Synthesis and Implementation of Procedural Controllers with Alarm Handling Capabilities

by

Ross John BAIRD

November 1999

A thesis submitted for the degree of Doctor of Philosophy of the University of London
and for the Diploma of Membership of the Imperial College

Department of Chemical Engineering and Chemical Technology
Imperial College of Science, Technology and Medicine
London SW7



Abstract

This thesis presents a formal method for the synthesis of provably correct procedural controllers for implementation in sequential operations in flexible manufacturing systems. It is motivated by the need for 100 percent accurate and safe computer control code for the operation of chemical process plants. This need has arisen due to the dramatic increase in the use of computer control within the chemical industry and the tightening of health and safety guidelines as well as operational efficiency targets.

The thesis extends previous work on Procedural Control Theory (PCT) (Sanchez, 1994; Alsop, 1996), which uses finite state machines to formally synthesise provably correct procedural controllers. The major novel contributions of the thesis are:

1. the development and theoretical proof of methods for use of a controllers in a hierarchy, and of multiple initial conditions for controllers,

2. the development of a structured approach for dealing with alarm and emergency handling aspects by way of partitioning a component's behaviour into normal and abnormal operating regions,

3. the development of ways to interface PCT with other emerging formalisms such as statecharts, and

4. the practical implementation of the above methods in a software engineering environment suitable for use by engineers.

Case studies representative of flexible manufacturing systems support the theoretical developments. The first case study was conducted on the Batch Pilot Plant in the Centre for Process Systems Engineering, Imperial College. It involved synthesis, translation and implementation of procedural controllers for both a process simulation and the actual pilot plant. The second, required the synthesis of procedural controllers for a Minerals Processing Screw Conveyor operating under normal and alarm conditions. A benchmarking study of a hypothetical milk pasteurisation plant, involving the verification of the synthesised procedural controllers against a simulation model is also presented plus a discussion regarding the use of the techniques by 33 Masters of Science students.

Acknowledgements

I wish to thank my supervisor, Professor Sandro Macchietto for his support and promotion of my work. I am indebted to Dr. Celeste Colantonio for the hours she has spent reading and rereading the manuscript. Her help is immensely appreciated.

I also wish to thank Dr. Guillermo Rotstein and Dr. Nick Alsop for their guidance in the early stages of the thesis and providing me with a strong foundation to build on. Thank you to Luis Parra-Mora and the 1999 Process Systems Engineering Masters of Science students, I appreciate the time and effort they contributed to provide a test-bed for my work.

My research was made possible thanks to the financial support of the Engineering and Physical Sciences Research Council (EPSRC) of the UK.

To all my friends in the centre with whom I have spent many a time wondering whether this thesis would ever get done, thank you. The fruitless discussions made life much more interesting.

I owe a lot to my friends in the canoeing world, especially those at ULU, in the Dreamers and the Pink Ladies for providing the balance to my life over the last three years and suffering with me when things got tough.

A finally, I want to thank my family, who have always supported and encouraged me in everything I do.

Contents

1 Introduction **13**
1.1 Aims of this Thesis 15
1.2 Thesis Outline 15

2 Literature Review **18**
2.1 Discrete Event Systems 18
2.2 Verification of Control Logic 18
2.2.1 Computer Hazard and Operability Studies 18
2.2.2 Model Checking 19
2.2.3 Combined Discrete/Continuous Verification 20
2.2.4 Control Code Verification 20
2.2.5 Petri Nets 20
2.2.6 Condition/Event Systems 21
2.3 Modelling of Control Logic 21
2.3.1 Structured Modelling of Process Systems 21
2.3.2 Modelling Processes Using Statecharts 21
2.3.3 Modular and Hierarchical Discrete-Event Systems 22
2.3.4 Industrial Implementation of Controllers 23
2.3.5 Automatic Code Generation 24
2.4 Synthesis of Control Logic 25
2.4.1 Supervisory Control Theory 25
2.4.2 Procedural Control Theory 26
2.5 Timed Discrete-Event Systems 27
2.6 Summary and Conclusions 28

3 Procedural Control Theory **32**
3.1 Buffer System Example 32
3.2 Synthesis of Procedural Controllers 33
3.2.1 Finite State Machines 33
3.2.2 Languages and the Asynchronous Product 35
3.2.3 Developing the Model and Specification Constraints 37
3.3 Evolution of Synthesis Methods 39
3.3.1 Controllability 40
3.4 Procedural Controller 42
3.5 Decomposition 42
3.5.1 Internal Consistency 42
3.5.2 Parallel Decomposition 43
3.5.3 Series Decomposition 43
3.5.4 Procedure Initiation and Inhibition Theory 44

4 Statecharts **45**
4.1 States . . . 46
4.2 Transitions . . . 46
4.3 Static Commands . . . 49
4.4 Static Commands and Initial States . . . 50

5 Hierarchical Procedural Controllers **52**
5.1 Introduction . . . 52
5.1.1 Aim . . . 53
5.2 Modular Procedural Control Theory . . . 53
5.2.1 Procedural Transitions . . . 55
5.3 Modelling Procedural Controllers as Procedural Components . . . 56
5.3.1 Consideration of Marked States in Procedural Controllers . . . 56
5.3.1.1 Termination of Procedural Controllers . . . 57
5.3.1.2 Marked States of Differing Significance . . . 57
5.4 Synthesising Higher Level Procedural Controllers . . . 58
5.4.1 Integrating Higher and Lower Level Procedural Controllers . . . 59
5.4.2 Properties of Hierarchical Procedural Controllers . . . 61
5.4.3 Example - CIP Heating Procedure . . . 64
5.5 Interlocking Components and Reusable Controllers . . . 65
5.5.1 Example - CIP Heating . . . 66
5.6 Summary . . . 67

6 Multiple Initial States **68**
6.1 Introduction . . . 68
6.2 Procedural Controllers with Multiple Initial States . . . 68
6.3 A New Approach for Process Modelling . . . 71
6.4 Synthesis of Controllers with Multiple Initial States . . . 71
6.4.1 Example - CIP Heating . . . 72
6.5 Initial States with Covered State Variables . . . 73
6.5.1 Theoretical Validation . . . 76
6.5.2 Implementation Issues . . . 76
6.6 Summary . . . 80

7 Partitioning **82**
7.1 Motivation and Scope . . . 82
7.1.1 Definitions . . . 83
7.1.2 Hierarchical Model Development . . . 84
7.2 Applying Statecharts to PCT . . . 84
7.3 Partitioned Elementary Components . . . 85
7.3.1 Synthesis of Procedural Controllers within Each Superstate . . . 86
7.4 Sub-Phase Coordination and Initiation . . . 90
7.5 Theoretical Specification Decomposition . . . 90
7.6 Summary and Conclusions . . . 92

8 Structured Implementation of Procedural Controllers **94**
8.1 Introduction . . . 94
8.1.1 Aim . . . 94
8.2 Control System Architecture . . . 95
8.2.1 Overview of *gPROMS* . . . 95
8.2.2 Process Models/Items . . . 96

8.2.3 Interpretors 97
8.2.3.1 Process Model Observers 98
8.2.3.2 Controller Tasks 98
8.2.4 Discrete-Event Controller 103
8.3 Summary and Conclusions 104

9 Computational Aspects in the Synthesis of Procedural Controllers 106
9.1 Introduction 106
9.2 Current Synthesis Approaches 107
9.3 State Based versus Language Based 107
9.4 Purely FSMs versus Elementary Components and Temporal Logic 108
9.5 Separate Model and Specification Synthesis versus Combined Synthesis . 109
9.6 Multiple and Covered Initial States 110
9.7 Synthesis Methods: Level versus Path 110
9.8 Synthesis Algorithm 111
9.8.1 Forced Transition Sequence 111
9.8.2 Never Transition List 112
9.8.3 Next Transition List 112
9.9 Summary and Conclusions 112

10 Guided Specification of Procedural Controllers 115
10.1 Introduction 115
10.2 Sequence Developer 116
10.2.1 Guided Modelling and Operation Specification 118
10.2.1.1 Component Definition and Instantiation 118
10.2.1.2 Component Libraries 118
10.2.1.3 Process Modelling 120
10.2.1.4 Operating Specification 121
10.2.2 Procedural Controller Synthesis 123
10.3 Results Display 123
10.3.1 Viewing Graphically 124
10.3.2 Structured Text Documentation 124
10.4 Automatic Control Code Generation 125
10.4.1 Generation of Process Interpretors 126
10.4.2 Procedural Controller Translation 127
10.4.2.1 Inhibits 127
10.4.2.2 Pre-Checks 127
10.4.3 Generation of General Logic 128
10.5 Summary and Conclusions 130

11 Case Studies 134
11.1 Introduction 134
11.2 Case Study 1 - The Batch Pilot Plant 135
11.2.1 Overview of the Batch Pilot Plant 135
11.2.1.1 Batch Pilot Plant Computer Control System 136
11.2.2 Clean-In-Place of Tank T1 136
11.2.2.1 Operating Procedure Overview 136
11.2.2.2 Equipment Overview 138
11.2.3 Procedural Controller Synthesis 138
11.2.3.1 Clean-T1 142

11.2.3.1.1 Water Rinse Operation 142
11.2.3.1.2 Detergent Rinse Operation. 149
11.2.3.1.3 Detergent Service Unit Operation. 149
11.2.4 Translation to *gPROMS* and PARACODE 154
11.2.5 Implementation in gPROMS 154
11.2.5.1 Water-Rinse 154
11.2.5.2 Detergent Service Unit Operation 157
11.2.5.3 Detergent Rinse Unit Operation 157
11.2.6 Implementation on the Batch Pilot Plant 158
11.2.6.1 Water Rinse Unit Operation 160
11.2.6.2 Detergent Service Unit Operation 161
11.2.6.3 Detergent Rinse Unit Operation 161
11.2.6.4 Summary of Case Study 1 162
11.3 Case study II - Mineral Processing Screw Conveyor 165
11.4 Case Study III - Benchmarking with an Industrial Example 171
11.5 Using Sequence Developer in a Teaching Role 171
11.6 Conclusions from the Case Studies 174
11.6.1 Quantitative Dynamics 174
11.6.2 Uncontrollable Transitions and Controller Size 175
11.6.3 Communication between procedural controllers 175
11.6.4 Feedback Failure and Soft Sensors 176
11.6.5 Integrated Controller and Process Design 178
11.6.6 Forcing Sequences under Partitioning 179

12 Conclusions and Future Work **180**
12.1 Significance 180
12.2 Summary 181
12.3 Future Directions 184
12.3.1 Grace Times 184
12.3.2 Shared Controllable Transitions 184
12.3.3 Non-Fatal Failure Procedures 184
12.3.4 Further Hierarchical Architecture Development 185
12.3.5 Further Development of the CAD Tool 185

A Sample Sequence Developer Input File **186**

B Sample *gPROMS* File **191**

C Sample PARACODE Source File **195**

References 198

List of Figures

2.1 How formal synthesis techniques produce a controller. 29

3.1 Buffer Schematic 33
3.2 Primitive Component FSMs for Motivating Example 34

4.1 Statechart States 46
4.2 State Transitions 47
4.3 Transition economy in Statecharts 47
4.4 Active State Selection 48
4.5 State Transitions 49
4.6 State Transitions 50
4.7 State Transitions 51

5.1 ISA-S88 Finite State Machine representing a sequence controller. 56
5.2 Augmented ISA-S88 Procedural Component FSM. 57
5.3 Augmented ISA-S88 Procedural Controller FSM with multiple completion states. 58
5.4 Batch Pilot Plant CIP Unit Schematic Diagram. 65
5.5 Hierarchical controller structure of the Batch Pilot Plant Detergent Service 67

6.1 Push Button Failure Procedural Controller 69
6.2 Using connectors as initial states 69
6.3 Synthesising Controllers containing Multiple Initial States 73
6.4 CIP Heating Controller with Multiple Initial States 74
6.5 How Partial State Knowledge helps to reduce procedural controller size. . 77
6.6 CIP Heating Controller with Partial-Knowledge Initial States 78
6.7 The implementable CIP Heating Controller with covered Initial States but no compression after an uncontrollable transition has occurred. 79
6.8 CIP Heating Controller with Partitioning and Multiple Initial States . . . 81

7.1 Statecharts of Common Process Structure 85
7.2 Examples of Partitioned Components 88
7.3 An example of a higher level procedural controller. 91

8.1 Modelling Architecture 96
8.2 Mapping between an uncontrolled process item, discrete physical model, intelligent discrete model and a PARACODE observer for a digital valve. 99
8.3 Mapping between a process model, discrete physical model, intelligent discrete model and a *gPROMS* observer for a digital valve. 100
8.4 Mapping between a process model, discrete physical model, intelligent discrete model and an observer for a continuous level sensor. 101

8.5 Mapping between a process model, discrete physical model, intelligent discrete model and an observer for a continuous level sensor. 102
8.6 PARACODE and *gPROMS* code for the controller task to open a valve. . 103
8.7 PARACODE and *gPROMS* code for a discret-event controller. 104

9.1 How *TCT* and *PLATEST* use the state space 110

10.1 Main Window of the *Sequence Developer* GUI. 116
10.2 How *Sequence Developer* interacts with the user, *Sequence Solver* and *DaVinci*.117
10.3 Sequence Developer's Features Hierarchy. 117
10.4 Sequence Developer's Component Definition. 119
10.5 Sequence Developer's Object Oriented Hierarchy. 119
10.6 Current Component Instances are displayed in Sequence Developer's Component Instances Window. 120
10.7 Load and saving Components using Sequence Developer's Component Library. 120
10.8 Defining Static Constraints using Sequence Developer. 122
10.9 Defining Dynamic Constraints using Sequence Developer. 122
10.10Results display and feedback in *Sequence Developer* 123
10.11Viewing *Sequence Developer*'s synthesis log. 124
10.12*Sequence Developer*'s PARACODE translation input window. 126
10.13An example of an automatically generated PARACODE Observer. 128
10.14Automatically generated procedural controller pre-checks 131
10.15Automatically generated procedural controller logic 133

11.1 The Batch Pilot Plant 137
11.2 Schematic of CIP and Tank T1 139
11.3 Batch Pilot Plant Elementary Components 141
11.4 Clean-T1 Sequence Hierarchy 143
11.5 Level profile and valve positions for *Water-Rinse* procedure simulation in *gPROMS*. 156
11.6 CIP Level, Concentration and Temperature profiles with valve and pump states for *Detergent-Service* procedure simulation in *gPROMS*. 157
11.7 CIP and T1 Level profiles with valve and pump states for *Detergent-Rinse* procedure simulation in *gPROMS*. 158
11.8 CIP and T1 Level profiles with valve and pump positions for the *Clean-T1* Operation simulation in *gPROMS* 159
11.9 Level profile and valve positions for *Water-Rinse* procedure operation on the Batch Pilot Plant. 160
11.10Temperature profile and valve positions for the *Detergent-Service* operation on the Batch Pilot Plant. 161
11.11CIP and T1 Level profiles with valve and pump states for *Detergent-Rinse* procedure operation on the Batch Pilot Plant. 162
11.12CIP and T1 Level profiles with valve and pump positions for the *Clean-T1* Operation on the Batch Pilot Plant 163
11.13Schematic of Screw Conveyor. 166
11.14Screw Conveyor Elementary Components 169
11.15PLANT X 172
11.16Tank 100 and the trucks' volume profiles for the Raw Milk Intake Procedure.173
11.17Lattice structure representing the operating specification for a failure monitoring procedure. 176

11.18CIP and T1 Level profiles with valve and pump states for *Detergent-Rinse* procedure operation on the Batch Pilot Plant. 177
11.19Proposed procedural component enabling easier communication between controllers. 178

List of Tables

6.1 CIP Heating example synthesis statistics for each single initial state and a multiple initial state controller. 75
6.2 Comparison of controller sizes of four single initial state controllers with a single multiple initial state controller. 75

7.1 CIP Heating example synthesis statistics for each controller created by partitioning or a single controller. 89

9.1 Summary of approaches to the aspects of controller synthesis 108

11.1 Summary of Techniques Demonstrated in the Case Studies 135
11.2 Summary of Instances used in the Batch Pilot Plant Case Study. 140
11.3 Models and Controller Synthesis Statistics for Cleaning T1. 153
11.4 Generated Code Statistics for Cleaning T1. 155
11.5 Partitioned models of the Screw Conveyor. 170

Nomenclature

C	Controller FSM
C_i	Controller FSM for subsystem i
H_i	The ith series process model
$L(M)$	Language generated by FSM M
$L_m(M)$	Marked language generated by FSM M
$L(C/M)$	Closed loop language generated by controller C and process M
$L(C_i \uparrow C_j/M)$	Closed loop language generated by parallel controllers C_i and C_j
$L(C_i \rightarrow C_j/M)$	Closed loop language generated by series controllers C_i and C_j
$L(C_1 \twoheadrightarrow C_2/M)$	Closed loop language generated by series controllers C_i and C_j with differing alphabets.
$L(C_i \triangle C_j/M)$	Closed loop language generated by hierarchical controllers C_i and C_j
M	Process model FSM
Q	Set of states
Q_0	Set of nominal initial states
Q_m	Set of marked states
q_i	The ith initial state
S	Specification FSM
S_i	Specification for subsystem i
V^{n_v}	Set of state variables
$(v_j)_q$	State variable j in state q
X	Set of controller states
X_m	Set of marked controller states
X_0	Set of controller's initial states
x_i	The ith initial controller state
X_w	Set of controller wait states

Greek Symbols

γ	State variable transition partial function, $\gamma : \Sigma \times V^{n_v} \rightarrow V^{n_v}$
δ	State transition partial function, $\delta : \Sigma \times Q \rightarrow Q$
ϵ	The null event
ξ	Controller state transition partial function, $\xi : \Sigma \times X \rightarrow X$
Σ	Set of transitions or alphabet of process events
Σ^*	The string set
Σ_i	The alphabet of subsystem i
Σ_{ij}	Set of controllable events in Σ_j equivalent to events in Σ_i
τ	The next transition

Chapter 1

Introduction

In recent years, the chemical process industries have been moving towards the manufacturing of low volume, high value-added products, targeting customers with very specific needs. This trend has created new challenges and flexible manufacturing plants are becoming widespread. These often semi-continuous or batch plants are able to produce a very large and changing number of products via a variety of plant configurations and operating specifications.

Batch processes rely heavily on regular switching between operations. Thus, operating procedures for normal startup and shutdown, abnormal operation, emergencies and interlocking are in abundance. Constant switching and the fact that a process is very rarely in a steady-state introduces a much higher risk of hazardous situations. It is claimed that up to 63% of batch plant shutdowns are caused by operational errors while for continuous plants the corresponding statistic is more in the region of 30-40% (Nimmo, 1995). A large percentage of these shutdowns were caused by incorrect operating procedures. To reduce the risk of these errors, operating procedures must be clearly and carefully defined (Bridges and Williams, 1997).

Plants are most at risk of an accident during change. To switch between products quickly, efficiently and safely, process computers have been introduced to automate procedures and have been shown to result in significant economic benefits (Nimmo, 1993; Lemetais, 1999). More importantly, their use for repetitive and tedious tasks contributes to the overall increase in safety of operations. Additional long term benefits include the ability to log information for analysis and the critical incident recording for diagnosis and evaluation of accidents (Edwards and Clark, 1999).

The process industry is now facing two often contradicting challenges, economic efficiency and safety. On one hand, as budgets tighten and customer demands become more specific, project managers are required to streamline design and commissioning stages. On the other hand, safety requirements also have become far stricter in recent years, requiring engineers to consider many new factors and to assure regulatory agencies that the automation code is *safe*. Safety Critical Systems such as those found in the chemical process industry, can have very dangerous consequences when an error occurs. While the

introduction of computers has reduced the likelihood of human errors to occur, it has also introduced a vast range of new potential hazards. Potential errors range from hardware failure and sensors sending invalid data to errors in the control code itself. This, coupled with the ongoing problems associated with incorrect operating specifications, provides a large potential for costly and possibly fatal errors to occur.

While the process hardware undergoes strict design and safety analysis before installation and operation, process control systems have not experienced the same scrutiny. Systematic Hazard and Operability studies (HAZOPs) nowadays are common place in the design environment of the process engineer but are not regularly performed in the domain of the control engineer. To alleviate the problem, the use of Computer HAZOP studies (CHAZOPs) have been proposed (Nimmo, 1994). However, quite often safety verification of operating procedures and process control remains an afterthought in process design and safety.

With the use of computer controllers widespread, the process industries are not alone in the dilemma of producing safe and correct control code. For example 90% of all baggage was misdirected in the initial trials of Denver Airport's computerised baggage handling system in 1996. The computer control community is now turning to formal methods to verify or synthesise computer sequences for operating procedures. This thesis focuses on one such method that is based around finite state machines (FSM). They form the basis for Procedural Control Theory (PCT) (Sanchez, 1994) which relies on the principles that with knowledge of the behaviour of each elementary component of a system, a model can be built of the overall process. To this, operating specifications can be applied and assessments made as to their feasibility and safety.

Extensions to the PCT framework were made to apply it to large scale processes (Alsop, 1996) under normal operation. However, a large proportion of operating procedures refer to actions to be taken when failures occur. Indeed, large economical gains can be made if a process is operated correctly when in an abnormal state (Nimmo, 1995). Correct operation in such circumstances is clearly important with regard to human and environmental safety.

In addition to the core theoretical developments that are required to provide a substantial framework for procedural controller synthesis, the needs of industry must be considered. For any technology to be successfully accepted it must show a clear path from theoretical development to actual implementation. While process control vendors have developed programming techniques and structures to aid the designer and translation algorithms have evolved for DESs, a gap still lies between the synthesis of a procedural controller and its seamless implementation on a process.

Taking this one step further, the design, synthesis, translation and implementation of operation procedures must appear clear and simple to the final end user. That is, a design engineer with limited exposure to the theory of PCT and DESs. For a technology such as PCT to be accepted outside academia (and indeed within) a user friendly method of using all tools is essential. Wren (1998) states that the man machine interface (MMI) and

presentation to decision makers holds the key to a project's success and not necessarily the theoretical frontiers it may have forged.

1.1 Aims of this Thesis

This thesis follows the early developments of PCT and aims to evolve it into a complete package that combines theoretical developments with implementation issues to gain industry acceptance. It does so in two areas: theoretically and practically.

Theoretically, it aims to further develop current theory to ease the synthesis of large systems and in particular the ability to handle abnormal behaviour. Building on previous decomposition theory, the development of hierarchical theory is needed to help implement the previous work. When considering abnormal behaviour, past methods have encountered problems of size. Here, the treatment of alarms seeks to recognise, as it is done in industry, the unique aspects that are associated with failures. This is in the sense that, under the influence of a failure, a system's behaviour is restricted to a subset of the possible behaviour before the failure occurred. A system's behaviour can be partitioned into normal and abnormal behaviour and operating procedures synthesised for each of them accordingly.

On the practical side, the recognition that industrial control languages are highly structured provides the key to the implementation of procedural controllers and similar synthesised controllers on real plants. The development of a methodology that ideally can be automated and provides a systematic translation from FSM into executable control code is investigated. It aims to exploit the structured properties of industrial sequential control software while maintaining a clear mapping to the procedural controllers and elementary components it is implementing.

In addition, recognising the importance of the needs of the target end user of this technology, the design of suitable implementation software is undertaken. The goals are to fully support the methodology in a practical environment, to provide an attractive interface for the user, and to increase the processing capabilities and thus the diversity of procedural controllers which can be quickly synthesised, verified and implemented.

1.2 Thesis Outline

To achieve the objectives discussed above, the thesis is divided into three areas: previous work and background theory, theoretical developments and implementation advances.

