compiler window.

System Requirements

The Synect Compiler requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

Other Synect tools require that you also have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

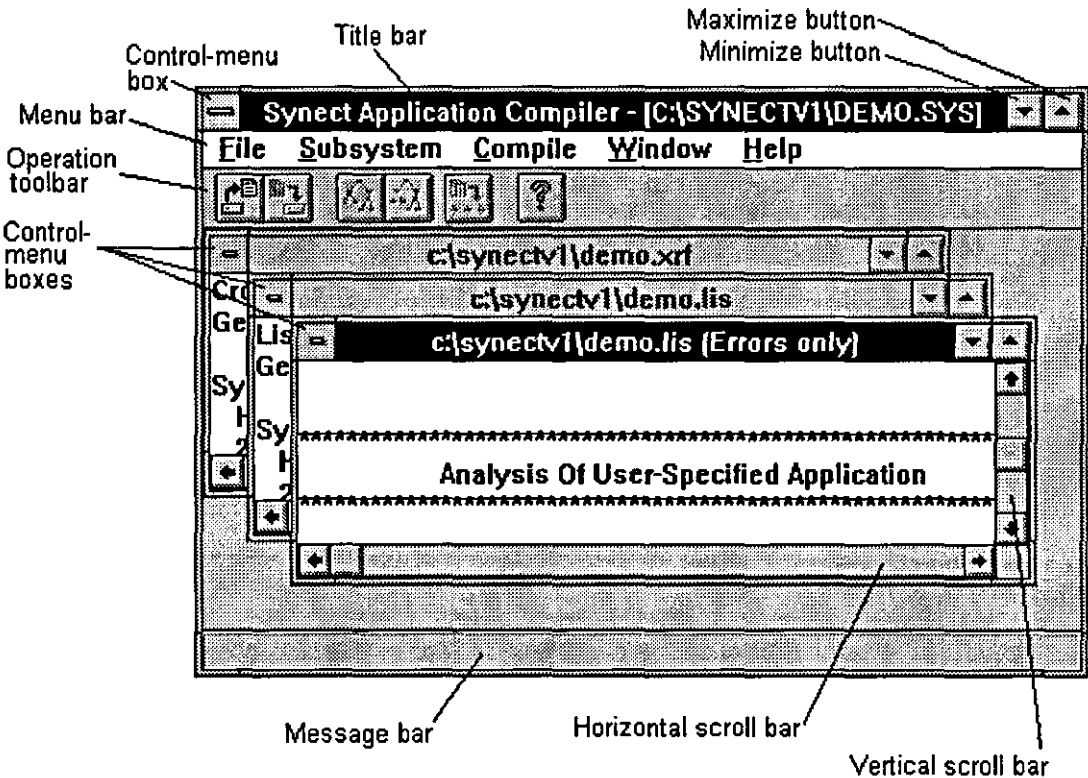
- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press ENTER.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

- 8 Read the installed file `readme.txt` for details of any changes to the product or documentation since the documentation was printed.

Starting the Compiler

Ensure that the Program Manager window is on display. If the Synect window isn't visible, use the Window menu to open it. Double-click on the Compiler icon or use the keyboard arrow key to select the icon and then press ENTER.

When using the Compiler, the window will typically look like the following:



Title bar
The outer window is the Synect Compiler window. The title bar therefore shows the product title `Synect Application Compiler` and the name of the application which has been loaded (if any).

The inner windows show text files which are produced when the Compiler compiles your application. The title bar of an inner window will be a filename appended with:

- `".xrf"` cross-reference
- `".lis"` the list file (warnings and errors)
- `".lis (Errors Only)"` the list file (errors only i.e. ignoring warnings)

Control-menu boxes

Allows you to restore, move, size, minimize, maximize or close the window.
Also allows you to make another window the active window.

Menu bar

Lists the available menus.

Minimize box

Allows you to shrink the Synect window to an icon at the bottom of the screen.

Maximize box

Allows you to enlarge the Synect window to fill the entire screen.

Operation toolbar

Contains the momentary buttons which can be used as menu shortcuts.

Message bar

When the cursor is moved over an Operation Toolbar button or when a menu option is highlighted, the message bar will show a brief description of the function of the button/menu option.

Horizontal and vertical scroll bars

Allow you to pan around the text file to change the portion which is displayed in the window.

On-Line Help



The on-line help is context-sensitive. So if you click on a help button in a dialog, you will automatically be shown the help information associated with that dialog. If no dialog is being displayed, you can choose [Help|Contents](#) to take you to the help contents page. The help information is shown in a separate window.

This page left intentionally blank

3 Basic Concepts

The Synect Application Editor enables you to specify a model of your application. See the Application Editor User Guide, chapter 3, Basic Concepts for more information relating to defining the model. The Compiler first checks your specification for consistency (see the Application Editor User Guide, Appendix A, Rules, for further details). If there are no errors, it then derives a mathematical model from your specification which it then examines for possible inefficiency in the design. The model may subsequently be:

- analysed by the Application Analyzer for behavioural properties.
- interpreted by the Simulator which can drive the STD Monitor application to animate the application.
- used by the C Code Generator to generate ANSI-standard C to implement the specification.

Subsystem Analyses

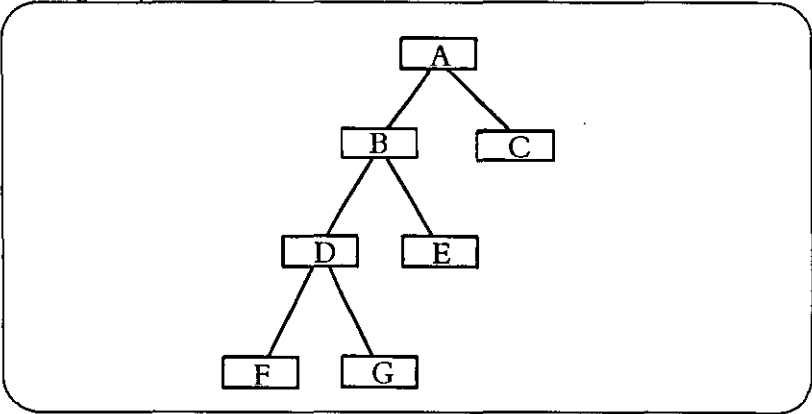
The Compiler also enables you to deal with a subset of your application. When used in conjunction with the Analyzer, this might help you identify:

- if a part of your application deadlocks (although the whole application may not deadlock).
- whether an object enforces the necessary constraints on its child objects.

It may also be useful for generating the control system code for a subset of the whole application, therefore facilitating stepwise integration.

Subsystem Only

The first of the alternatives is to consider only that portion of the hierarchy including and below the specified object (i.e. the object, its children, grand-children, etc.). This makes the specified object the root object of the new hierarchy. In the following example, using the Subsystem Only facility to nominate object D as the new root object will result in an Object Hierarchy consisting only of objects D, F and G.



**Subsystem
Except**

The other alternative is to exclude a nominated object from the hierarchy. This will automatically exclude its children, grandchildren, etc.. In the example above, nominating object D as the object to exclude will result in an Object Hierarchy consisting of objects A, B, C and E.

Understanding The Compiler Output

As described in chapter 5, Compiling, the Compiler provides traceability to show how it derived the mathematical model from your specification. This information is mostly self-explanatory but the less intuitive aspects are described below.

**Dealing
With Line
Breaks**

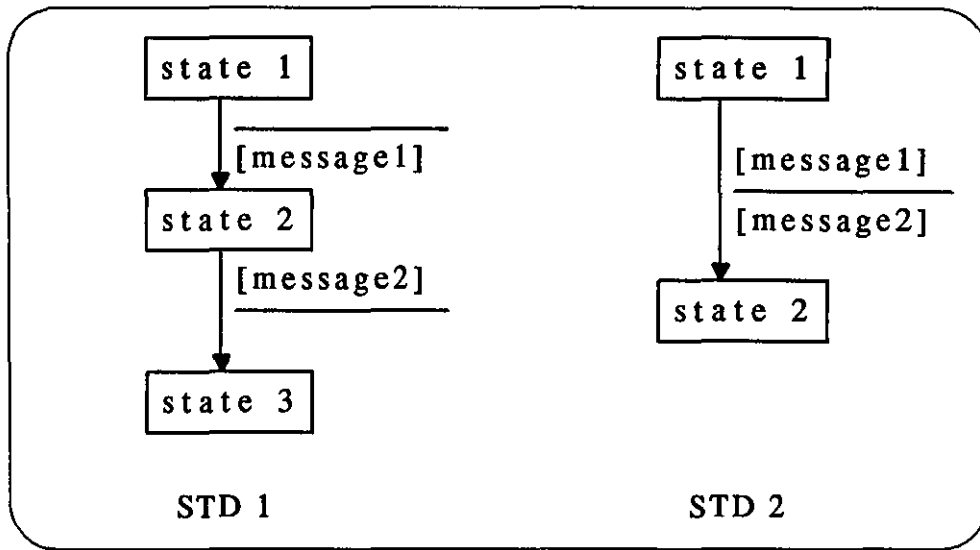
State names and object names can have embedded line breaks i.e. they can be formatted to be displayed over several lines. In order to be able to show any name on one line, line breaks are shown using the "|" character. For example, the following state from an STD would be represented as at|home|position:

a t
home
position

**Synchronous
Messages**

A synchronous message causes the sending transition and the receiving transition to fire as one, thereby synchronising the 2 transitions. If the receiving STD is not in a position to receive the message, the transition in the sending STD cannot fire.

Under specific circumstances, the Compiler may cause a state to be by-passed. Consider the following 2 STDs which communicate via synchronous messages:



When STD "STD 1" is in state "state 1" and it fires the transition to change state to "state 2", it sends synchronous message "[message1]" to STD "STD 2". When STD "STD 2" is in state "state 1" and it receives synchronous message "[message1]", it changes state to "state 2" and sends synchronous message "[message2]" to STD "STD 1". When STD "STD 1" is in state "state 2" and it receives synchronous message "[message2]", it changes state to "state 3". Considering that the definition of a synchronous message is that the transitions fire as one, it follows that the result of the above sequence must be that STD "STD 1" changes state from "state 1" to "state 3". The only way this can be achieved is for the Compiler to replace the individual transitions by a single transition between states "state 1" and "state 3". This will result in the Analyzer reporting that state "state 2" is an unreachable state. When monitoring STD "STD 1" using the STD Monitor, you'll see that when the transition between states "state 1" and "state 2" is enabled, the transition between states "state 2" and "state 3" is also enabled. When the transition fires, STD "STD 1" will change from state "state 1" directly to state "state 3", by-passing state "state 2" altogether.

Asynchronous Messages An asynchronous message is buffered by the sender. When the message is in the buffer, it is said to be *pending*. If the message is already pending, the transition in the sending STD cannot fire.

The Compiler deals with an asynchronous message by creating an STD with states: "#pending" and "#not_pending" denoting whether the message is already in the buffer. If you use asynchronous messages in your application, you will see references to STDs in addition in the ones you specified when you examine the Compiler listing file or cross-reference file. The additional STD names will be the same as the asynchronous message names.

Variables

The Compiler deals with a variable by creating an STD with a state for each value between the variable's minimum and maximum. If you use variables in your application, you will see references to STDs in addition in the ones you specified when you examine the Compiler listing file or cross-reference file. The additional STD names will be the same as the variable names.

The difference between each variable's minimum and maximum (its range) should be kept as small as possible. Otherwise, the Compiler may not be able to compile the application or the Analyzer may not be able to cope with the number of combinations of system state.

4 Open And Save An Application

When you first start the Synect Compiler, very few of the menu items or buttons are enabled. This is because you must first open an application to compile.

When you saved your application from the Application Editor, it created 2 files. One of these files is used solely by the Application Editor. The other file has the same filename you specified for your application but with the extension ".sys". It is this file which the Compiler uses.

Opening An Application



To open an application, choose the File|Open Application menu option. The standard file open dialog will then be started, allowing you to specify the name of the file from which the application is to be loaded.

Saving An Application



The File|Save As menu option allows you to save your application to a different file. If you've changed the application by excluding one or more objects, the Compiler will not allow you to compile until you've saved the revised application to file (choose a different filename from the one you originally opened). This is to avoid the possibility of generating misleading results when interpreting the Compiler output information.

This page left intentionally blank

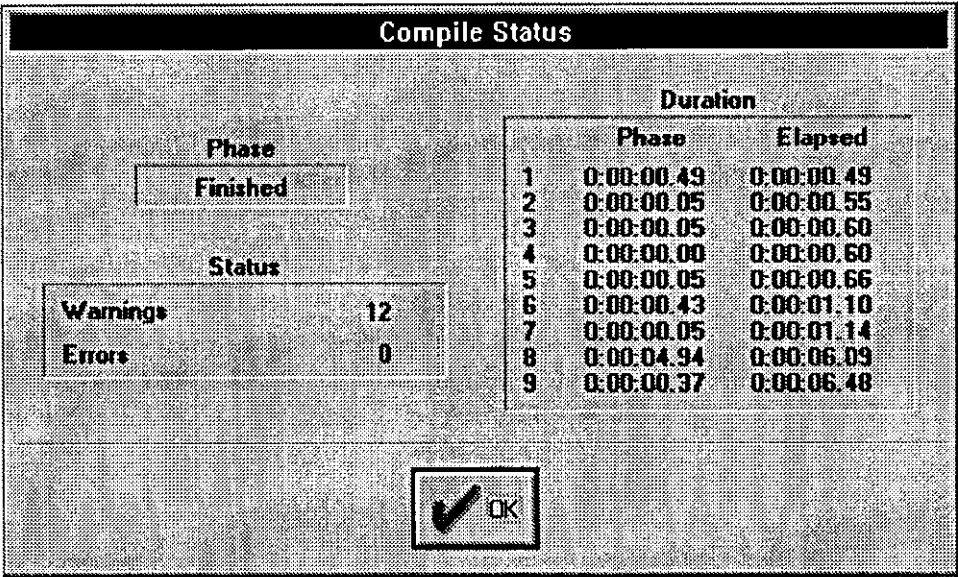
5 Compiling The Application

This chapter describes how to compile an application and the information which you'll see as the application is being compiled. It also describes the contents of the two files which are displayed in read-only windows when the compilation is completed.

Starting The Compilation



When you've opened your application, choose the `Compile|Compile` menu option to begin compilation. The `Compile Status` dialog will be displayed, showing the progress. When completed, this dialog will look like the following:



During compilation, the "OK" button will be a "Cancel" button. Clicking on the "Cancel" button will abort the compilation. When the compilation has completed (the "Cancel" button has been replaced with the "OK" button), click on the "OK" button to close the `Compile Status` dialog.

Phase 1 of the compilation checks to ensure that application conforms to the rules defined in the `Application Editor User Guide`, `Appendix A, Rules`. If errors are found, the number of errors will be displayed on the `Compile Status` dialog and the compilation will be aborted. The list file will show details of the errors found.

List File

Following a compilation, a list file is created with the same name as your application but with extension ".lis". This is displayed in two windows in the Compiler. One of the windows shows the entire list file whereas the other just shows errors. If the application was found to violate any of the rules defined in the Application Editor User Guide, Appendix A, Rules, and this resulted in an error, the list file will show only limited information because the compilation will have been aborted.

Analysis of User Specification

The first set of information in the list file relates to your specification i.e. the Object Hierarchy, messaging and STDs. Inconsistencies, such as a message declared as sent from an object but not declared as received by any other, are listed in this section. Warnings may also be listed to notify you of, for example, transitions which have no conditions or actions. Refer to the Application Editor User Guide, Appendix A, Rules, for details of the checks which are performed.

A warning report starts with "***WARNING**", an error report starts with "***ERROR***". The name of the object follows and then the name of the STD.

- If the report refers to a state, the state name is listed, followed by the details of the warning or error.
- If the report refers to a transition, the transition number within the STD is shown followed by the details of the warning or error. The transition number can be ignored – the transition's start and end states are listed immediately below the report.

Analysis of Compiled Application

The next part of the list file is an analysis of the mathematical model generated by the Compiler. Transitions in the mathematical model link STD transitions together. Synect assumes that the application is a reactive system i.e. that the application exists to respond to events received from the real world (real world inputs) by taking the specified course of action – typically by exerting some influence on the system being controlled (real world outputs). As such, warnings will be listed if, for example, a transition is found which references no real world inputs and no real world outputs.

The report starts with "***WARNING**" and is followed by the mathematical model transition number. The transition number is listed for traceability reasons and can usually be ignored because the STD start states and end states are listed immediately below.

Cross-Reference File

Provided that the compilation was successful, a cross-reference file is created with the same name as your application but with extension ".xrf". This is displayed in a window in the Compiler. The contents of this file show how the Compiler has derived the mathematical model from your specification. You don't need to use this information to be able to use the Synect tools - it is provided for traceability.

**State
Reference
Notation**

In the cross-reference file, the general format of a reference to an STD state is:

`<object>.<std> <state>`

For example, object "Robot" with STD "main" and state "at|home" (remembering that the "|" character represents a line break) would be shown as:

`Robot.main at|home`

A sub-sequence STD's state would be referred to as:

`<object>.<std>.<sub-sequence std> <state>`

For example, object "Robot" with STD "main" and sub-sequence STD "getting|raw|part" in state "START" would be shown as:

`Robot.main.getting|raw|part START`

**Transition
Reference
Notation**

In the cross-reference file, the general format of a reference to an STD transition is:

`<object>.<std> <STD transition number>`

A reference to an STD transition in a sub-sequence STD would be referred to as:

`<object>.<std>.<sub-sequence std> <STD transition number>`

The transition's start states and end states are listed below the references to the STD transitions.

**Model
Dimensions**

The first set of information in the cross-reference file refers to the dimensions of the derived mathematical model. It shows the number of places (equivalent to STD states) and transitions (equivalent to combined STD transitions).

Initial States

The initial states are then listed, using the format defined above.

**Place to
State
Cross-**

The STD state from which each place in the mathematical model has been derived is then listed. This is shown in the following format:

<place number> <reference to STD state>

**Transition to
STD
Transition
Cross-**

The final set of information in the file shows the STD transitions from which each transition in the mathematical model has been derived.

6 Using Subsystem Analyses

The Compiler enables you to deal with a subset of the overall application. Chapter 3, Basic Concepts, describes why you might consider using this facility. This chapter describes how to use the facility and explains what the Compiler does.

Subsystem Only



The Subsystem Only facility enables you to consider only that portion of the Object Hierarchy including and below a nominated object. Choose the **Subsystem|Only** menu option to cause the Compiler to start the New Root Object dialog. This lists each of the object names. Click on the entry which is to be the new root object and then click on "OK". Choose the **File|Save As** menu option to start the standard file save dialog and specify a new filename into which the modified application is to be saved.

Having used the Subsystem Only facility, you must save the application before being able to start the compilation. You should specify a filename other than the one originally loaded. This is to prevent possible confusion when using the compilation results. If this were not enforced, the compiler output would relate to a subset of the whole application but this might not be apparent because the filename would correspond with that of the whole application.

The advantage of using this facility in the Compiler rather than the cut and paste facility in the Application Editor is that the Compiler attempts to ensure that the application is consistent. It knows that a root object cannot have any messages to or from a parent because, by definition, a root object has no parent. Consequently, it removes any parent messages from the new root object's external interface definition and from any transitions in its STDs.

Subsystem Except



The Subsystem Except facility enables you to exclude a portion of the Object Hierarchy including and below a nominated object. Choose the **Subsystem|Except** menu option to cause the Compiler to start the Object To Exclude dialog. This lists each of the object names. Click on the entry which is to be excluded and then click on "OK". Choose the **File|Save As** menu option to start the standard file save dialog and specify a new filename into which the modified application is to be saved.

Having used the Subsystem Except facility, you must save the application before being able to start the compilation. You should specify a filename other than the one originally loaded. This is to prevent possible confusion when using the compilation results. If this were not enforced, the compiler output would relate

to a subset of the whole application but this might not be apparent because the filename would correspond with that of the whole application.