Chapter 2 provides a review of existing literature regarding sequential operating procedures. It reviews a variety of approaches including HAZOP analysis, verification techniques such as Model Checking, Condition/Event Systems and Petri Nets. The synthesis techniques of Supervisory Control Theory and Procedural Control Theory are discussed as well as approaches to include time, modularity and hierarchy in system models. It is decided that Procedural Control Theory provides an opportunity to meet the objectives of

the thesis through its approach to build complex models from simple elementary models. The Statechart modelling formalism is selected as a possible method for implementing a hierarchical structure within the PCT framework. Implementation methods are reviewed with regard to assessing what is required of a software package for this area.

An introduction to the basic theory of PCT is given in Chapter 3. More detailed descriptions are available elsewhere (Alsop, 1996; Sanchez *et al.*, 1999) however, this summary outlines the key details required for this thesis. Likewise, Chapter 4 reviews Statecharts, introducing the main concepts that are required for the remainder of the thesis. A more detailed report can be found in Harel (1987).

Chapter 5 begins the theoretical contribution of this work. The hierarchical implementation of procedural controllers is formalised. First, a discussion of how procedural controllers are represented in the hierarchy is presented with an aim to remain as generic as possible to facilitate the widest possible application. It is shown how procedural controllers can be represented as elementary components in higher level procedural controllers. The reuse of procedural controllers is discussed with rules and tests outlined. The chapter is concluded with the theoretical proof of the methods presented.

Theoretical development continues in Chapter 6 with the introduction of the concept of multiple initial states. The chapter investigates the proposal that many process systems can start from a variety of initial situations highlighting the importance of a procedural controller starting from the correct initial state. A new approach to modelling DESs is suggested where the initial states become part of the specification. Theoretical proof with an illustrative example are given.

Using the developments of the two previous chapters, Chapter 7 tackles the problem of synthesising procedural controllers that contain alarm handling capabilities. It is motivated by the industrial requirement for handling normal and abnormal behaviour of a process. Statecharts are introduced to the PCT framework as a way of modelling the process behaviour. Using the hierarchical theory from Chapter 5, Partitioning Theory is developed to consider elementary components decomposed into normal and abnormal behaviour. Then, the theory of Chapter 6 is incorporated to obtain the full theoretical machinery required.

The practical developments of this work are presented beginning at Chapter 8. It builds on previous attempts to provide a structured implementation architecture to facilitate the translation and implementation of procedural controllers. The chapter outlines how procedural controllers are synthesised in the discrete domain but implemented in a combined discrete/continuous domain. This requires a three-tiered architecture which has an additional interpretation level in between the two domains. The chapter emphasises the importance that simulation or implementation code that is produced should maintain a clear mapping to the models and controllers it implements.

In chapter 9, the development of C++ software to implement PCT is outlined. The chapter begins with a brief critical review of previous packages and an analysis of what is

required of a new package. In particular the chapter presents the algorithms required to incorporate the use of sequencing and multiple initial states within the solution software.

The development of a graphical user interface (GUI) is discussed in chapter 10. The need of a user friendly package to gain acceptance among industry is addressed here. The GUI, developed using JAVA, incorporates a high degree of error checking and requires a limited amount of user interaction to efficiently complete the control design task. Features of the program are discussed in the chapter, including guided process modelling and specification, component libraries as well as synthesis feedback logs and a variety of results formats.

The developments of the previous six chapters are brought together in chapter 11 where two significant case studies are presented. The first one uses the Imperial College Batch Pilot Plant to demonstrate the application of this work to a plant typical of a pharmaceutical, fine chemicals or food manufacturing plant. Procedural Controllers are synthesised for the Cleaning In Place operation of a feed preparation tank, one of the most complex in the operation of the plant, and an operation indicative of the complexity of most operations. Results are presented which show that the synthesis task is achieved easily. The second uses a minerals processing plant to illustrate some complex specifications and to provide a comparison with other approaches to the synthesis and verification of discrete event systems. The chapter also discusses the use of the theory and software by 33 students as part of a Masters of Science project.

The thesis concludes in chapter 12 with suggestions regarding the future direction of the research and the development of the software package.

Chapter 2

Literature Review

2.1 Discrete Event Systems

Research on development of controllers for Discrete Event Systems (DESs) has focused on the verification of operating specifications against process models, the efficient modelling of operating specifications, and the synthesis of provably correct controllers based on knowledge of both the model and specification. This literature review concentrates mainly in the area of state machines and their use in synthesis and verification of the control logic. It begins with a review of verification techniques in section 2.2. Modelling techniques are then discussed in section 2.3 including structural methods such as statecharts, industrial control specification methods and standard engineering practice currently used. It is followed by a review of synthesis methods in section 2.4. Specifically, methods based on Supervisory Control Theory and Procedural Control Theory are discussed. Section 2.6 concludes the chapter.

2.2 Verification of Control Logic

Verification determines whether a given operating procedure or control logic satisfies a set of specifications when applied to control a given process. In other words, it seeks to establish whether the controlled process behaves consistently with the operating specifications.

2.2.1 Computer Hazard and Operability Studies

The call for HAZOP studies to include computer controllers associated with a process has gained momentum recently (Nimmo, 1994). Schubach (1997) calls for an inclusion of computers in standard HAZOP procedures. It presents an evolution of HAZOP procedures which applies the techniques to the whole control system considering the sensor, actuators, system hardware plus the sequential logic. Chung *et al.* (1998) are in agreement with this and based on a survey of 300 incidents or near misses in the process industries have compiled a *Life Cycle Question Library* which endeavours to force the designers to

consider all situations associated with computer controlled plants. This work forms part of an overall verification approach (Yang and Chung, 1998) which describes verification as a 4 step process. *Process Representation*, in this case by way of Process Control Event Diagrams (PCEDs), defines the process in the discrete domain. Verification is broken into two steps. First verification of the ideal control logic is done when no items are considered to fail. Then each item is systematically analysed regarding its effect on the process should it fail. Both verification steps involve calculations based on the continuous dynamic properties of the process (such as temperature changes). Finally, life cycle questions of their previous work are applied. Recognising the complexity of large systems and the enormity of manually performing the assessment, software is available to help carry out the tasks.

2.2.2 Model Checking

Moon *et al.* (1992) applied verification techniques to process systems by using the model checking approach proposed by Burch *et al.* (1992). It is a temporal logic based methodology which requires a *model* of the process (in Finite State Machine (FSM) form) being tested and *assertions* about the behaviour of the system with respect to safety and operability (Moon and Macchietto, 1993). The model checker assesses whether the system satisfies each assertion, providing a counterexample if an error exists. Verification of concurrent operating procedures where the asynchronous product of the procedures(modelled as FSMs) was taken and assertions tested (Moon, 1994) (Marikar *et al.*, 1998). Again, the complete model needed to be constructed. All these approaches rely on a flat FSM modelling structure which encounters state explosion difficulties when dealing with large systems. Probst and Powers (1994) used hierarchical and modular FSMs to model larger processes incorporating failures and then verified them using model checking. A realistically sized screw conveyor transport problem was verified using symbolic model checking (Probst *et al.*, 1997) which bases itself on the idea that formulae can replace sets of states and so reduce the size and complexity of process model being verified. The formulae are represented in Boolean form and are used for verification as part of *ordered binary decision diagrams*. The specifications are written in *computational-tree logic* (CTL) which comprises standard and temporal logic expressions. A software tool, SMV (Rausch and Krogh, 1998), performs verification as to whether the specification in CTL is satisfied (true) with respect the model. However, it was shown that the use of binary logic diagrams can lead to incorrect verification through hazards and races (Park and Barton, 1996) owing to the fact that logic diagrams assume parallel processing yet most computer systems only achieve a quasi-parallel state of processing. A similar model checking approach was applied to a large scale chemical process problem (Park and Barton, 1997) in which state explosion was avoided by modelling the process implicitly as a system of Boolean equations with temporal logic representing the specifications. These were then converted to an MILP feasibility problem on which branch and bound techniques were

used to solve the problem.

2.2.3 Combined Discrete/Continuous Verification

Recently, there have been some efforts towards the verification of hybrid systems characterised by a mixture of continuous variables and discrete events. Klein and Wehlan (1996) proposed the use of Boolean Difference Calculus where Boolean Algebra is used to describe the dynamics of DESs. MILP methods were also applied to hybrid continuous and discrete systems by Dimitriadis *et al.* (1996) to verify safety of a system. The avoidance of forbidden or dangerous states was considered but no consideration was made regarding achieving goal states. Verification of discrete controllers on combined discrete/continuous process simulation was done with a view to implementation on real processes.

2.2.4 Control Code Verification

A large effort has focussed on the analysis and verification of implementable control code too. Applications in railway signalling (Gibbs, 1994) and aircraft guidance systems have exploited languages such *B* (Lano *et al.*, 1998) and *MALPAS* (Palmer, 1995) to assure the control code is safe and executable.

2.2.5 Petri Nets

A popular state machine based formalism used for verification of control logic (Luder *et al.*, 1996) is the Petri Net framework. Within the Petri Net framework, models of process components are first developed and then paths are verified to assure safe operation. The models contain three basic elements: *places, transitions* and *arcs. Places*, signify elementary component states. They are linked by *transitions* which occur when a component changes state. To model component interaction *Arcs* between the elementary components enable and disable the transitions.

Like most state machine based methodologies, Petri Nets encounters the problem of state explosion when modelling large processes. Two approaches within Petri Nets have reduced the size of this problem. By constructing them without aggregating the elementary components (Rausch *et al.*, 1996) or basing them on the specifications (Yamalidou *et al.*, 1990), a large degree of decomposition is retained. Alternatively, the system is decomposed into subsystems and controllers implemented for each of these individually (Feldmann *et al.*, 1996).

A strength of Petri Nets is the allowance of concurrent operations. However, this provides problems in implementation when two specifications compete for a device. Feldman *et al.* (1996) countered this by decomposing the system into subsystems and implementing complex controllers in areas of conflict. This also alleviated risks of state explosion.

A modelling language, *GRAFCET* (Årzén, 1994), based on Petri Net theory implements controllers as supervisory control systems.

2.2.6 Condition/Event Systems

Condition/Event (Sreenivas and Krogh, 1991) systems are somewhat a combination of the concepts of model checking and Petri nets. With an aim to build systems from smaller subsystems, the importance of this technique is in the way it deals with how these subsystems interact. Other approaches are based on either conditioning transitions or synchronising of events. CESs recognise that a process consists of events (be they issued by the control systems or sensed on the process) and conditions associated with the occurrence of these events. With the knowledge of a systems conditions and events, a model and controller can be synthesised. A system is defined in modules and interaction is by way of the conditions and events. For example, a condition signal in one module can disable or enable state transitions in another module.

General Condition/Event systems model processes exhibiting both discrete and continuous behaviour and thus several subclasses have been developed. These include Discrete CES (Kowalewski *et al.*, 1998), Boolean CES (Pinzon *et al.*, 1997) and Net CES (Pinzon and Hanisch, 1998). CES and Discrete CESs are modelled block diagrams and signal flows conforming with general practice in systems theory. Net CESs use bounded Petri Nets to represent the dynamic behaviour of a system.

2.3 Modelling of Control Logic

2.3.1 Structured Modelling of Process Systems

The division of a process state space into three regions has been recognised in the past (Klein and Wehlan, 1996; Wonham, 1996). These regions, the *'normal operating range'*, the *'admissible error range'*, and the *'non-admissible error range'* (Klein and Wehlan, 1996), are determined by the operating specifications. The aim of procedural controllers is to control a process within the *'normal operating range'* attempting to avoid the *'admissible error range'*, while never allowing it to enter the *non-admissible error range'*.

As discussed in later sections, current synthesis methods acknowledge that this structure exists within a process. However, no method exploits the structure explicitly during the model building process. Alternatively, methods based on statecharts enable the structural modelling of a process but no means for formal synthesis are provided.

2.3.2 Modelling Processes Using Statecharts

Statecharts (Harel, 1987), provide a means to capture hierarchy and modularity and introduce the benefits of a formal representation for concurrent behaviour (Huizing *et al.*, 1988). Statecharts can be summarised by the following concept (Harel, 1987; von der Beeck, 1994; Huizing and de Roever, 1991).

$$Statecharts = FSM + depth + orthogonality + broadcast\_communication \quad (2.1)$$

A brief overview of statecharts and the properties that are of particular interest to this work is given in Appendix 4. For a detailed introduction to statecharts consult (Harel, 1987) for the original models or their more recent review (Harel and Naamad, 1996).

Since their inception, many variants of statecharts have emerged. A discussion of their similarities and differences is presented by von der Beeck (1994). All variations keep the four main properties of statecharts: state hierarchy, conditional transitions, static commands and multiple initial states. Hierarchical states allow the existence of *substates* of other *superstates*. States are classified into OR or AND states. Within an OR-state, if the system is in that state then it is in only one of the substates at a time. Within an AND-state, the system is simultaneously in each of the substates. This introduces concurrency. Linking these states are transitions that can have enabling conditions and commands. Transitions can also be contained within a state. These are renamed static commands. Every time a state is entered the static commands are executed. To cope with hierarchical states, multiple initial states are allowed. *Connectors, history* and *default* states are used to determine the active initial state.

In a non-trivial application to an air traffic collision avoidance system, Leveson *et al.* (1994) noted that two or three levels of state hierarchy were sufficient. Graphical representation was simplified with features such as transition tables.

Most DES theory assumes that transitions are instantaneous. Therefore, it is highly unlikely that two transitions occur simultaneously (Probst *et al.*, 1997). However, implementation has shown that while computer related transitions satisfy this assumption, transitions involving process plant equipment take a finite time to complete (Baird, 1997). The alternative proposed is to use a grace time but this makes the controller blind to uncontrollable events for a finite time, which could also have costly consequences. The hierarchy of statecharts enables the modelling of transitions that take a finite length of time and monitoring of uncontrollable transitions simultaneously. Transition priorities based on the originating state (Pnueli and Shalev, 1991) determine which transition is executed.

2.3.3 Modular and Hierarchical Discrete-Event Systems

As the problem of state explosion arose in the DES community, suggested methods for combating it were developed. The first ideas were to break the process model into concurrently operating modular processes (Wonham, 1996) or sequential decomposition (Alsop, 1996) and the development of cascaded models (Sreenivas and Krogh, 1991). These methods showed that several modular controllers could be operated in parallel or sequence.

As statecharts gained in popularity as a methods for specification, formal methods

were developed which embraced the concepts of hierarchical state machines. A variant of statecharts, asynchronous hierarchical state machines (AHSMs), introduced by Brave and Heymann (1993), did not allow synchronisation between parallel components by forbidding shared events. Reachability analysis tools were developed for AHSMs, showing that they are able to retain the properties of the FSMs they replace, while exponentially reducing the required computational time.

Hierarchical Multilevel State (HMS) machines (Gabrielian and Franklin, 1991) present an amalgamation of several different approaches to produce what is claimed to be a simple specification language with graphical representation which is verifiable and easily translated into control code. In a similar way to Petri Net theory, elementary components are modelled independently. Transitions were allowed to execute concurrently and several states could be active at any time. Applications focused on PLC implementation (Gabrielian, 1994) which is quite understandable as the nature of transition execution shares many similarities with Ladder Logic. HMS machines used a "tableau-based theorem proving method" to verify safety of a process (Gabrielian and Iyer, 1991). However, this did not consider whether goal-states were achieved. A model checking approach was incorporated using temporal interval logic (TIL) to expand the HMS into single level state machines. Rather than enumerate the whole model, a set of models H_i were developed such that H_1 was independent of all other machines and the behaviour of H_i for $i > 1$ was only dependent on $H_1, .., H_{i-1}$.

Hierarchical systems were developed within SCT (Hubbard and Caines, 1998). An approach was taken whereby a low level process model was synthesised and the hierarchy applied with the knowledge of the behaviour of the system and based on some rules. This made it very process model and specification dependent and also did not address the question of state explosion at the lower level. The concept of *vocalisation* (Wong and Wonham, 1996*a*) followed a similar approach in that the lower level process model was generated first. Key states in the lower model were then "vocalised" (i.e. transmitted) to create a higher level model. The concept of hierarchical consistency was introduced to ensure the controllers cooperated.

Conversely, in PCT the priority was to avoid state explosion. By following guidelines at the synthesis stage, a process and specification were decomposed before a process model was synthesised. This enabled smaller reduced models to be synthesised and removed the need for the overall process model to be synthesised at any stage (Alsop, 1996).

2.3.4 Industrial Implementation of Controllers

Configuration of industrial control systems has been made easier through the use of structured packages for the synthesis of controllers. The IEC 1131 standard and PLCOpen projects (Tinham, 1996) (Maler, 1999) aim to unify the industrial control community with a range of common specification languages. These promise to increase re-usability and portability of code across projects and platforms. The standard advocates the use of

five specification styles: *Instruction Lists*, *Structured Text*, *Ladder Logic*, *Function Block Diagrams* and *Sequential Function Charts*. Ladder logic, found in PLCs (Clements-Jewery and Jeffcoat, 1996) and structured text languages such as PARACODE (APV, 1994) and BATCH 90 (Elsag-Bailey, 1992) for distributed control systems (DCSs) are tailored to the task of process control. Function Block Diagrams and Sequential Function Charts lead the new wave of graphically configurable software beginning to replace these languages. All these techniques are structured to minimise errors and help produce comprehensive controllers, however they offer no formal means of analysis or design.

Since the development of the ISA-S88 industry standards (ISA, 1995) most control systems facilitate the decomposition of process procedures into *recipes, unit procedures, operations* and *phases*. Moreover, structured decomposition after the stage of *unit procedures* by separating *normal operating* procedures from *abnormal operating* procedures has also been facilitated. Phases are defined for stages of operation, one separate phase being the emergency phase. Similar approaches were developed where emergency procedures are defined in separate sequences, exploiting the process structure.

Unit procedures can also be run in parallel. However, since no formal verification of nonconflicting languages exists, this option is not widely exploited. It is also only pseudo concurrency as computers in reality must execute all actions in a sequence. Control code is stepped through in a sequence, either by lines of typed code or ordered function blocks graphically represented. The order and timing of such sequences can greatly determine the behaviour of the plant.

The *synchrony hypothesis* (Pnueli and Shalev, 1991), which assumes the controller is infinitely faster than its environment, is implicitly adopted when implementing controllers. Grace times are often used (Alsop, 1996) (Baird, 1997) (Elsag-Bailey, 1992) to hold controllers at certain states while process plant equipment completes the command that the controller has initiated. Therefore, the assumption of instantaneous execution of transitions (von der Beeck, 1994), where a string of controllable transitions is assumed to occur instantaneously, is not completely accurate within the context of industrial controllers.

Industrial control systems have developed to incorporate *concurrency* and *priority* while promoting structure that reduces complexity. Techniques have also been developed within user groups that overcome problems and exploit timing properties when implementing controllers on operating plants. However, implementation of methods that formally model or analyse process systems or verify their operating specifications is limited.

2.3.5 Automatic Code Generation

With the use of formal methods for control systems specification room for error was reduced to translating the specification into the actual control code to be implemented. Embracing the IEC 1131 standard several techniques have emerged approaching the problem from different angles. Methods include mapping Statecharts to PLC code (Bates *et al.*, 1998)and Petri Nets to programs in C++ (Yao and He, 1997). Others chose to develop

their own specification interfaces to produce a variety of vendor input languages (Duran and Batocchio, 1999; Sakao *et al.*, 1997). The variety of translation facilities available prove that it is possible to automatically generate control code from models. The common ground that all methods share is that they establish a rigid structure within the code that they produce. While each uses a unique structure, there are some common attributes. Establishing a powerful structure that could be applied to all industrial systems would facilitate rapid development and reproduction of operating procedures (Sanchez *et al.*, 1996).

2.4 Synthesis of Control Logic

Rather than building manually the logic for the controller and then verifying this logic using an algorithmic procedure, one could try to develop a logic that is correct by construction using CAD tools. This leads to the design of *provably correct controllers.* The following subsections outline three techniques following this path. An alternative review with practical comparisons of four synthesis techniques for DESs is given elsewhere (Pinzon *et al.* ,1997)97.

2.4.1 Supervisory Control Theory

The Supervisory Control Theory (SCT) proposed by Ramadge and Wonham (1987) forms the foundation for many other developments in modelling and control of DESs. SCT introduces a well defined feedback control setting where transitions are labelled as *controllable* and *uncontrollable.* Controllable transitions are those over which the process computer has some kind of control such as opening and closing valves. Uncontrollable transitions cannot be effected by the process computer. They are process responses that can only be observed. For example, levels rising and falling.

In the SCT framework, a process is controlled using a *supervisor* that enables/disables controllable transitions in the process. Supervisors were formally characterised in SCT as 1) *complete*, 2)*proper* and 3)*controllable.* A complete supervisor is able to track any potential event of the process under control. A proper supervisor assures that the process will reach its goal state. A controllable supervisor ensures the process will not enter a forbidden state of operation. That is, from any process state no transitions exist that lead to forbidden states and are enabled by the supervisor.

Considering these properties within the context of the structural modelling previously discussed, the controllability property (Wonham, 1996) ensures a process can stay within the normal range. SCT proposed a *extended non-admissible error range* which includes all states where control cannot restrict transitions to the *non-admissible error range.* This results in the synthesis of a controller that exists only within the *normal operating range.*

The design of a supervisor proceeds as follows. First the process and specifications are modelled using FSMs. Then, a minimal restricted supervisor is computed. This supervisor

achieves the specifications without unnecessary restrictions to the process behaviour.

As SCT was applied to larger processes, modularity was considered. Individual supervisors were generated for each specification then intersected to produce a supervisor for all specifications. This did not reduce the size of the process model though. As with all FSM based DES theory, SCT encountered the problem of state explosion when tackling larger applications. A hierarchical decomposition theory was outlined by Wonham (1996) whereby critical states of the system were *vocalised* to a higher state. A second model of the vocalised stated was generated as the higher controller. Specifications were applied in the same manner as before and a supervisor constructed. As multiple controllers were operating in parallel, the property of hierarchical consistency was developed to assure that they were synchronised. Vocalisation still required the base model to be constructed. It simplified system specification for the designer, however the computational load was not reduced.

In a practical application, Leduc (1996) decomposed a system under SCT. The aggregation of the modular supervisors was proven to be controllable. However, nonblocking was not proven. This was due to a large degree of interaction between components. Completely decoupled systems were proven to be controllable and nonblocking under modular decomposition within SCT.

Timed Discrete Event Systems (TDESs) were modelled within SCT by Brandin and Wonham (1994). Uncontrollable time events, *ticks*, were introduced to quantify time within the model. Other methods stemmed from these such as Timed Transition Models (Ostroff, 1997) where transitions were permitted to occur between a certain window of time once enabled. This allowed delays and time outs to be modelled without explicitly constructing an element for the task.

Within TDESs, *forcible* events were introduced. Both controllable and uncontrollable transitions could be forcible. As tick transitions were uncontrollable, they could not be disabled under SCT. Forcible transitions were defined as being able to preempt tick transitions. Although eluded to, preempting of non-tick transitions was not pursued. Thus the controllability analysis remained the same.

2.4.2 Procedural Control Theory

Procedural Control Theory (PCT) embraced the FSM modelling framework and the broad base that SCT had established to produce techniques more focused on the control of chemical processes. A more detailed introduction to the principles of PCT is given in Appendix 3 as well as in Sanchez (1994), Rotstein *et al.* (1995), Alsop (1996) and Sanchez *et al.* (1999).

Procedural Control Theory departed from SCT in two main areas: *controllability* and *forcing transitions*. By allowing a controller to force events and preempt other events occurring, PCT was able to provide a less conservative definition of controllability. States which had uncontrollable transitions leading to forbidden states but also at least one con-

trollable transition leading to an accepted state (normally removed by SCT) remained within the controllable behaviour as a controllable transition could preempt the uncontrollable transition. This *conditional controllability* was found to be far more applicable to chemical processes.