The advantage of using this facility in the Compiler rather than the cut and paste facility in the Application Editor is that the Compiler attempts to ensure that the application is consistent. It takes each of the excluded object's parent messages and amends the object which was its parent to remove them from the external interface definition and STD transitions.







Appendix A Menus

File	
Open Application	Open an existing application
Save As	Save the application with a new name
Exit	Finish running the Application Compiler
Subsystem	
Only	Consider only a part of the object hierarchy
Except	Exclude part of the object hierarchy
Compile	
Compile	Compile the application
Window	
Cascade	Cascade open windows
Tile	Tile open windows
Arrange Icons	Arrange iconic windows along bottom
Close Windows	Close all open windows
Help	
Contents	Help table of contents
Using help	Help on using online Help
About	About Synect Application Compiler

This page left intentionally blank

Appendix B Toolbar Buttons

The Application Compiler provides an Operation Toolbar displayed horizontally across the top of the screen. It contains buttons which can be used as shortcuts instead of pulling down the corresponding menu and selecting the relevant item. The buttons will be greyed-out if the corresponding function is unavailable at that time.

	<u>Menu Equivalent</u>	<u>Usage</u>
	File Open Application	Load an application from a file on disk
	File Save As	Save the application to a new file on disk
	Subsystem Only	Consider only a part of the object hierarchy
	Subsystem Except	Exclude part of the object hierarchy
	Compile Compile	Compile the application
	Help Contents	Display help information

Synect

Analyzer User Guide

Version 1.2

**Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough
Cleveland, TS14 8JY
England**

**Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk**

© Copyright 1994, 1995, 1996 Hopkinson Computing Limited. All rights reserved.

Synect is a registered trademark of Hopkinson Computing Limited

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Information in this User Guide is subject to change without notice and does not represent a commitment on the part of Hopkinson Computing Limited.

The software described in this User Guide is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. No part of this User Guide may be reproduced or transmitted in any form or by any means, electronic or otherwise, including photocopying and recording, for any purpose, without the express written permission of Hopkinson Computing Limited.

Document History

10 June 1996

Re-issue reflecting Analyzer V1.1 - new cover sheet and chapter 1.

28 October 1996

Re-issue reflecting Analyzer V1.2 - new cover sheet only.

Contents

Chapter 1	Introducing Synect	1
	User Interface	1
	The Method	1
	Synect Documentation	1
	The Tools	2
	Document Conventions	2
Chapter 2	Getting Started	3
	System Requirements	3
	Installation	3
	Starting the Analyzer	4
	On-Line Help	5
Chapter 3	Basic Concepts	7
	How It Works	7
	Reachability Tree	7
	Limitations	8
	Output Information From The Analyzer	9
	Deadlocks	9
	Unreachable States	10
	Dead Transitions	10
Chapter 4	Open An Application	11
	Opening An Application	11
Chapter 5	Generating The Reachability Tree	13
	Starting The Generation Of The Reachability Tree	13
	Analyzer List File	14
	Deadlocks	14
	Unreachable States	14
	Dead Transitions	14
Chapter 6	Querying Deadlocks	15
	The Deadlocks Dialog	15
	Navigating	15
	Saving Information To File	15
Chapter 7	State Search Querying	17
	The State Search Dialog	17
	Specifying The Query	17

	Executing The Query	17
	Viewing The Results	17
	Saving Information To File	18
	Returning To Define Query Mode	18
Appendix A	Menus	19
Appendix B	ToolBar Buttons	21

1 Introducing Synect

Synect is a set of software tools which helps the designer of a control system to produce a specification which is clear, precise and free of errors. Synect combines the ease of use of a graphical user interface, with a widely used diagrammatic notation and the power of mathematical modelling.

User Interface

The Synect™ tools run on the Microsoft® Windows™ operating system. As such, you need to know how to use Windows before you use Synect. In particular, you will need to know how to use the mouse to click, double-click and drag. You will also need to know how to interact with menus, dialogs and how to move, resize and close windows. For information, refer to the Microsoft Windows User's Guide.

The Method

Synect uses an object-based method to enable you to model the system you want to control. It is useful to have some knowledge of object-based/object-oriented analysis and design techniques before attempting to use Synect in earnest.

Synect Documentation

Each of the Synect tools has an associated User Guide. This User Guide explains how to use each of the functions available in the Analyzer. It also explains why you might want to use the function.

Each application has context-sensitive on-line help. The Analyzer on-line help contains a "How Do I?" section, including a "How Do I Get Started?" sub-section for first-time users.

A Tutorial is also provided which offers a worked example and shows how each of the tools is used with the example application.

The Tools

The tools which make up the Synect toolset are:

Application Editor	graphical means of defining the application.
Compiler	check the specification for consistency and possible warnings and generate a mathematical model of the application.
Analyzer	ability to check for design errors such as deadlock (where the system "hangs") and unwanted state combinations.
Simulator	provides the ability to interactively "drive" the application or replay past behaviour of the live control system.
STD Monitor	animates the specification (used in conjunction with the Simulator or the live control system).
C Code Generator	generate ANSI-standard C code to implement the application.
Neuron C Generator	generate Neuron C to run on one or more nodes on an Echelon LonWorks network to implement a distributed control solution.
Ladder Logic Generator	generate relay ladder logic to run on a programmable controller.

Document Conventions

The User Guide adopts the following conventions:

<i>application name</i>	text that you type or that you see on the screen.
KEY NAME	keyboard keys, such as ENTER, CTRL or DEL.
Menu Choice	a menu option, such as File Exit denoting choose the Exit command from the File menu.
<i>description</i>	description of a term with a specific meaning.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the Analyzer.
- how to start the Analyzer.
- the Analyzer window.

System Requirements

The Synect Analyzer requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

The Analyzer also requires that you have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

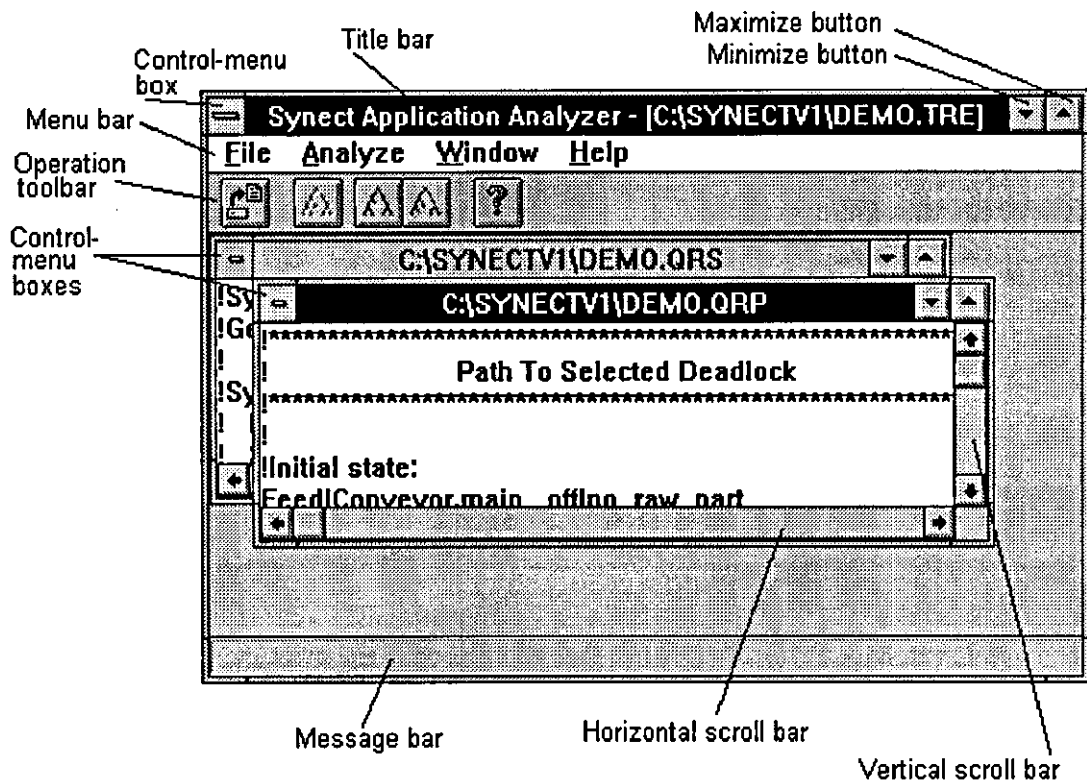
- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press `ENTER`.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

- 8 Read the installed file `readme.txt` for details of any changes to the product or documentation since the documentation was printed.

Starting the Analyzer

Ensure that the Program Manager window is on display. If the Synect window isn't visible, use the Window menu to open it. Double-click on the Analyzer icon or use the keyboard arrow key to select the icon and then press ENTER.

When using the Analyzer, the window will typically look like the following:



Title bar

The outer window is the Synect Analyzer window. The title bar therefore shows the product title `Synect Application Analyzer` and the name of the application which has been loaded (if any).

The inner windows show text files which are produced by the Analyzer. The title bar of an inner window will be a filename with one of the following extensions:

`".als"`

Analyzer list file - produced when the Analyzer explores the system states.

<code>".qrs"</code>	query result summary - either the system state at each deadlock or the system states satisfying the query criteria.
<code>".qrp"</code>	query result path - either the path from the start state to the currently selected system state.

Control-menu boxes

Allows you to restore, move, size, minimize, maximize or close the window.
Also allows you to make another window the active window.

Menu bar

Lists the available menus.

Minimize box

Allows you to shrink the Synect window to an icon at the bottom of the screen.

Maximize box

Allows you to enlarge the Synect window to fill the entire screen.

Operation toolbar

Contains the momentary buttons which can be used as menu shortcuts.

Message bar

When the cursor is moved over an Operation Toolbar button or when a menu option is highlighted, the message bar will show a brief description of the function of the button/menu option.

Horizontal and vertical scroll bars

Allow you to pan around the text file to change the portion which is displayed in the window.

On-Line Help



The on-line help is context-sensitive. So if you click on a help button in a dialog, you will automatically be shown the help information associated with that dialog. If no dialog is being displayed, you can choose [Help|Contents](#) to take you to the help contents page. The help information is shown in a separate window.

This page left intentionally blank

3 Basic Concepts

The Synect Application Editor enables you to specify a model of your application. See the Application Editor User Guide, chapter 3, Basic Concepts for more information relating to defining the model. The Compiler derives a mathematical model from your specification. The Analyzer explores the different combinations of state that your application can reach and enables you to query this information.

How It Works

The Analyzer enables you to interactively examine the behavioural properties of the application. These fall into 2 categories:

deadlock	where the application will "hang". For example because one part is waiting for a second part which is waiting for the first part.
state search	"can the application get into a specified combination of states".

The analysis results can be saved to file and displayed in windows in the Analyzer.

Reachability Tree

To be able to support these analyses, the Analyzer must first explore all of the different states which the application can reach. It does this by generating the *Reachability Tree*. If you have a large application, or many concurrent activities, this could take a long time or may even fail due to lack of memory. The following steps should be taken to keep the application manageable:

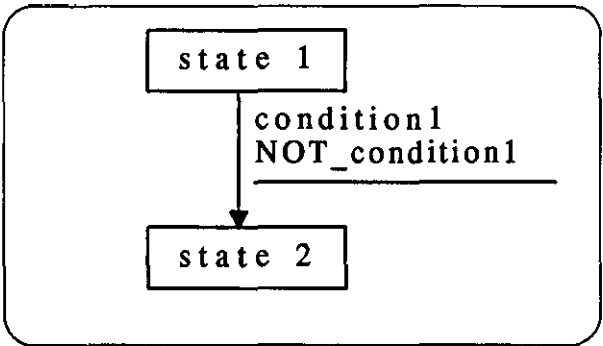
- use synchronous messaging where possible.
- if variables are used, make the range (max - min) as small as possible.
- make sure that all objects in the object hierarchy are necessary for the application.

If you used the Compiler to investigate a subset of your whole application, you can use the Analyzer to investigate the behaviour of the subsystem. This is the preferred method of analysing applications because:

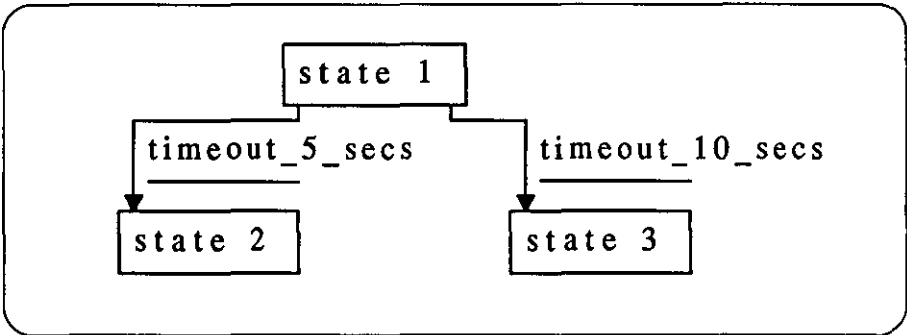
- the Analyzer is more likely to be able to generate the Reachability Tree for the subsystem than for a large application.
- it offers greater confidence in the overall design because it has been subjected to more detailed investigation.
- there may be greater confidence in the re-usability of a subsystem because its has been investigated in isolation.

Limitations

When the Analyzer explores the different paths which the application can follow in order to arrive at all of the possible combinations of system state which it can reach, it ignores the real-world inputs. This could lead to erroneous analyses. In the following example, the transition has 2 real world inputs which control whether the transition can fire. However, the second condition is the logical opposite of the first. Consequently, they can never both be TRUE and, in the target control system, this transition would never fire. The Analyzer, however, ignores the real world inputs and therefore assumes that the transition can fire.



In the next example, the real world inputs read a timer. The transition from state "state 1" to state "state 2" will be enabled after 5 seconds whereas the transition from state "state 1" to state "state 3" would be enabled after 10 seconds. However, the real world input reading the 5 second timeout would always be enabled before the real world input reading the 10 second timeout. State "state 3" would therefore be unreachable. The Analyzer, however, ignores the real world inputs and would therefore explore the path followed by firing the transition from state "state 1" to state "state 3". In practice, the fact that the Analyzer ignores timing information can help you to find errors which only manifest themselves due to subtle changes in the time ordering of real world events.

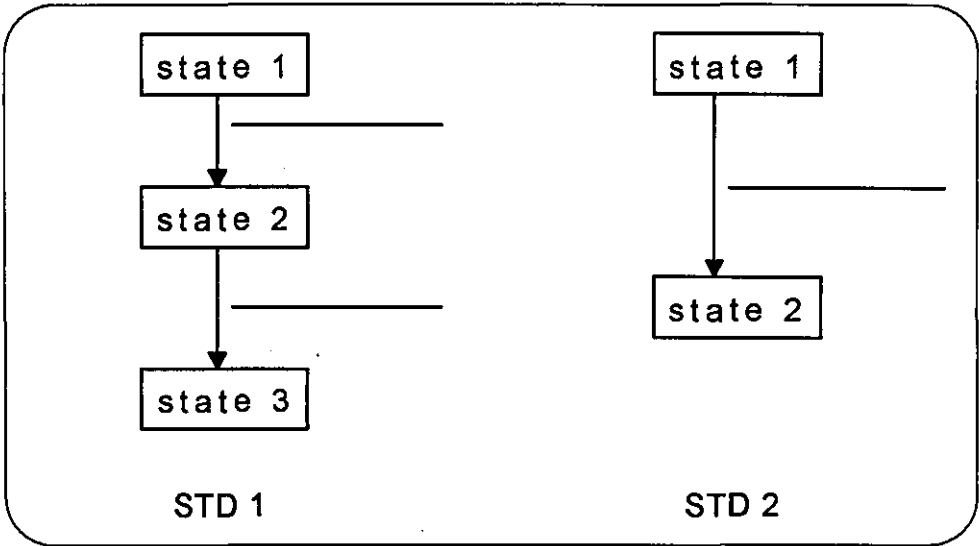


Output Information From The Analyzer

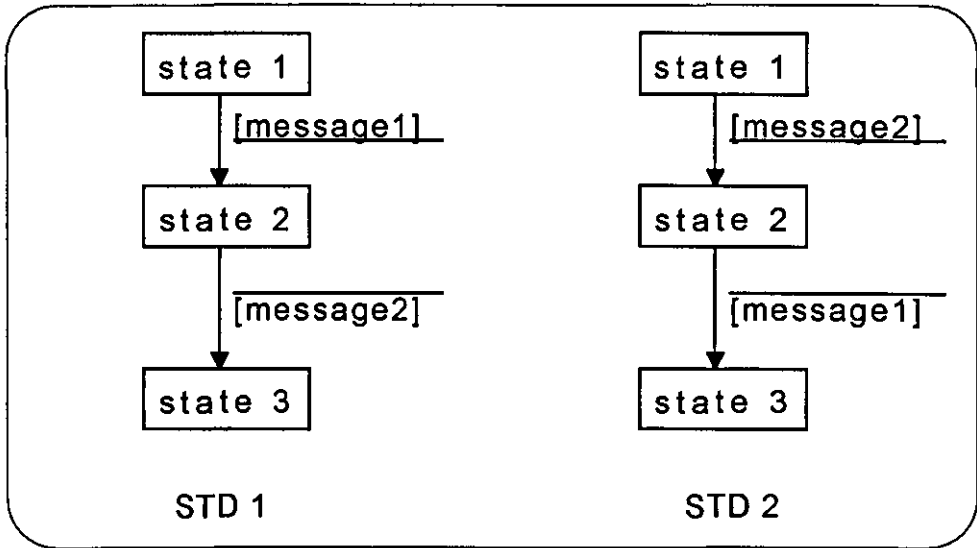
Having generated the Reachability Tree, the Analyzer creates a list file which it displays in a window. This contains summary information about deadlocks which were discovered, unreachable states and dead transitions.

Deadlocks

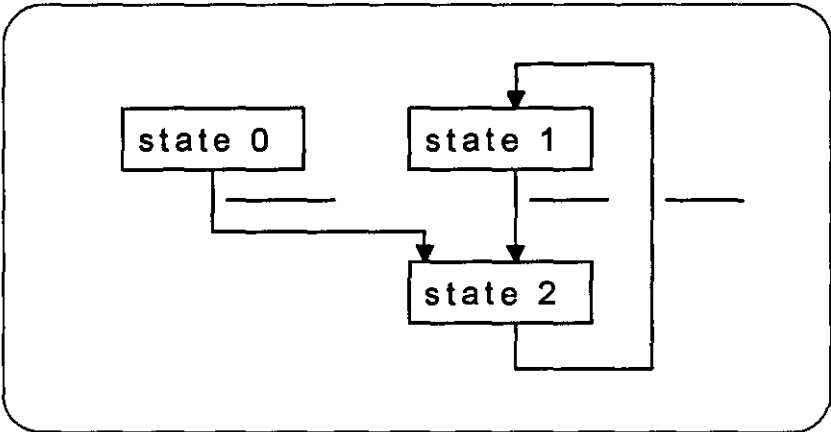
A *deadlock* is a combination of system states from which the application cannot proceed. Once the application enters that state combination, it "hangs". In the following example, the application deadlocks when STD "STD 1" reaches state "state 3" and STD "STD 2 reaches state "state 2" - there are no transitions which can fire:



In the next example, the application deadlocks with STD "STD 1" in state "state 1" and STD "STD 2" in state "state 1" because each STD is waiting for a message which the other can't send:

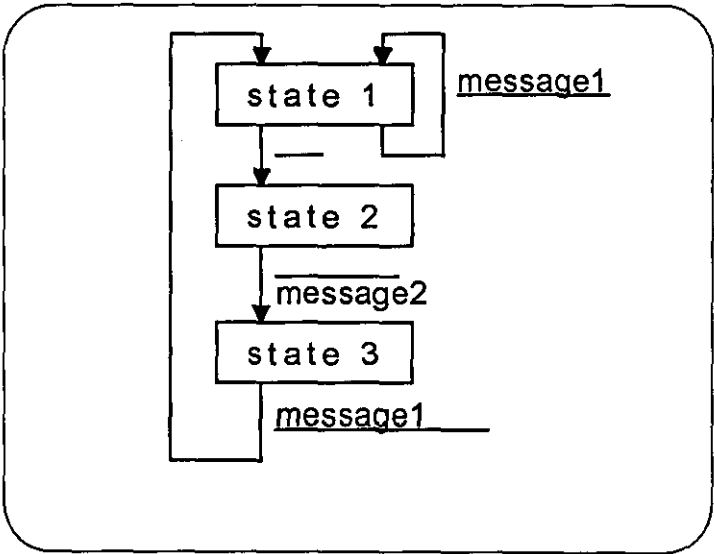


Unreachable States A state which an STD can never enter is referred to as an *unreachable state*. In the following example, assuming that the STD starts in state "state 1", state "state 0" is an unreachable state:



See also the Compiler User Guide, chapter 3, Basic Concepts, subsection Understanding The Compiler Output for a description of how synchronous messages can result in a state being unreachable.

Dead Transition A transition in an STD which the application never fires is referred to as a *dead transition*. In the following example, the message "message1" can be received in state "state 1" or state "state 3". If the sender of this message always waits until the message "message2" has been received before sending "message1", the transition starting in state "state 1" waiting for message "message1" is a dead transition:



4 Open An Application

When you first start the Synect Analyzer, very few of the menu items or buttons are enabled. This is because you must first open an application to analyze or open a Reachability Tree previously generated by the Analyzer.

When you compiled your application using the Compiler, it created a file with the name of your application but with the extension ".net". This file is used by the Analyzer.

Opening An Application



To open an application, choose the `File|Open Compiled Application` menu option. The standard file open dialog will then be started, allowing you to specify the name of the file from which the application is to be loaded. By default, the dialog will list files with extension ".net". To load a Reachability Tree, click on the `List Files of Type` drop-down listbox and click on entry `Reach Trees (*.tre)` to see the available Reachability Tree files.

This page left intentionally blank

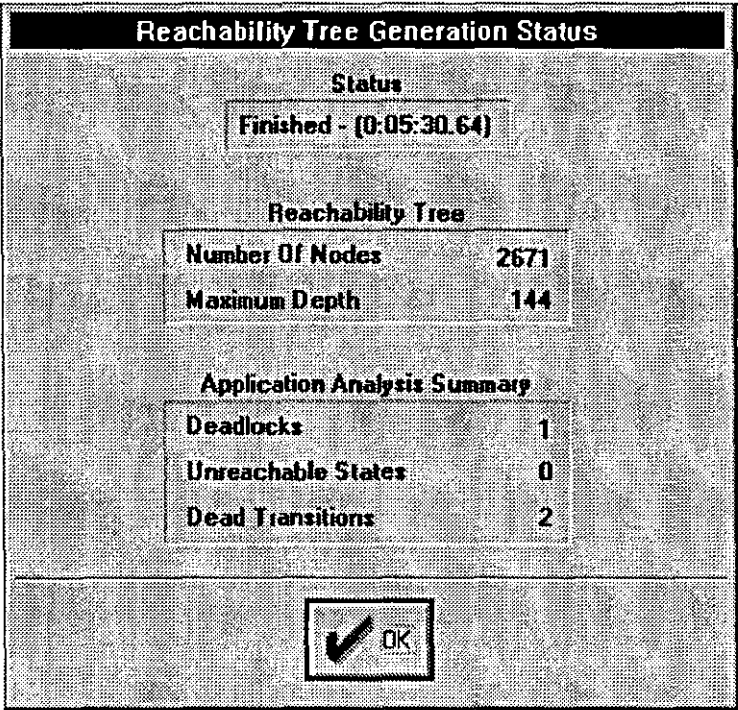
5 Generating The Reachability Tree

The Reachability Tree is a pre-requisite for the interactive analyses which the Analyzer supports. This chapter describes how to generate the Reachability Tree and the information you'll see as it is being generated.

Starting The Generation Of The Reachability Tree



When you've opened your application, choose the `Analyze|Generate Tree` menu option to begin generation of the Reachability Tree. The Reachability Tree Generation Status dialog will be displayed, showing the progress. When completed, this dialog will look like the following:



During generation of the Reachability Tree, the "OK" button will be a "Cancel" button. Clicking on the "Cancel" button will abort the generation. When the Reachability Tree has been generated, the Analyzer will save it to disk, look for unreachable states and dead transitions and write the number found in the dialog. The details are written in the list file which is displayed in a window. Clicking on the "OK" button closes the dialog.

Analyzer List File

Following generation of the Reachability Tree, a list file is created with the same name as your application but with extension ".als". This is displayed in a window in the Analyzer. The list file details the deadlocks, unreachable states and dead transitions which were summarised in the Reachability Tree Generation Status dialog.

Deadlocks

The first set of information in the list file shows the deadlocked system states. The number of deadlocks found is shown and then the state of each of the STDs at each of the deadlocks. The Reachability Tree node number information can be ignored. See the Compiler User Guide, chapter 5, Compiling The Application for information about the format of a reference to an STD's state.

Unreachable States

The next set of information shows any STD states which the application will never be in.

Dead Transitions

Finally, the list file shows any transitions in the mathematical model which the application never fires. For each of these references, the transition number is shown (and can be ignored) and the transition's start states and end states are shown.

6 Querying Deadlocks

Having generated or opened the Reachability Tree, you can query deadlocks which may have been identified. This chapter describes how to use the Deadlocks Dialog to query the deadlocks and save information to disk.

The Deadlocks Dialog



To start the Deadlocks Dialog, choose the `Analyze|Deadlocks` menu option. This dialog contains a listbox in which the deadlocked application state is displayed, buttons for navigating around the available deadlocked states, and buttons for saving information to file.

If your application contains no deadlocks, the message "No deadlocks found" will be displayed, the listbox will be empty and the only enabled buttons will be the "OK" and "Help" buttons.

Navigating

If your application contains 1 or more deadlocks, the message "Deadlock 1 of n" will be displayed where "n" is the number of different deadlocked application states found. The listbox will contain the application state at the first deadlock found. The buttons below the listbox are as follows:



Show the first deadlocked state.



Show the previous deadlocked state.



Show the next deadlocked state.



Show the final deadlocked state.

You can use these buttons to navigate around the available set of deadlocks.

Saving Information To File

The "Save All Found" button will save, to a file you specify, the application state at each of the deadlocks. It will then display this file in a window in the Analyzer. The default file extension is ".qrs" (query result summary).

The "Save This" button will save, to a file you specify, the path from the initial state to the current deadlock state. The changes of state from one step to the next are denoted by the state being prefixed with "=>" and postfixed with "<=". The file will be displayed in a window in the Analyzer. The path it takes may not be the only path to this deadlock. The default file extension is ".qrp" (query result path).

The "Save This As Event Log" button will save, to a file you specify, the path from the initial state to the current deadlock state as an event log file. This file

may then be loaded into the Simulator and, in conjunction with the STD Monitor, can be used to animate your specification to show you how the application reaches the deadlock from its initial state. The path it takes may not be the only path to this deadlock. The default file extension is ".sel" (Synect event log).

7 State Search Querying

Having generated or opened the Reachability Tree, you can query whether the application can reach a specified set of states and, if so, find out how. This chapter describes how to use the State Search Dialog to search for specified combinations of state and how to save the information to disk.

The State Search Dialog



To start the State Search Dialog, choose the `Analyze|State Search` menu option. This dialog can be thought of in 2 parts. The upper part of the dialog is concerned with expressing the query. The lower part of the dialog is concerned with the results from the search specified by the query.

Specifying The Query

At the top left of the dialog is a list box showing the STDs defined in your application. Double-click on one of these and the list box at the top right of the dialog shows the states in the chosen STD. You can then select a state and click on the "Add" button to add this STD/state combination to the query which is shown in the Query list box. Alternatively, you can double-click on the state to add it to the query. Refer to the Compiler User Guide, chapter 3, Basic Concepts for details of how asynchronous messages and variables are represented as STDs.

To remove an STD/state combination from the query, select the appropriate entry in the Query list box and then click on the "Delete" button.

Having specified the query, you can save it to file by clicking on the "Save" button. Correspondingly, you can load in a previously defined query by clicking on the "Load" button. The default file extension is ".qri" (query input). The file format is such that a section of a query result file can be used as the basis of a query input file.

Executing The Query

When the query contains the desired STD/states to be sought, click on the "Search" button at the bottom of the dialog to execute the query.

Viewing The Results

The query results are displayed in the lower part of the dialog.. This is very similar to the presentation of results in the Deadlocks Dialog. The results part of the dialog contains a listbox in which the sought application state is displayed, buttons for navigating around the matches found, and buttons for saving information to file.

If no application states were found satisfying the query, the message `No matches found` will be displayed, the listbox will be empty and the only enabled buttons

will be the "OK" and "Help" buttons and the "Edit" button in the query part of the dialog.

If 1 or more matches were found, the message `Match 1 of n` will be displayed where "n" is the number of matches found. The listbox will contain the application state at the first match found. The buttons below the listbox are as follows:



Show the first deadlocked state.



Show the previous deadlocked state.



Show the next deadlocked state.



Show the final deadlocked state.

You can use these buttons to navigate around the matches found.

**Saving
Information
To File**

The "Save All Found" button will save, to a file you specify, the application state at each match. It will then display this file in a window in the Analyzer. The default file extension is ".qrs" (query result summary).

The "Save This" button will save, to a file you specify, the path from the initial state to the state at the current match. The changes of state from one step to the next are denoted by the state being prefixed with "=>" and postfixed with "<=". The file will be displayed in a window in the Analyzer. The path it takes may not be the only path to this match. The default file extension is ".qrp" (query result path).

The "Save This As Event Log" button will save, to a file you specify, the path from the initial state to the state at the current match as an event log file. This file may then be loaded into the Simulator and, in conjunction with the STD Monitor, can be used to animate your specification to show you how the application reaches the state at the current match from its initial state. The path it takes may not be the only path to this match. The default file extension is ".sel" (Synect event log).

**Returning To
Define Query
Mode**

To return to the mode where the query can be edited, click on the Edit button in the query part of the dialog.






Appendix A Menus

File	
Open Compiled Application	Open a compiled application
Exit	Finish running the Application Analyzer
Analyze	
Generate Tree	Generate the Reachability Tree
Deadlocks	Show deadlocked application states
State Search	Search for specified application state
Window	
Cascade	Cascade open windows
Tile	Tile open windows
Arrange Icons	Arrange iconic windows along bottom
Close Windows	Close all open windows
Help	
Contents	Help table of contents
Using help	Help on using online Help
About	About Synect Application Analyzer

This page left intentionally blank

Appendix B Toolbar Buttons

The Application Analyzer provides an Operation Toolbar displayed horizontally across the top of the screen. It contains buttons which can be used as shortcuts instead of pulling down the corresponding menu and selecting the relevant item. The buttons will be greyed-out if the corresponding function is unavailable at that time.

	<u>Menu Equivalent</u>	<u>Usage</u>
	File Open Compiled Application	Load a compiled application from a file on disk
	Analyze Generate Tree	Generate the Reachability Tree
	Analyze Deadlocks	Show deadlocked application states
	Analyze State Search	Search for specified application state
	Help Contents	Display help information

Synect

STD Monitor User Guide

Version 1.3

**Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough
Cleveland, TS14 8JY
England**

**Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk
WWW: <http://www.hopkinsn.demon.co.uk>**

© Copyright 1994, 1995, 1996, 1997 Hopkinson Computing Limited. All rights reserved.

Synect is a registered trademark of Hopkinson Computing Limited

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Information in this User Guide is subject to change without notice and does not represent a commitment on the part of Hopkinson Computing Limited.

The software described in this User Guide is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. No part of this User Guide may be reproduced or transmitted in any form or by any means, electronic or otherwise, including photocopying and recording, for any purpose, without the express written permission of Hopkinson Computing Limited.

Document History

10 June 1996

Re-issue reflecting STD Monitor V1.1 - new cover sheet, contents and chapters 1, 3 and 4.

28 October 1996

Re-issue reflecting STD Monitor V1.2 - changes to cover sheet and chapter 3.

6 October 1997

Re-issue reflecting STD Monitor V1.3 - changes to cover sheet and chapter 3.

Contents

Chapter 1	Introducing Synect	1
	User Interface	1
	The Method	1
	Synect Documentation	1
	The Tools	2
	Document Conventions	2
Chapter 2	Getting Started	3
	System Requirements	3
	Installation	3
	Starting the STD Monitor	4
	On-Line Help	5
Chapter 3	Basic Concepts	7
	Colour Conventions	7
	Object Hierarchy Window	7
	STD Window	7
	Synect Server Data Source	8
	Connecting To The Synect Simulator	8
	Connecting To The Live Control System	8
Chapter 4	Open An Application	9
	Opening An Application	9
Chapter 5	Displaying A Diagram In A Window	11
	Displaying An STD	11
	Tracking An STD's Current State	11
	Zooming	11
	Resizing A Window	11
Appendix A	Menus	13
Appendix B	ToolBar Buttons	15

This page left intentionally blank

1 Introducing Synect

Synect is a set of software tools which helps the designer of a control system to produce a specification which is clear, precise and free of errors. Synect combines the ease of use of a graphical user interface, with a widely used diagrammatic notation and the power of mathematical modelling.

User Interface

The Synect™ tools run on the Microsoft® Windows™ operating system. As such, you need to know how to use Windows before you use Synect. In particular, you will need to know how to use the mouse to click, double-click and drag. You will also need to know how to interact with menus, dialogs and how to move, resize and close windows. For information, refer to the Microsoft Windows User's Guide.

The Method

Synect uses an object-based method to enable you to model the system you want to control. It is useful to have some knowledge of object-based/object-oriented analysis and design techniques before attempting to use Synect in earnest.

Synect Documentation

Each of the Synect tools has an associated User Guide. This User Guide explains how to use each of the functions available in the STD Monitor. It also explains why you might want to use the function.

Each application has context-sensitive on-line help. The STD Monitor on-line help contains a "How Do I?" section, including a "How Do I Get Started?" sub-section for first-time users.

A Tutorial is also provided which offers a worked example and shows how each of the tools is used with the example application.

The Tools

The tools which make up the Synect toolset are:

Application Editor	graphical means of defining the application.
Compiler	check the specification for consistency and possible warnings and generate a mathematical model of the application.
Analyzer	ability to check for design errors such as deadlock (where the system "hangs") and unwanted state combinations.
Simulator	provides the ability to interactively "drive" the application or replay past behaviour of the live control system.
STD Monitor	animates the specification (used in conjunction with the Simulator or the live control system).
C Code Generator	generate ANSI-standard C code to implement the application.
Neuron C Generator	generate Neuron C to run on one or more nodes on an Echelon LonWorks network to implement a distributed control solution.
Ladder Logic Generator	generate relay ladder logic to run on a programmable controller.

Document Conventions

The User Guide adopts the following conventions:

<code>application name</code>	text that you type or that you see on the screen.
<code>KEY NAME</code>	keyboard keys, such as ENTER, CTRL or DEL.
<code>Menu Choice</code>	a menu option, such as File Exit denoting choose the Exit command from the File menu.
<i>description</i>	description of a term with a specific meaning.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the STD Monitor.
- how to start the STD Monitor.
- the STD Monitor window.

System Requirements

The Synect STD Monitor requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

Other Synect tools also requires that you have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

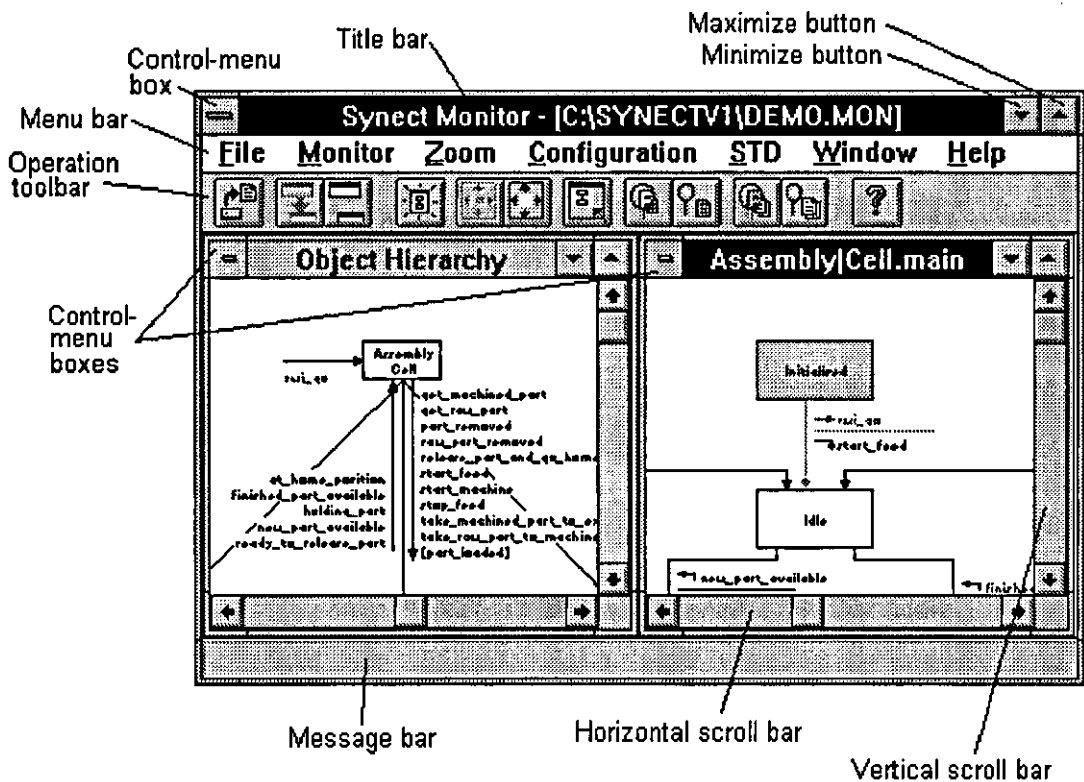
- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press ENTER.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

- 8 Read the installed file `readme.txt` for details of any changes to the product or documentation since the documentation was printed.

Starting the STD Monitor

Ensure that the Program Manager window is on display. If the Synect window isn't visible, use the Window menu to open it. Double-click on the STD Monitor icon or use the keyboard arrow key to select the icon and then press ENTER.

When using the STD Monitor, the window will typically look like the following:



Title bar
The outer window is the Synect STD Monitor window. The title bar therefore shows the product title `Synect Monitor` and the name of the application which has been loaded (if any).

The inner windows show either the Object Hierarchy or a state transition diagram. The title bar of an inner window will therefore be either `Object Hierarchy` or the name of the state transition diagram.

Control-menu boxes

Allows you to restore, move, size, minimize, maximize or close (except for the Object Hierarchy window) the window. Also allows you to make another window the active window or open the control panel (Synect window only).

Menu bar

Lists the available menus.

Minimize box

Allows you to shrink the Synect window to an icon at the bottom of the screen.

Maximize box

Allows you to enlarge the Synect window to fill the entire screen.

Operation toolbar

Contains the momentary buttons which can be used as menu shortcuts.

Message bar

When the cursor is moved over an Operation Toolbar button or when a menu option is highlighted, the message bar will show a brief description of the function of the button/menu option.

Horizontal and vertical scroll bars

Allow you to pan around the diagram to change the portion which is displayed in the window.

On-Line Help



The on-line help is context-sensitive. So if you click on a help button in a dialog, you will automatically be shown the help information associated with that dialog. If no dialog is being displayed, you can choose [Help/Contents](#) to take you to the help contents page. The help information is shown in a separate window.