Placing these definitions within the structured modelling approach of section 2.3.1, an *extended non-admissible error range* was defined where a transition to the *non-admissible error range* cannot be prevented. For example, if a state has an uncontrollable transition leading to the *non-admissible error range* and a controllable transition leading elsewhere, the controllable transition could be forced to preempt the uncontrollable transition. Thus, the system remains under control and the state does not need to become part of the *extended non-admissible error range*. This approach has been supported (Klein and Wehlan, 1996) as process systems regularly operate in both the *normal operating range* and the *admissible error range*. However, while recognising that this structure exists, PCT did not exploit it while performing synthesis of a controller for the whole process.

A *Procedural Controller* was formally defined by Rotstein *et al.* (1995) which facilitated its translation directly into industrial control code. It required that from any state, either a single controllable transition existed or only uncontrollable transitions exited. Mouille (1997) then altered the definition to consider failures, so that failure events could not be preempted by controllable transitions.

PCT also encountered state explosion problems and methods of decomposition were addressed by using flags to communicate between controllers (Camillocci, 1995). Alsop (1996) showed that a process could be broken into smaller independent processes for the purposes of control. This was based on the realisation that during an operation not all process components play a role. Theory was developed to establish the serial and parallel components that were required. However, to apply the theory exactly was computationally complex. A similar solution was proposed in SCT which skirted this problem and could be applied to PCT (Lee and Wong, 1998). These subprocesses were allowed to share uncontrollable transitions. Controllable transitions could not be shared as that constituted controllers competing to drive a device. Series decomposition was also developed where a sequence could be broken into smaller manageable sequences by placing subgoals part way through it. Initiation and inhibition properties were also formally established by Alsop (1996) to assure that the process and controller started in the same state and that conflicting controllers did not execute at the same time.

These theories were exploited in several large scale manufacturing case studies (Gallo, 1996) (Presto Project P4, 1996) (Alsop *et al.*, 1996).

2.5 Timed Discrete-Event Systems

In an attempt to capture quantitative dynamics within a DES, Timed DESs have become popular. Wonham (1996) introduced timed FSMs to supervisory control theory where

time was quantified by *ticks*. Each tick transition represented a unit in time. Specifications were then able to include constraints regarding when an event could or should occur. For systems with small amounts of timing constraints this was very effective. However, as systems with longer timing constraints are tackled the models become considerably larger and their application less clear. Further development of supervisory control of timed discrete event systems has shown the benefits on small scale projects and married this development with hierarchical developments within SCT (Brandin, 1998) (Wong and Wonham, 1996*b*).

Quantitative dynamics are also present in Petri nets. In a case study, Kristoffersen *et al.* (1999) implement timed automata in the software package UPPAAL. By using a variable to measure time, the model size was kept under control. However, by explicitly modelling component behaviour, individual models seemed unnecessarily complex and also limited in their use. Components with possibly varying quantitative dynamic needed to be defined accordingly. As such a tank model needed to include two states to signify filling as it had two possible rates.

2.6 Summary and Conclusions

This literature review has shown how in recent times, the call for an assurance of the computer control system of a process to achieve its goals safely and efficiently has gained momentum. HAZOP analysis, now well established in general process design, is being extended to the control systems. A thorough library of questions relating to control systems has been established focusing on design, safety and efficiency. The questions provide a systematic approach to process system specification, forcing the designer to establish a clear specification of the desired operation.

A large degree of emphasis has been placed on the definition of systems and the methods of specifications. Batch processing operation specification has seen the development of the ISA-S88.01 standard which outlines procedures for modelling and specifying operating procedures. From the computing community the IEC 1131-1 standard for computer control programming has defined the preferred programming architectures that should be used. Both these standards endeavour to create a systematic method of specification of operating procedures, so to reduce the risk of error. In both industry and academia, graphical specification methods have also emerged such as STATEMATE, visualSTATE and Sequential Function Charts to ease the pain of specification. These have aided clear understanding and representation of desired behaviour.

Verification has been approached from several angles. Model checking algorithms have been developed but they can only handle problems of moderate size as they require enumerating the whole model. Symbolic model checking developed that increased the power but are still limited by size. Their solution method of Boolean Logic Diagrams also encounters difficulties when considering concurrent process operations. Petri Nets

enable model decomposition and powerful use of concurrent operating procedures. Graphical specification packages aid in behaviour definition, however decomposition methods result in complex conditional specifications and communication between models. Condition/Event systems follow a similar path to Petri Nets and encounter similar difficulties. Computational packages such as *SMV* and *Grafcet* enable verification of complex systems but require specialist knowledge of the program and its input language.

Alternatively to verification, formal synthesis methods have been proposed. Formal synthesis takes the general approach that is illustrated by Figure 2.1. Working from the bottom upward, a controller for a system is synthesised by intersecting the process model with the operating specification. The behaviour that is common to both the process model and the operating specification constitutes the controller. To create the overall process model, models of the elementary components of a system are combined and the modelling constraints based on the process's open loop behaviour are applied. To create the operating specification, again the models of the elementary components are combined and then the constraints regarding the desired operation of the process are applied. Two approaches to formal synthesis are Supervisory Control theory and Procedural Control Theory.

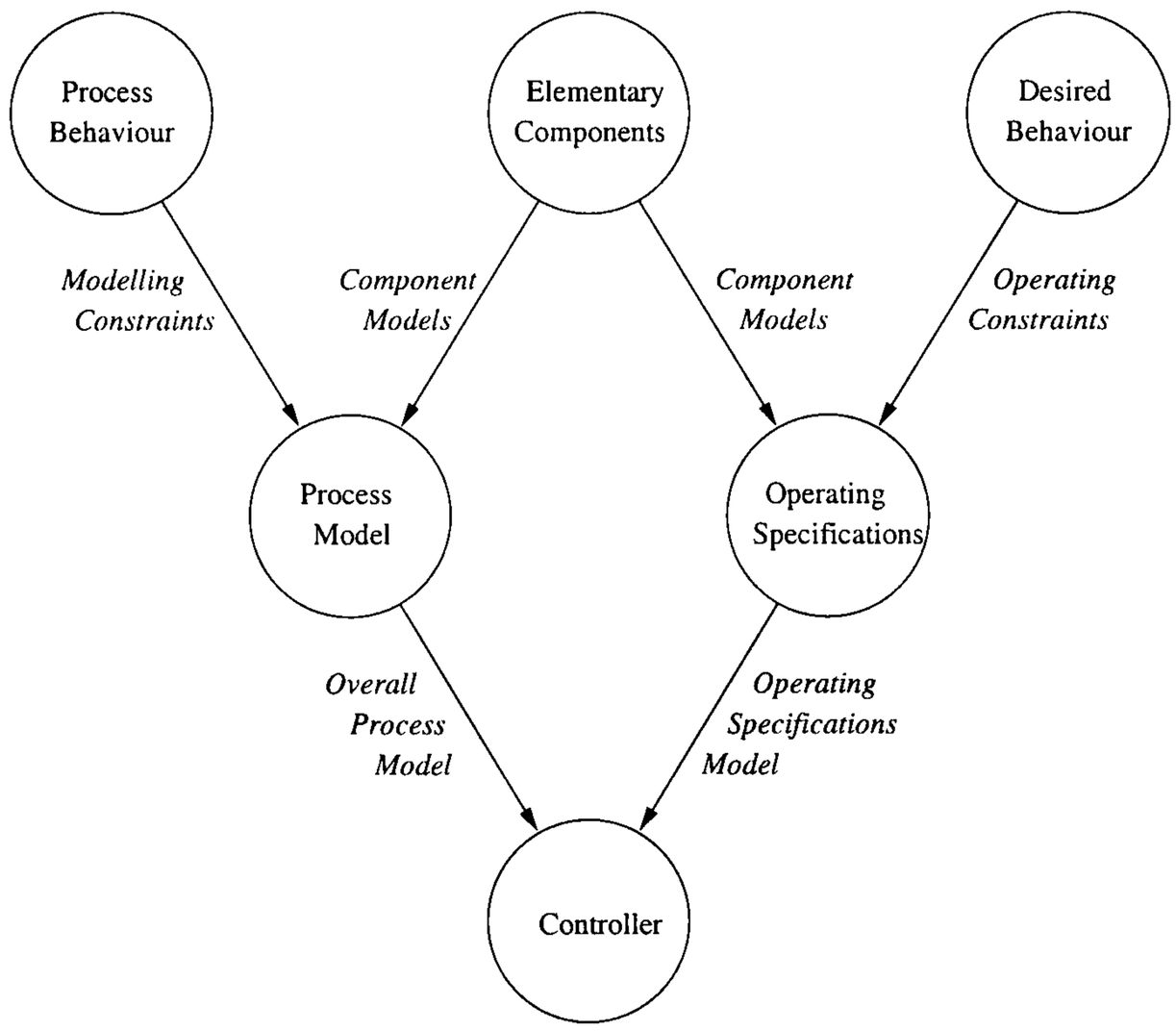


Figure 2.1: How formal synthesis techniques produce a controller.

Supervisory Control Theory (SCT) first proposed methods based on language theory. FSMs generating the process model language and the operating specification language were constructed and controllers synthesised as the language that result by intersecting them. The definition of transitions as either *controllable* or *uncontrollable* and controllability analysis were instrumental in creating a strong grounding for formally synthesised process supervisors. However, creating process model and specification FSMs was not easy requiring confidence in FSM theory. The definition of controllability within SCT is also restrictive and limits its application to process industry problems.

Building on the basic concepts of SCT, Procedural Control Theory (PCT) allows the synthesis of provably correct procedural controllers. Model and specification building were formalised by requiring FSMs representing the behaviour of each elementary component of a process to be created. By combining these simple models, complex process models are constructed. The concept of controllability was expanded making it more suitable for process systems and a *Procedural Controller* was defined that was a controller that could be directly applied to a process plant. Both SCT and PCT have software which has developed in parallel with the theory but like other approaches reviewed, to use them requires specialist knowledge regarding the theory.

Most DESs were first formulated with qualitative time measurement. Methods were developed that include time explicitly in the areas of Supervisory Control Theory, Petri Nets, Condition Event Systems and Timed Transition Models. In most of these areas, applications have been related to electronics industry such telecommunications and computer hardware specification. Timing details of the process being modelled remained constant and thus explicit timing was applied effectively. In the process industry, quantitative time can not be applied in the same way. Dynamics of a process vary and thus models involving explicit time steps become large and unmanageable when qualitative time measures have been adequate.

In all methods of specification, concepts of modularity and hierarchical decomposition have become common place. The industry standard advocates a hierarchical decomposition and the use of modularity to clarify specification. This is mirrored in industrial applications and facilitates the use of subroutines and decomposition of specification into separate tasks. Likewise, formal specification methods led by Harel (1987) followed the route of hierarchical decomposition. Methods based on Statecharts rely heavily of the hierarchical structure to reduce the size of problems. Thus, structured methods of specification definition have been developed. While this approach has contributed to reducing errors, it does not verify that the specification is valid and will ensure safe and efficient achievement of the operating goals.

Formal synthesis and verification techniques used modular decomposition to combat state explosion in process models. Hierarchical decomposition was used in SCT to decompose the specification. This however did not reduce the model size thus limiting its use. PCT established a modular decomposition technique which enabled the synthesis of reduced models and proved that they could be operated in combinations of series and parallel. Hierarchical implementation was done in practical examples however, formal hierarchical synthesis was not developed. Therefore, manual intervention is required, which can lead to error. Decomposition also has limitations and as more complex elementary components are modelled, especially those incorporating failures, state explosion against becomes an issue.

Engineering practice and experience in the development of procedural code suggest that it may be beneficial to formalise the special features associated with failure events. Fur-

thermore, characterising different regions of operation as normal and abnormal suggests that in fact failures should be treated differently from other system events. When a failure occurs, the system goals change (even if only temporarily), and a new set of operating specifications come into effect. Moreover, the feasible behaviour of the system is modified (e.g. a failed pump can only be off and cannot switch on). In essence, a designer specifies a separate procedural controller for each failure situation within the overall procedural controller. Industrial synthesis methods recognise this and result in logic that separates the procedures developed for normal operation from those active after failures have occurred. Incorporating this structural approach to the analysis of failure events into a formal and verifiable synthesis framework is yet to be addressed by current research.

Recent work has discussed the translation of models in such structured languages and verification using formal methods though they are in their infancy and encounter trouble with size. Additionally, the specifications produced by most packages are flat and lack any structure that is consistent with the two standards mentioned above. This makes the resulting logic more difficult to understand and translate into implementation code. These translation algorithms rarely consider the structure of the target code or produce code that resembles the process models and specification they use. Neither do they translate procedures that have been formally synthesised or verified. A need exists for a complete package that provides the ability to easily carry out process modelling and problem specification, formal synthesis or verification and code translation for structured implementation with minimal human intervention.

Chapter 3

Procedural Control Theory

This chapter outlines the theory developed by Sanchez (1994) to construct provably correct procedural controllers for DESs. PCT shares some of its foundation theory with Supervisory Control Theory. However, further development by Rotstein *et al.* (1995) and Alsop (1996) have made it applicable to large scale processes and produced implementable control code.

An example system is first introduced in section 3.1, which will be used through this appendix. An outline of the synthesis theory then follows. In section 3.3, approaches to the use of the theory are presented. Controller properties are discussed in section 3.3.1 and the chapter is completed with a description of decomposition theory in section 3.5.

3.1 Buffer System Example

The system chosen is an extension of the buffer system example presented by Sanchez (1994). A schematic of the system is shown in Figure 3.1.

Water is fed into the tank using valve FV and discharged using valve DV. Valve FV can fail open or fail closed. Valve DV is digital. Two Level sensors, each of two states, indicate the water level in the tank. A digital push button has been added to provide means for operator interaction. When pressed it signals an emergency situation and therefore the need to shutdown the system. The button can be pressed at any time.

The operating specifications for the system are as follows:

- In the initial state of the system, the tank is empty, the push button is released and both valves are closed.
- During normal operation:
    1. Valve FV is opened.
    2. The tank is filled to the normal level.
    3. Valve FV is closed.
    4. Valve DV is opened.
    5. The tank is discharged until it is empty.

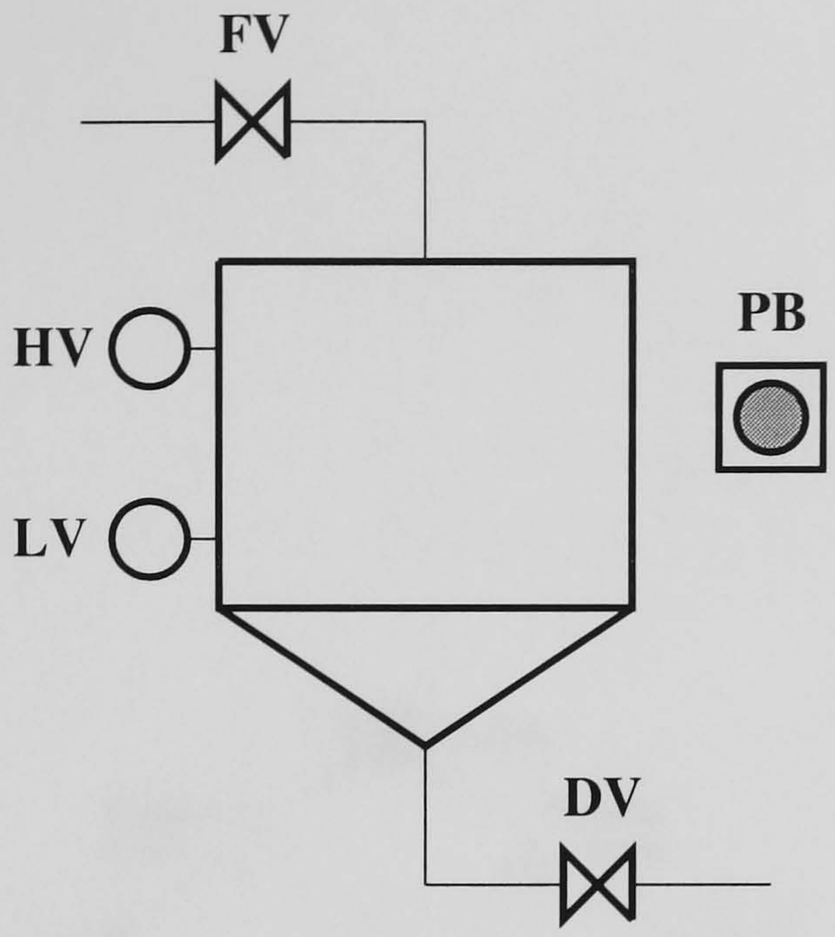


Figure 3.1: Schematic of the Buffer Tank Studied.

6. Valve DV is closed and no further actions occur.

- Emergency action to be taken:

  1. If valve FV fails closed.
     - Close Valve DV and take no further action.
  2. If valve FV fails open.
     - Open Valve DV and take no further action.
  3. If both valves are working properly and the push button is pressed.
     - Close Valve FV.
     - Close Valve DV and take no further action.

- Forbidden situations are:

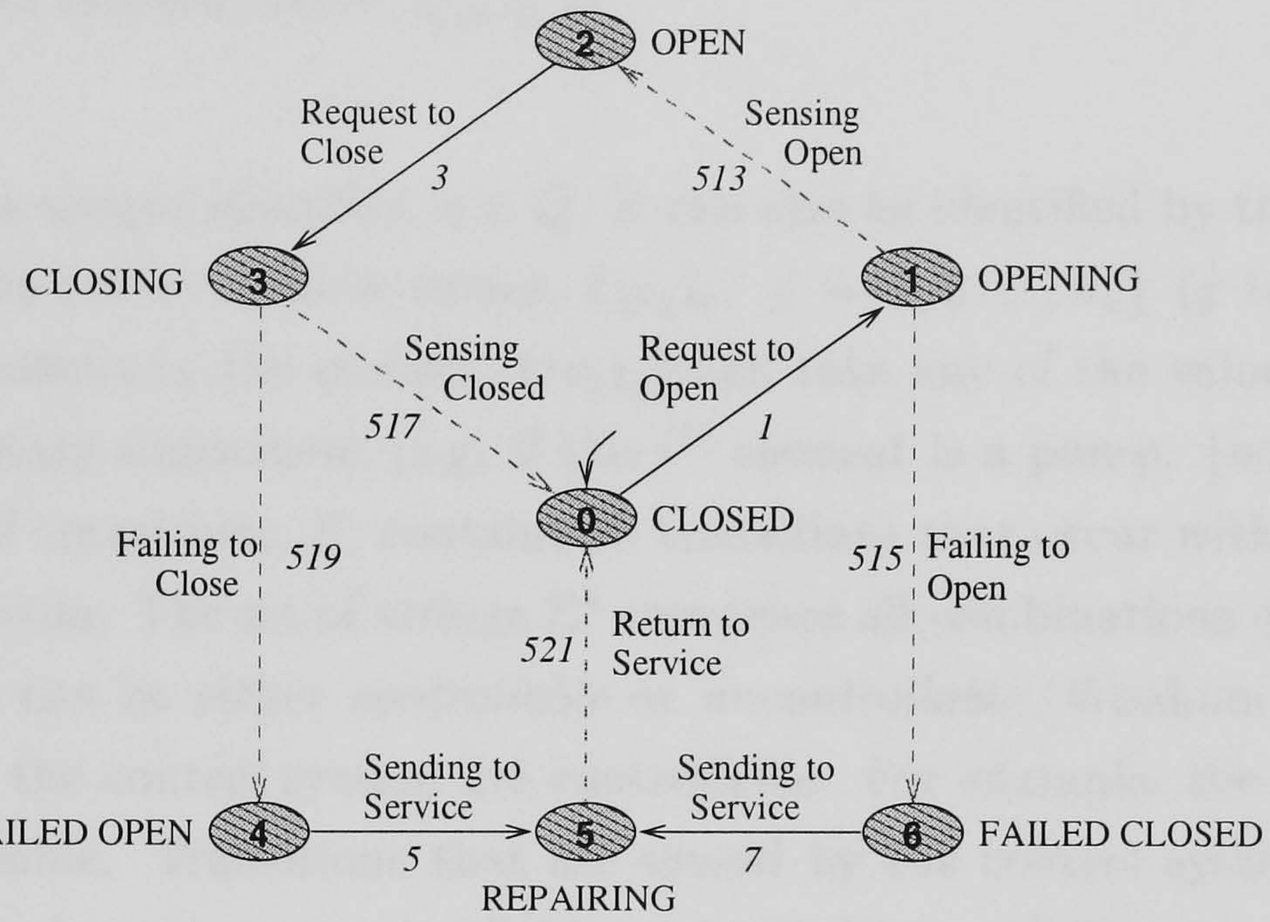
  1. During normal operation, never open both Valve FV and Valve DV simultaneously.

3.2 Synthesis of Procedural Controllers

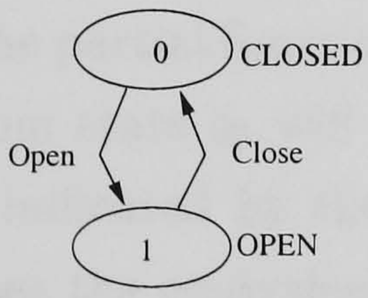
3.2.1 Finite State Machines

The synthesis of procedural controllers is a bottom up process. Each elementary component is modelled individually as an FSM. Examples are shown in figure 3.2. The FSM reflects the behaviour of the system by having finite states to represent every state of the system. The states are linked by the transitions of each of the elementary components. Every transition signifies a change of state. The FSMs used to model a process are represented by a 7-tuple:

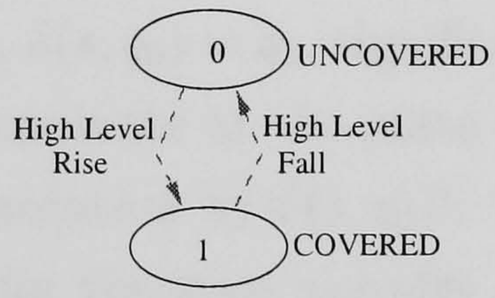
$$M = \{Q, V^{n_v}, \Sigma, \delta, \gamma, q_0, Q_m\} \tag{3.1}$$



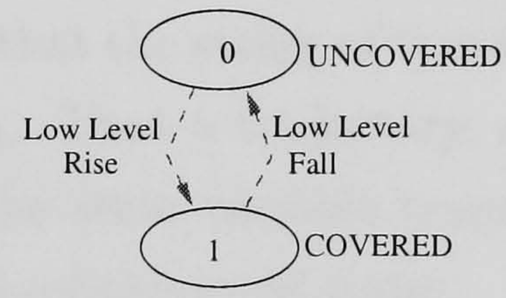
(a) Valve FV



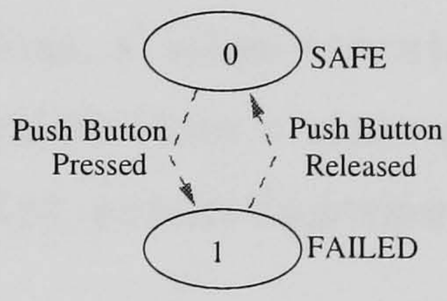
(b) Valve DV



(c) High Level Sensor



(d) Low Level Sensor



(e) Push Button

Figure 3.2: Primitive Component FSMs for Motivating Example

where

Q	is the set of states, $q \in Q$
V^{n_v}	is the set of state vectors of which each has a unique association with one state $q \in Q$, $\{(v_j)_q,\ j = 1, 2 \ldots, n_v\}$
n_v	is the number of state variables defining state q
Σ	is the set of transitions, $\sigma \in \Sigma$
δ	is the state transition partial function, $\delta : \Sigma \times Q \rightarrow Q$
γ	is the state variable transition partial function, $\gamma : \Sigma \times V^{n_v} \rightarrow V^{n_v}$
q_0	is the nominal initial state, $q_0 \in Q$
Q_m	is the set of marked states, $Q_m \subseteq Q$

Each state has a unique identifier, $q \in Q$. It can also be identified by the corresponding combination of the state variable values, $\{(v_j)_q,\ j = 1, 2 \ldots, n_v\}$ (j is the number of elementary components in the model). $\{(v_j)_q\}$ can take any of the values that are valid for the jth elementary component (e.g. if the jth element is a pump: $\{on, off\}$).