This page left intentionally blank

3 Basic Concepts

The Synect Application Editor enables you to specify a model of your application. See the Application Editor User Guide, chapter 3, Basic Concepts for more information relating to defining the model. The STD Monitor can animate your specification, obtaining its data from either the Simulator or the live control system. It can therefore be used for the following:

- as an aid to testing the specified logic.
- as a presentation aid during customer reviews.
- as a diagnostic tool to obtain an inside view of the state of a live control system.

Colour Conventions

The STD Monitor displays the Object Hierarchy in a window and, when obtaining its data from the Simulator, colour codes the real world inputs to show whether they are being tested and, if so, whether or not they are enabled. The STD Monitor also displays STDs which you nominate in windows and colour codes the current state, the transitions and, when obtaining its data from the Simulator, the real world inputs.

Object Hierarchy Window

When the STD Monitor is obtaining its data from the Simulator, a real world input will be shown in red in the object hierarchy window if it has been tested and is not enabled. It will be shown in green if it has been tested and is enabled. Otherwise it is shown in the default colour (usually blue). When the Simulator advises that a transition has been fired, all real world inputs in the Object Hierarchy window will be redrawn in their default colour. The process then repeats - as a real world input is tested, its status is shown by the appropriate colour in the Object Hierarchy window.

STD Window

The current STD state is shown in grey.

The last transition fired is drawn in yellow. Transitions which are enabled with respect to the structure of the application (in other words those which will be able to fire provided that all of their real world inputs are enabled) are drawn in green. Transitions which satisfy both of these criteria are drawn in magenta.

It follows that if a transition is enabled with respect to the structure of the application, any conditions which refer to real world inputs must be being tested to see if the transition can be fired. When data is being obtained from the Simulator, these conditions will therefore be drawn in red if they are not enabled or green if they are enabled.

Synect Server Data Source

The STD Monitor obtains its dynamic information from another Windows application, the Synect Server, via Windows Dynamic Data Exchange (DDE). During development, the server will probably be the Synect Simulator. But the server may be a Windows application communicating with the target control system so that the STD Monitor displays the current state of the control system.

Connecting To The Synect Simulator

If no mapping file has been specified in the command line, the STD Monitor will automatically attempt to connect to a Synect Server after opening an application. Menu options are also provided for manual connection and disconnection. The STD Monitor must be connected to a Synect Server for the diagrams to be animated.

Connecting To The Live Control System

To instruct the STD Monitor to obtain its data from a Synect Server which is connected to the live control system, it must be told the DDE service name, topic name and, for each STD to be animated, the name of the STD followed by the DDE item name. You specify this information in a mapping file and let the STD Monitor know the filename when it starts by passing it as a parameter.

For example, assume that the Synect Neuron C Generator has been used to generate the code for STD Switch.main on node "sw" and STD Machine.main on node "mach" on an Echelon LonWorks network. Create a file (or use the file written by the Neuron C Generator), such as "test.llm" with the following contents (do not include spaces anywhere and be aware that the STD Monitor will perform a case-sensitive read of the file):

```
DDE_SERVER=LMSRVR1
DDE_TOPIC=Netvar
STD=Switch.main,sw.nvo_Switch_main
STD=Machine.main,mach.nvo_Machine_main
```

Click on the STD Monitor icon in Program Manager to select it and then choose the File|Copy menu option to create a copy (putting the copy in the same program group as the original). The copy will now be selected. Choose the File|Properties menu option to start the Program Item Properties dialog. Change its name to, for example, "STD Live Monitor" and at the end of the command line add a space and then the name of the file. In this example the command line should then look something like:

```
c:\synectv1\syn_mon.exe test.llm
```



Start this new program item in the usual way. Open the application by choosing the File|Open Application menu option. Choosing the Monitor|Connect menu option will cause the STD Monitor to read the contents of file "test.llm" and attempt to connect to the DDE server. The STDs can then be displayed to show the current state of each STD in the live control system.

To display the state of an STD which is running in a PLC, using code generated by the Synect ladder logic generator, the STD Monitor needs to know the integer value associated with each state. This is achieved via entries in the mapping file of the following form:

```
STATE=Open,10
```

4 Open An Application

When you first start the Synect STD Monitor, very few of the menu items or buttons are enabled. This is because you must first open an application.

When you saved your application from the Application Editor, it created a file with the name of your application but with the extension ".mon". This file is used by the STD Monitor.

Opening An Application



To open an application, choose the `File|Open Application` menu option. The standard file open dialog will then be started, allowing you to specify the name of the file from which the application is to be loaded.



Having loaded the application, the Object Hierarchy will be displayed in a window and most of the menu options and buttons will be enabled. If the STD Monitor is not being used in conjunction with the live control system (i.e. there was no filename parameter when the application was started), it will also attempt to connect to a Synect Server. If this is unsuccessful, for example because the Simulator is not running, you will need to manually initiate this connection when the Synect Server becomes available by choosing `Monitor|Connect`. If the Synect Server application terminates, you'll need to perform this operation again. Once the STD Monitor has connected to the Synect Server, the `Monitor|Disconnect` menu option is enabled to allow you to prevent further animation of the diagrams.

This page left intentionally blank

5 Displaying A Diagram In A Window

This chapter describes the choices which are available for displaying the Object Hierarchy and STDs. It assumes that you've already opened an application.

Displaying An STD



To create a new window (or windows) displaying an STD (STDs), choose the STD|Display menu option. The Select STD To Display dialog will be started, enabling you to nominate which STDs you want to display (multiple selections are allowed). Click on "OK" to close the dialog and the STD Monitor will create a new window for each STD selected.

Tracking An STD's Current State



By default, the STD Monitor will automatically attempt to move an STD within a window to ensure that the current state is visible. To disable this facility, choose STD|Current State Not Necessarily Visible. To re-enable, choose STD|Current State Always Visible. This pair of options applies to the currently active STD window.

Zooming



To enlarge the diagram in the currently active window, use the Zoom|In menu option. To reduce the size of the diagram, use the Zoom|Out menu option. Choose Zoom|Reset to reset to the original magnification.



Similarly, to enlarge all diagrams, use the Zoom|In All menu option. To reduce the size of all diagrams, use the Zoom|Out All menu option. Choose Zoom|Reset All to reset all windows back to their original magnification.

Choose Configuration|Zoom Factor to change the amount by which the zooming functionality magnifies or reduces the diagram.

Resizing A Window



In addition to the usual functionality for resizing and moving windows, the STD Monitor also offers the option of resizing a window to fit the diagram which it displays. Choose STD|Resize Window To Size Of Drawing to resize the window to be the smallest possible which still enables the entire diagram to be drawn in the window without the need for scroll bars.

This page left intentionally blank













Appendix A Menus

File	
Open Application	Open an existing application
Exit	Finish running Synect STD Monitor
Monitor	
Connect	Establish communication link with a Synect Data Server
Disconnect	Terminate the communication link with the Synect Data Server
Zoom	
In	Enlarge the diagram
Out	Reduce the size of the diagram
Reset	Reset the size of the diagram to its original size
In All	Enlarge all diagrams
Out All	Reduce the size of all diagrams
Reset All	Reset the size of all diagrams to their original size
Configuration	
Zoom Factor	Amount by which to enlarge/reduce the diagram when using the Zoom menu options.
STD	
Display	Create a new window to display an STD
Current State Always Visible	Pan STD in window to make current state always visible
Current State Not Necessarily Visible	Don't pan STD in window in order to make current state always visible
Resize Window To Size Of Drawing	Resize window to enclose drawing
Window	
Cascade	Cascade open windows
Tile	Tile open windows
Arrange Icons	Arrange iconic windows along bottom
Close All STD Windows	Close all STD windows
Help	
Contents	Help table of contents
Using help	Help on using online Help
About	About Synect STD Monitor

This page left intentionally blank

Appendix B Toolbar Buttons

The STD Monitor provides an Operation Toolbar displayed horizontally across the top of the screen. It contains buttons which can be used as shortcuts instead of pulling down the corresponding menu and selecting the relevant item. The buttons will be greyed-out if the corresponding function is unavailable at that time.

	<u>Menu Equivalent</u>	<u>Usage</u>
	File Open Application	Open an existing application
	Monitor Connect	Establish communication link with a Synect Data Server
	Monitor Disconnect	Terminate the communication link with the Synect Data Server
	STD Display	Create a new window to display an STD
	STD Current State Always Visible	Pan STD in window to make current state always visible
	STD Current State Not Necessarily Visible	Don't pan STD in window in order to make current state always visible
	STD Resize Window To Size Of Drawing	Resize window to enclose drawing
	Zoom In	Enlarge the diagram in the active window
	Zoom Out	Reduce the size of the diagram in the active window
	Zoom In All	Enlarge all diagrams
	Zoom Out All	Reduce the size of all diagrams
	Help Contents	Display help information

Synect

Simulator User Guide

Version 1.6

**Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough
Cleveland, TS14 8JY
England**

**Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk**

© Copyright 1994, 1995, 1996 Hopkinson Computing Limited. All rights reserved.

Synect is a registered trademark of Hopkinson Computing Limited
Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Information in this User Guide is subject to change without notice and does not represent a commitment on the part of Hopkinson Computing Limited.

The software described in this User Guide is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. No part of this User Guide may be reproduced or transmitted in any form or by any means, electronic or otherwise, including photocopying and recording, for any purpose, without the express written permission of Hopkinson Computing Limited.

Document History

17 December 1995

Re-issue reflecting Simulator V1.1 - new cover sheet and Appendix C, DDE Services.

18 March 1996

Re-issue reflecting Simulator V1.2 - new cover sheet, contents, and changes to most sections.

10 June 1996

Re-issue reflecting Simulator V1.5 - new cover sheet and chapter 1.

28 October 1996

Re-issue reflecting Simulator V1.6 - new cover sheet and chapters 5 & 7.

Contents

Chapter 1	Introducing Synect	1
	User Interface	1
	The Method	1
	Synect Documentation	1
	The Tools	2
	Document Conventions	2
Chapter 2	Getting Started	3
	System Requirements	3
	Installation	3
	Starting the Simulator	4
	On-Line Help	6
Chapter 3	Basic Concepts	7
	How It Works	7
	Event Logging	7
	Synect DDE Server Data Source	8
	Synect DDE Client	8
Chapter 4	Open An Application	9
	Opening An Application	9
Chapter 5	Interactively Driving The Application	11
	Using The Simulator Control Dialog	11
	Changing The Simulator Mode	12
	Changing The Selected Event Log Position	12
	Using The Real World Input Status Dialog	13
	Using The Real World Outputs Dialog	14
	Behaviour when adding a new entry	14
	Using The Simulator As A DDE Client	15
	Create the mapping file	15
	Load the mapping file	16
	Connect to the DDE server	16
	Drive the DDE server	16
Chapter 6	Create, Save And Replay An Event Log	17
	Creating A New Empty Event Log	17
	Saving An Event Log	17
	Replaying An Event Log	17

Chapter 7	Configuration	19
	Configuring Timing Attributes	19
	Playback Mode	19
	Record Mode	19
	Configuring Event Log Attributes	20
	Configuring The Handling Of RWI And RWO Dialogs	20
Appendix A	Menus	23
Appendix B	ToolBar Buttons	25
Appendix C	DDE Services	27

1 Introducing Synect

Synect is a set of software tools which helps the designer of a control system to produce a specification which is clear, precise and free of errors. Synect combines the ease of use of a graphical user interface, with a widely used diagrammatic notation and the power of mathematical modelling.

User Interface

The Synect™ tools run on the Microsoft® Windows™ operating system. As such, you need to know how to use Windows before you use Synect. In particular, you will need to know how to use the mouse to click, double-click and drag. You will also need to know how to interact with menus, dialogs and how to move, resize and close windows. For information, refer to the Microsoft Windows User's Guide.

The Method

Synect uses an object-based method to enable you to model the system you want to control. It is useful to have some knowledge of object-based/object-oriented analysis and design techniques before attempting to use Synect in earnest.

Synect Documentation

Each of the Synect tools has an associated User Guide. This User Guide explains how to use each of the functions available in the STD Monitor. It also explains why you might want to use the function.

Each application has context-sensitive on-line help. The Simulator on-line help contains a "How Do I?" section, including a "How Do I Get Started?" sub-section for first-time users.

A Tutorial is also provided which offers a worked example and shows how each of the tools is used with the example application.

The Tools

The tools which make up the Synect toolset are:

Application Editor	graphical means of defining the application.
Compiler	check the specification for consistency and possible warnings and generate a mathematical model of the application.
Analyzer	ability to check for design errors such as deadlock (where the system "hangs") and unwanted state combinations.
Simulator	provides the ability to interactively "drive" the application or replay past behaviour of the live control system.
STD Monitor	animates the specification (used in conjunction with the Simulator or the live control system).
C Code Generator	generate ANSI-standard C code to implement the application.
Neuron C Generator	generate Neuron C to run on one or more nodes on an Echelon LonWorks network to implement a distributed control solution.
Ladder Logic Generator	generate relay ladder logic to run on a programmable controller.

Document Conventions

The User Guide adopts the following conventions:

<code>application name</code>	text that you type or that you see on the screen.
<code>KEY NAME</code>	keyboard keys, such as ENTER, CTRL or DEL.
<code>Menu Choice</code>	a menu option, such as File Exit denoting choose the Exit command from the File menu.
<i>description</i>	description of a term with a specific meaning.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the Simulator.
- how to start the Simulator.
- the Simulator window.

System Requirements

The Synect Simulator requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

Other Synect tools also requires that you have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

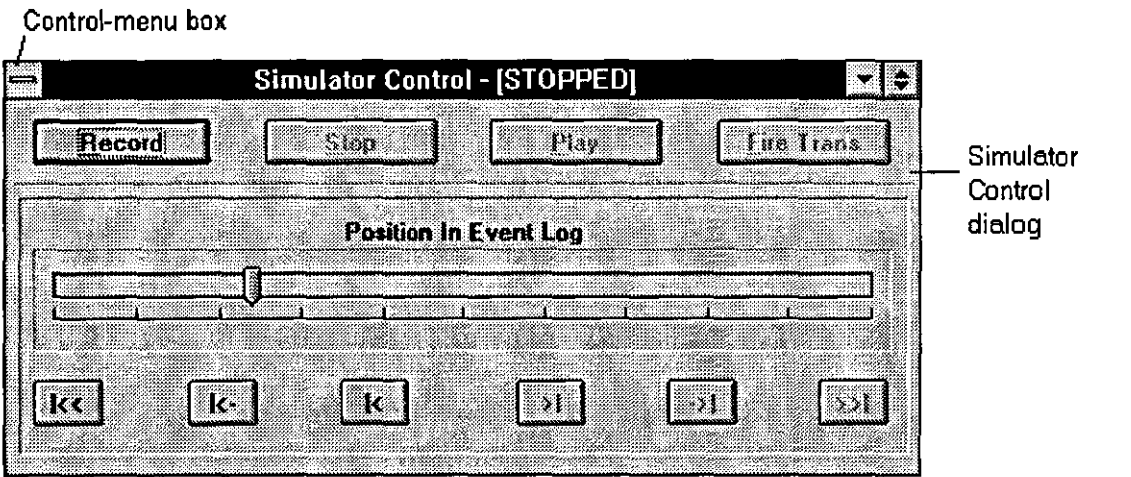
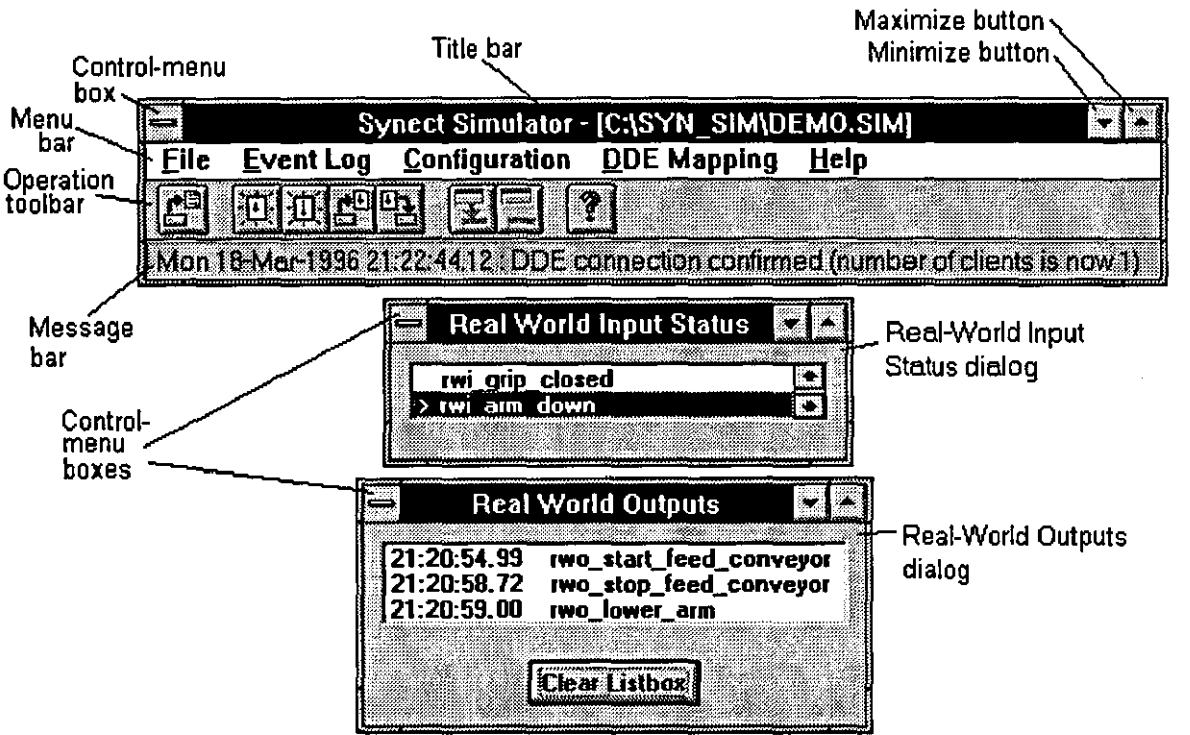
- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press ENTER.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

- 8 Read the installed file `readme.txt` for details of any changes to the product or documentation since the documentation was printed.

Starting the Simulator

Ensure that the Program Manager window is on display. If the Synect window isn't visible, use the Window menu to open it. Double-click on the Simulator icon or use the keyboard arrow key to select the icon and then press ENTER.

When using the Simulator , the window will typically look like the following:



Title bar

The window at the top of the diagram is the Synect Simulator window. The title bar therefore shows the product title `Synect Simulator` and the name of the application which has been loaded (if any).

Control-menu boxes

Allows you to restore, move, size, minimize, maximize or close (except for the Real World Input Status dialog or Simulator Control dialog) the window. Also allows you to make another window the active window or open the control panel. The control-menu box belonging to each of the dialogs also allows you to determine whether the dialog is to be a topmost window.

Menu bar

Lists the available menus.

Minimize box

Allows you to shrink the window to an icon at the bottom of the screen.

Maximize box

Allows you to enlarge the window to fill the entire screen.

Operation toolbar

Contains the momentary buttons which can be used as menu shortcuts.

Message bar

When the cursor is moved over an Operation Toolbar button or when a menu option is highlighted, the message bar will show a brief description of the function of the button/menu option. Also gives information regarding connection and disconnection of DDE clients.

Real-World Input Status dialog

Allows you to simulate the status of real-world inputs by selecting and de-selecting corresponding entries in the listbox.

Real-World Outputs dialog

Shows you when real-world outputs are invoked.

Simulator Control dialog

The Simulator's control panel. In its normal form, this dialog only gives you access to the Record, Stop, Play and Fire Trans buttons. In its maximised form, you also have access to the event log controls, these being the slider and the navigation buttons.