The alphabet of transitions, Σ, contains all transitions that occur within any of the elementary components. The set of strings Σ^* comprises all combinations of the transitions in Σ. Transitions can be either *controllable* or *uncontrollable* (Wonham, 1996). Transitions initiated by the control system are controllable. For example, the request to open a valve is controllable. Transitions that are sensed by the control system and which it has no direct control over are uncontrollable. A level rising or falling is an uncontrollable transition.

The state partial function, δ, defines the relationship between states for each transition. The partial function, $\delta(s, q_0) = q_1$, signifies that the string of transitions, s, when executed from state q_0 will terminate at the state q_1. That a trajectory, s exists from a state q_0, is indicated by the notation by $\delta(s, q_0)!$. The state variable transition partial function γ does the equivalent for the state variable descriptions of states.

FSMs have one initial state, q_0. From there PCT attempts to find trajectories to the goal or marked states, Q_m.

3.2.2 Languages and the Asynchronous Product

Each FSM contains an alphabet of transitions, Σ. When several transitions from Σ are linked together they form a string of transitions or word, also known as a process trajectory. Within each FSM several strings can be made. The collection of all the strings from an FSM is known as the language generated by the FSM. For a FSM, M, the language produced is defined as:

$$L(M) = \{s \in \Sigma^* |\ \delta(s, q_0)!\} \tag{3.2}$$

The marked language of an FSM, containing only strings which terminate at a marked state, is defined by:

$$L_m(M) = \{s \in \Sigma^* |\ \delta(s, q_0) = Q_m\} \tag{3.3}$$

Combining a set of elementary component FSMs and their respective languages produces all the possible combinations of the elementary component state variables and a language containing all possible transitions. Known as the asynchronous product of languages it uses the intersection of languages. First, we present the intersection of two languages:

$$L_1 \cap L_2 = \{s \in \Sigma^* | s \in L_1 \wedge s\ L_2\} \tag{3.4}$$

To create the asynchronous product, the inverse projection operator is used. This is more easily understood if the projection operator is first defined. The projection operator $\mathcal{P}_{\Sigma_p}$ deletes from strings in a language all occurrences of events not in Σ_p (Wonham, 1996). The inverse projection $\mathcal{P}^{-1}_{\Sigma_p}$ adds to all strings in a language all transitions that are not in Σ_p.

$$\begin{aligned}
\mathcal{P}_{\Sigma_p} &: \Sigma^* \rightarrow \Sigma_p^* \\
\mathcal{P}_{\Sigma_p}(\epsilon) &= \epsilon \\
\mathcal{P}_{\Sigma_p}(\sigma) &= \begin{cases} \sigma & \text{if } \sigma \in \Sigma_p \\ \epsilon & \text{if } \sigma \notin \Sigma_p \end{cases} \\
\mathcal{P}_{\Sigma_p}(s\sigma) &= \mathcal{P}_{\Sigma_p}(s)\mathcal{P}_{\Sigma_p}(\sigma) \quad s \in \Sigma^*, \sigma \in \Sigma
\end{aligned} \tag{3.5}$$

$$\mathcal{P}^{-1}_{\Sigma_p} L_p = \{s \subseteq \Sigma^* / \mathcal{P}_{\Sigma_p} s \in L_p\} \tag{3.6}$$

The inverse projection operator then allows the asynchronous product to synchronise on alphabets outside each of the elementary components alphabets. An important property is that the operation $\mathcal{P}^{-1}_{\Sigma_p}$ on the two languages L_1 and L_2 over alphabet Σ_p is closed under intersection. That is:

$$\mathcal{P}^{-1}_{\Sigma_p}(L_1 \cap L_2) = \mathcal{P}^{-1}_{\Sigma_p} L_1 \cap \mathcal{P}^{-1}_{\Sigma_p} L_2. \tag{3.7}$$

The asynchronous product of FSMs is defined by:

$$L(M) = \mathcal{P}^{-1}_{\Sigma_1} L(e_1) \cap \mathcal{P}^{-1}_{\Sigma_2} L(e_2) \ldots \cap \mathcal{P}^{-1}_{\Sigma_{nv}} L(e_{n_v}) \tag{3.8}$$

From this, a model and a specification are produced which are then used to produce a controller.

3.2.3 Developing the Model and Specification Constraints

The designer is guided to first develop modelling and operating specification constraints separately. When considering the interaction of several elementary components, not all possible combinations make sense. Typically, there may be states that do not physically exist. For example, on the buffer tank, it is not possible for the low level sensor, LV, to be uncovered and the high level sensor, HV, to be covered. These states must be removed from the asynchronous product to create a correct system model. Within a state, the state variable values are used to describe it (in the example, {FV, DV, LV, HV, PB}). To remove a state, predicate logic is used as demonstrated by equation 3.9. The covering symbol, ∞, is used to indicate "all possible values of a state variable" . The state and all transitions to it are removed from the model.

$$(\infty, \infty, Uncovered, Covered, \infty) = FALSE \tag{3.9}$$

If elementary components interact, some transitions may not be possible. In the example, the level cannot rise if the feed valve is closed. To remove transitions temporal logic is used. The source state is described by predicate logic. Temporal logic is then added to forbid the transition from occurring. Equation 3.10 is the formula forbidding the level rising in the example when the feed valve is closed. The symbol $\bigcirc$, *next*, indicates the transitions that must not follow the specified state. The state of the discharge valve, levels and push button do not matter and thus are *covered.*

$$(Closed, \infty, \infty, \infty, \infty) \rightarrow \bigcirc[\tau \neq Low\ Level\ Rise\ \overline{\vee}\ High\ Level\ Rise] \tag{3.10}$$

When developing a corresponding system model, the transitions can only be uncontrollable transitions and can only be removed. The formulae representing modelling constraints are then transformed into equivalent FSMs and then intersected using equation 3.11 with the asynchronous product of all component models to produce the correct overall system model.

$$L(M) = L(p_1) \cap L(p_2) \ldots \cap L(p_{n_i}) \cap \mathcal{P}_{\Sigma_1}^{-1} L(e_1) \cap \mathcal{P}_{\Sigma_2}^{-1} L(e_2) \ldots \cap \mathcal{P}_{\Sigma_{n_v}}^{-1} L(e_{n_v}) \tag{3.11}$$

where p_i is the corresponding FSM for the ith modelling constraint formula.

When developing the specification, only *controllable* transitions can be manipulated. States and transition forbidden in the operating specification are removed using predicate and temporal logic respectively as before. Transitions can also be *forced.* Forced transitions are implemented in a similar way to forbidden transitions as shown by equation 3.12. It describes the request to open the feed valve when in the initial state.

$$(Closed, Closed, Uncovered, Uncovered, Released) \rightarrow \bigcirc[\tau = Open\ Feed\ Valve] \quad (3.12)$$

It is possible to force a sequence of transitions. Equation 3.13 represents the specification that when the tank is full, the feed valve must be shut and the drain valve opened. When a transition is forced all other transitions originating from that state are removed.

$$\begin{aligned} (Open, Closed, Covered, Uncovered, Released) &\rightarrow \bigcirc[\tau = Close\ FV] \\ &\rightarrow \bigcirc[\tau = Open\ DV] \end{aligned} \quad (3.13)$$

In the specification, controllable transitions can also be forbidden using the same method as for the model, as shown by equation 3.10. Uncontrollable transitions can *never* be forced or forbidden by an operating specification. Forced transitions occur before any other transitions occur. This was based on the following assumption (Rotstein *et al.*, 1995).

Assumption 3.1 *Controllable transitions can always be executed* before *uncontrollable transitions.*

The predicate and temporal logic formulae are also converted into FSMs and intersected with the asynchronous product of the component models to produce the specification.

$$L(S) = L(r_1) \cap L(r_2) \ldots \cap L(r_{n_s}) \cap \mathcal{P}_{\Sigma_1}^{-1} L(e_1) \cap \mathcal{P}_{\Sigma_2}^{-1} L(e_2) \ldots \cap \mathcal{P}_{\Sigma_{n_v}}^{-1} L(e_{n_v}) \quad (3.14)$$

where r_i is the corresponding FSM for the ith specification constraint formula.
A process model and operating specifications may be intersected to produce a candidate controller, C.

$$L(C) = L(S) \cap L(M) \quad (3.15)$$

The closed looped behaviour of the candidate controller C on the process M is then defined as:

$$L(C/M) = L(C) \cap L(M). \quad (3.16)$$

The candidate controller at this stage is not necessarily a procedural controller or guaranteed to yield controllable closed loop behaviour when applied to the process model. Its applicability to these properties must be tested as is discussion in following Sections 3.3.1 and 3.4.

3.3 Evolution of Synthesis Methods

In implementing synthesis methods, Sanchez (1994) chose to develop two separate FSMs, one for the process model and one for the operating specifications. Both start with the asynchronous product of the component models and then have forbidden states and transitions removed. This method thus synthesised two larger FSMs to obtain the controller. The two FSMs are then intersected and controllability analysis is performed before a controller was produced. A controller is defined by a 5 tuple given by:

$$C = \{X, \Sigma, \xi, x_0, X_m\} \tag{3.17}$$

where

X	is the set of controller states, $x \in X$
Σ	is the set of process transitions, $\sigma \in \Sigma$
ξ	is the state transition partial function, $\xi : \Sigma \times X \rightarrow X$
x_0	is the initial controller state, $x_0 \in X$
X_m	is the set of marked controller states, $X_m \subseteq X$

The controller monitors uncontrollable transitions from the process and sends requests for controllable transitions occur. Within SCT and Sanchez's PCT, a controller ignores state variable values relying solely on transitions.

In an alternative approach, Mouille (1997) builds the controller state by state. Starting with the initial state, all constraints are tested on exiting transitions and possible destination states, whether they are associated with the process model or the operating specification. If a permitted new state or transition is found, it is added and the procedure repeated until no more are found. This way, a state that is forbidden or to which no transitions are permitted, is never created. Mouille's method only requires the whole asynchronous product to be created if there are no constraints at all. This reduces the computational complexity and thus allowed controllers for larger systems to be synthesised.

Mouille's approach relied heavily on state variable values. Thus, the controller now must be redefined as a 7 tuple:

$$C = \{X, Y^{n_v}, \Sigma, \xi, \psi, x_0, X_m\} \tag{3.18}$$

where the two additions are:

Y^{n_v}	is the set of state variables, $\{(y_j)_x,\ j = 1, 2 \ldots, n_v\}$
ψ	is the state variable transition partial function, $\psi : \Sigma \times Y^{n_v} \rightarrow Y^{n_v}$

The candidate controller was synthesised using Algorithm 9.1 which is presented in Chapter 9.

Mouille's method only considered specifications forcing individual transitions. Specifying a sequence of controllable transitions as in the work of Sanchez where not possible. It was also not possible to have two state with the same state variable values but with different transitions leading to and exiting from them.

3.3.1 Controllability

After applying the constraints, the FSM produced is not guaranteed to stay within a specific subset of desired states or to reach one of the marked goal states until it has under gone some further tests.

States that cannot be reached from the initial state or from which a marked state cannot be reached are undesirable and are removed by finding the *reachable, coreachable* and *trim* structure within the FSM.

In an FSM, M, the set of reachable states, Q_r, includes those states where a trajectory exists from the initial state enabling them to be reached. If all states within M can be reached, M is reachable. The reachable states, Q_r are:

$$Q_r = \{q \in Q \mid \exists s \in \Sigma^*,\ \delta(s, q_0) = q\} \tag{3.19}$$

The set of states in M from which a marked state is reachable is called the coreachable set of states. If all states within M are coreachable then M is coreachable. Q_{cr} is defined as:

$$Q_{cr} = \{q \in Q \mid \exists s \in \Sigma^*,\ \delta(s, q) \in Q_m\} \tag{3.20}$$

The set of reachable and coreachable states within M is the trim structure of M. If all states are reachable and coreachable, then M is trim.

Next the controller produced must undergo tests for *completeness* and *controllability*. Completeness established within SCT by Ramadge and Wonham (1987) and extended under the PCT framework (Rotstein *et al.*, 1995) assures that every trajectory within the model is accounted for by the controller. Trajectories are either tracked within the controller or preempted by controllable transitions within the controller.

Definition 3.1 Completeness *A controller C is complete with respect to a given process P if, for any string $s \in \Sigma^*$ and $\sigma \in \Sigma_u$, the conditions,*

$$\xi(s, x_0)! \text{ and } \delta(s\sigma_u, q_0)! \tag{3.21}$$

imply that one of the following is true:

1. $\xi(s\sigma_u, x_0)!$
2. $\exists \sigma_c \in \Sigma_c \text{ s.t. } \delta(s\sigma_c, q_0)! \wedge \xi(s\sigma_c, x_0)!$

Controllability tests the ability of the controller to maintain the process within the specific subset of states desired. The mathematical definition is shown by definition 3.2. It tests that for each state, either (option 1) all uncontrollable transitions terminate in states within the controller FSM or (option 2) that there is at least one controllable transition leaving the state that terminates within the controller FSM.

Definition 3.2 Controllability. *A language K is controllable with respect to a language L if it can be partitioned into two languages K_1 and K_2, where $K_1 \cup K_2 = K$ such that:*

1. $K_1\Sigma_u \cap L \subseteq \overline{K}$
2. $\forall s \in K_2, \exists \sigma_c \in \Sigma_c$ s.t. $s\sigma_c \in \overline{K} \wedge s\Sigma_u \cap \overline{K} = \emptyset$

where $\overline{K}$ is the prefix closure of the language K. The prefix closure of a language K is the set of prefixes of all strings within a language K. If $\overline{K} = K$, then the language is closed, all FSMs produce closed languages. The prefix closure, $\overline{K}$ is defined by:

$$\overline{K} = \{s \in \Sigma^* | \ \exists t \in \Sigma^*, st \in K\} \tag{3.22}$$

Nonblocking assures that every partial trajectory in a language can be completed to a string contained in the marked language. That is, that the FSM will always terminate at a marked state. A language generated by controller C on a system M is nonblocking if:

$$L(C/M) = \overline{L_m(C/M)} \tag{3.23}$$

Nonblocking is achieved if the following are true:

Theorem 3.1 *If C and M are trim, the closed loop system generated by C on M is nonblocking iff $L_m(\mathcal{S}_u^w C)$ and $L_m(M)$ are nonconflicting.*

Proof *See (Sanchez, 1996).*

where two languages are *nonconflicting* if any common prefix can be completed to a common string.

Theorem 3.2 *The closed loop system generated by C on M is nonblocking if C is trim, complete and in agreement with respect to M.*

Proof *See (Alsop, 1996).*

Conformance to specification tests whether a controller will achieve the requirements of the specification. Mathematically, this means that the closed loop language of the controller on the system is a subset of the specification language.

$$L(C/M) \subseteq L(S) \tag{3.24}$$

Alsop (1996) showed that this is true if $L(C) \subseteq L(S)$.

3.4 Procedural Controller

The above synthesis method and checks enable us to produce a controller but not necessarily a procedural controller. What is synthesised is the *Maximal Superstructure of Procedural Controllers* that conform to the specification. From it procedural controllers can be extracted. These are defined as follows:

Definition 3.3 *A procedural controller is a controller,*

$$C = \{X, \Sigma, \xi, x_0, X_m\} \tag{3.25}$$

in which for each $x \in X$ such that $\xi(\sigma, x)!$, one of the following is true:

1. $\sigma \in \Sigma_u \wedge \forall \sigma_c \in \Sigma_x \Rightarrow \xi(\sigma_c, x)$ *is undefined.*

2. $\sigma \in \Sigma_c \wedge \forall \sigma' \in \Sigma, \sigma' \neq \sigma \Rightarrow \xi(\sigma', x)$ *is undefined.*

Procedural controllers are then able to be directly translated into industrial control code.

3.5 Decomposition

Some decomposition methods have been developed by Alsop (1996) within the framework of PCT. Three cases were studied: internal consistency, parallel decomposition and series decomposition.

3.5.1 Internal Consistency

Internal consistency is the formalisation of a decomposition method practised regularly when implementing controllers. Practically, it says that elementary components that do not affected the process and are not effect by the process do not need to be included in the model of the process. such a decomposition, called a Class Ia system, is mathematically defined by:

$$\mathcal{P}_a^{-1} L(G_a) \cap L(M)\Sigma_a \subseteq L(M) \tag{3.26}$$

$$L(S_b) = L(G_b) \tag{3.27}$$

$$L_m(G_b) = L(G_b) \tag{3.28}$$

Here, process M, is divided into the part of the process being used, G_a, and the unaffected part, G_b. Equation 3.26 states that G_a is internally consistent with M. Equation 3.27 states that the specification of S_b is *free* and therefore every state of G_b is allowed. Equation 3.28 makes every state of G_b a marked state and thus the sequence can terminate with G_b in any state.

3.5.2 Parallel Decomposition

Parallel decomposition divides a process into subsystems operating concurrently. These Class Ib systems must conform to one condition, that the only shared transitions are uncontrollable.

$$\Sigma_a \cap \Sigma_b \subseteq \Sigma_u \tag{3.29}$$

This assures that the controllers produced operate independently of each other. This is to say that they do not compete to force or inhibit the same controllable transition, while they are permitted to observe any uncontrollable transitions (shared or not). These controllers are termed from here on *independent.* If marked languages produced by the subsequent controllers are nonconflicting and conform to their respective specification then so too does the whole system.

3.5.3 Series Decomposition

Series decomposition has been used in a variety of applications. In PCT, series decomposition simplifies the problem of specification by dividing operating procedures into smaller manageable phases of operation. Subgoals are defined within the operating specification which are states that the process must pass through on the way to the overall operation goal. The system is then decomposed between these subgoals. For example, a system with a single subgoal would be divided as below.

$$H_1 = \{Q, V^{n_v}, \Sigma_{-\sigma}, \delta, \gamma, q_0, \{q_1\}\} \tag{3.30}$$
$$H_2 = \{Q, V^{n_v}, \Sigma_{-\sigma}, \delta, \gamma, q_1, Q_m\} \tag{3.31}$$

Likewise, the specification is also split into phases. For the above case, two specifications, S_1 and S_2 were produced. The full specification S is given by:

$$L(S) = L(S_1) \cup L(S_1).\sigma_1.L(S_2) \tag{3.32}$$

where $\sigma_1 \notin \Sigma$ is the event that signals that the system has reached the subgoal. Series decomposition conforms to the following theorems.

Theorem 3.3 *For class II systems, the closed loop behaviour generated by M under series control from C_1 and C_2 conforms to specification S if C_1 generates a closed loop behaviour on H_1 which conforms to specification S_1 and similarly C_2 on H_2 and S_2.*

Proof *See Alsop (1996)*

Theorem 3.4 *For class II systems, the closed loop behaviour generated by M under series control from C_1 and C_2 is nonblocking if C_1 generates nonblocking behaviour on H_1 and similarly C_2 on H_2.*

Proof *See Alsop (1996)*

Thus, procedural controllers in series can be synthesised independently according to these rules and then operated together safely.

3.5.4 Procedure Initiation and Inhibition Theory

If a procedural controller begins execution when the system it is controlling is not in the correct initial state, problems of system deadlock, livelock or uncontrollability may arise. Procedure initiation theory (Alsop, 1996), assured that a procedural controller's initial state was satisfied before it began execution. System pre-checks were defined as,

$$L(C_x/M) = \Sigma^*.\sigma_x.L(C_x) \cap L(\mathcal{S}_{\sigma_x}^{Q_x} M) \tag{3.33}$$

where $\sigma_x \notin \Sigma$ is the transition signifying the starting of the procedural controller. $\mathcal{S}_{\sigma_x}^{Q_x}$ is the selflooping of the starting transition at the initial state in the model. The first term in equation 3.33 allows any string of transitions to occur before the controller is started by the execution of σ_x and the synthesised sequence of controlled transitions is to execute. The second tracks each of the transitions during open loop operation but only allows the execution of σ_x when the system is in the initial state. The intersection of these two terms assures a safe closed loop procedural controller.

When a system is decomposed and several procedural controllers are controlling a system, interaction between the controllers must be considered. If controllers conflict with each other's operation, they must be operated mutually exclusively. These rules follow on from the decomposition theory introduced above. As uncontrollable transitions can be shared between controllers, inhibition theory pertains mainly to the sharing of controllable events. Moreover, it assesses whether controllers will compete for the use of a controllable component or will drive a controllable component in opposite directions and thus cause the controllers to become livelocked. For a detailed explanation consult Alsop (1996).

Chapter 4

Statecharts

FSMs represent a DES through exact states which correlate with each discrete physical state. They have been used in applications of relatively small size to synthesise and validate operating procedures for safety critical systems. For larger applications, additional methods were required whereby the process was decomposed into manageable subsystems (Alsop, 1996). FSMs grow exponentially with the inclusion of each extra elementary component. For models consisting of 10 two-state components, the FSM has at least 1024 states. This is difficult for a designer to understand and use effectively. Computational complexity also becomes a problem as the FSMs reach sizes of 10^6 states. Decomposition methods have made progress towards reducing the effect of state explosion. However, systems exist that cannot be decomposed to small enough subsystems that can be synthesised easily. Further development of decomposition methods or investigation of alternative modelling techniques are required.

First introduced by Harel (1987), Statecharts provide some solutions to the problem of state explosion. By using hierarchy and modularity within a single statechart, the representation of a system is far less cluttered and more easily understood. Computational complexity is also reduced by minimising redundancy. In addition, they introduce the benefits of a formal representation for concurrent behaviour (Huizing and de Roever, 1991). Statecharts can be summarised by the following equation (Harel, 1987; von der Beeck, 1994; Huizing and de Roever, 1991).

$$Statecharts = FSM + depth + orthogonality + broadcast\_communication \tag{4.1}$$

A brief overview of statecharts and the properties that are of particular interest to this work is given here. For a detailed introduction to statecharts consult (Harel, 1987) for the original models or their more recent review (Harel and Naamad, 1996).

4.1 States

Statecharts promote hierarchy and modularity by including new states describing the system's behaviour. This way statecharts provide easier comprehension of the system. However, they do sacrifice the exact knowledge of every components state at every stage of operation.

Depth is introduced in statecharts by the use of a hierarchy of super and sub-states. A state within a state is termed a *substate*. A state containing a *substate* is termed a *superstate*. At the lowest and most detailed level exist *basic* states. Basic states do not have any substates.

All states that are not basic states are either *OR-states* or *AND-states*. Within an OR-state, the system can reside in one and only one of the immediate *substates*. An *AND-state* is partitioned into several parallel states and the system resides simultaneously in all of them. This introduces the property of concurrency, that is the system is concurrently in each of the states within the AND-state.

Figure 4.1 shows the different types of states. The highest level superstate, A, is an AND-state containing the two OR-states of B and C. Within B are the basic state D and the AND-state E. E contains the OR-states of F and G. Within F and G are the basic states of H, I, J and K. The basic states of L, M and N are contained within the state C.

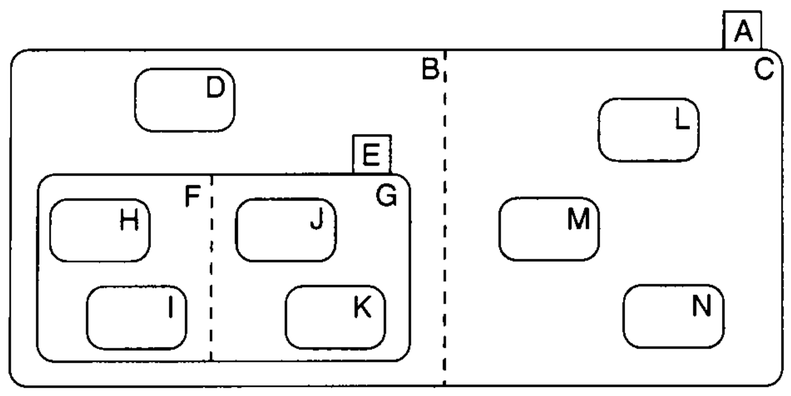


Figure 4.1: Types of States within Statecharts.