On-Line Help



The on-line help is context-sensitive. So if you click on a help button in a dialog, you will automatically be shown the help information associated with that dialog. If no dialog is being displayed, you can choose [Help|Contents](#) to take you to the help contents page. The help information is shown in a separate window.

3 Basic Concepts

The Synect Application Editor enables you to specify a model of your application. See the Application Editor User Guide, chapter 3, Basic Concepts for more information relating to defining the model. The Compiler derives a mathematical model from your specification. The Simulator executes the model, enabling you to interactively drive your application or replay past behaviour. The Simulator sends information to other Windows packages for display in the required format. It can also receive information from other Windows packages so that it can be driven remotely. These facilities enable the Simulator to drive plant sensors and actuators via a third party interface (such as the Echelon LonWorks DDE Server).

How It Works

The Simulator executes the mathematical model derived by the Compiler from your specification. It determines which transitions might be able to fire and tests the status of their real-world inputs. From this set of transitions, it constructs a list of the transitions which have all their real world input conditions enabled and picks one of them to be fired. It then fires the transition thereby changing the states of the appropriate STDs. The cycle then repeats.

The real-world input status is determined from the Real World Input Status dialog. Click on an entry to toggle whether the corresponding real-world input is to be interpreted as enabled.

The Simulator Control dialog acts as the Simulator's control panel. It enables you to control the mode of operation of the Simulator and to navigate around the event log.

Event Logging

Every time a transition is fired or a real world input is tested and is found to have changed state, a time-stamped entry is written to an event log. The event log is held in memory but may be saved to file on disk or reloaded from disk for replay. The event log is linear by default which means that when it is full, the Simulator must stop. Alternatively, the event log may be configured to be circular so that when full, the oldest record is overwritten.

The Simulator can also be used to replay an event log which has been generated by the Synect Analyzer or by a C program created by the Synect C Code Generator.

Synect DDE Server Data Source

The Simulator acts as a Windows Dynamic Data Exchange (DDE) server to provide data for compatible Windows products, such as the STD Monitor and Wonderware's InTouch SCADA product. The other products behave as DDE clients, establishing advise loops with the Simulator. Whenever an entry is written to the event log, or you change the currently selected position within the event log, the Synect clients are notified of the new information.

The Simulator also allows DDE clients to change the simulated status of real-world inputs by allowing them to interact with the Real World Input Status dialog. This provides the ability for your application to be driven via a third party user interface, whilst still offering the benefits of the Simulator's event logging functionality.

Appendix C, DDE Services, describes the information which the Simulator makes available.

Synect DDE Client

The Simulator can also act as a DDE client, notifying the server when real-world outputs are invoked and enabling the server to control the simulated status of real-world inputs in the Real World Input Status dialog. For example, the Simulator could drive the Echelon LonWorks DDE Server so you could run the control logic whilst driving the plant sensors and actuators.

4 Open An Application

When you first start the Synect Simulator, very few of the menu items or buttons are enabled. This is because you must first open an application.

When you compiled your application using the Compiler, it created a file with the name of your application but with the extension ".sim". This file is used by the Simulator.

Opening An Application



To open an application, choose the File|Open Application menu option. The standard file open dialog will then be started, allowing you to specify the name of the file from which the application is to be loaded.

Having loaded the application, the Simulator Control dialog, Real World Input Status dialog and Real World Outputs dialog will be displayed. An empty event log will be created, capable of holding 50 entries and linear (i.e. the Simulator will stop recording when it contains 50 entries rather than overwriting the oldest entry).

The Simulator window will have been created as a small window at the top right of your screen before being maximised. If the application is loaded with the frame window maximised, the dialogs created will be positioned around the screen such that they do not overlap. The frame window can then be restored to its normal (un-maximised) form to effectively move it out of the way to the top right of the screen.

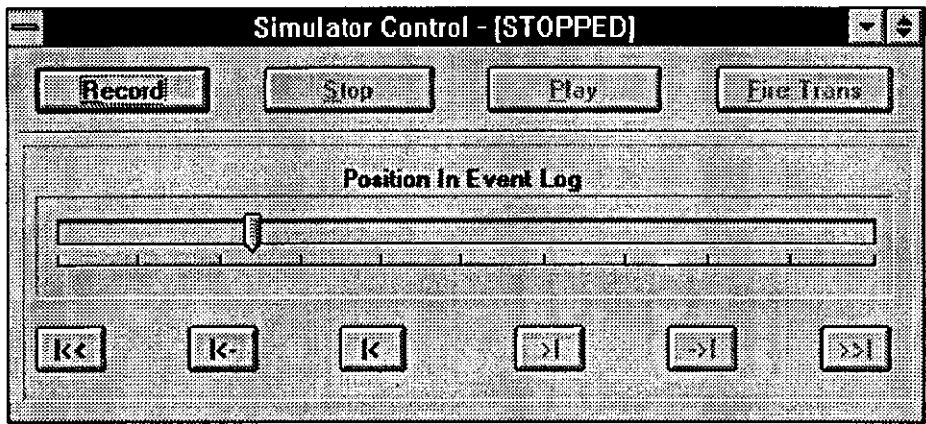
This page left intentionally blank

5 Interactively Driving The Application

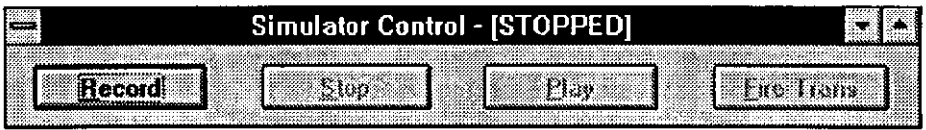
Having loaded your application, you can now interactively drive the application, simulating the behaviour of the system your application is to control and monitoring how your application reacts. You should load the application into the STD Monitor (see the STD Monitor's User Guide for details) or another Synect-compatible product so that you can observe the changing states of the STDs. Using the Simulator as a DDE server or DDE client, the status of real-world inputs can be determined from plant sensors and the real-world outputs can be used to control plant actuators.

Using The Simulator Control Dialog

In its maximised form, the Simulator Control dialog looks like the following:



In its normal (un-maximised) form, the Simulator Control dialog looks like the following:



By default, this dialog is a topmost window. This is so that it will be drawn on top of the STD Monitor windows. To make it a non-topmost window, use the dialog's "Always On Top" System Menu option to toggle this attribute.

This dialog can be thought of in 2 parts. The upper part contains buttons to control the mode of operation of the Simulator. The lower part is used to observe and change the current position in the event log.

Changing The Simulator Mode

The buttons and their uses are as follows:

Record

Starts recording, writing records into the event log from the current position. If this would lead to unsaved event log contents being overwritten, a warning message will be displayed before recording begins. The Simulator mode changes to RECORDING which is reflected in the dialog's title. The Stop button will be enabled, all others being disabled. Most of the menu options and toolbar buttons will be disabled.

Stop

Stops recording or playback. The Simulator mode changes to STOPPED which is reflected in the dialog's title. Enables the Record button if the event log is not full or is circular. Enables the Play button if not at the end of the event log and event log records exist after the current position.

Play

Starts playback of the event log from the current position. The Simulator mode changes to PLAYING which is reflected in the dialog's title. The Stop button will be enabled, all others being disabled. Most of the menu options and toolbar buttons will be disabled.

Fire Trans

If the Simulator mode is RECORDING, and the Timing Attributes dialog has been used to set the Record Mode to Pause Before Firing Transition, this button will be enabled when the Simulator is ready to fire a transition. This provides the facility to single-step whilst recording. Clicking this button will cause the Simulator to fire the transition and it will pause when it is ready to fire the next transition.

Changing The Selected Event Log

The relative position of the slider thumbnail denotes the current record in the event log. When the Simulator mode is RECORDING or PLAYING, the slider thumbnail is moved accordingly. When the Simulator mode is STOPPED, the slider button may be dragged to change the current event log position. Alternatively, the following buttons may be used:



Go to the beginning of the event log



Select the record just before the most recent record relating to the firing of a transition. Use this to step over the records which contain individual tests of real world input state.



Select the previous event log record



Select the next event log record



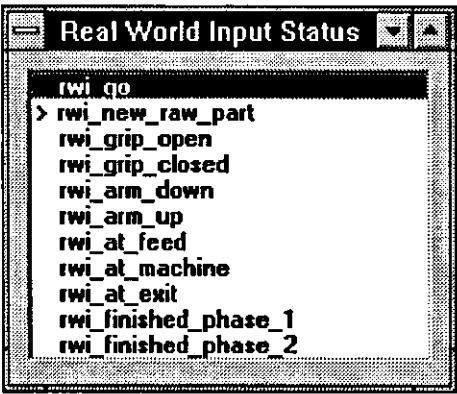
Select the record just before the next record relating to the firing of a transition. Use this to step over the records which contain individual tests of real world input state.



Go to the end of the event log

Using The Real World Input Status Dialog

The Real World Input Status dialog looks like the following:



By default, the dialog is a topmost window. This is so that it will be drawn on top of the STD Monitor windows. To make it a non-topmost window, use the dialog's "Always On Top" System Menu option to toggle this attribute.

The dialog contains a listbox in which each real world input is listed. To simulate the real world input being enabled, click on the corresponding item in the list box so that it is highlighted. To simulate it being disabled again, click on the entry again so that it is not highlighted. By default, the Simulator will reset the status of the real-world input (i.e. unhighlight it) after firing a transition which references it. Also by default, the Simulator will not automatically reset the real-world input if it is being controlled by another application via a DDE link. It assumes that you are manually controlling each real-world input until a DDE message is received with the real-world input's status. You can change this behaviour by means of the Real World Inputs And Real World Outputs dialog.

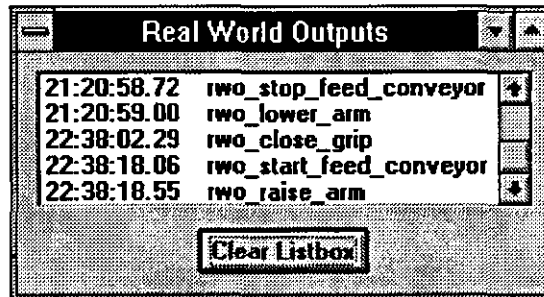
The real world input name is prefixed with ">" when it is being tested. This enables you to quickly identify which real world inputs the Simulator is waiting for before being able to fire a transition.

The Simulator only uses the information from this dialog when it is recording.

This dialog may be driven by a DDE client (see Appendix C, DDE Services, for details) or by a DDE server.

Using The Real World Outputs Dialog

The Real-World Outputs dialog looks like the following:



By default, the dialog is a topmost window. This is so that it will be drawn on top of the STD Monitor windows. To make it a non-topmost window, use the dialog's "Always On Top" System Menu option to toggle this attribute.

This dialog contains a list box which is initially empty. As a real-world output is invoked, a time-stamped entry will be written into the listbox showing the real-world output's name. To erase the contents of the listbox, click on the Clear Listbox button. The Real World Inputs And Real World Outputs dialog can be used to change the maximum number of entries in the listbox. When this number has been reached, the oldest entry will be removed to make room for the new entry. The default value is 100 and the allowable range of values is 5 to 1000.

Behaviour when adding a new entry

There are two ways of using the information in the real world outputs dialog's listbox. The first is to see new entries as they are being added to observe which real world outputs are being invoked. In this case, the listbox should be automatically scrolled so that the new entry is visible.

Alternatively, you might want to examine the history by scrolling back through the listbox. This suggests that the listbox should not be automatically scrolled otherwise the portion of entries being examined would be replaced with the most recent portion.

To compromise, if the most recent entry is visible in the listbox when a new entry is to be added, the listbox will be scrolled to make the new entry visible. If the new entry is not visible (i.e. you have scrolled it off the bottom of the listbox), the range of entries being displayed in the listbox will remain on display after the new entry is added. This has no effect on the removal of the oldest entry when a new entry is to be added and the listbox already contains the

maximum number of entries. In this case, the portion of the listbox entries you are examining will, in time, be removed from display as new entries are added.

Using The Simulator As A DDE Client

The Simulator can talk to a DDE Server so that the server can enable and disable real world inputs in the Real World Input Status dialog and can also be notified when real-world outputs are invoked.

Create the mapping file

Use a text editor (such as Notepad), to create a mapping file, such as "test.map" with the following contents:

```
! Mapping file for Synect Simulator
! Applicable for the demo application supplied with the product

STOP=rwo_open_grip
STOP=rwo_raise_arm

DDE_SERVER=syn_serv
DDE_TOPIC=syn_topic

DDE_ITEM=gripper_item
RWI=rwi_grip_closed,NOT=open
RWI=rwi_grip_open,open
RWO=rwo_open_grip,open_gripper
RWO=rwo_close_grip,close_gripper

DDE_ITEM=arm_elevation_item
RWI=rwi_arm_up,<1
RWI=rwi_arm_down,>1
RWO=rwo_raise_arm,0
RWO=rwo_lower_arm,2
```

Blank lines and those beginning with "!" are treated as comments and ignored.

Do not leave spaces around the "=" or "," and ensure that the keywords (DDE_ITEM, RWI, RWO, etc) are in uppercase.

The lines beginning "STOP=" specify the names of real-world outputs which will be invoked when the Simulator stops recording. For example, if you are controlling a motor via two real-world outputs, one to start it and the other to stop it, you would probably want to stop the motor when you stop the Simulator recording. You can have as many of these "STOP=" lines as you want.

The lines beginning "DDE_SERVER=" and "DDE_TOPIC=" specify the service and topic for the DDE conversation.

There then follows sets of entries relating to the items within the topic from which the Simulator will obtain its real-world inputs and to which it will notify when real-world outputs are invoked. Typically, a DDE_ITEM will be associated with either real-world inputs or real-world outputs but not both.

In the first "DDE_ITEM=" block above, the Simulator will establish an advise loop (hot link) with the server such that it will be notified if the item

"gripper_item" changes value. When the Simulator receives a new value, it examines its value. It first compares its value with the string "open" (case sensitive). If it doesn't match, it will set real-world input "rwi_grip_closed" to true (the entry in the Real-World Input Status dialog will be highlighted), otherwise it will set it to false. It then tests the value against "open" again. If it matches, the Simulator sets the real-world input "rwi_grip_open" to true, otherwise it sets it to false.

When the real-world output "rwo_open_grip" is invoked, the Simulator will send the string "open_gripper" to DDE item "gripper_item". When real-world output "rwo_close_grip" is invoked, the Simulator sends the string "close_gripper".

Now consider the second "DDE_ITEM=" block. When the Simulator receives a new value for DDE item "arm_elevation_item", it attempts to read the string as a number. If the number is less than 1.0 (the number in the file can be a floating point or integer number), the real-world input rwi_arm_up will be set to true (otherwise it will be set to false). If the number is greater than 2.0, the real-world input rwi_arm_down will be set to true (otherwise false).

The relational operators supported for numerics are <, <=, >= and >.

When the real-world output "rwo_raise_arm" is invoked, the Simulator will send the string "0" to DDE item "arm_elevation_item". When real-world output "rwo_lower_arm" is invoked, the Simulator sends the string "2".

Load the mapping file

Choose DDE Mapping|Load Mapping File to load the mapping file. This will result in a list file being produced with the same name as the mapping file but extension ".mpo" (for mapping output). This shows what the Simulator will use. The file format is such that it can be used as the mapping file if required.

Connect to the DDE server



Loading the mapping file will automatically cause the Simulator to attempt to connect to the DDE server. Alternatively, choose DDE Mapping|Connect to DDE Server. To disconnect from the DDE server, choose Mapping|Disconnect from DDE Server.

Drive the DDE server

As the DDE server notifies the Simulator of changing values of DDE items, the entries in the Real World Input Status dialog can be seen to change (i.e. whether or not they are highlighted). Click on the record button on the Simulator Control dialog to run the control logic. As real-world outputs are invoked, the DDE server will be sent the appropriate messages as described above.

6 Create, Save And Replay An Event Log

Event logs can be generated by the Analyzer, by a Synect-generated control program or by the Simulator itself. This chapter describes the options regarding creating a new empty event log within the Simulator, how to save it to disk and how to load an event log from disk to replay its contents.

Creating A New Empty Event Log

When an application is loaded into the Simulator, an empty event log is automatically created with the capacity to store 50 entries and configured to be linear (i.e. the Simulator will stop recording when it's full rather than overwriting the oldest entry).



If you want to reset the application to its initial state but you've been using a circular event log which has overwritten its oldest entry, choose **Event Log|New - Initial States** to create an empty event log, starting at the application's initial state.



Having been driving the Simulator interactively, you might reach a point where the application could follow one of a number of paths (depending on the order in which real-world inputs are satisfied). In this circumstance, ensure that the Simulator mode is Stopped and save the event log to disk (see section Saving An Event Log, below). Choose **Event Log|New - Current States** to create an empty event log, starting at the application's current state. You may then follow the first path and save the resulting event log. To explore one of the other paths, load in the first event log (which ends at the application state from which you want to start) and choose **Event Log|New - Current States** again to create a new empty event log starting at the same application state. You can then repeat this set of actions as often as required to obtain event logs which record the behaviour having followed each of the alternative paths available.



Saving An Event Log



The Simulator mode (shown in the title of the Simulator Control dialog) must be STOPPED before the event log can be saved to disk. Choose **Event Log|Save** to start the standard File Save dialog from which you can specify the name of the file in which you want to save the event log.

Replaying An Event Log



To load an event log for replay, the Simulator mode must be STOPPED. Choose **Event Log|Open** to start the standard File Open dialog. By default, the dialog will list files with extension ".sel". These are event logs which were generated by the Simulator or the Analyzer. To list event logs generated by a

Synect C control program, click on the `List Files of Type` drop-down listbox and click on entry `C Event Logs (*.cel)` to see the available C event log files.

Having loaded an event log, the selected position is set at the end of the event log. You will need to move the selected position further back in the event log before the "Play" button will be enabled.

By default, the Simulator will read the next entry from the event log every 3 seconds. To use the relative timing stored in the event log, choose `Configuration|Timing` to start the Timing Attributes dialog (see chapter 7, Configuration, for details).

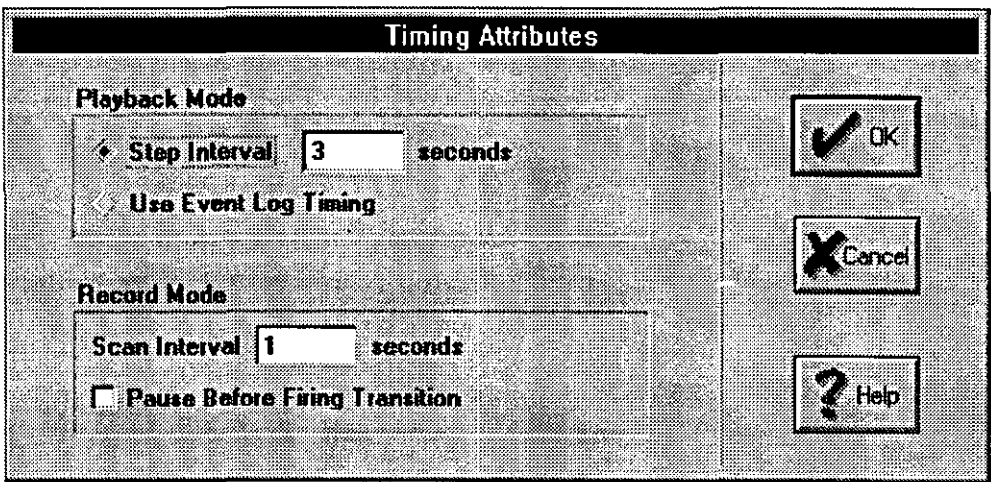
7 Configuration

This chapter describes the configurable attributes of the Simulator. These fall into three categories:

- timing attributes.
- event log attributes.
- how the Real World Input Status dialog and the Real World Outputs dialog are handled.