4.2 Transitions

The introduction of a hierarchy of states facilitates the definition of more powerful transitions. Transitions have the following entities: 1) a *source* state, 2) a *terminating* state, 3) *enabling* conditions, and 4) *executed* commands. The first two properties are self explanatory. The enabling conditions and executed commands are optional properties. It is possible to have none or several of both properties. If a state variable's value cannot be determined from the source state, an enabling condition is used to assess its value. Several enabling conditions can be combined within a transition to create an AND-condition. When the transition is enabled, the system moves to the next state. It does so through

the executed commands. These are controllable commands that are executed because of the enabling conditions. Therefore while the states do not display as much information as in a FSM, a statechart allows the transitions more power to compensate and improve the design process.

Figure 4.2 shows examples of statechart transitions. Transition *t1* is enabled by the uncontrollable condition x. When it is true, the system moves to state B and the controllable executed command y occurs. Transition *t2* is an AND-condition where both f and g must be true before the transition is enabled. When it is enabled executed commands, h and i occur and the system moves to state C.

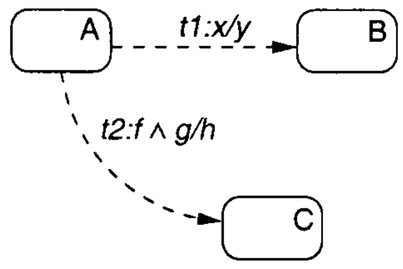


Figure 4.2: Transitions within Statecharts.

Embracing a state hierarchy allows for economy of transitions as well. Instead of several transitions from a group of states all terminating at a common state, a single transition from a superstate containing the originating states can be used. Transitions can be at any level of the hierarchy and traverse levels as well. This is demonstrated in figure 4.3. Three transitions with identical enabling conditions and executed commands all terminate at the common state D but have different source states. The standard representation would be as in figure 4.3(a). Forming a superstate E around states A, B and C, a single transition can replace the original three.

If a transition does not terminate at a basic state, three methods can determine the active substate: *default* states, *history* states and *connectors*. Default states are defined when the model is synthesised and are the active state when no other state is specified. They are shown by a default transition which originated from a filled circle and terminates

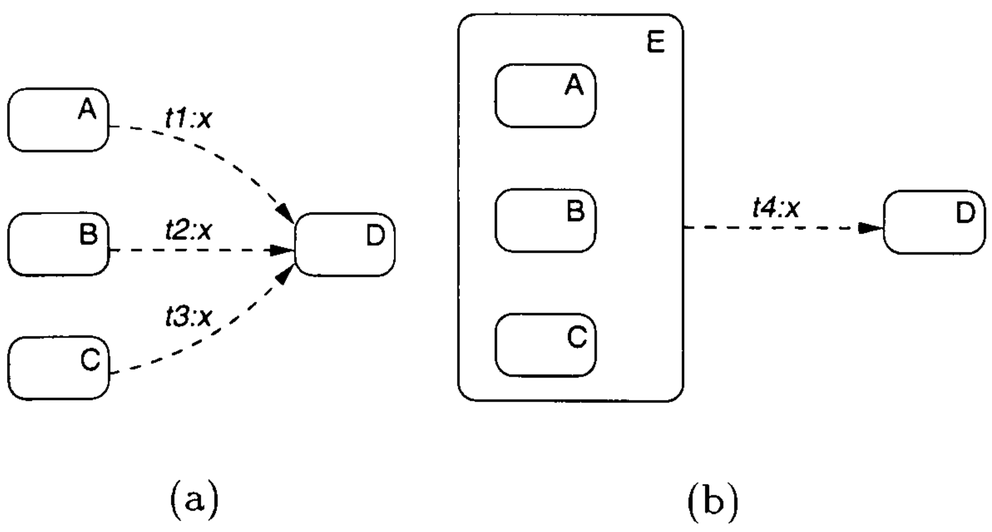


Figure 4.3: Transition economy in Statecharts

at the default state. History states return the system to the substate which it was in when the system last left the superstate. History states have priority over default states and are shown by a circle with an H inside. Connectors, depicted by a circle with a C inside, conditionally select the destination state within a superstate. Transitions lead to the connector and then conditional transitions lead from the connector to the possible active states.

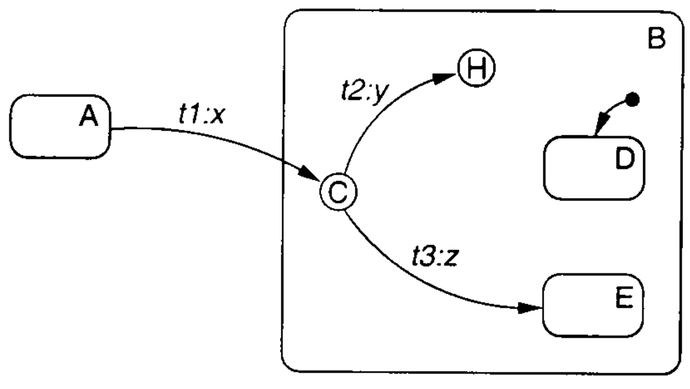


Figure 4.4: Active State Selection within Statecharts.

Figure 4.4 shows a transition, *t1*, enabled by the condition x from state A to state B. The transition terminates at a connector. Two transitions with conditions start at the connector. If condition y is true, transition $t2$ is enabled which terminates at the history state. The new active state will be the state that the system was most recently in. If condition z is true, transition $t3$ is enabled and the new active state is E. If neither condition is true, then the default transition is enabled and the new active state is D.

4.3 Static Commands

States can also contain *static commands*. These commands are automatically executed when the system enters the state. Static commands are formatted in the same way as transitions and thus can be considered as transitions that have been placed within the state for economy of specification size. Using static commands, a procedural controller can be condensed to a group of wait states linked by uncontrollable transitions. The use of static commands to condense a model to its key states has some similarities with state vocalisation (Wonham, 1996).

Figure 4.5, shows the statechart representation of the normal operation of the buffer example first introduced in chapter 3. Within each state, static commands are present. They are of the simplest form containing only executed commands. Therefore, the controllable transitions *s2: /Close FV* and *s3: /Open DV* are executed as soon as the system enters the state *empty tank*. As mentioned above, the specification has been condensed to three wait states containing static commands. The transitions linking each state are uncontrollable transitions with only enabling conditions.

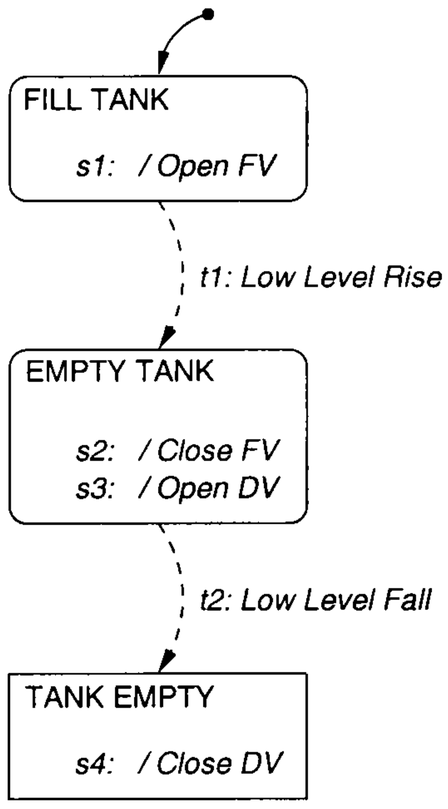


Figure 4.5: Normal Operating Procedure for the Buffer System.

When static commands contain enabling conditions as well as executed commands, statecharts become very powerful. Because statecharts do not explicitly track every individual state variable all of the time, the model may not know whether a transition is possible. Enabling conditions provide a formal way of dealing with this problem. In figure 4.6, the state *push button failed* does not know whether valves FV and DV are open or closed. Rather than creating an FSM of four states to track three values, two static commands with enabling conditions are used. If valve FV is open then static command $s5$ is enabled and the executed command *Close FV* occurs. Likewise with DV and static command $s6$. If the enabling conditions are not true then the static command is not enabled and the next command is assessed.

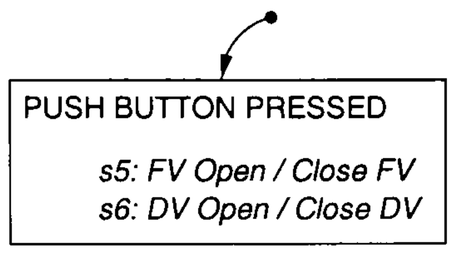


Figure 4.6: Statechart for Push Button Emergency.

4.4 Static Commands and Initial States

When a static command contains only executable commands, it assumes that these commands are possible. Therefore, the state of the respective state variable is known. For example in figure 4.5, the initial state is assumed to be when both valves are closed and the level low. The static commands do not need to check them. However, in the hierarchy of statecharts some knowledge can be lost. When the system enters a new superstate, not all state variables may be known.

When an FSM is defined, the initial and subsequent state variable values are known. If there are several possible initial states, then several FSMs must be created. These can be identical in structure except that their initial states differ. With the use of static commands, statecharts are able to have a single initial state which can correspond to several different initial states in FSM terminology. When the push button is pressed in the buffer example, there are four possible combinations of the valve states. These can all be covered by a single state as shown in figure 4.6. This strength of statecharts is very important in applications within chemical processes as they contain a large number of uncontrollable events which must be checked when a controller is initialised. Traditionally, these elementary components had to be in a specific state before a procedural controller would start. Most processes can start with uncontrollable components, such as a tank level, in any one of several states. This flexibility of statecharts allows a single controller to manage this property whereas PCT would require a separate controller for each combination of states.

The complete specification of the buffer system example is shown in figure 4.7.

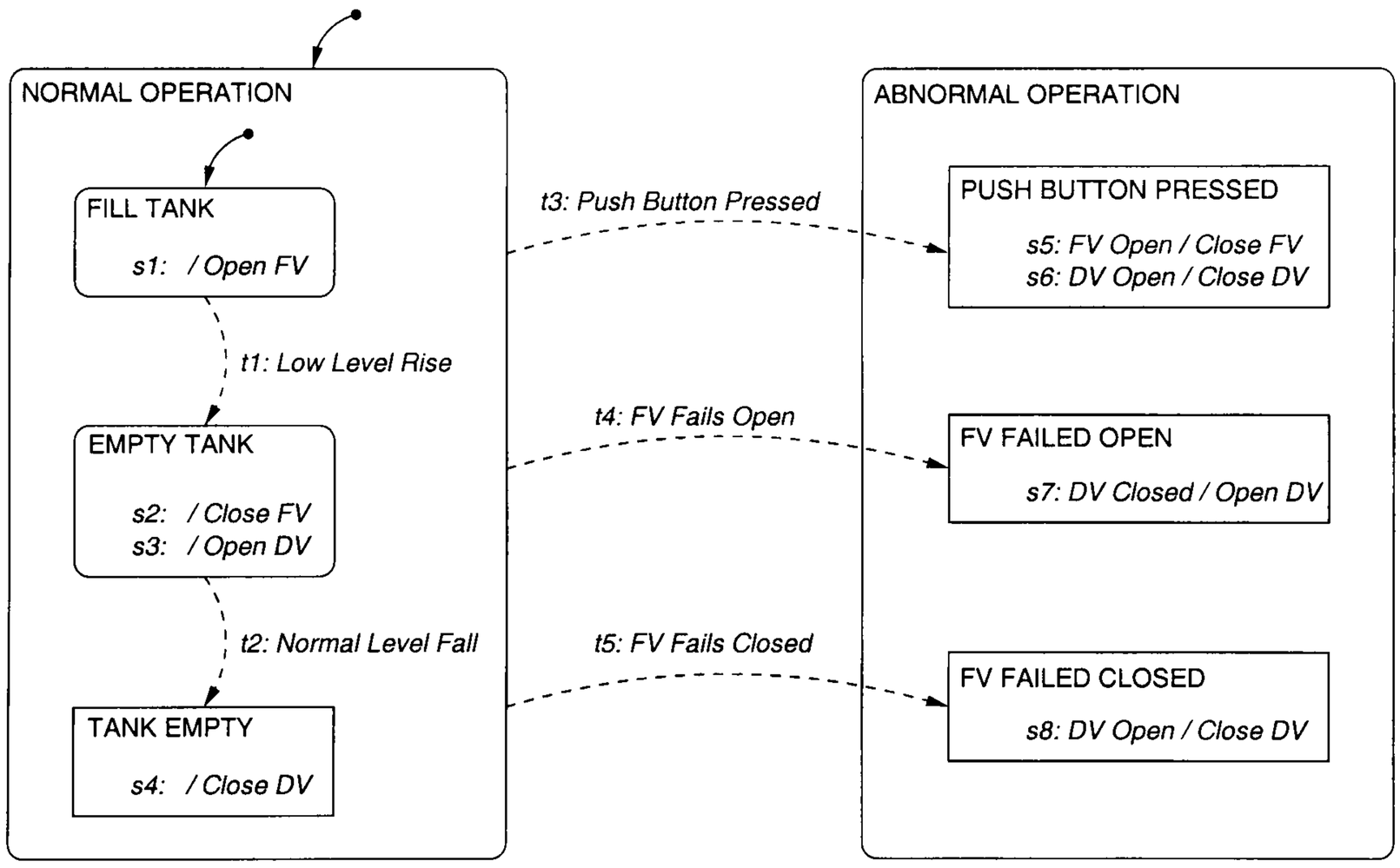


Figure 4.7: Statechart for Buffer System Example.

Chapter 5

Hierarchical Procedural Controllers

5.1 Introduction

In the synthesis and validation of controllers for DESs, state explosion is the largest obstacle in the way of development (Camillocci, 1995). Both in computation and designer understanding, it limits the size of problems that can be tackled. Now techniques exist (Alsop, 1996; Lee and Wong, 1998) which enable decomposition of large problems into manageable subsystems, making formal synthesis possible. However, the coordination of the subsystems and their respective controllers has not been formally addressed. This leaves the behaviour of the controller produced by recombining these subsystems without a guarantee of being *provably correct* or verified.

In a series of papers (Wong and Wonham, 1996*a*; Wong and Wonham, 1996*b*), a method for hierarchical supervisory control was presented. The concept of hierarchical consistency was introduced highlighting the importance of communication between different levels of a controller hierarchy. They used *vocalisation* to acknowledge (or vocalise) important states in the lower controller to be monitored by the higher supervisor. However, this method requires the synthesis of the complete process model at the lowest level and still faces the problem of state explosion. It also requires each case to be considered individually.

Decomposition rules exist within PCT (Alsop, 1996) that remove the need to synthesise whole process models. However, their implementation structure was established *a priori*. That is to say that the execution order, controller initiation and completion were established in the definitions. Both PCT and SCT, use a two tier hierarchy that confine elementary components to the lowest level.

This leaves three areas of interest that require further development for a thorough formal framework to be established. These are:

1. Procedural controllers should have the ability to assess whether it is appropriate that they execute or not when called and not be synthesised as part of a predetermined

controller hierarchy.

2. The desirable goal of achieving provably correct procedural controllers is not met if higher level coordinating controllers are not formally synthesised. Therefore, formal synthesis is required at all levels of the controller hierarchy.

3. Multi-tiered controller hierarchies with elementary components at any level would increase the number of problems to which PCT is applicable.

5.1.1 Aim

This chapter aims to uphold the formal synthesis rules of PCT while increasing its application by:

1. creating a basis for modelling generic procedural controller behaviour so that it can be used to formally synthesise higher level coordinating procedural controllers,

2. developing techniques that create autonomous procedural controllers thus increasing their robustness and reusability,

3. introducing procedural controllers containing a combination of elementary components and other procedural controllers, and

4. establishing theoretical proofs for the application of each of the above developments.

5.2 Modular Procedural Control Theory

In this section, established modular procedural control theories for pre-checking, series decomposition and parallel decomposition (Alsop, 1996) are extended to consider the open loop behaviour of a process before and after controller execution. This enables their proper use in modular procedural controllers.

In PCT, the language produced by two controllers operating in parallel is:

$$L(C_x \uparrow C_y/M) = \mathcal{P}_{\Sigma_x}^{-1} L(C_x) \cap \mathcal{P}_{\Sigma_y}^{-1} L(C_y) \cap L(M) \tag{5.1}$$

where,

$$L(C_x) = L(S_x) \cap L(H_x) \tag{5.2}$$

$$L(C_y) = L(S_y) \cap L(H_y) \tag{5.3}$$

and

$$L(M) = \mathcal{P}_{\Sigma_x}^{-1} L(H_x) \cap \mathcal{P}_{\Sigma_y}^{-1} L(H_y). \tag{5.4}$$

If the alphabets Σ_x and Σ_y are disjoint, parallel operation is trivial and the equation is suitable. However, if transitions are shared between controllers, only when both controllers are operating will they be permitted to occur. This is due to the controllers not considering the open loop behaviour of the process before they begin execution. To include this and thus to enable the controllers to operate in parallel but having different initial conditions, pre-checks are added.

Open loop behaviour is modelled by the term Σ_i^*. It represents any transition strings that occur when controller i is not operating. An external command, modelled by the external transition σ_{s_i}, is also required to start the controller. It is an uncontrollable transition that models the assessment of the current process state when an external source requests that controller i be started.

$$L(C_x \uparrow C_y/M) = \mathcal{P}^{-1}_{\Sigma_x+\sigma_{sx}}\{\Sigma_x^*.\sigma_{s_x}.L(C_x)\} \cap \mathcal{P}^{-1}_{\Sigma_y+\sigma_{sy}}\{\Sigma_y^*.\sigma_{s_y}.L(C_y)\} \cap L(\mathcal{S}_x^x \mathcal{S}_y^y M) \quad (5.5)$$

Likewise, for a process M under series control from C_1 and C_2, the closed loop behaviour is defined by equation 5.6 and with pre-checks by equation 5.7:

$$L(C_1 \rightarrow C_2/M) = L(C_1).\sigma_{c_1}.L(C_2) \cap L(\mathcal{S}^{q_1}_{\sigma_{c_1}} M) \quad (5.6)$$

$$L(C_1 \rightarrow C_2/M) = \Sigma^*.\sigma_{s_1}.L(C_1).\sigma_{c_1}.L(C_2) \cap L(\mathcal{S}^{q_0}_{\sigma_{s_1}} \mathcal{S}^{q_1}_{\sigma_{c_1}} M). \quad (5.7)$$

The order of C_1 followed by C_2 was predetermined by the equations and there are no pre-checks regarding C_2. This is because C_2 is synthesised with the knowledge that it should always immediately follow C_1.

After controllers have finished executing, no open loop behaviour is considered. Therefore, the controllers are required to terminate in synchronisation. Again, any shared transitions are forbidden when only one controller is executing. External completion transitions, communicating the controllers completion to external monitors, are also absent from the equations.

Equations 5.5 and 5.7 are now extended to include the termination of controllers and open loop behaviour after controllers have completed execution. The language generated by two parallel controllers with differing initial states and differing marked states is given by:

$$\begin{aligned} L(C_x \uparrow C_y/M) &= \mathcal{P}^{-1}_{\Sigma_x+\sigma_{sx}+\sigma_{cx}}\{\Sigma_x^*.\sigma_{s_x}.L(C_x).\sigma_{c_x}.\Sigma_x^*\} \\ &\cap \mathcal{P}^{-1}_{\Sigma_y+\sigma_{sy}+\sigma_{cy}}\{\Sigma_y^*.\sigma_{s_y}.L(C_y).\sigma_{c_y}.\Sigma_y^*\} \\ &\cap L(\mathcal{S}^x_{\sigma_{sx}} \mathcal{S}^y_{\sigma_{sy}} \mathcal{S}^x_{\sigma_{cx}} \mathcal{S}^y_{\sigma_{cy}} M), \end{aligned} \quad (5.8)$$

and by two controllers in series with differing initial and marked states:

$$L(C_1 \rightarrow C_2/M) = \Sigma^*.\sigma_{s_1}.L(C_1).\sigma_{c_1}.\sigma_{s_2}.L(C_2).\sigma_{c_2}.\Sigma^* \cap L(\mathcal{S}^{q_1}_{\sigma_{s_1}} \mathcal{S}^{Q_{m_1}}_{\sigma_{c_1}} \mathcal{S}^{q_2}_{\sigma_{s_2}} \mathcal{S}^{Q_{m_2}}_{\sigma_{c_2}} M). \quad (5.9)$$

Developing equation 5.9, the first term on the right hand side can be divided into two terms:

$$\begin{aligned} L(C_1 \rightarrow C_2/M) &= \Sigma^*.\sigma_{s_1}.L(C_1).\sigma_{c_1}.\sigma_{s_2}.\Sigma^*.\sigma_{c_2}.\Sigma^* \\ &\cap \Sigma^*.\sigma_{s_1}.\Sigma^*.\sigma_{c_1}.\sigma_{s_2}.L(C_2).\sigma_{c_2}.\Sigma^* \\ &\cap L(\mathcal{S}^{q_1}_{\sigma_{s_1}} \mathcal{S}^{Q_{m_1}}_{\sigma_{c_1}} \mathcal{S}^{2}_{\sigma_{s_2}} \mathcal{S}^{Q_{m_2}}_{\sigma_{c_2}} M) \end{aligned} \quad (5.10)$$

since

$$\begin{aligned} L(C_1) &\subseteq \Sigma^* \\ L(C_2) &\subseteq \Sigma^*. \end{aligned}$$

Traditionally, series controllers have been synthesised from the same transition alphabet. However, this is a conservative approach which is not necessary. Instead of equation 5.10, equation 5.11 uses the inverse projection operator, $\mathcal{P}^{-1}$, to allow two controllers from differing alphabets to synchronise on transitions that are not common.

$$\begin{aligned} L(C_1 \twoheadrightarrow C_2/M) &= \mathcal{P}^{-1}_{\Sigma_1+\sigma_{s_1}+\sigma_{c_1}}\{\Sigma_1^*.\sigma_{s_1}.L(C_1).\sigma_{c_1}.\sigma_{s_2}.\Sigma_1^*.\sigma_{c_2}.\Sigma_1^*\} \\ &\cap \mathcal{P}^{-1}_{\Sigma_2+\sigma_{s_2}+\sigma_{c_2}}\{\Sigma_2^*.\sigma_{s_1}.\Sigma_2^*.\sigma_{c_1}.\sigma_{s_2}.L(C_2).\sigma_{c_2}.\Sigma_2^*\} \\ &\cap L(\mathcal{S}^{q_1}_{\sigma_{s_1}} \mathcal{S}^{Q_{m_1}}_{\sigma_{c_1}} \mathcal{S}^{2}_{\sigma_{s_2}} \mathcal{S}^{Q_{m_2}}_{\sigma_{c_2}} M). \end{aligned} \quad (5.11)$$

On closer inspection, equation 5.11 (series operation) is very similar to equation 5.8(parallel operation). The difference lies in the presence of start and completion transitions for each procedural controller and on their placement in the first two terms of the equations. Before continuing, an explanation of these external transitions and their role in hierarchical procedural control theory must be given.

5.2.1 Procedural Transitions

Modular procedural control theory, as presented above, uses start transitions, σ_{s_i}, and completion transitions, σ_{c_i}. These are external transitions not in the controllers' alphabets ($\sigma_{s_i}, \sigma_{c_i} \notin \Sigma_i$), that are forced or observed by another controller. Other than their inclusion in the above equations, their formal implementation has not yet been addressed. The group of external transitions that control the operation of a procedural controller

are termed *Procedural Transitions*. These transitions can include start and completion transitions (mentioned above) plus pause, hold, stop and abort, to name a few.

Although an informal implementation method was presented in a practical application by Camillocci (1995), there was a lack of theoretical support. The approach created a hierarchy of controllers communicating through flag transitions which are analogous to the procedural transitions presented here.

Meanwhile, the ISA-S88.01 international standard has defined a procedure hierarchy and rules for specification decomposition of batch processes providing an avenue for generic implementation of procedural components.

It shall now be shown how a procedural controller's behaviour can be generalised and modelled as an FSM with a transition alphabet containing procedural transitions.

5.3 Modelling Procedural Controllers as Procedural Components

Like the elementary component instances they control, procedural controllers' behaviour can be discretised and modelled using FSMs. The ISA-S88.01 standard presents a comprehensive model, shown as an FSM in Figure 5.1, from which a suitable model can be derived. Procedural transitions, introduced in the previous section, can be seen on the figure. This model of the procedural controller can be treated like any elementary component model and implemented in a higher level procedural controller. From here on, elementary components representing procedural components will be referred to as procedural components.

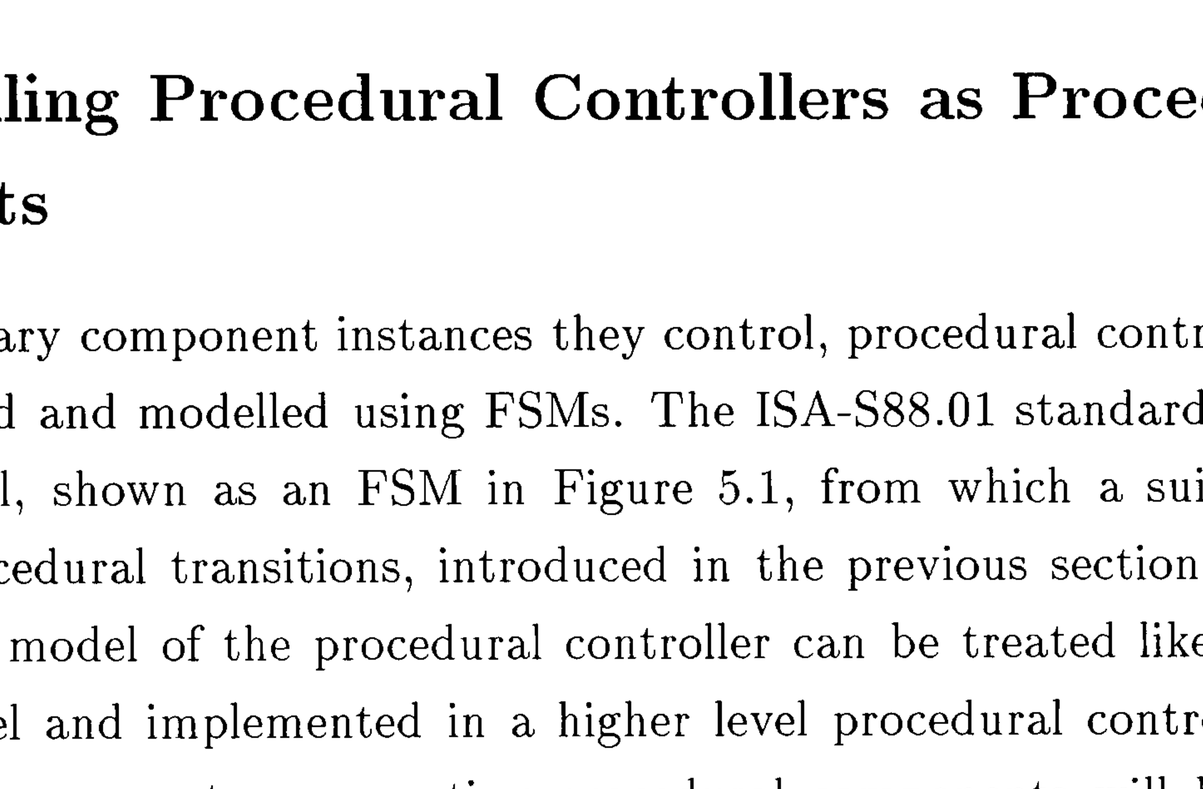


Figure 5.1: ISA-S88 Finite State Machine representing a sequence controller.

5.3.1 Consideration of Marked States in Procedural Controllers

The structure in Figure 5.1 has a single *complete* state reached by an uncontrollable transition. This means that the procedural controller terminates automatically on reaching

a marked state. It assumes that all marked states have the same significance. This fact raises two questions:

1. How can procedural controllers that are not self-terminating (i.e. safety monitoring) be modelled as procedural components?

2. How can procedural controllers with marked states of differing significance be modelled?

5.3.1.1 Termination of Procedural Controllers

There are currently two approaches to the termination of sequential code. Marked states in SCT are points where the supervisor can be terminated by external sources. On the other hand, marked states in PCT are points where the controller's aim has been achieved and the controller terminates itself. Both situations arise in sequential control applications, thus, each approach has merit and should be included in a comprehensive synthesis approach.

As the aim of this work is to create a generic structure, the basic procedural component shown in Figure 5.1 is augmented with the addition of a *Terminated* state, as shown in Figure 5.2. From the *Complete* state the controller may be terminated externally or it may return (by an uncontrollable transition) to the *Running* state. If the controller is self-terminating, the terminating transition may be redundant or can be used to reset monitoring variables.

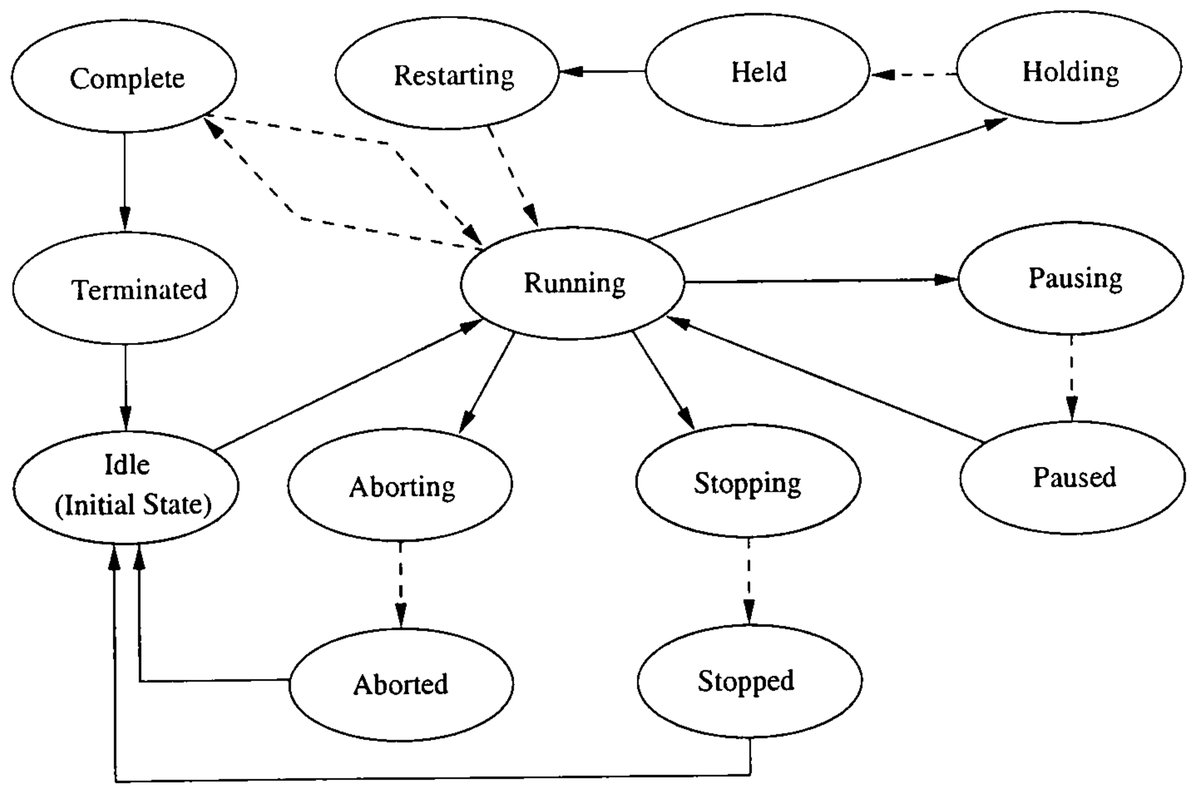


Figure 5.2: Augmented ISA-S88 Procedural Component FSM.

5.3.1.2 Marked States of Differing Significance

A procedural controller may often contain several marked states with differing significance. For example, one marked state could signify the procedural controller's goal having been achieved, while another could signify that a failure has occurred and the controller has reached a corresponding state to terminate in. This information needs to be

communicated to the higher level coordinating controller. In order to take this fact into account, the further augmented procedural component shown in Figure 5.3, is proposed. It contains several *Complete* states, in this case, called *Successfully Complete* and *Failed Complete*. By knowing which marked state has been reached, the higher level controller can take the appropriate action.

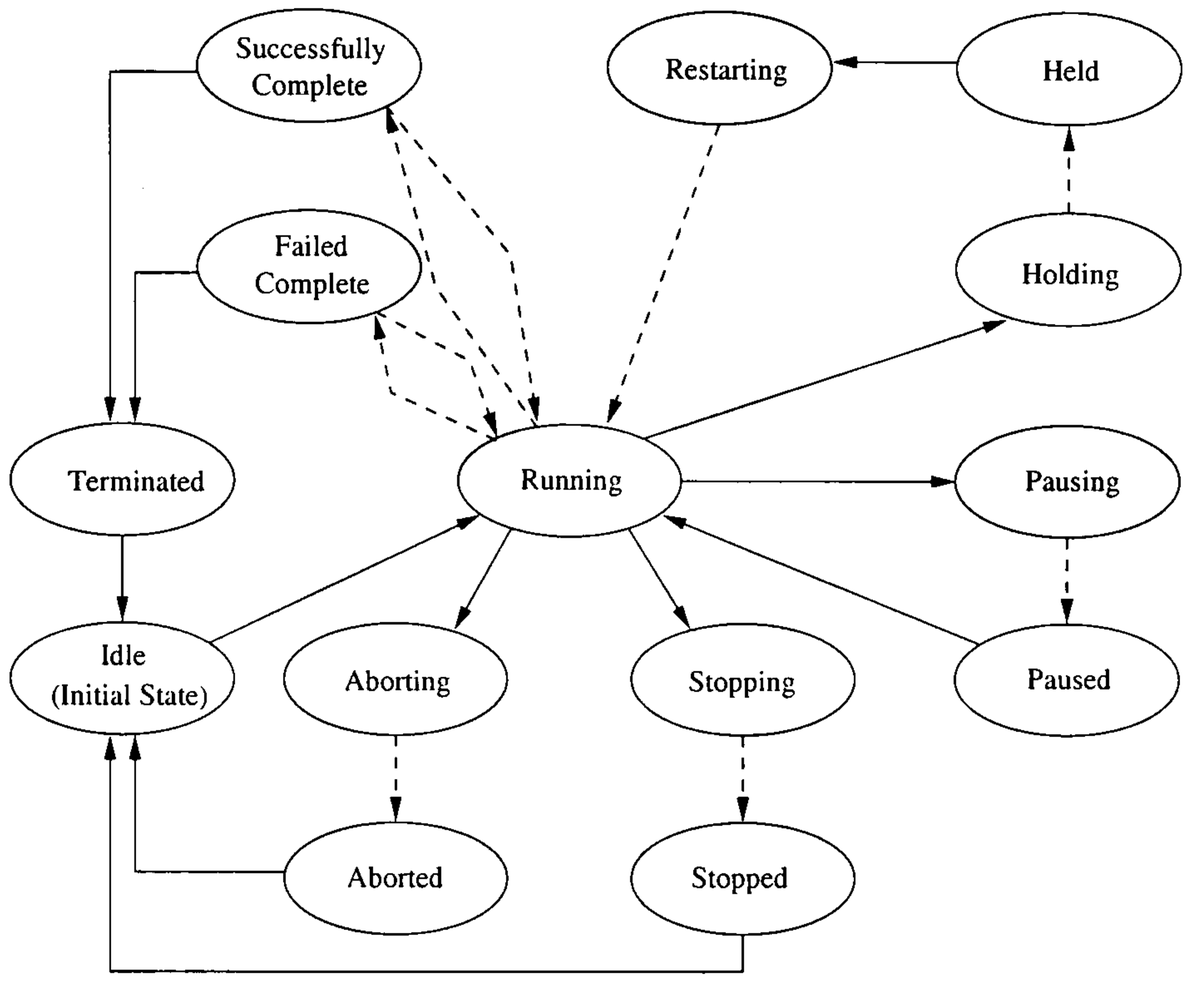


Figure 5.3: Augmented ISA-S88 Procedural Controller FSM with multiple completion states.

Each procedural controller loses some of its generality with this modification and becomes more like a unique procedural component. That is to say that the number and significance of the marked states in each procedural controller will have to be considered when creating its procedural component. However, the definition of each procedural component is done by following a generic set of instructions. Marked states in procedural controllers rarely are grouped into more than two types and indeed, when Partitioning Theory (introduced in Chapter 7) is used, there is rarely more than one group.

5.4 Synthesising Higher Level Procedural Controllers

To synthesise a higher level procedural controller coordinating the sequential operation of several procedural controllers, the asynchronous product of several procedural components provides the higher level controller model (no modelling constraints exist). Previous synthesis techniques (requiring the synthesis of the complete asynchronous product of elementary components before applying constraints) cannot synthesise procedural controllers containing the procedural components introduced in section 5.3. However this is not a problem, since new computation techniques which are presented later in Chapter 9, do not require synthesis of the asynchronous product and apply modelling and specification constraints from the first stages of controller synthesis. This enables the procedural component instances of section 5.3 to be included.

Operating specifications are determined by either inhibition theory, regarding which controllers are permitted to operate concurrently, or rules decided by the designer regarding execution order of the lower level procedural controllers. Using these specifications, a provably correct higher level procedural controller is synthesised. In doing so, no flag variables are required as the use of procedural components eliminates their need.

The practise is propagated through the complete controller hierarchy by creating a procedural component for each higher level procedural controller and synthesising an even higher level procedural controller, with as many hierarchical levels as desired. In other words, the procedural controller at each level is a procedural component of the complete controller.

5.4.1 Integrating Higher and Lower Level Procedural Controllers

It will now be shown how procedural components are incorporated into the modular procedural control theory presented in section 5.2. This will be named Hierarchical Procedural Control Theory (HPCT).

The languages generated by procedural controllers operating in parallel and series are defined by equations 5.8 and 5.11. When a higher level procedural controller is synthesised, a language of procedural transitions is generated to specify the execution order of the lower level procedural controllers. This language is added to modular control theory to produce the language generated under hierarchical control. In the following discussion, the terms on the right hand side of the equations will be referenced as either *controller terms* (the first two terms) or *modelling terms* (the third term).
For parallel operation, equation 5.8 becomes:

$$\begin{aligned} \underline{L(C_x \uparrow C_y / M)} \;=\; & \mathcal{P}_{hier}^{-1} L(C_{hier}) \\ \cap\; & \mathcal{P}_x^{-1}\{\Sigma_x^*.\sigma_{s_x}.L(C_x).\sigma_{c_x}.\Sigma_x^*\} \\ \cap\; & \mathcal{P}_y^{-1}\{\Sigma_y^*.\sigma_{s_y}.L(C_y).\sigma_{c_y}.\Sigma_y^*\} \\ \cap\; & L(\mathcal{S}^x_{\sigma_{s_x}} \mathcal{S}^y_{\sigma_{s_y}} \mathcal{S}^x_{\sigma_{c_x}} \mathcal{S}^y_{\sigma_{c_y}} M), \end{aligned} \tag{5.12}$$

which represents the language generated by two procedural controllers operated in parallel by a higher level procedural controller. In the above, $\mathcal{P}_x^{-1}$ is an abbreviation for $\mathcal{P}^{-1}_{\Sigma_x + \sigma_{s_x} + \sigma_{c_x}}$.

The difference between the two modes of operation, parallel and series, lies in the use of the start and completion transitions in the controller terms. For parallel operation (equation 5.8), each controller term only contains the start and completion transitions for that controller. This is because during parallel operation each procedural controller is not concerned when the others starting or completing their execution. For series operation (equation 5.11), where controller execution order is very important, all controller terms contain the start and completion transitions of all controllers operating in series.

However, in the new approach presented here (equation 5.12, the higher level procedural controller contains the information regarding the execution order of the lower controllers. Therefore, parallel or series operation is imposed by the higher level procedural controller and need not be included in the controller terms of the lower level controllers.

For series operation of two procedural controllers under the control of a higher level procedural controller, the order of procedural transitions, $\sigma_{s_1}.\sigma_{c_1}.\sigma_{s_2}.\sigma_{c_2}$ is contained within $L(C_{hier})$. Then, equation 5.11 results in:

$$\begin{aligned} \underline{L(C_1 \rightarrow C_2/M)} &= \mathcal{P}_{hier}^{-1} L(C_{hier}) \\ &\cap \mathcal{P}_1^{-1}\{\Sigma_1^*.\sigma_{s_1}.L(C_1).\sigma_{c_1}.\Sigma_1^*\} \\ &\cap \mathcal{P}_2^{-1}\{\Sigma_2^*.\sigma_{s_2}.L(C_2).\sigma_{c_2}.\Sigma_2^*\} \\ &\cap L(\mathcal{S}^1_{\sigma_{s_1}}\mathcal{S}^2_{\sigma_{s_2}}\mathcal{S}^1_{\sigma_{c_1}}\mathcal{S}^2_{\sigma_{c_2}}M), \end{aligned} \tag{5.13}$$

which is identical to equation 5.12. From now on, equation 5.12 and 5.13 will be referenced as the language generated by two procedural controllers under the control of a higher level procedural controller. So far, the discussion has only considered the operation of two lower level procedural controllers coordinated by a higher level controller. We now consider the general case of a higher level controller, C_{hier}, coordinating several procedural controllers. The set of lower level controllers is denoted by C^I, where $C^I = \{C_1, C_2, C_3..., C_i\}$ and i is the number of lower level controllers. Thus the language generated by one higher level controller coordinating several controllers is:

$$\begin{aligned} L(C_{hier} \triangle C^I/M) &= \mathcal{P}_{hier}^{-1} L(C_{hier}) \\ &\cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i).\sigma_{i_c}.\Sigma_i^*\} \right) \\ &\cap \left(\bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}}\mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}}M) \right) \end{aligned} \tag{5.14}$$

(5.15)

and its associated marked language is defined by:

$$\begin{aligned} L_m(C_{hier} \triangle C^I/M) &= \mathcal{P}_{hier}^{-1} L_m(C_{hier}) \\ &\cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L_m(C_i).\sigma_{i_c}.\Sigma_i^*\} \right) \\ &\cap \left(\bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}}\mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}}M) \right) \end{aligned} \tag{5.16}$$

where n is the number of low level controllers.

The inverse projection operator, $\mathcal{P}^{-1}_{hier}$, in the above, enables the high level controller to synchronise on events in the lower controllers, conforming to the rules of parallel decomposition and hierarchical decomposition, $\Sigma_{hier} \cap \Sigma_i \subseteq \Sigma_u$.

5.4.2 Properties of Hierarchical Procedural Controllers

This section shows that the properties of PCT are preserved when a process is placed under the control of a hierarchical procedural controller. First, we introduce lemmas relating the decomposed controllers, C_i, and the decomposed models, H_i. Then, the properties of conformance to specification and nonblocking are shown to hold under hierarchical PCT.

Lemma 5.1 $\bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} M) = \mathcal{P}^{-1}_{hier} L(H_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}^{-1}_{\Sigma_i + \sigma_{s_i} + \sigma_{c_i}} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} H_i) \right)$

Proof

From equation 5.4,

$$L(M) = \mathcal{P}^{-1}_{\Sigma_x} L(H_x) \cap \mathcal{P}^{-1}_{\Sigma_y} L(H_y).$$

Considering the starting transitions required for hierarchical control:

$$L(\mathcal{S}^{Q_{0x}}_{\sigma_{sx}} \mathcal{S}^{Q_{0y}}_{\sigma_{sy}} M) = \mathcal{P}^{-1}_{\Sigma_x + \sigma_{sx}} L(\mathcal{S}^{Q_{0x}}_{\sigma_{sx}} H_x) \cap \mathcal{P}^{-1}_{\Sigma_y + \sigma_{sy}} L(\mathcal{S}^{Q_{0y}}_{\sigma_{sy}} H_y),$$

and now the completion transitions gives,

$$L(\mathcal{S}^{Q_{0x}}_{\sigma_{sx}} \mathcal{S}^{Q_{0y}}_{\sigma_{sy}} \mathcal{S}^{Q_{mx}}_{\sigma_{cx}} \mathcal{S}^{Q_{my}}_{\sigma_{cy}} M) = \mathcal{P}^{-1}_{\Sigma_x + \sigma_{sx} + \sigma_{cx}} L(\mathcal{S}^{Q_{0x}}_{\sigma_{sx}} \mathcal{S}^{Q_{mx}}_{\sigma_{cx}} H_x) \cap \mathcal{P}^{-1}_{\Sigma_y + \sigma_{sy} + \sigma_{cy}} L(\mathcal{S}^{Q_{0y}}_{\sigma_{sy}} \mathcal{S}^{Q_{my}}_{\sigma_{cy}} H_y).$$

Now, generalising for a model decomposed into several sub-models,

$$\bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} M) = \left(\bigcap_{i=1}^{n} \mathcal{P}^{-1}_{\Sigma_i + \sigma_{s_i} + \sigma_{c_i}} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} H_i) \right).$$

Intersecting the hiearachical model, $\mathcal{P}^{-1}_{hier} L(H_{hier})$ with both sides,

$$\mathcal{P}^{-1}_{hier} L(H_{hier}) \cap \left(\bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} M) \right) = \mathcal{P}^{-1}_{hier} L(H_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}^{-1}_{\Sigma_i + \sigma_{s_i} + \sigma_{c_i}} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} H_i) \right).$$

However, since

$$\mathcal{P}^{-1}_{hier} L(H_{hier}) \cap \left(\bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} M) \right) = \bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} M) \ ,$$

then

$$\bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} M) = \mathcal{P}^{-1}_{hier} L(H_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}^{-1}_{\Sigma_i + \sigma_{s_i} + \sigma_{c_i}} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} H_i) \right).$$

Lemma 5.2 $\Sigma_i^*.\sigma_{i_s}.L(C_i/H_i).\sigma_{i_c}.\Sigma_i^* = \Sigma_i^*.\sigma_{i_s}.L(C_i).\sigma_{i_c}.\Sigma_i^* \cap L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} H_i)$

Proof

From equation 3.16,

$$L(C/M) = L(C) \cap L(M).$$

Likewise,

$$L(C_i/H_i) = L(C_i) \cap L(H_i).$$

Selflooping the starting transition, σ_{s_i}, at the initial state on both sides gives,

$$\sigma_{s_i}.L(C_i/H_i) = \sigma_{s_i}.L(C_i) \cap L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} H_i),$$

and the completion transition at the marked states,

$$\sigma_{s_i}.L(C_i/H_i).\sigma_{c_i} = \sigma_{s_i}.L(C_i).\sigma_{c_i} \cap L(\mathcal{S}^{Q_{0_i}}_{\sigma_{s_i}} \mathcal{S}^{Q_{m_i}}_{\sigma_{c_i}} H_i).$$

Finally, allowing for open loop behaviour before and after the controller operates gives,

$$\Sigma_i.\sigma_{s_i}.L(C_i/H_i).\sigma_{c_i}.\Sigma_i = \Sigma_i.\sigma_{s_i}.L(C_i).\sigma_{c_i}.\Sigma_i \cap L(\mathcal{S}^{Q_{0_i}}_{\sigma_{s_i}} \mathcal{S}^{Q_{m_i}}_{\sigma_{c_i}} H_i).$$

Lemma 5.3 $L(C_{hier} \triangle C^I/M) = \mathcal{P}^{-1}_{hier} L(C_{hier}/H_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i/H_i).\sigma_{i_c}.\Sigma_i^*\} \right)$

where $\mathcal{P}_i^{-1}$ *is an abbreviation for* $\mathcal{P}^{-1}_{\Sigma_i + \sigma_{s_i} - \sigma_{c_i}}$.