Configuring Timing Attributes

Choose Configuration|Timing to start the Timing Attributes dialog, shown below:



This dialog can be thought of in two parts. One part relates to when the Simulator is recording an event log and the other relates to when the Simulator is replaying an event log.

Playback Mode

When the Simulator is replaying an event log, it steps through the entries in the event log. The interval between reading successive entries can be either fixed (you specify the number of seconds) or correspond to the interval when the entries were originally written to the event log.

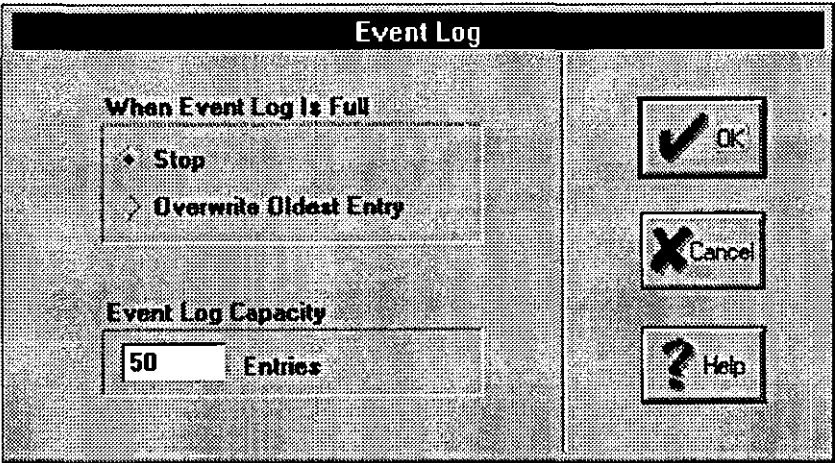
Record Mode

When the Simulator is recording, it runs your application logic at the interval you specify (in seconds).

You can instruct the Simulator to pause before firing a transition. In this case, the "Fire Trans" button on the Simulator Control dialog will be enabled when the Simulator is ready to fire a transition. The transition will be fired when you click on the "Fire Trans" button.

Configuring Event Log Attributes

Choose Configuration|Event Log to start the Event Log dialog, shown below:



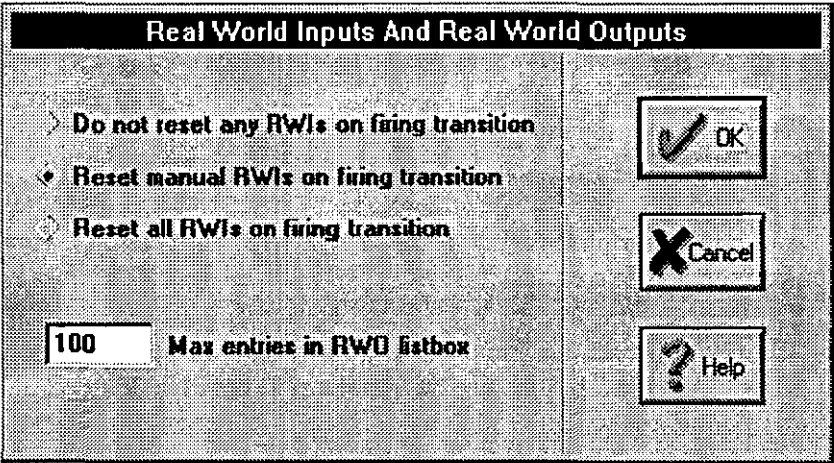
This dialog can be thought of in 2 parts.

The first part relates to the behaviour of the Simulator when recording and it fills the event log. The options are for the Simulator to stop or to overwrite the oldest entry.

The second part relates to the size of the event log in terms of number of entries. If this number is reduced such that information will be lost, a warning message is displayed.

Configuring The Handling of RWI and RWO Dialogs

Choose Configuration|RWI And RWO Dialogs to start the Real World Inputs And Real World Outputs dialog, shown below:



This dialog can be thought of in two parts.

The first part relates to the treatment of real-world inputs following the firing of a transition. By default, the "Reset manually set RWIs on firing transition" radiobutton is checked. The interpretation of this radiobutton can best be described by an example. In the demo application, the real-world input "rwi_go" must be enabled for the first transition to fire. When running the Simulator, the "rwi_go" entry in the Real World Input Status dialog would be highlighted by clicking on it.

With the "Reset manually set RWIs on firing transition" radiobutton checked, the Simulator will reset the real-world input status having fired the transition which referenced it. With the radiobutton unchecked, you must click on the "rwi_go" entry in the listbox again to remove the highlight. Beware that, whilst this makes it easier to drive the application around manually, it may not adequately represent how the target plant equipment will behave. In particular, a race condition could more easily be overlooked.

With the "Reset manually set RWIs on firing transition" radiobutton or "Do not reset any RWIs on firing transition" radiobutton checked, the Simulator will not reset those real-world inputs which are determined via a DDE link. On initialisation, the Simulator assumes that each real-world input is being determined by the user until a DDE message is received which relates to the real-world input.

With the "Reset all RWIs on firing transition" radiobutton checked, the Simulator will also reset those real-world inputs which are determined via a DDE link.

The second part of the dialog relates to the Real World Outputs dialog. The number of entries in the listbox is constrained to the maximum which is specified by the "Max entries in RWO listbox" editbox. When this number of entries has been reached and another entry is to be added, the oldest entry will first be removed. The default is 100 entries but you can change this to any value between 5 and 1000.

This page left intentionally blank









Appendix A Menus

File	
Open Application	Open an existing application
Exit	Finish running Synect Simulator
Event Log	
New - Initial States	Create a new event log starting at initial states
New - Current States	Create a new event log starting at current states
Open	Open an existing event log
Save	Save current event log to file
Configuration	
Timing	Configure the speed at which the Simulator runs
Event Log	Configure event log capacity and behaviour when full
RWI And RWO Dialogs	Configure how RWI/RWO dialog boxes are handled
DDE Mapping	
Load Mapping File	Load mapping between RWI/RWO and external DDE server
Connect To DDE Server	Connect to DDE server
Disconnect From DDE Server	Disconnect from DDE server
Help	
Contents	Help table of contents
Using help	Help on using online Help
About	About Synect Simulator

This page left intentionally blank

Appendix B Toolbar Buttons

The Simulator provides an Operation Toolbar displayed horizontally across the top of the screen. It contains buttons which can be used as shortcuts instead of pulling down the corresponding menu and selecting the relevant item. The buttons will be greyed-out if the corresponding function is unavailable at that time.

	<u>Menu Equivalent</u>	<u>Usage</u>
	File Open Application	Open an existing application
	Event Log New - Initial States	Create a new event log starting at initial states
	Event Log New - Current States	Create a new event log starting at current states
	Event Log Open	Open an existing event log
	Event Log Save	Save current event log to file
	DDE Mapping Connect To DDE Server	Connect to DDE server
	DDE Mapping Disconnect From DDE Server	Disconnect from DDE server
	Help Contents	Display help information

This page left intentionally blank

Appendix C DDE Services

Windows Dynamic Data Exchange (DDE) provides a mechanism for applications to exchange data. The Simulator and the STD Monitor use DDE so that the Simulator can tell the STD Monitor the current state of the application, which transitions are enabled etc.. The Simulator also provides other DDE topics so that third party products, such as Wonderware's InTouch SCADA package, can be driven from the Simulator.

The service name is Synect. The topic and item names are as follows:

Topic: SynectMonitorData

Item: CurrentInfo

This topic and item is reserved for use with the STD Monitor application.

Topic: SynectApplicationNames

Item: "STDNames"

This topic allows a client to find out the names of the STDs in the application. The client issues a one-off request for this data (i.e. advise loop is not supported). The names are returned separated by carriage return (ASCII 13 decimal) and linefeed (ASCII 10 decimal), the whole string being terminated with a null (ASCII 0). For example, if an application consisted of 3 STDs "STD1", "STD2" and "STD3", the information returned would be:

STD1<cr><lf>STD2<cr><lf>STD3<nul>

Topic: SynectApplicationNames

Item: "RWINames"

This topic allows a client to find out the names of the real-world inputs in the application. The client issues a one-off request for this data (i.e. advise loop is not supported). The names are returned separated by carriage return (ASCII 13 decimal) and linefeed (ASCII 10 decimal), the whole string being terminated with a null (ASCII 0).

Topic: SynectApplicationNames

Item: "RWONames"

This topic allows a client to find out the names of the real-world outputs in the application. The client issues a one-off request for this data (i.e. advise loop is not supported). The names are returned separated by carriage return (ASCII 13 decimal) and linefeed (ASCII 10 decimal), the whole string being terminated with a null (ASCII 0).

Topic: SynectSTDStates

Item: Name of STD

This topic allows a client to find out the names of the states in the specified STD. The client issues a one-off request for this data (i.e. advise loop is not supported). The names are returned separated by the tab character (ASCII 9 decimal), the whole string being terminated with a null (ASCII 0).

Topic: SynectStateData**Item: Name of STD**

This topic allows a client to find out the current state of the specified STD. The client should establish an advise loop - the Simulator will then notify the client whenever the state changes. The format of the STD name is exactly as listed in the Select STD To Display dialog in the STD Monitor application. The Simulator returns a data handle from which the client copies the state name.

Topic: SynectRWICommandedStateData**Item: Name of real world input**

This topic allows a client to interact with the RWI Status listbox used in the Simulator. The client should establish an advise loop - the Simulator will then notify the client if the user clicks on an RWI in the listbox. The client may also poke a new value, in which case, the RWI Status listbox will be updated accordingly. The commanded status of the RWI is specified as a 2 byte string with the first character being character '0' to denote unset or character '1' to denote set and the second character being zero (string terminator).

Topic: SynectRWOInvokedData**Item: Name of real world output**

This topic allows a client to be notified whenever a real world output is invoked. The client should establish an advise loop - the Simulator will then notify the client when the RWO is invoked. The RWO status is returned as a boolean in the same format as specified for SynectRWICommandedStateData topic above. After the RWO has been invoked, the Simulator will return the status denoting the set state. The client must then poke the RWO to reset the status to the unset state.

A weakness in this scheme becomes apparent when the user changes the current position in the event log (by means of the Simulator Control dialog). The state of each STD is available at each record in the event log but no information exists about the invocation of RWOs. Consequently, if the user changes the current position in the event log, clients will be notified of the current state but no notification regarding RWOs will be given.

Synect

C Code Generator User Guide

Version 1.6

**Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough
Cleveland, TS14 8JY
England**

**Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk**

© Copyright 1994, 1995, 1996, 1997 Hopkinson Computing Limited. All rights reserved.

Synect is a registered trademark of Hopkinson Computing Limited

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Information in this User Guide is subject to change without notice and does not represent a commitment on the part of Hopkinson Computing Limited.

The software described in this User Guide is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. No part of this User Guide may be reproduced or transmitted in any form or by any means, electronic or otherwise, including photocopying and recording, for any purpose, without the express written permission of Hopkinson Computing Limited.

Document History

10 June 1996

Re-issue reflecting C Code Generator V1.3 - new cover sheet and chapter 1.

28 October 1996

Re-issue reflecting C Code Generator V1.4 - changes to cover sheet and chapter 5.

30 April 1997

Re-issue reflecting C Code Generator V1.5 & V1.6 - changes to cover sheet, contents and chapters 2 to 5.

Contents

Chapter 1	Introducing Synect	1
	User Interface	1
	The Method	1
	Synect Documentation	1
	The Tools	2
	Document Conventions	2
Chapter 2	Getting Started	3
	System Requirements	3
	Installation	3
	Starting the C Code Generator	4
	On-Line Help	4
Chapter 3	Basic Concepts	5
	Scope Of The Code Produced	5
	Data-oriented	5
	Code-oriented	5
	Files Generated	5
	Data-oriented	5
	Code-oriented	6
	The Configuration File (code-oriented variant only)	6
	Format Of The Mapping File	6
Chapter 4	Open An Application	7
	Opening An Application	7
Chapter 5	Code Options	9
	Interrupt-Driven Or Scan-Based	9
	Interrupt-Driven	9
	Scan-Based	9
	Diagnostic Options	10
	Event Logging	10
	Host Messaging	10
	Generating The Code	10

This page left intentionally blank

1 Introducing Synect

Synect is a set of software tools which helps the designer of a control system to produce a specification which is clear, precise and free of errors. Synect combines the ease of use of a graphical user interface, with a widely used diagrammatic notation and the power of mathematical modelling.

User Interface

The Synect™ tools run on the Microsoft® Windows™ operating system. As such, you need to know how to use Windows before you use Synect. In particular, you will need to know how to use the mouse to click, double-click and drag. You will also need to know how to interact with menus, dialogs and how to move, resize and close windows. For information, refer to the Microsoft Windows User's Guide.

The Method

Synect uses an object-based method to enable you to model the system you want to control. It is useful to have some knowledge of object-based/object-oriented analysis and design techniques before attempting to use Synect in earnest.

Synect Documentation

Each of the Synect tools has an associated User Guide. This User Guide explains how to use each of the functions available in the C Code Generator. It also explains why you might want to use the function.

Each application has context-sensitive on-line help. The C Code Generator on-line help contains a "How Do I?" section, including a "How Do I Use The C Code Generator?" sub-section for first-time users.

A Tutorial is also provided which offers a worked example and shows how each of the tools is used with the example application.

The Tools

The tools which make up the Synect toolset are:

Application Editor	graphical means of defining the application.
Compiler	check the specification for consistency and possible warnings and generate a mathematical model of the application.
Analyzer	ability to check for design errors such as deadlock (where the system "hangs") and unwanted state combinations.
Simulator	provides the ability to interactively "drive" the application or replay past behaviour of the live control system.
STD Monitor	animates the specification (used in conjunction with the Simulator or the live control system).
C Code Generator	generate ANSI-standard C code to implement the application.
Neuron C Generator	generate Neuron C to run on one or more nodes on an Echelon LonWorks network to implement a distributed control solution.
Ladder Logic Generator	generate relay ladder logic to run on a programmable controller.

Document Conventions

The User Guide adopts the following conventions:

<code>application name</code>	text that you type or that you see on the screen.
<code>KEY NAME</code>	keyboard keys, such as ENTER, CTRL or DEL.
<code>Menu Choice</code>	a menu option, such as File Exit denoting choose the Exit command from the File menu.
<i>description</i>	description of a term with a specific meaning.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the C Code Generator.
- how to start the C Code Generator.
- the C Code Generator window.

System Requirements

The Synect C Code Generator requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

Other Synect tools also requires that you have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

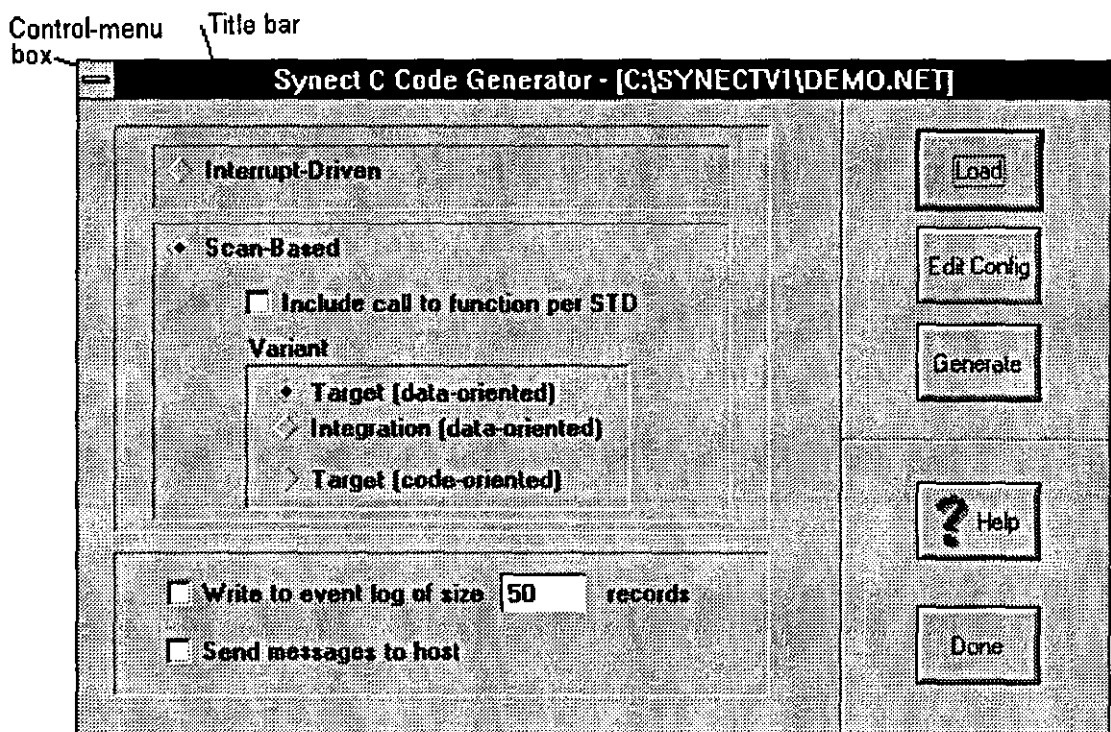
- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press `ENTER`.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

- 8 Read the installed file `readme.txt` for details of any changes to the product or documentation since the documentation was printed.

Starting the C Code Generator

Ensure that the Program Manager window is on display. If the Synect window isn't visible, use the Window menu to open it. Double-click on the C Code Generator icon or use the keyboard arrow key to select the icon and then press ENTER.

When using the C Code Generator, the window displays the following dialog:



Title bar

The title bar shows the product title `Synect C Code Generator` and the name of the application which has been loaded (if any).

Control-menu box

Allows you to move or close the window. Also allows you to open the control panel or obtain information about the C Code Generator.

On-Line Help

Click on the "Help" button to take you to the help contents page. The help information is shown in a separate window.

3 Basic Concepts

The Synect Application Editor enables you to specify a model of your application. See the Application Editor User Guide, chapter 3, Basic Concepts for more information relating to defining the model. The Compiler derives a mathematical model from your specification. The C Code Generator generates an ANSI standard C program from the mathematical model.

Scope Of The Code Produced

The C Code Generator can produce code in two different forms. The first form is data-oriented, where the relationships between states and transitions are held in arrays. The second form is code-oriented, resulting in an "if" statement for each transition.

Data-oriented The C Code Generator partitions the code it produces into several files. It does not have sufficient information to be able to produce a complete implementation. This is partly due to the generated code being platform-independent. Some of the files therefore contain stubs which must be completed or replaced with the appropriate software. For example, the C Code Generator creates a function for each real-world input, where the function simply returns TRUE. It is for the developer to code the mechanism by which the status of the real-world input is determined (by polling a sensor, for example).

The "engine" generated by the C Code Generator is data-driven. The C code for the engine is, however, also supplied. This code must not be manually altered - to do so may introduce inconsistencies into the software or violate the logic defined in your specification.

Code-oriented The C Code Generator partitions the code into a C source file and a header file (with extension ".ch1"). A configuration file with extension ".cin" provides the facility for specifying #define macros to replace real-world input or real-world output function calls. It also enables an include filename.

Files Generated

The name of each file generated is derived from the name of the application but with different extensions for each file:

Data-oriented	".c" (C source)	The engine.
	".h" (C include)	Include file.

	<code>".env"</code> (C source)	Environment functions. This file contains stubs for a function which is called on completion of each scan (scan-based code only) and a function which is called if a deadlock is detected.
	<code>".rwi"</code> (C source)	Real-world inputs. This file contains a stub for each real-world input. If the interrupt-driven code was requested, this file also contains a stub for the interrupt-level invocation of each real-world input.
	<code>".rwo"</code> (C source)	Real-world outputs. This file contains a stub for a function called at initialisation which should initialise the environment as required. It also contains a stub for each real-world output.
	<code>".std"</code> (C source)	Produced if the Include call to function per STD checkbox was checked. Contains the function for each STD so that you can add code which is to be executed when an STD is in a particular state.
Code-oriented	<code>".c"</code> (C source)	The engine.
	<code>".ch1"</code> (C include)	Include file.
	<code>".cin"</code>	Configuration file.