Proof

From equation 5.14,

$$L(C_{hier} \triangle C^I/M) = \mathcal{P}^{-1}_{hier} L(C_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i).\sigma_{i_c}.\Sigma_i^*\} \right) \cap \left(\bigcup_{i=1}^{n} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} M) \right)$$

Using Lemma 5.1,

$$= \mathcal{P}^{-1}_{hier} L(C_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i).\sigma_{i_c}.\Sigma_i^*\} \right)$$
$$\cap \mathcal{P}^{-1}_{hier} L(H_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1} L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} H_i) \right)$$

From equation 3.7,

$$= \mathcal{P}^{-1}_{hier}\{L(C_{hier}) \cap L(H_{hier})\}$$
$$\cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}(\{\Sigma_i^*.\sigma_{i_s}.L(C_i).\sigma_{i_c}.\Sigma_i^*\} \cap L(\mathcal{S}^{Q_{0_i}}_{\sigma_{i_s}} \mathcal{S}^{Q_{m_i}}_{\sigma_{i_c}} H_i)) \right)$$

From lemma 5.2 ,

$$= \mathcal{P}^{-1}_{hier}\{L(C_{hier}/H_{hier})\}$$
$$\cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i/H_i).\sigma_{i_c}.\Sigma_i^*\} \right)$$

Theorem 5.1 *For hierarchical systems, the closed loop behaviour generated by* C_{hier} *and* C_i *on* M *conforms to specification* S *if* C_{hier} *generates a closed loop behaviour on* H_{hier} *that conforms to specification* S_{hier} *and similarly* C_i *generates a closed loop behaviour on* H_i *that conforms to specification* S_i.

Proof

From Σ *we develop the specification* S.

However, S *is too complex to implement, therefore we decompose* S *into a set of smaller specifications,* $S^I = \{S_1, S_2, S_3, ..., S_i\}$. *To coordinate how the specifications in* S^I *interact a coordinating specification,* S_{hier} *is introduced.*

From parallel decompisition theory,

$$L(S) = \mathcal{P}^{-1}_{\Sigma_a} L(S_a) \cap \mathcal{P}^{-1}_{\Sigma_b} L(S_b)$$

And from series decomposition theory,

$$L(S)' = L(S_a) \cap L(S_a).\sigma.L(S_b)$$

where $L(S)'$ is the new adjusted specification that includes the flag event, $\sigma \notin \Sigma$, to coordinate the operation of the two specifications in series. Such that,

$$L(S) = \mathcal{P}_{\Sigma}(L(S)')$$

In hierarchical control theory,

$$L(S)' = \mathcal{P}^{-1}_{\Sigma_{hier}} L(S_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i).\sigma_{i_c}.\Sigma_i^*\} \right) \tag{5.17}$$

where $\Sigma_i \in \Sigma$,
S_i specify the operation of the process components,
$\Sigma_{hier} \cap \Sigma = \emptyset$, and
S_{hier} specifies how the specifications, S_i interact.

So, for a hierarchal control structure to conform to specification, it must satisfy,

$$L(C_{hier} \triangle C^I / M) \subseteq L(S)'$$

from which it follows that

$$\begin{aligned} \mathcal{P}_{\Sigma} L(C_{hier} \triangle C^I / M) &\subseteq \mathcal{P}_{\Sigma} L(S)' \\ &\subseteq L(S) \end{aligned}$$

Using lemma 5.3,

$$L(C_{hier} \triangle C_i / M) = \mathcal{P}^{-1}_{\Sigma_{hier}} L(C_{hier}/H_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}(\{\Sigma_1^*.\sigma_{i_s}.L(C_i/H_i).\sigma_{i_c}.\Sigma_i^*\} \right),$$

by conformance to specification,

$$\subseteq \mathcal{P}^{-1}_{\Sigma_{hier}} L(S_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(S_i).\sigma_{i_c}.\Sigma_i^*\} \right)$$

and substituting equation 5.17,

$$\subseteq L(S)' \qquad \square$$

Theorem 5.2 *For hierarchical systems, the closed loop behaviour generated by C_{hier} and C_i on M is nonblocking if,*

1. *C_{hier} generates nonblocking behaviour on H_{hier} and similarly C_i generates nonblocking behaviour on H_i*
2. *$\mathcal{P}^{-1}_{\Sigma_{hier}} L_m(C_{hier}/H_{hier})$ and $\mathcal{P}^{-1}_{\Sigma_i}\{\Sigma_i^*.\sigma_s.L_m(C_i/H_i).\sigma_c.\Sigma_i^*\}$ are nonconflicting for all $C_i \in C^I$.*

Proof

From lemma 5.3,

$$L(C_{hier} \triangle C^I / M) = \mathcal{P}^{-1}_{hier} L(C_{hier}/H_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i/H_i).\sigma_{i_c}.\Sigma_i^*\} \right),$$

if they are nonblocking, then

$$= \overline{\mathcal{P}_{hier}^{-1} L_m(C_{hier}/H_{hier})} \cap \left(\bigcap_{i=1}^{n} \overline{\mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i/H_i).\sigma_{i_c}.\Sigma_i^*\}} \right),$$

if they by nonconflicting, then

$$= \overline{\mathcal{P}_{hier}^{-1} L_m(C_{hier}/H_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1}\{\Sigma_i^*.\sigma_{i_s}.L(C_i/H_i).\sigma_{i_c}.\Sigma_i^*\} \right)}.$$

by lemma 5.3,

$$= \overline{L(C_{hier} \triangle C^I/M)} \qquad \square$$

If all alphabets, $\Sigma_{hier}, \Sigma_1, \Sigma_2, \Sigma_i, .., \Sigma_n$, are disjoint then the property of nonconflicting holds trivially. Otherwise, it must be tested for numerically.

In PCT and SCT, all elementary components are confined to the lowest level of the controller hierarchy and any higher level controllers comprise of procedural components or vocalised states respectively. However, practical work (Camillocci, 1995) showed that this need not be the case. Elementary components and procedural components can be combined in a procedural controller, as procedural and elementary components are treated in the same way during controller synthesis. The theory presented this section applies to all combinations of component types as it will be shown in the following by an example in the following section.

5.4.3 Example - CIP Heating Procedure

To illustrate the use of procedural components and further developments, an example will be used in this and the following chapters. Part of the Batch Pilot Plant in the Centre for Process Systems Engineering at Imperial College, is a Clean in Place (CIP) unit which is used for heating the cleaning fluid in a phase of a larger operation. For a full description, refer to chapter 11. Figure 5.4 shows a schematic diagram of the CIP Unit. The heating phase requires the temperature of a caustic solution to be raised to a target temperature (60 C) by recirculating it through heat exchanger HE3 using pump P6. To do so, a valve route (involving AV1-10, AV1-14, AV1-16, AV1-25, AV1-15, AV1-22, AV1-24, AV1-20 and DDV1-8) must be set and then returned to a safe position after the phase is completed. Therefore, the procedural controllers *Open Route* and *Close Route* are created. Two procedural controllers were formally synthesised (one each for *Open Route* and *Close Route*) containing elementary components representing the valves. A procedural component was then defined for each procedural controller. The (higher level) phase controller was then synthesised from the elementary component instances *P6*, *IS1-1*, *IS1-2* and *IT1-17* (level indicators and a temperature sensor, respectively) plus the two procedural component instances. Since *Open Route* and *Close Route* were formally synthesised and are thus provably correct, and the CIP Heating procedural controller was formally synthesised. Using the theory presented in section 5.4.2, we can assert that the CIP Heating procedural controller is also provably correct.

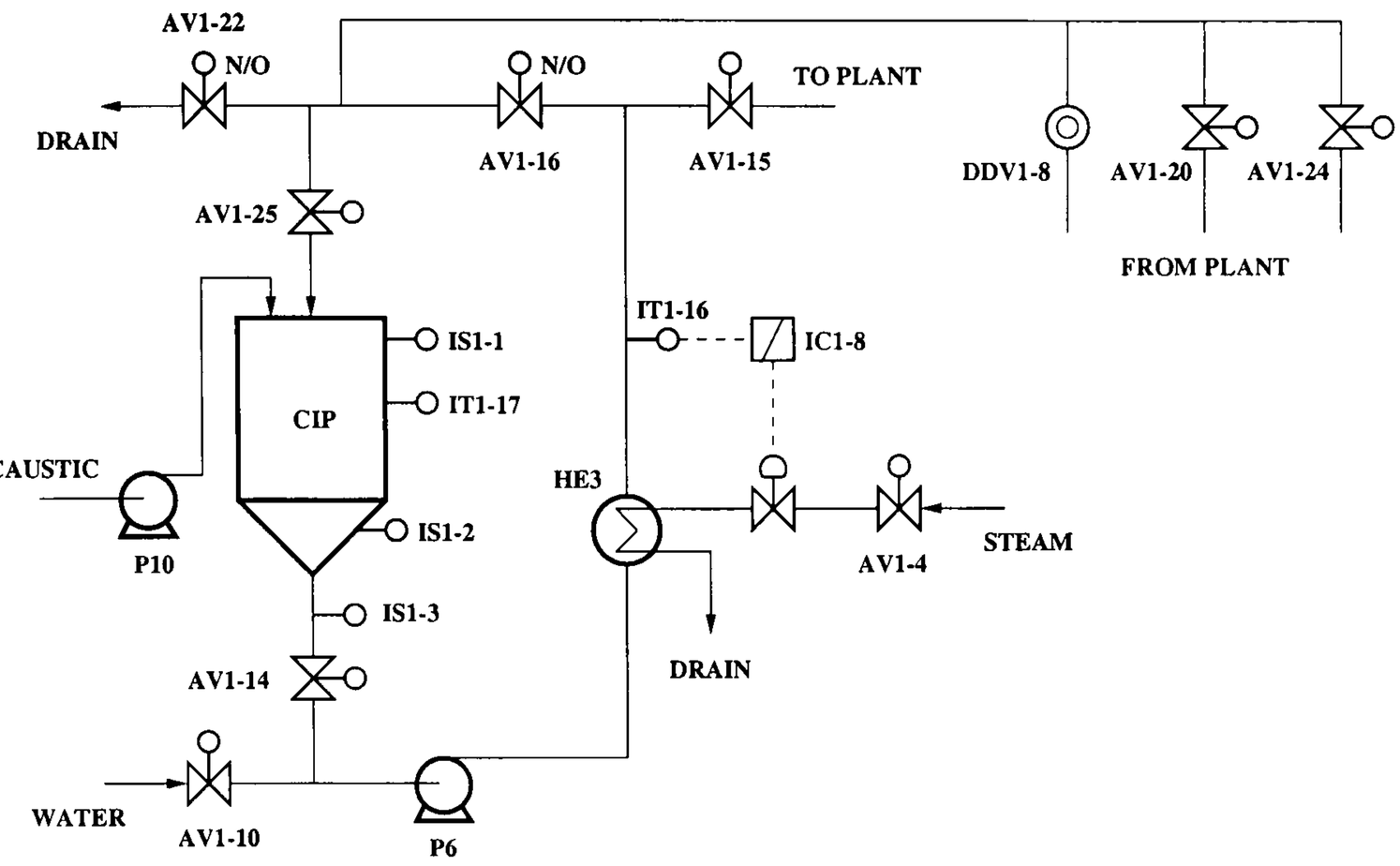


Figure 5.4: Batch Pilot Plant CIP Unit Schematic Diagram.

5.5 Interlocking Components and Reusable Controllers

Camillocci (1995) synthesised procedural controllers to be used within a specific higher level procedural controller. A more powerful procedural controller would be produced if it was synthesised so as to work independently. This makes the controller autonomous in the sense that it can check its own applicability when called. To do so, all elementary component instances involved in the procedure must be included in the synthesis possibly leading to state explosion. However, this does not occur if a significant number of the instances are interlocked while the controller executes as interlocked component instances do not increase controller size. Appropriate synthesis techniques for this case are introduced and described in chapter 9.

Including more instances in each controller does increase the risk of violating PCT decomposition rules, in particular, when component instances containing controllable transitions are shared. Interlocked instances bring uncertainty because, if the controllable transitions of an interlocked instance are not present in the controller, they are prohibited and therefore cannot be forced by another controller. The transitions are not in the language but are present in the corresponding alphabet.

Common practice, to avoid uncertainty, is to forbid controllers to share a component instance containing controllable transitions. However, theorem 5.3 below states that an exception to this occurs when controllers interlock a common component.

Theorem 5.3 *Multiple procedural controllers may share component instances which contain controllable transitions iff,*

1. *no instructions exist in any controller requesting the execution of controllable transitions in shared instances while multiple controllers are operating.*

That is $\forall \sigma \in \Sigma_a \cap \Sigma_b, \sigma \in \Sigma_{ic}$. Σ_{ic} *is the alphabet of transitions contained in component instances that are interlocked during the concurrent operation of controllers* a *and* b.

Proof *Since any language L can be broken into the concatenation of several languages, $L_1.L_2.L_3$, the proof need only be for two controllers operating in parallel because the controller that starts first can be divided into the required languages and likewise the controller that finishes last.*

From parallel decomposition theory, for two controllers (with alphabets Σ_a and Σ_b) operating in parallel to be nonblocking $\mathcal{P}_{\Sigma_a}^{-1}$ and $\mathcal{P}_{\Sigma_b}^{-1}$ must be nonconflicting and $\Sigma_a \cap \Sigma_b \subseteq \Sigma_u$.

Consider the higher controller C_H that calls the lower controller C_L. The language generated by C_H can be written as $L(C_{H_1}).L(C_{H_2}).L(C_{H_1})$ where, $L(C_{H_2})$ is the language of strings that can occur in $L(C_H)$ while C_L is operating. C_L generates the language $L(C_L)$. Now, $\exists\sigma|\sigma \in \Sigma_H \cap \Sigma_L \wedge \sigma \in \Sigma_c$. Therefore, C_H and C_L must not operate in parallel by the rules of parallel decomposition.

However, while $\sigma \in \Sigma_H \cap \Sigma_L$, if $\sigma \notin L(C_{H_2}) \wedge \sigma \notin L(C_L)$ then the two controllers can operate in parallel because both controllers forbid the controllable transition's occurrence while they are in parallel operation. The associated check for nonconflict regarding uncontrollable transitions still applies.

This can then be extended to multiple controllers operating in parallel and hence, multiple controllers can operate in parallel with shared controllable component instances iff the shared controllable transitions are forbidden.

5.5.1 Example - CIP Heating

Consider the CIP Heating procedure again. During operation, it interlocks valve *AV1-24* closed. When the procedure is run on its own there are no extra considerations. However, it forms part of a complete detergent service as shown by the hierarchy of controllers in Figure 5.5. For the whole detergent preparation procedure, AV1-24 must remain closed. Therefore, the detergent preparation operation controller interlocks AV1-24 closed, as does the CIP-Heating phase controller. Practically, this is thorough checking but it violates traditional rules of PCT as $\Sigma_{hier} \cap \Sigma_{lower} \subseteq \Sigma_{con}$. The clarification in theorem 5.3 accommodates this case as $\Sigma_{hier} \cap \Sigma_{lower} \subseteq \Sigma_{ic}$. Where Σ_{ic} is the alphabet of interlocked controllable transitions.

The pre-check in the CIP Heating controller in this case is redundant and generates some additional logic and code. However, the benefits of thorough pre-checking and autonomous procedural controllers are considered to outweigh the small increase in code size. In fact, the autonomous nature of the CIP Heating procedure means that it can be used with any of the cleaning procedures, thus resulting in a significant reduction in code generated and development time required.

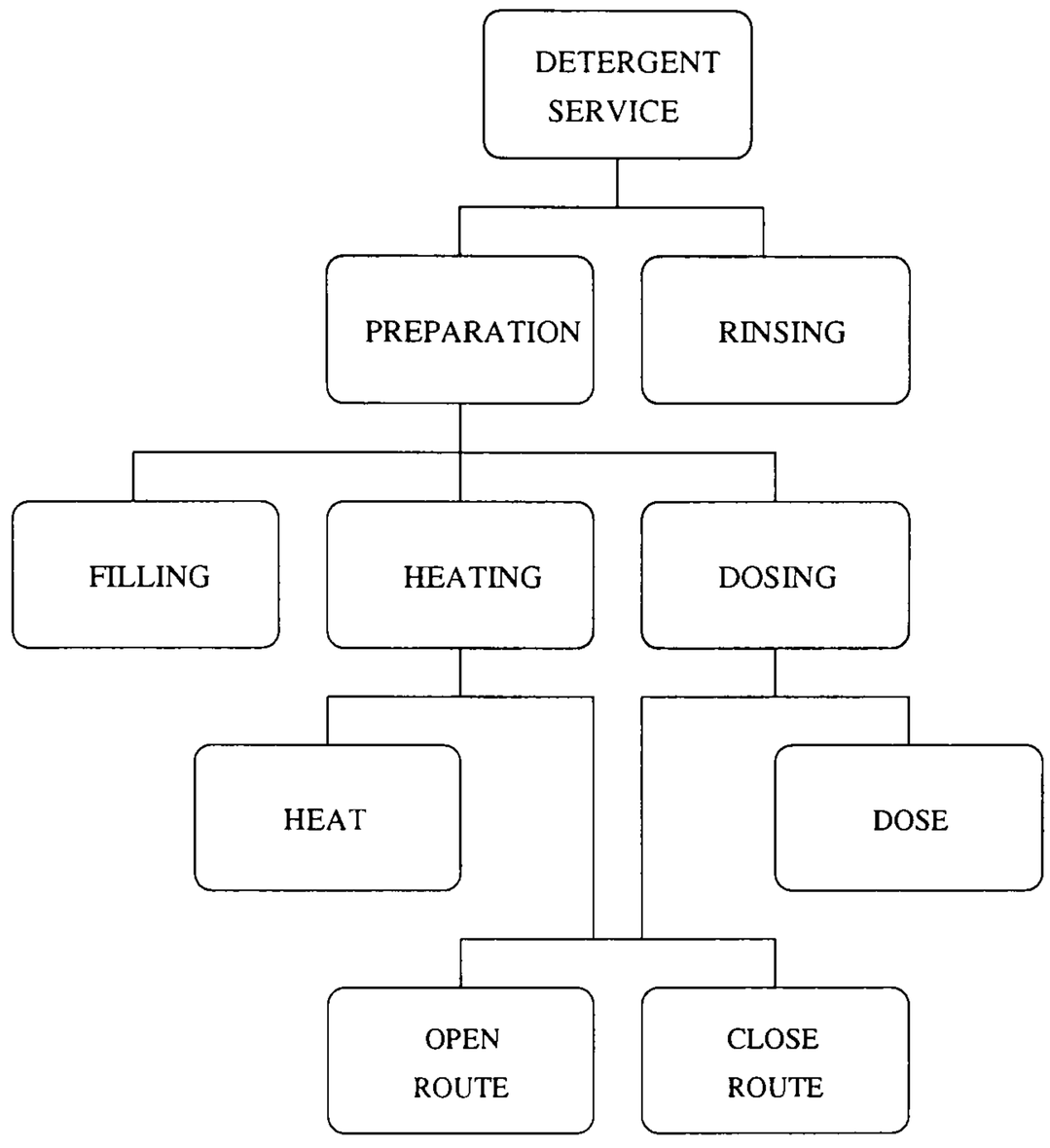


Figure 5.5: Hierarchical controller structure of the Batch Pilot Plant Detergent Service

5.6 Summary

This chapter has brought together a variety of approaches to modelling large processes as discrete event systems and developed them into hierarchical procedural controllers. Drawing from international standards, structured models of procedural controllers that form the basis for a generic hierarchical modelling approach were established.

Contributing to the reduction of size and synthesis effort, strategies have been presented to increase the robustness and reusability of procedural controllers with new theoretical clarification being given.

Further theoretical development for the decomposition of large systems and their hierarchical implementation once synthesised, was also presented.

Chapter 6

Multiple Initial States

6.1 Introduction

Process decomposition creates many small procedures which increases the complexity of the coordinating controllers that sit above them in the hierarchy. Ultimately, this may still lead to state explosion, albeit at a higher level in the controller hierarchy.

Quite often, several of the lower level controllers created will operate on the same process with very similar operating specifications, only differing in their initial states. Combining controllers into a single controller would reduce the risk of state explosion as well as ease the understanding and the maintenance of the controllers.

This chapter provides logical reasoning for why multiple initial states are needed in procedural controllers. It achieves this by reducing the size and number of controllers required to meet a control objective. A theoretical proof is developed, showing that procedural controllers with multiple initial states satisfying all the criteria of PCT can be synthesised. These controllers contribute to reducing the risk of state explosion and ease the designer's burden by preventing a control objective being spread over several controllers when a single one is sufficient.

6.2 Procedural Controllers with Multiple Initial States

Allowing an FSM to have multiple initial states enables several procedural controllers to be condensed into one. Figure 6.1 illustrates the benefits with reference to two possible initial states, A and B of the simple push button of the buffer tank example in chapter 3. Normally, two procedural controllers would be developed. If state A is the initial state then state B would be removed when the trim structure is synthesised, as it can not be reached from A. On the other hand, if B is the initial state, A is unreachable and removed. Rather than creating two separate procedural controllers, an FSM with multiple initial states and the ability to select the active state is desired.

Selecting the correct *active* initial state is done by means of the Statechart concept of *connectors*. As discussed in chapter 4, connectors are pseudo initial states added to a state

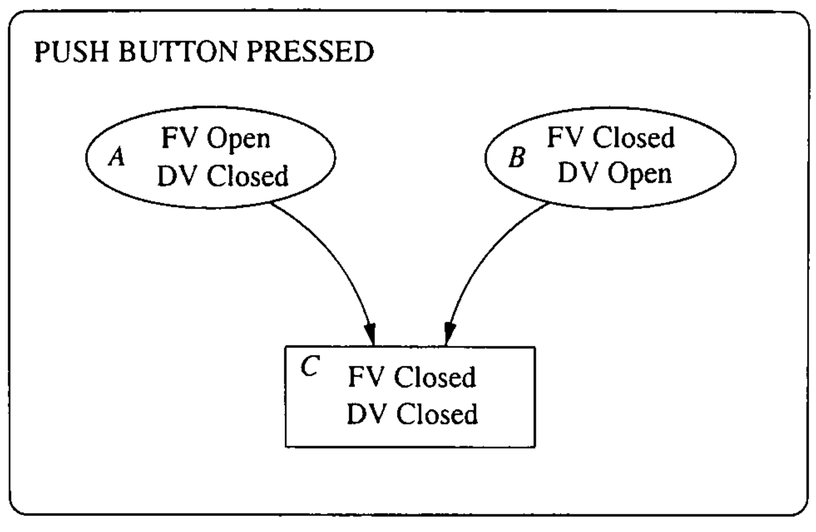


Figure 6.1: An Example of the benefits of Multiple Initial States using a Procedural Controller for when a Push Button is pressed.

machine in order to decide which is the active initial state. From a connector, conditional transitions, like pre-checks, go to the respective initial states. Figure 6.2 demonstrates how a connector state is applied to the machine of Figure 6.1.