The Configuration File (code-oriented variant only)

To edit the configuration file, select the code-oriented variant by checking the "Target (code-oriented)" checkbox and then pressing the "Edit Config" button. This starts a simple text editor with the configuration file loaded. If no configuration file exists, a new file is created and populated with commented out entries.

Format Of The Mapping File Consider the following example:

```
HEADER_FILENAME=user_hh.h
RWI=rwi_go,switch_contact==MADE
RWO=rwo_start_feed_conveyor,motor_m2_command=RUN
```

It is worth noting the following points:

- The format of a line defining the code to be substituted for a real world input is:
`RWI=real_world_input_name,code_to_substitute`
- The format of a line defining the code to be substituted for a real world output is:
`RWO=real_world_output_name,code_to_substitute(without the semicolon)`
- The C Code Generator performs a case-sensitive read of the mapping file and will not ignore spaces unless to the right of a comma.

4 Open An Application

When you first start the Synect C Code Generator, none of the options are enabled and the "Generate" button is disabled. This is because you must first open an application.

When you compiled your application using the Compiler, it created a file with the name of your application but with the extension ".net". This file is loaded into the C Code Generator.

Opening An Application

To open an application, click on the "Load" button. The standard file open dialog will then be started, allowing you to specify the name of the file from which the application is to be loaded. The dialog will list files with extension ".net".

Having loaded the application, the radio buttons and checkboxes will be enabled and the "Generate" button will be enabled.

This page left intentionally blank

5 Code Options

This chapter describes the different options which will determine how your application logic is invoked and alternatives relating to testing and diagnostic capabilities.

Interrupt-Driven Or Scan-Based

The first choice is between interrupt-driven and scan-based code. The interrupt-driven alternative may be more efficient in terms of processor loading and should lead to faster response times, but requires that the target environment supports an interrupt-driven approach. It also requires more involvement from the developer.

Interrupt-Driven

If the interrupt-driven alternative is chosen, each real world input will result in 2 functions being generated. The first of these is used at non-interrupt level and simply allows the engine to determine the last known status of the real-world input at any time. The second is the function which is to be invoked at interrupt level when the real-world input becomes enabled (i.e. returns TRUE). Because the code generated is independent of the target platform, the control of access to critical sections of code needs to be implemented by the developer.

Scan-Based

If the scan-based version is chosen, you can check the `Include call to function per std` checkbox to instruct the C Code Generator to add code such that you can perform routine functionality depending on the state of an STD. For the data-oriented C code, a separate function is created for each STD and a parameter notifies the function of the STD's current state. This function is called every scan. For the code-oriented variant, each STD state is assumed to have its own function to be called.

The code variants available are as follows:

- | | |
|-----------------------------|--|
| Target (data-oriented) | The code which you will run in the target environment. |
| Integration (data-oriented) | This is an interactive variant, using <code>printf</code> statements to inform you of the current system state, which transitions are enabled etc. and allowing you to decide which transition to fire. This variant is intended for use where you want to integrate parts of your controlled system but simulating the remainder and "driving" it manually. |
| Target (code-oriented) | This option generates the alternative form of C code, producing "switch" and "if" statements. |

Diagnostic Options

Event logging and host messaging are available for data-oriented variants, but not for the code-oriented variant. Within the data-oriented variants, event logging and host messaging are supported by both scan-based and interrupt-driven implementations.

Event Logging

If you check the "Write to event log" checkbox, the generated code will log the testing of real world inputs and the firing of transitions to an event log file. The event log will be circular such that when it is full it will overwrite the oldest entry. The event log's capacity is 50 records by default but you can change this via the editbox.

The event log maintained by the control program can then be loaded into the Simulator to replay the sequence of activities which occurred.

Host Messaging

If you check the "Send messages to host" checkbox, the generated code will call functions to send information to a host device when a real world input is tested and when a transition is fired. The body of these functions is not defined because it is dependent on the target environment. This functionality is provided to enable a Synect Server running on a PC to obtain information about the behaviour of the live control system such that it can be made available for presentation to operational staff. Presentation of the information could be via the Synect STD Monitor or a third party product, such as a SCADA product.

Generating The Code

Click on the "Generate" button to create the files containing the C code conforming to the options you specified.

Synect

Distributed Neuron C Code Generator User Guide

Version 2.2

**Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough
Cleveland, TS14 8JY
England**

**Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk**

© Copyright 1994, 1995, 1996 Hopkinson Computing Limited. All rights reserved.

Synect is a registered trademark of Hopkinson Computing Limited
Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Information in this User Guide is subject to change without notice and does not represent a commitment on the part of Hopkinson Computing Limited.

The software described in this User Guide is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. No part of this User Guide may be reproduced or transmitted in any form or by any means, electronic or otherwise, including photocopying and recording, for any purpose, without the express written permission of Hopkinson Computing Limited.

Neuron C Consultant: Dr. Rob Harrison

Document History

10 June 1996

Re-issue reflecting Neuron C Generator V2.0 - complete re-write.

28 October 1996

Re-issue reflecting Neuron C Generator V2.1 and V2.2 - changes to cover, contents, chapters 2, 3, 4 & 5.

Contents

Chapter 1	Introducing Synect	1
	User Interface	1
	The Method	1
	Synect Documentation	1
	The Tools	2
	Document Conventions	2
Chapter 2	Getting Started	3
	System Requirements	3
	Installation	3
	Starting the Neuron C Code Generator	4
	On-Line Help	5
Chapter 3	Basic Concepts	7
	How It Works	7
	Mapping A Transition To A When Statement	7
	Include Files	8
	The Mapping File	9
	Format Of The Mapping File	9
Chapter 4	Open An Application	11
	Opening An Application	11
Chapter 5	Code Options	13
	Defining Nodes	13
	Adding A New Node	13
	Deleting A Node	13
	Changing A Node's Name	13
	Renaming A Node Id	13
	Defining Object To Node Mapping	14
	Edit The Mapping File	14
	Generate The Source Code And Include Files.....	14

This page left intentionally blank

1 Introducing Synect

Synect is a set of software tools which helps the designer of a control system to produce a specification which is clear, precise and free of errors. Synect combines the ease of use of a graphical user interface, with a widely used diagrammatic notation and the power of mathematical modelling.

User Interface

The Synect™ tools run on the Microsoft® Windows™ operating system. As such, you need to know how to use Windows before you use Synect. In particular, you will need to know how to use the mouse to click, double-click and drag. You will also need to know how to interact with menus, dialogs and how to move, resize and close windows. For information, refer to the Microsoft Windows User's Guide.

The Method

Synect uses an object-based method to enable you to model the system you want to control. It is useful to have some knowledge of object-based/object-oriented analysis and design techniques before attempting to use Synect in earnest.

Synect Documentation

Each of the Synect tools has an associated User Guide. This User Guide explains how to use each of the functions available in the Neuron C Code Generator. It also explains why you might want to use the function.

Each application has context-sensitive on-line help. The Neuron C Code Generator on-line help contains a "How Do I?" section, including a "How Do I Use The Neuron C Code Generator?" sub-section for first-time users.

A Tutorial is also provided which offers a worked example and shows how each of the tools is used with the example application.

The Tools

The tools which make up the Synect toolset are:

Application Editor	graphical means of defining the application.
Compiler	check the specification for consistency and possible warnings and generate a mathematical model of the application.
Analyzer	ability to check for design errors such as deadlock (where the system "hangs") and unwanted state combinations.
Simulator	provides the ability to interactively "drive" the application or replay past behaviour of the live control system.
STD Monitor	animates the specification (used in conjunction with the Simulator or the live control system).
C Code Generator	generate ANSI-standard C code to implement the application.
Neuron C Generator	generate Neuron C to run on one or more nodes on an Echelon LonWorks network to implement a distributed control solution.
Ladder Logic Generator	generate relay ladder logic to run on a programmable controller.

Document Conventions

The User Guide adopts the following conventions:

<code>application name</code>	text that you type or that you see on the screen.
<code>KEY NAME</code>	keyboard keys, such as ENTER, CTRL or DEL.
<code>Menu Choice</code>	a menu option, such as File Exit denoting choose the Exit command from the File menu.
<i>description</i>	description of a term with a specific meaning.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the Neuron C Code Generator.
- how to start the Neuron C Code Generator.
- the Neuron C Code Generator window.

System Requirements

The Synect Neuron C Code Generator requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

Other Synect tools also requires that you have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

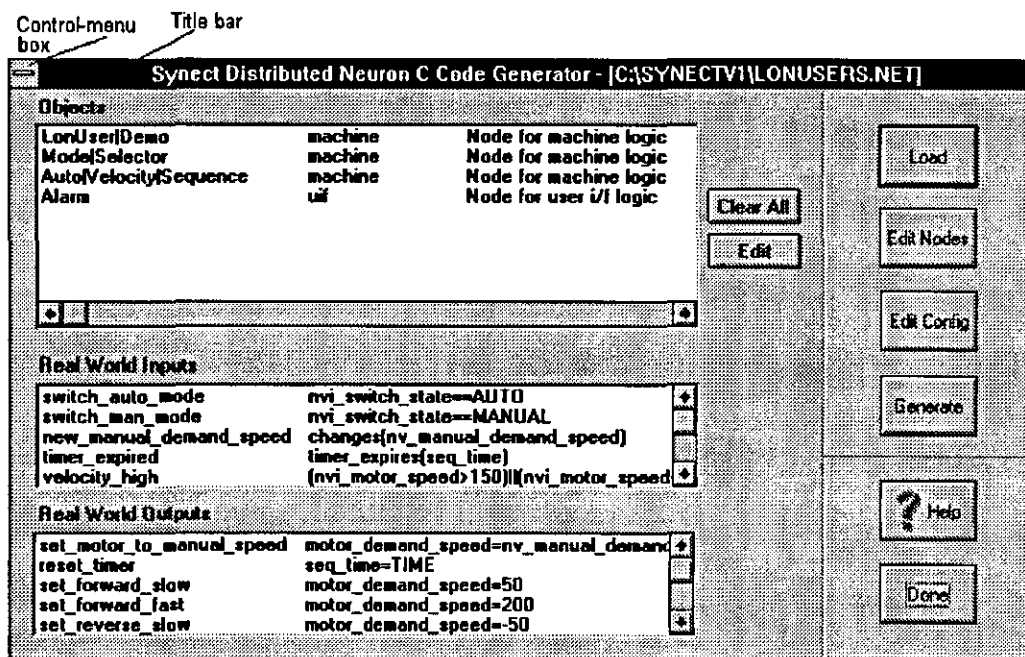
- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press ENTER.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

- 8 Read the installed file `readme.txt` for details of any changes to the product or documentation since the documentation was printed.

Starting the Neuron C Code Generator

Ensure that the Program Manager window is on display. If the Synect window isn't visible, use the Window menu to open it. Double-click on the Neuron C Code Generator icon or use the keyboard arrow key to select the icon and then press ENTER.

When using the Neuron C Code Generator, the window displays the following dialog:



Title bar

The title bar shows the product title `Synect Distributed Neuron C Code Generator` and the name of the application which has been loaded (if any).

Control-menu box

Allows you to move or close the window. Also allows you to open the control panel or obtain information about the Neuron C Code Generator.

Objects listbox

Reading from left to right, each row of the listbox contains an object name, a node id and a node name. This listbox shows, for each object defined in the

object hierarchy, the node to which it is allocated and the name of the node. In the above example, the "Alarm" object is the only object assigned to the "uif" node - all of the other objects are assigned to node "machine". The name associated with node "uif" is "Node for user i/f logic".

Real World Inputs listbox

Reading from left to right, each row of the listbox contains a real world input name and its corresponding Neuron C code. This listbox shows, for each real world input, the Neuron C code which will be substituted when the "Generate" button is pressed.

Real World Outputs listbox

Reading from left to right, each row of the listbox contains a real world output name and its corresponding Neuron C code. This listbox shows, for each real world output, the Neuron C code which will be substituted when the "Generate" button is pressed.

On-Line Help

Click on the "Help" button to take you to the help contents page. The help information is shown in a separate window.

This page left intentionally blank

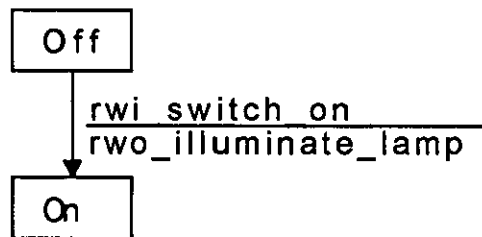
3 Basic Concepts

The Synect Application Editor enables you to specify a model of your application. See the Application Editor User Guide, chapter 3, Basic Concepts for more information relating to defining the model. The Compiler derives a mathematical model from your specification. The Neuron C Code Generator generates a C program from the mathematical model which can then be compiled and run on an Echelon LonWorks Neuron target environment.

How It Works

Mapping A Transition To A When Statement

Consider a node on the network which is to be responsible for the logic corresponding to STD "Switch.main". Assume that this STD contains a transition as follows:



This transition specifies that when the STD is in state "Off" and the condition "rwi_switch_on" detects that the user has pressed the appropriate button, the real world output "rwo_illuminate_lamp" is invoked to apply power to the lamp and the STD changes state to state "On".

Each STD has a corresponding integer variable in the generated Neuron C code to remember the current state of the STD. The variable for STD "Switch.main" will be "nvo_Switch_main". Assume that constants are defined for states "Off" and "On" to be 1 and 2 respectively. The transition can be mapped to the following when statement:

```

when (nvo_Switch_main == Off
    && rwi_switch_on() == TRUE)
{
    nvo_Switch_main = On;
    rwo_illuminate_lamp();
}
  
```

Assume further that the real world input "rwi_switch_on" is determined by reading an i/o pin and that an i/o pin controls whether the lamp is illuminated as per the following declarations:

```

IO_3  input  bit    switch_io_line
IO_2  output bit    lamp_io_line
  
```

Defining that "rwi_switch_on" is to be translated to "io_in(switch_io_line)==1" and "rwo_illuminate_lamp" is to be translated to "io_out(lamp_io_line, 1)", the when statement becomes:

```
when (nvo_Switch_main == Off
      && io_in(switch_io_line)==1)
{
  nvo_Switch_main = On;
  io_out(lamp_io_line,1);
}
```

In order to optimise the use of when clause table space when the .nc source file is compiled using the Echelon compiler, the individual conditions within the when statement are moved to a function:

```
boolean synect_t0 ()
{
  return (nvo_Switch_main == Off
          && io_in(switch_io_line)==1);
}

when (synect_t0())
{
  nvo_Switch_main = On;
  io_out(lamp_io_line,1);
}
```

Include Files There are two different aspects to include files:

- Files which are read by the Neuron C Code Generator and included into the Neuron C source code (into files with extension ".nc").
- Include files which the Neuron C Code Generator creates which are referenced by `#include` directives in the Neuron C source code.

There are two files for each target node which fall into the first category:

- The first of these contains header information which is copied to the start of the Neuron C source code. This will typically contain compiler directives such as `#pragma` and `#define`.
- The second contains the initialisation code for the `when reset` statement. The contents of the file are copied into the Neuron C source code to be executed in the `when reset` task body.

The Neuron C Code Generator creates four include files for each node. These have the same filename as the node name and their extensions are:

- ".h0" - contains a `#define` to convert any variable names longer than 16 characters to a unique name which is 16 characters in length. This overcomes the Echelon compiler constraint limiting variable names to 16 or less characters in length.

- ".h1" - contains a `#define` for each state name.
- ".h2" - contains a `network input int nvi_var_name` statement for each STD which the node needs to reference which is hosted on a different node.
- "h3" - contains the function definitions containing the individual conditions to be evaluated in when statements (see section "Mapping A Transition To A When Statement" above).

The Mapping File

The mapping file stores the configured data which defines for each target node:

- The node's id and name.
- The objects which are to be mapped to the node.
- The name of the include file to be copied into the header of the generated Neuron C.
- The name of the include file to be copied into the task body of the `when reset` statement.
- The Neuron C which is to be substituted for each real world input.
- The Neuron C which is to be substituted for each real world output.

The mapping file has the same name as the application but with extension ".nod". If you have not created the mapping file when the Neuron C Generator loads an application, it will issue the following warning message (but will still carry on):

```
Couldn't open processing node definition file <file_name>
```

When the "Edit Config" button is pressed, a new version of the mapping file will be created if the configuration has changed. In order to minimise the risk of typing errors, the file will contain commented-out references to objects which are not mapped to a node and to real-world inputs and real-world outputs not referenced by any object. When the "Generate" button is pressed, a new version of the mapping file is always created. The original is copied to the file with same name but extension ".no1".

Format Of The Mapping File

The following mapping file content was used for the example screen layout shown in Chapter 1:

```
NODE=machine
NAME=Node for machine logic
HEADER_FILENAME=machine.hh
WHEN_RESET_FILENAME=machine.hwr
OBJECT=LonUser|Demo
OBJECT=Mode|Selector
OBJECT=Auto|Velocity|Sequence
RWI=switch_auto_mode,nvi_switch_state==AUTO
RWI=switch_man_mode,nvi_switch_state==MANUAL
RWI=new_manual_demand_speed,changes(nv_manual_demand_speed)
RWO=set_motor_to_manual_speed,motor_demand_speed=nv_manual_demand_speed
```

```

RWO=reset_timer,seq_time=TIME
RWO=set_forward_slow,motor_demand_speed=50
RWI=timer_expired,timer_expires(seq_time)
RWO=set_forward_fast,motor_demand_speed=200
RWO=set_reverse_slow,motor_demand_speed=-50
RWO=set_reverse_fast,motor_demand_speed=-200
RWO=stop_motor,motor_demand_speed=0

NODE=uif
NAME=Node for user i/f logic
HEADER_FILENAME=uif.hh
WHEN RESET_FILENAME=uif.hwr
OBJECT=Alarm
RWI=velocity_high,(nvi_motor_speed>150)|| (nvi_motor_speed<-150)
RWO=set_buzzer_on,io_out(buzzer_io_line,ON)
RWI=velocity_ok,(nvi_motor_speed<=150)&&(nvi_motor_speed>=-150)
RWO=set_buzzer_off,io_out(buzzer_io_line,OFF)

```

It is worth noting the following points:

- By convention, the code to be copied into the header of the generated code is stored in a file with extension ".hh".
- By convention, the code to be copied into the `when reset` task body of the generated code is stored in a file with extension ".hwr".
- The format of a line defining the code to be substituted for a real world input is:

```
RWI=real_world_input_name,code_to_substitute
```

- The format of a line defining the code to be substituted for a real world output is:

```
RWO=real_world_output_name,code_to_substitute(without the
semicolon)
```

- The Neuron C Code Generator performs a case-sensitive read of the mapping file and will not ignore spaces unless to the right of a comma.

4 Open An Application

When you first start the Synect Neuron C Code Generator, the "Edit Nodes", "Edit Config" and "Generate" buttons are disabled. This is because you must first open an application.

When you compiled your application using the Synect Compiler, it created a file with the name of your application but with the extension ".net". This is the file which is to be loaded into the Neuron C Code Generator.

Opening An Application

To open an application, click on the "Load" button. The standard file open dialog will then be started, allowing you to specify the name of the file from which the application is to be loaded. By default, the dialog will list files with extension ".net".

If you have previously generated code for this application, a mapping file will have been created containing the node names and the mapping of objects to each of these nodes. When you load the application, this file is automatically read and the listboxes populated accordingly. The mapping file is a file whose name corresponds to the application name and with extension ".nod".

If this is the first time you've loaded this application into the Neuron C Code Generator, you'll get a warning message to notify you that the mapping file couldn't be found. Simply click on the "Ok" button and continue as normal.

Having loaded the application, the listboxes and buttons will be enabled.

This page left intentionally blank

5 Code Options

This chapter describes the options which affect how the Neuron C code is generated once the Neuron C Code Generator has been started. The Neuron C Code Generator provides the ability to configure the mapping information interactively.

Interactively, you can use the Neuron C Code Generator to:

- Define the number of nodes.
- Define the id and name of each node.
- Define, for each object in the Object Hierarchy, the node to which it is to be mapped i.e. which node will host the code corresponding to that object's STDs.
- Edit the mapping file (.nod).
- Generate the source code and include files.

Defining Nodes

Click on the "Edit Nodes" button to start the Define/Edit Nodes dialog to add or delete nodes. This dialog can also be used for changing a node's name or to start the Rename Node Id dialog to rename the node id.

Adding A New Node

Click in the Node Id editbox and type a new name of up to eight characters. The Node Id must be a valid DOS filename. Click in the Node Name editbox and type a descriptive name of up to 49 characters (free format) and then click on the "Add" button.

To use an existing node id and name as the basis for a new node (to minimise typing), click on the corresponding entry in the listbox before clicking in the Node Id editbox.

Deleting A Node

Click on the corresponding entry in the listbox and then click on the "Delete" button.

Changing A Node's Name

Click on the corresponding entry in the listbox and then click in the Node Name editbox. Use the keyboard to edit the name as required and then click on the "Edit" button.

Renaming A Node Id

Click on the corresponding entry in the listbox and then click on the "Rename" button to start the Rename Node Id dialog. Type the new node id and then click on the "Ok" button.

Defining Object To Node Mapping

To specify the node to which an object is to be mapped, either double-click on the object in the Objects listbox or single-click on it and click on the "Edit" button to the right of the Objects listbox. This will start the Select Node dialog.

The Select Node dialog lists the nodes which you have defined. Either double-click on the desired entry or single-click on it and then click on the "Ok" button.

To erase all object to node mappings, click on the "Clear All" button to the right of the Objects listbox.

Note that all object to node mapping must be defined before Neuron C code can be generated.

Edit The Mapping File

Click on the "Edit Config" button to start a simple text editor to edit the mapping file (".nod" file). Before starting the editor, a new file will be created if the configuration has changed (or no mapping file was loaded). In this case, the original will be copied to a file with the same name but extension ".no1".

The editor is very similar to the Windows Notepad text editor. Features such as cut and paste can be used to edit the commented-out information which the Code Generator writes to the mapping file, thus minimising the risk of introducing typing errors.

On exiting the editor, the Code Generator re-populates the listboxes from the mapping file.

Generate The Source Code And Include Files

Clicking on the "Generate" button will cause the Code Generator to validate the mapping. If errors are found, such as objects not mapped to a node, you will be notified via a message box. If the mapping is valid, the Code Generator will create the source code and include files described in section 3, Basic Concepts.

The Code Generator also generates a file with extension ".llm" (LonWorks Live Monitoring). This file can be used by the Synect STD Monitor in conjunction with Echelon's DDE Server to monitor the live control system. See the STD Monitor user guide, chapter 3, Basic Concepts, Connecting To The Live Control System for details.

Synect

Allen Bradley PLC5 Ladder Logic Code Generator User Guide

Version 2.2

**Hopkinson Computing Limited
29 Deepdale, Pine Hills, Guisborough
Cleveland, TS14 8JY
England**

**Tel/Fax: +44 (0) 1287 638606
email: synect@hopkinsn.demon.co.uk
WWW: <http://www.hopkinsn.demon.co.uk>**

© Copyright 1994, 1995, 1996, 1997 Hopkinson Computing Limited. All rights reserved.

Synect is a registered trademark of Hopkinson Computing Limited
Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Information in this User Guide is subject to change without notice and does not represent a commitment on the part of Hopkinson Computing Limited.

The software described in this User Guide is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. No part of this User Guide may be reproduced or transmitted in any form or by any means, electronic or otherwise, including photocopying and recording, for any purpose, without the express written permission of Hopkinson Computing Limited.

Document History

10 November 1996

Re-issue reflecting Code Generator V2.0 - complete re-write to use configuration file and user-specified register for the state variable rather than a bit per Petri net place.

31 January 1997

Re-issue reflecting Code Generator V2.1 - add breakpoint and pause functionality.

1 October 1997

Re-issue reflecting Code Generator V2.2 - comments written in format appropriate for RSLogix5 programming software.

Contents

Chapter 1	Introducing Synect	1
	User Interface	1
	The Method	1
	Synect Documentation	1
	The Tools	2
	Document Conventions	2
Chapter 2	Getting Started	3
	System Requirements	3
	Installation	3
	Starting the A-B PLC5 Code Generator	4
	On-Line Help	5
Chapter 3	Basic Concepts	7
	How It Works	7
	Configuration File	7
Chapter 4	Open An Application	9
	Opening An Application	9
Chapter 5	Code Options	11
	Edit The Configuration File	11
	Generate The Source Code Files.....	11

This page left intentionally blank

1 Introducing Synect

Synect is a set of software tools which helps the designer of a control system to produce a specification which is clear, precise and free of errors. Synect combines the ease of use of a graphical user interface, with a widely used diagrammatic notation and the power of mathematical modelling.

User Interface

The Synect™ tools run on the Microsoft® Windows™ operating system. As such, you need to know how to use Windows before you use Synect. In particular, you will need to know how to use the mouse to click, double-click and drag. You will also need to know how to interact with menus, dialogs and how to move, resize and close windows. For information, refer to the Microsoft Windows User's Guide.

The Method

Synect uses an object-based method to enable you to model the system you want to control. It is useful to have some knowledge of object-based/object-oriented analysis and design techniques before attempting to use Synect in earnest.

Synect Documentation

Each of the Synect tools has an associated User Guide. This User Guide explains how to use each of the functions available in the Allen-Bradley PLC5 Ladder Logic Code Generator. It also explains why you might want to use the function.

Each application has context-sensitive on-line help. The Allen-Bradley PLC5 Ladder Logic Code Generator on-line help contains a "How Do I?" section, including a "How Do I Use The Ladder Logic Code Generator?" sub-section for first-time users.

A Tutorial is also provided which offers a worked example and shows how each of the tools is used with the example application.

The Tools

The tools which make up the Synect toolset are:

Application Editor	graphical means of defining the application.
Compiler	check the specification for consistency and possible warnings and generate a mathematical model of the application.
Analyzer	ability to check for design errors such as deadlock (where the system "hangs") and unwanted state combinations.
Simulator	provides the ability to interactively "drive" the application or replay past behaviour of the live control system.
STD Monitor	animates the specification (used in conjunction with the Simulator or the live control system).
C Code Generator	generate ANSI-standard C code to implement the application.
Neuron C Generator	generate Neuron C to run on one or more nodes on an Echelon LonWorks network to implement a distributed control solution.
Ladder Logic Generator	generate relay ladder logic to run on a programmable controller.

Document Conventions

The User Guide adopts the following conventions:

<code>application name</code>	text that you type or that you see on the screen.
<code>KEY NAME</code>	keyboard keys, such as ENTER, CTRL or DEL.
<code>Menu Choice</code>	a menu option, such as File Exit denoting choose the Exit command from the File menu.
<i>description</i>	description of a term with a specific meaning.

2 Getting Started

This chapter describes:

- the hardware and software requirements which you need to be able to use Synect.
- how to install the Allen-Bradley Ladder Code Generator.
- how to start the Allen-Bradley Ladder Code Generator.
- the Allen-Bradley Ladder Code Generator window.

System Requirements

The Synect Allen-Bradley Ladder Code Generator requires that you use:

- a 486 (or better) running Windows 3.1.
- VGA monitor in 800 x 600 mode (or higher resolution).
- a mouse or other pointing device (such as a trackball).

Other Synect tools also requires that you have the following:

- 8 MByte RAM.
- very large permanent swap file (recommended size is 20 MByte).
- at least 10 MByte free disk space per application.

Installation

If you haven't installed Synect yet, you'll need to do that first. Check that your computer satisfies the requirements listed above before starting the installation. The first floppy disk contains file `install.txt` which contains any updates to the installation process - you should read this file before installing the software.

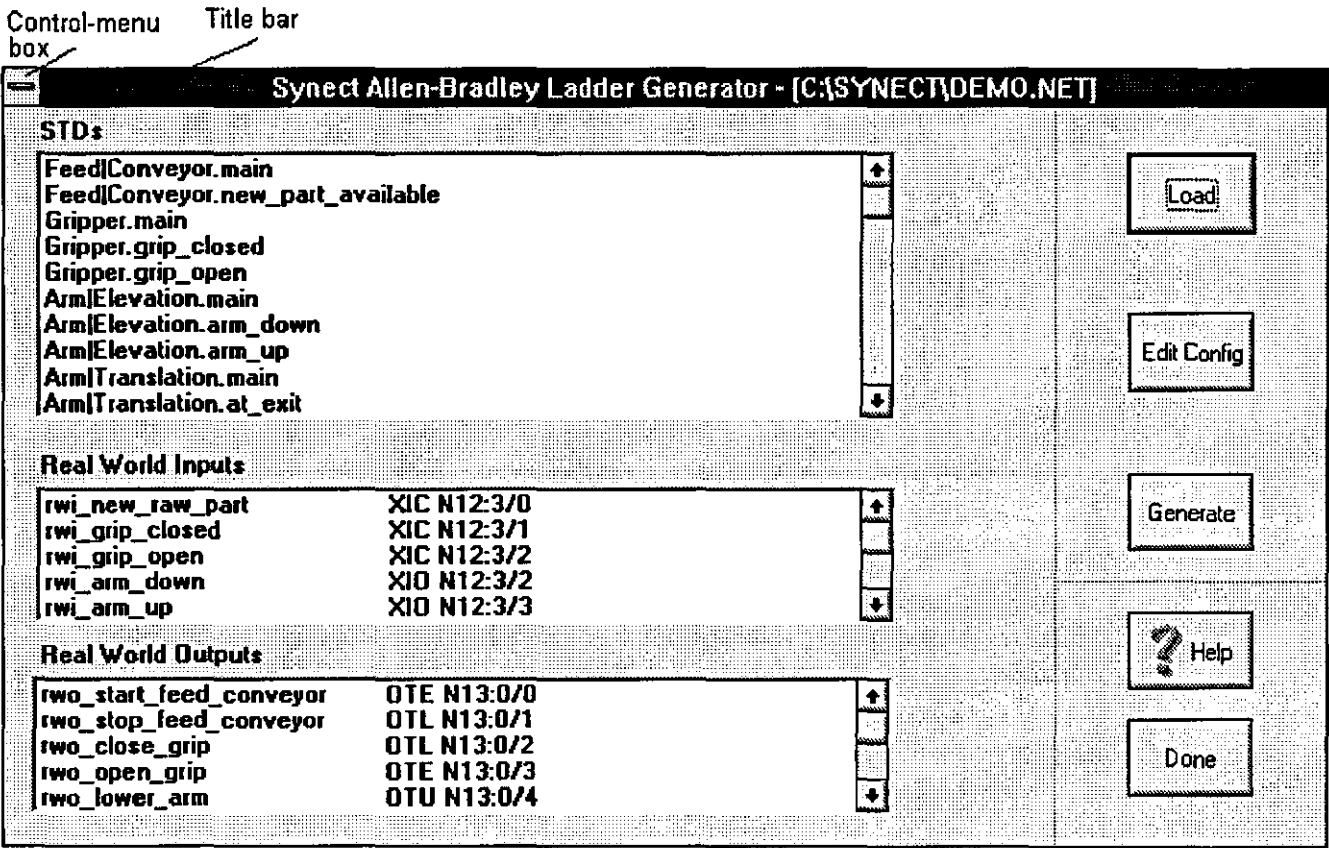
- 1 Put floppy disk 1 into your floppy disk drive.
- 2 Start Windows (by typing `win` at the DOS prompt if necessary).
- 3 Choose `File|Run` from the Program Manager.
- 4 Type `a:\install` then press ENTER.
- 5 The installation program will now guide you through the installation process. Simply answer the questions to specify which Synect tools you want to install.
- 6 When installation is complete, you can remove the floppy disk from the drive. You should now have a Program Manager group containing an icon for each of the installed Synect tools.
- 7 Plug the dongle into the computer's parallel port.

- 8 Read the installed file `readme.txt` for details of any changes to the product or documentation since the documentation was printed.

Starting the A-B PLC5 Code Generator

Ensure that the Program Manager window is on display. If the Synect window isn't visible, use the Window menu to open it. Double-click on the Allen-Bradley PLC5 Code Generator icon or use the keyboard arrow key to select the icon and then press ENTER.

When using the Allen-Bradley PLC5 Code Generator, the window displays the following dialog:



Title bar

The title bar shows the product title `Synect Allen Bradley Ladder Generator` and the name of the application which has been loaded (if any).

Control-menu box

Allows you to move or close the window. Also allows you to open the control panel or obtain information about the Allen-Bradley PLC5 Code Generator.

STDs listbox

Each row of the listbox contains an STD name.

Real World Inputs listbox

Reading from left to right, each row of the listbox contains a real world input name and its corresponding Allen-Bradley PLC5 ladder code. This listbox shows, for each real world input, the Allen-Bradley PLC5 ladder code which will be substituted when the "Generate" button is pressed.

Real World Outputs listbox

Reading from left to right, each row of the listbox contains a real world output name and its corresponding Allen-Bradley PLC5 ladder code. This listbox shows, for each real world output, the Allen-Bradley PLC5 ladder code which will be substituted when the "Generate" button is pressed.

On-Line Help

Click on the "Help" button to take you to the help contents page. The help information is shown in a separate window.

This page left intentionally blank

3 Basic Concepts

The Synect Application Editor enables you to specify a model of your application. See the Application Editor User Guide, chapter 3, Basic Concepts for more information relating to defining the model. The Compiler derives a mathematical model from your specification. The Allen-Bradley PLC5 Ladder Logic Generator generates a ladder logic program from the mathematical model which can then be loaded into Allen-Bradley programming software, such as RSLogix5.

How It Works

Although the Ladder Logic Code Generator appears in the form of a dialog, its behaviour is specified entirely by configuration file. The configuration file defines the filenames to which the ladder logic will be written, the PLC registers to be used as state variables, etc.. It is assumed that the configuration file will be derived automatically from an external source by a project-specific utility (outside the scope of Synect). This will probably a master database from which the PLC database and Human Machine Interface (HMI) database are derived. The Code Generator therefore does not attempt to generate symbol names, address or instruction comments. But it does write rung comments associated with each rung in a form suitable for loading into the RSLogix5 programming software.

Configuration File The file is stored in a file with the same name as the application and extension ".abc" (Allen-Bradley Configuration).

The first two entries in the file define the DDE server and topic from which the STD Monitor can obtain data about the live control system. This is used in the ".plm" file which the Code Generator writes (see the STD Monitor User Guide for more details about monitoring the live control system).

```
DDE_SERVER=<dde_server_name>
DDE_TOPIC=<dde_topic_name>
```

The next entries are concerned with defaults for state values. When the ladder code is generated, each state will be assigned a value. For example, state "Closed" might be value 100 and state "Open" might be value 110. It would be tedious to have to specify the value for each state so the Code Generator provides two methods to make this easier.

The first method is that it can interpret state names. For example, if you always format your states as STATE_10|Open (where the "|" character denotes carriage return), you can tell the Code Generator that the state value is prefixed by "STATE_". In this example, state "Open" would be assigned the value 10.

The second method is to define a default start value and increment for state values. So if you specify the start value as 100 and the increment as 10, and the STD contains states "Open" and "Closed", state "Open" would be assigned value 100 and state "Closed" would be assigned value 110.

This configuration of state values would be achieved with the following three lines in the configuration file:

```
STATE_VALUE_PREFIX=STATE_
DEFAULT_START_STATE_VALUE=100
DEFAULT_STATE_VALUE_INCREMENT=10
```

There then follows a section for each STD. An example is as follows:

```
STD=Feed|Conveyor.main
STATE_REGISTER=N100:44
LADDER_FILE=file2.a5c
```

The first line defines that until another "STD=" entry is found, subsequent specifications refer to STD Feed|Conveyor.main. The second line defines the register which is to hold the current state value for this STD. The third line defines the file into which the ladder logic for this STD is to be written.

Optionally, debug code can be automatically generated for the STD. The debug code allows you to specify any number of registers into which you can manually write the state values at which you wish the STD to pause. The breakpoint bit is the flag which the ladder logic sets to denote that a breakpoint has been encountered. The single step bit is equivalent to specifying a breakpoint at every state. This is specified as follows:

```
SINGLE_STEP_BIT=B3/0
BREAKPOINT_BIT=B3/1
BREAKPOINT_REGISTER=N300:1
BREAKPOINT_REGISTER=N300:2
BREAKPOINT_REGISTER=N300:3
```

Finally, the file contains the definition for each real world input and each real world output. For example:

```
RWI=rwi_new_raw_part,XIO N55:77
RWI=rwi_grip_closed,XIC N44:33
RWO=rwo_start_feed_conveyor,OTL N43:32
RWO=rwo_stop_feed_conveyor,OTE N66:99
```

It is worth noting the following points:

- The format of a line defining the code to be substituted for a real world input is:
RWI=real_world_input_name,code_to_substitute
- The format of a line defining the code to be substituted for a real world output is:
RWO=real_world_output_name,code_to_substitute
- The Allen-Bradley PLC5 Code Generator performs a case-sensitive read of the mapping file and will not ignore spaces unless to the right of a comma.

4 Open An Application

When you first start the Synect Allen-Bradley PLC5 Ladder Logic Code Generator, the "Edit Config" and "Generate" buttons are disabled. This is because you must first open an application.

When you compiled your application using the Synect Compiler, it created a file with the name of your application but with the extension ".net". This is the file which is to be loaded into the Allen-Bradley PLC5 Ladder Logic Code Generator.

Opening An Application

To open an application, click on the "Load" button. The standard file open dialog will then be started, allowing you to specify the name of the file from which the application is to be loaded. By default, the dialog will list files with extension ".net".

If you have previously generated code for this application, a configuration file will already have been created. When you load the application, this file is automatically read and the listboxes populated accordingly. The configuration file is a file whose name corresponds to the application name and with extension ".abc".

If this is the first time you've loaded this application into the Allen-Bradley PLC5 Ladder Logic Code Generator, you'll get a warning message to notify you that the configuration file couldn't be found. Simply click on the "Ok" button and continue as normal.

Having loaded the application, the listboxes and buttons will be enabled.

This page left intentionally blank

5 Code Options

This chapter describes the options which affect how the ladder logic code is generated once the Allen-Bradley PLC5 Ladder Logic Code Generator has been started. All of these options are specified in a text configuration file. This file can be edited with the Allen-Bradley PLC5 Ladder Logic Code Generator running via the simple built-in editor.

Interactively, you can use the Allen-Bradley PLC5 Ladder Logic Code Generator to:

- Edit the configuration file (.abc).
- Generate the source code files.

Edit The Configuration File

Click on the "Edit Config" button to start a simple text editor to edit the configuration file ("abc" file). Before starting the editor, a new file will be created if no configuration file was loaded when the application was loaded.

The editor is very similar to the Windows Notepad text editor. Features such as cut and paste can be used to edit the commented-out information which the Code Generator writes to the configuration file, thus minimising the risk of introducing typing errors.

On exiting the editor, the Code Generator re-populates the listboxes from the mapping file.

Generate The Source Code Files

Clicking on the "Generate" button will cause the Code Generator to validate the configuration. If errors are found, such as no register defined for an STDs state variable, you will be notified via a message box. If the configuration is valid, the Code Generator will create the source code files described in section 3, Basic Concepts.

The Code Generator also generates a file with extension ".plm" (PLC Live Monitoring). This file can be used by the Synect STD Monitor in conjunction with a third party DDE Server to monitor the live control system. See the STD Monitor user guide, chapter 3, Basic Concepts, Connecting To The Live Control System for details.