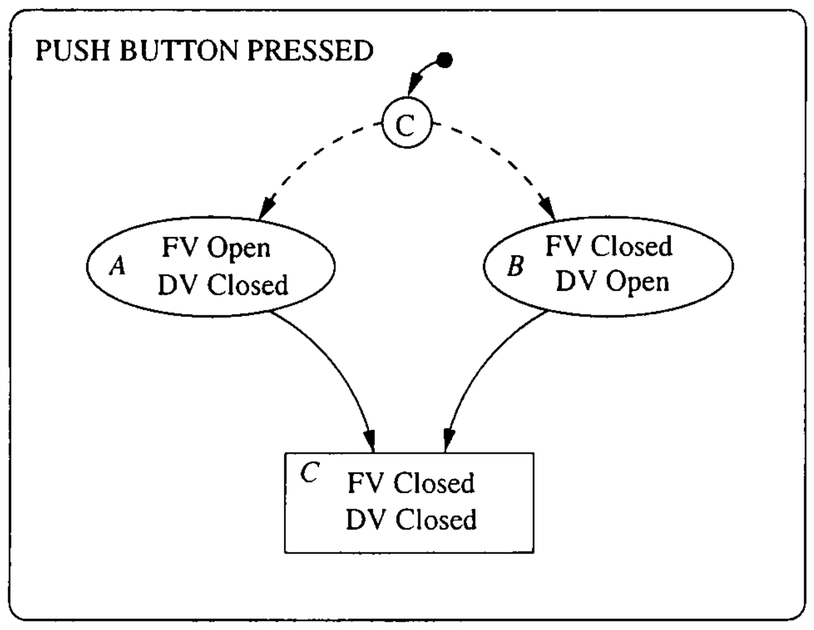


Figure 6.2: Using connectors to create a pseudo initial state.

The pre-check theory of Alsop (1996) used similar concepts to check initial states in PCT. However, pre-check theory has only been applied to procedural controllers with single initial states. It is possible to apply pre-checking to procedural controller with multiple initial states but first let us consider the definition of a procedural controller with pre-checking.

The closed loop behaviour of a system M under control of C_x with pre-checks is defined by equation 6.1.

$$L(C_x/M) = \Sigma^*.\sigma_x.L(C_x) \cap L(\mathcal{S}^{Q_x}_{\sigma_x} M). \tag{6.1}$$

Now consider a procedure with the initial states q_1 and q_2, and operating specifications S_1 and S_2. The specifications differ only by their initial states. To synthesise a controller, subsystem models, H_1 and H_2, are required:

$$H_1 = \{Q, V^{n_v}, \Sigma, \delta, \gamma, q_1, Q_m\} \quad (6.2)$$
$$H_2 = \{Q, V^{n_v}, \Sigma, \delta, \gamma, q_2, Q_m\}. \quad (6.3)$$

Note that the only difference between the models is their initial state. Specifications are then applied, S_1 to H_1 and S_2 to H_2, and controllers C_1 and C_2 synthesised:

$$C_1 = \{X, \Sigma, \xi, x_1, X_m\} \quad (6.4)$$
$$C_2 = \{X, \Sigma, \xi, x_2, X_m\}, \quad (6.5)$$

resulting in the associated closed loop behaviour:

$$L(C_1/H_1) = L(C_1) \cap L(H_1) \quad (6.6)$$
$$L(C_2/H_2) = L(C_2) \cap L(H_2). \quad (6.7)$$

Adding pre-checks for a single initial state yields,

$$L(C_1/H_1) = \Sigma^*.\sigma_1.L(C_1) \cap L(\mathcal{S}_{\sigma_1}^{q_1} H_1) \quad (6.8)$$
$$L(C_2/H_2) = \Sigma^*.\sigma_2.L(C_2) \cap L(\mathcal{S}_{\sigma_2}^{q_2} H_2). \quad (6.9)$$

Considering only C_1 gives,

$$\begin{aligned} L(C_1/H_1) &= \Sigma^*.\sigma_1.L(C_1) \cap L(\mathcal{S}_{\sigma_1}^{q_1} H_1) \\ &= \Sigma^*.\sigma_1.L(C_1) \cap L(\mathcal{S}_{\sigma_1}^{q_1} M) \;\; Since H_1 = \mathcal{S}_{\sigma_1}^{q_1} M \\ &= \Sigma^*.\sigma_1.L(C_1) \cap L(\mathcal{S}_{\sigma_1}^{q_1} \mathcal{S}_{\sigma_2}^{q_2} M). \end{aligned}$$

Likewise, for C_2,

$$L(C_2/H_2) = \Sigma^*.\sigma_2.L(C_2) \cap L(\mathcal{S}_{\sigma_1}^{q_1} \mathcal{S}_{\sigma_2}^{q_2} M).$$

The above shows that the same process model can be used to synthesise multiple procedural controllers and, therefore, the initial state is a property that belongs to the specification and not to the model. To do so, the start transitions associated with each initial state must be selflooped at the respective states in the process model. This now raises the question of whether process models should have multiple initial states in their definition or indeed any initial state at all.

6.3 A New Approach for Process Modelling

In DES theory, a process M can be modelled as an FSM described by the 7-tuple:

$$M = \{Q, V^{n_v}, \Sigma, \delta, \gamma, q_0, Q_m\} \tag{6.10}$$

where, q_0 and Q_m are the set of initial states and marked states respectively. From a formal viewpoint, a DES can be thought of as a dynamic system. This is an entity with a state space and a state transition structure. Because q_0 and Q_m are defined as part of the operating specification and used at the stage of controller synthesis, there is no need to consider explicitly the initial and marked states in the process model. Then M can be represented as the 5-tuple:

$$M = \{Q, V^{n_v}, \Sigma, \delta, \gamma\} \tag{6.11}$$

According to this new representation, H_1 and H_2 in equations 6.2 and 6.3 are equal and $M = H_1 = H_2$. Then operating specifications that differ only by their initial states can be applied to a single process model, M.

6.4 Synthesis of Controllers with Multiple Initial States

Defining individual controllers on a single model for each initial state (of which at any stage only one will actually execute) is an arduous and somewhat redundant task. Synthesis of a procedural controller that generated the language of the combined controllers results in a large reduction in controller size, let alone the effort required during synthesis.

In order to simplify the synthesis procedure, a controller FSM containing multiple initial states and its associated language is now defined as:

$$C = \{X, \Sigma, \xi, X_0, X_m\} \tag{6.12}$$

where

X_0 is the set of controller initial states, $x_i \in X_0$

The language $L(C)$ generated by the FSM C is the set of every possible transition string from every initial state in X_0:

$$L(C) = \{s \in \Sigma^* | \ \xi(s, x_i)! \wedge x_i \in X_0\}. \tag{6.13}$$

Accordingly,

$$L_m(C) = \{s \in \Sigma^* | \ \xi(s, x_i) = X_m \wedge x_i \in X_0\}. \tag{6.14}$$

Equations 6.13 and 6.14 are evolutions of equations 3.2 and 3.3 and are equivalent to the union of several single initial state controller FSMs:

$$L(C) = \bigcup_{i=1}^{n} L(C_i), \tag{6.15}$$

$$L_m(C) = \bigcup_{i=1}^{n} L_m(C_i), \tag{6.16}$$

where

n is the number of single initial state controllers.

C_i is the procedural controller with a single initial state x_i.

To determine which state is the active initial state, pre-checks must be added to the controller. Now, several start transitions are possible from the pseudo initial state and so equation 6.1 is changed to give:

$$L(C_x/M) = \Sigma^*.\Sigma_\sigma.L(C_x) \cap L(\mathcal{S}_{\sigma_x}^{Q_x} M) \tag{6.17}$$

where Σ_σ is the alphabet of start transitions and $\Sigma_\sigma \cap \Sigma = \varnothing$.

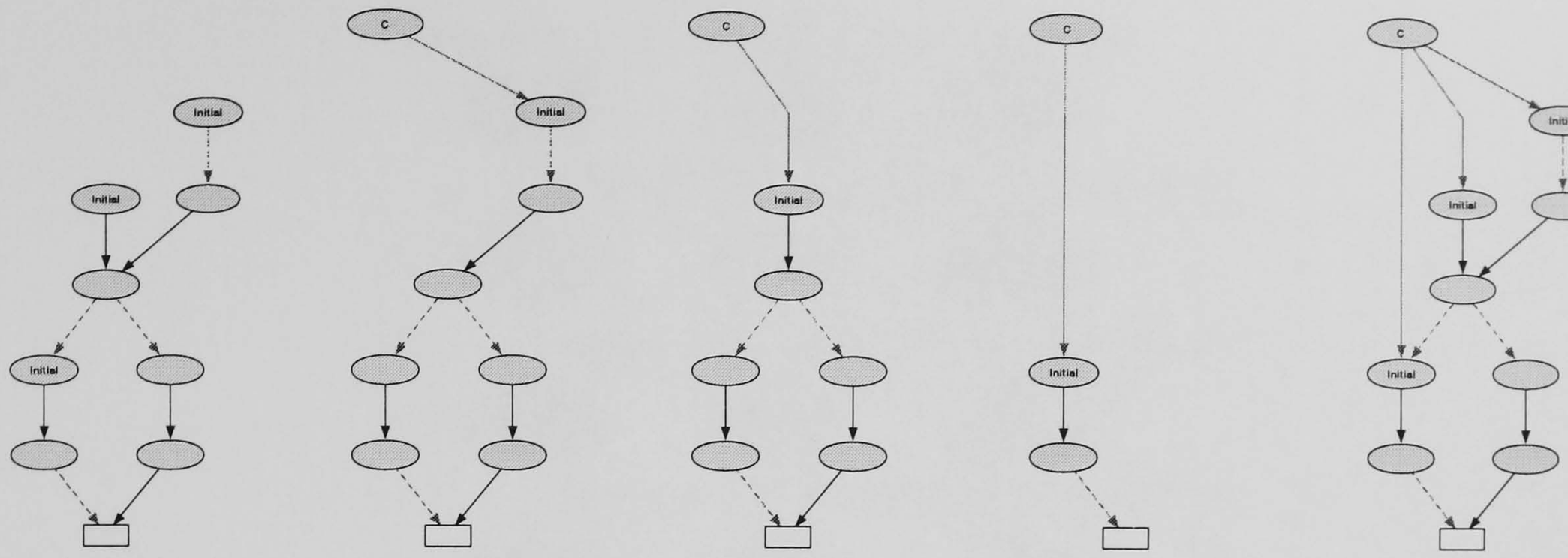
A graphical example is given in Figure 6.3. The FSM shown in Figure 6.3(a) has 9 states, three of which are potential initial states. Traditionally, this FSM was implemented by three FSMs as shown in Figure 6.3(b), (c) and (d), together comprising 19 states. Before even considering the additional effort associated with organising these controllers, the number of states synthesised has doubled. It is observed that the FSM in Figure 6.3(a) generates the same language as the union of the three subsequent FSMs. Figure 6.3(e) shows how only one of the initial states will be satisfied at any application of the FSM, thus breeding easier understanding. It is observed graphically, that the pseudo initial state is the same as a connector in StateCharts.

6.4.1 Example - CIP Heating

To illustrate the benefits of using a single process model and controller with multiple initial states, recall the *CIP-Heating* procedure. A key component instance for this procedure is the temperature sensor $IT1-17$. It is an analogue sensor which has been discretised into 4 states: *Extra Cold*, *Cold*, *Warm* and *Hot*. The procedure marked state includes $IT1-17$ as Hot.

The temperature of the fluid in the CIP tank is assumed to be *Cold* initially. However, it is acceptable for the procedure to start when it is *Extra Cold* or *Warm*. Therefore, there are three possible initial states.

Traditionally, three separate procedural controllers are synthesised, each with the number of constraints and states listed in tables 6.1 and 6.2 respectively. A coordinating



(a) C_{TOT} (b) C_1 (c) C_2 (d) C_3 (e) C'_{TOT}

Figure 6.3: Synthesising Controllers containing Multiple Initial States

procedural controller is also required to execute the correct procedural controller. The total number of states generated in the model is 6252 and a procedural controller of 85 states is produced. One procedural controller with multiple initial states replaces the four controllers mentioned above, as shown in Figure 6.4. It required a model of 2048 to be generated and synthesised a procedural controller of 42 states. The following benefits resulted:

- Only one procedural controller is synthesised whose size is comparable to each of the others with the consequent reduction of the modelling, specification and synthesis tasks to approximately a quarter.
- No coordinating controller is required.
- An overall reduction in the number of states is obtained as shown by table 6.2.

However, the controller still contains parallel trajectories of identical transition strings. These create needless redundancy, larger controllers and in turn more control code than necessary. This is because PCT calculates every state variable value in every state explicitly. The next section addresses the overcoming this drawback.

6.5 Initial States with Covered State Variables

An approach is now proposed to minimise parallel transition string redundancy by allowing states where not all state variable values are explicitly known. The size of controllers can be greatly reduced if during their execution state variables are covered but not explicitly known. The controller meets all the formal criteria of PCT while being more easily understood.

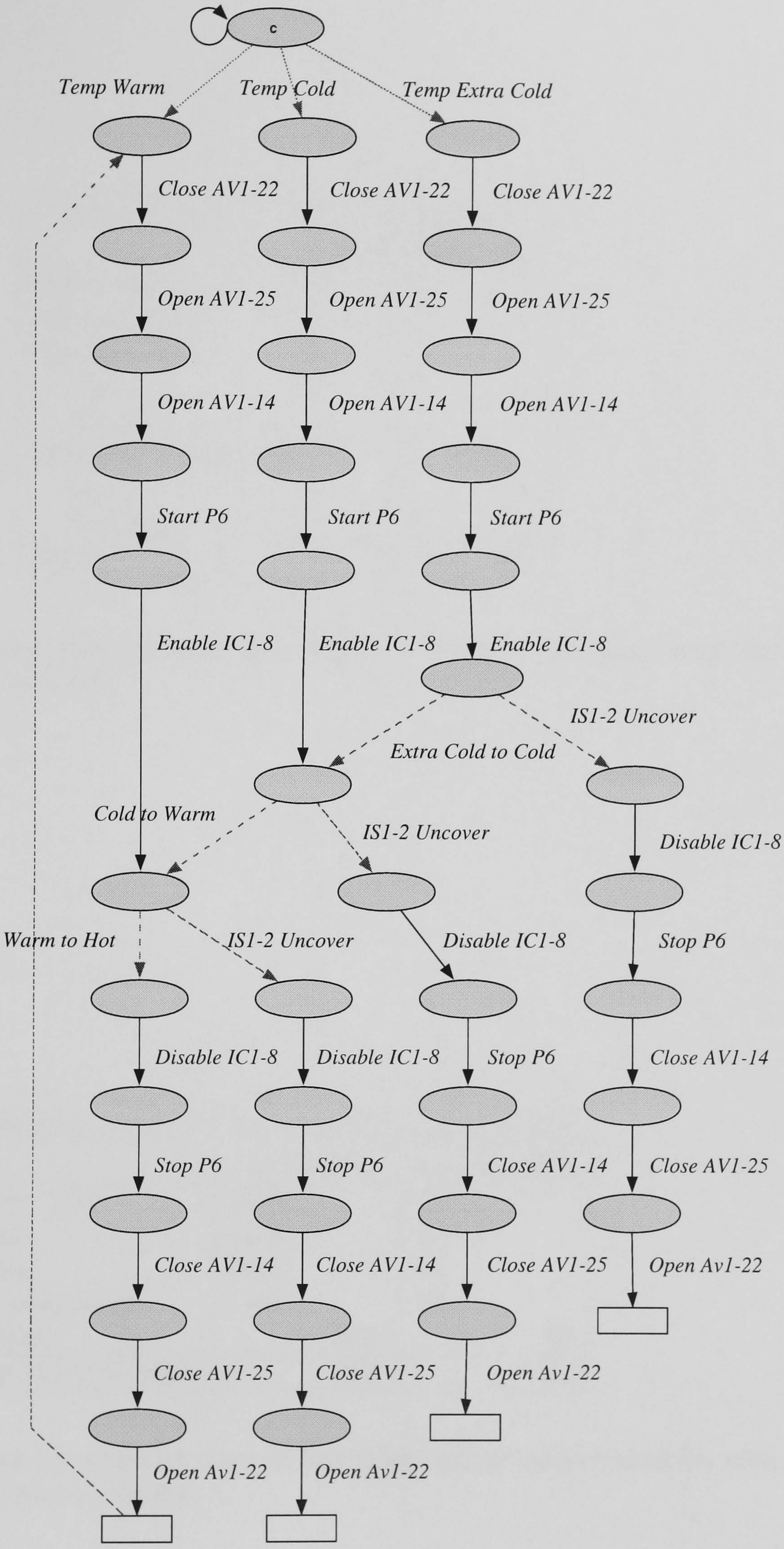


Figure 6.4: CIP Heating Controller with Multiple Initial States

Constraint Type	Number	
	Single	Multiple
Initial States	1	3
Marked States	2	2
Specifications		
Static	5	5
Dynamic	6	6
Modelling Constraints		
Static	0	0
Dynamic	3	3
Total	17	19

Table 6.1: CIP Heating example synthesis statistics for each single initial state and a multiple initial state controller.

CONTROLLER	No. of States in Model	No. of States in Controller
Very Low	2048	32
Low	2048	25
Okay	2048	18
Coordinating	108	10
Total	6252	85
Multi Initial State	2048	42

Table 6.2: Comparison of controller sizes of four single initial state controllers with a single multiple initial state controller.

Pre-check algorithms are required when a controller is initiated to ensure that all elementary components are in the correct state. Initially, the controller has no knowledge of the state variable values, as signified by the first state of the FSM in Figure 6.5(a). The ∞ symbol represents any value of the elementary components instance. Pre-checks are performed by transition 101 and then the controller begins execution of the operating procedure. Likewise, a case with multiple initial states is shown in Figure 6.5(b).

In a procedural controller, the state variable values of each elementary component instance are explicitly known in every state. A relaxation of this rule is proposed which allows an elementary component instance state variable to have any of several values in each state of the procedural controller. This is termed Partial State Knowledge (PSK).

Partial State Knowledge enables identical parallel transition trajectories to be condensed into a single trajectory if elementary component instance states can be represented by a covered variable. This results in smaller procedural controllers which are easier to understand. For example, Figure 6.5(b) can be reduced to Figure 6.5(c) because the state of the first elementary component instance can be represented by a partially covered variable (*Medium/Low*) until the fifth state in Figure 6.5(c).

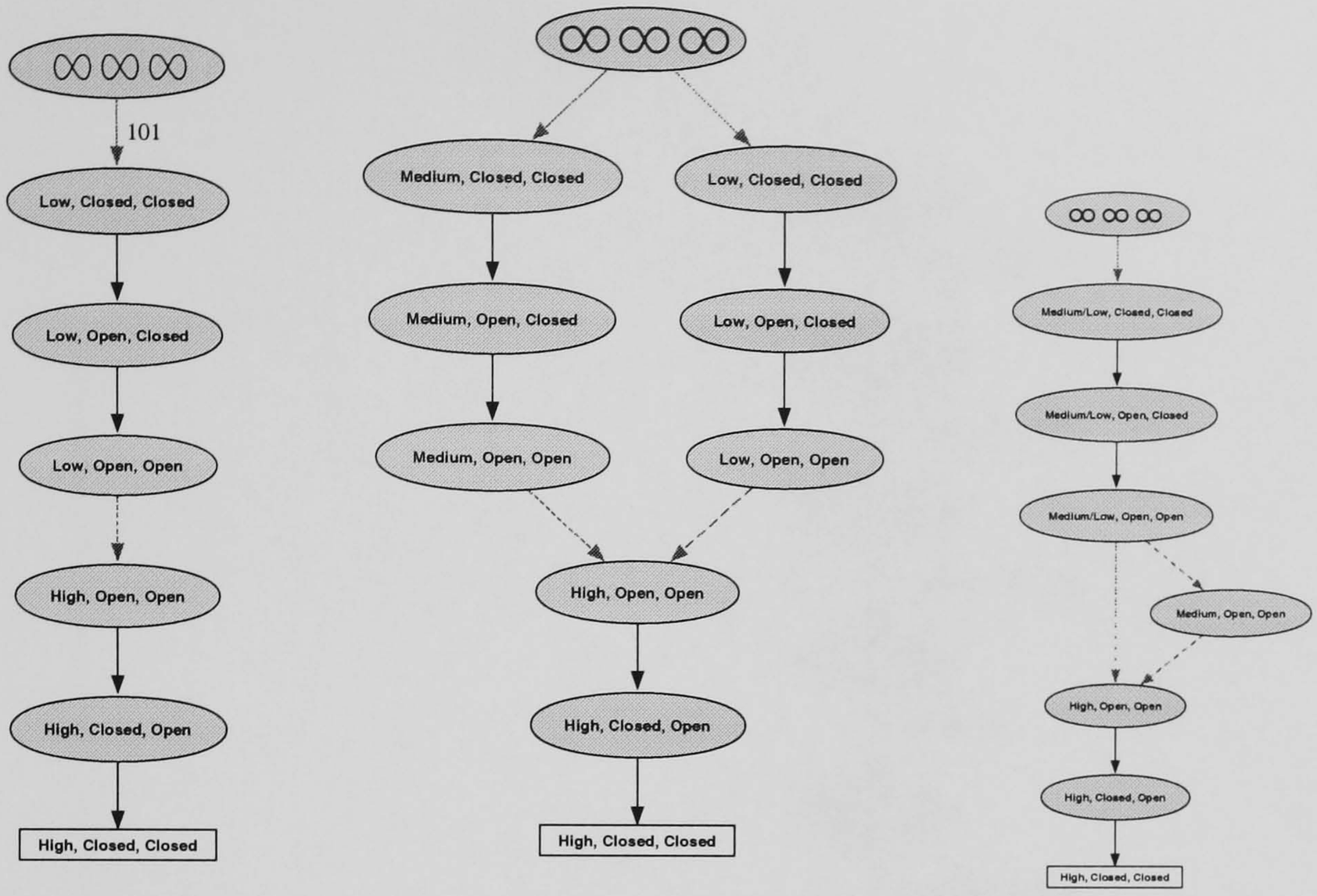
In the CIP Heating example of section 6.4.1, the temperature value was not explicitly required. It would have been sufficient to know that it was *not Hot*. The operating specifications for the next five transitions were identical for each of the three initial states. On sensing that *IS1-2* has become covered or *IT1-17* becoming hot, the transitions that followed were identical. Thus, the 42 state controller of Figure 6.4 could be reduced to a 14 state FSM as shown in Figure 6.6.

6.5.1 Theoretical Validation

The theoretical basis for PCT is founded in language theory. FSMs are one method of representation but it is the language generated by them that is most important. Therefore, since the FSMs in Figures 6.4 and 6.6 generate the same languages, they achieve the same results. The FSM in Figure 6.6 is an acceptable FSM for the purpose of calculating the closed loop behaviour and other PCT requirements. The difficulty arises when considering how to synthesise such an FSM from elementary components.

6.5.2 Implementation Issues

PCT's strength lies in the fact that very complex models and provably correct controllers that consider every possible state of the system can be synthesised from simple elementary components. To accomplish this incorporating covered states without endangering this property requires sophisticated defining and implementation. So far, good progress has been made, as part of this thesis, in software to enable such a synthesis procedure from an initial state until the first uncontrollable transition occurs. Thus, Figure 6.4 can be reduced to Figure 6.7. While the reduction to the minimal network of Figure 6.6 has not been implemented yet, this is already a significant step forward.



(a) (b) (c)

Figure 6.5: How Partial State Knowledge helps to reduce procedural controller size.

Combined with developments presented in the next chapter as part of Partitioning Theory, Partial State Knowledge procedural controllers are very powerful in their current implementation. Partitioning Theory decomposes a procedural controller into smaller controllers using failures as the boundaries. That is, when a component fails or recovers, the active controller is stopped and another is started. Frequent switching between controllers makes the initial states of each one very important. It also increases the number of possible initial states of each controller.

The use of multiple initial states reduces the number of these smaller controllers and Partial State Knowledge reduces their size. As the number of failures that can occur at any stage increases, the number of potential initial states also increases, making multiple initial states very useful. Moreover, emergency procedures tend to be the same for several initial states. This would create procedural controllers with several identical parallel transition trajectories which are perfect candidates for the use of PSK.

In the CIP Heating procedure, the transitions of level switch *IS1-2* between *covered* and *uncovered* are failure transitions. The operating specification is partitioned into two regions: *Region A, IS1-2* covered and *Region B, IS1-2* uncovered. The procedural controller for region A implements a partial knowledge procedural controller with the temperature sensor IT1-17 covered. In region B, a partial knowledge procedural controller is synthesised where all states are possible initial states. Region A has nine component instances

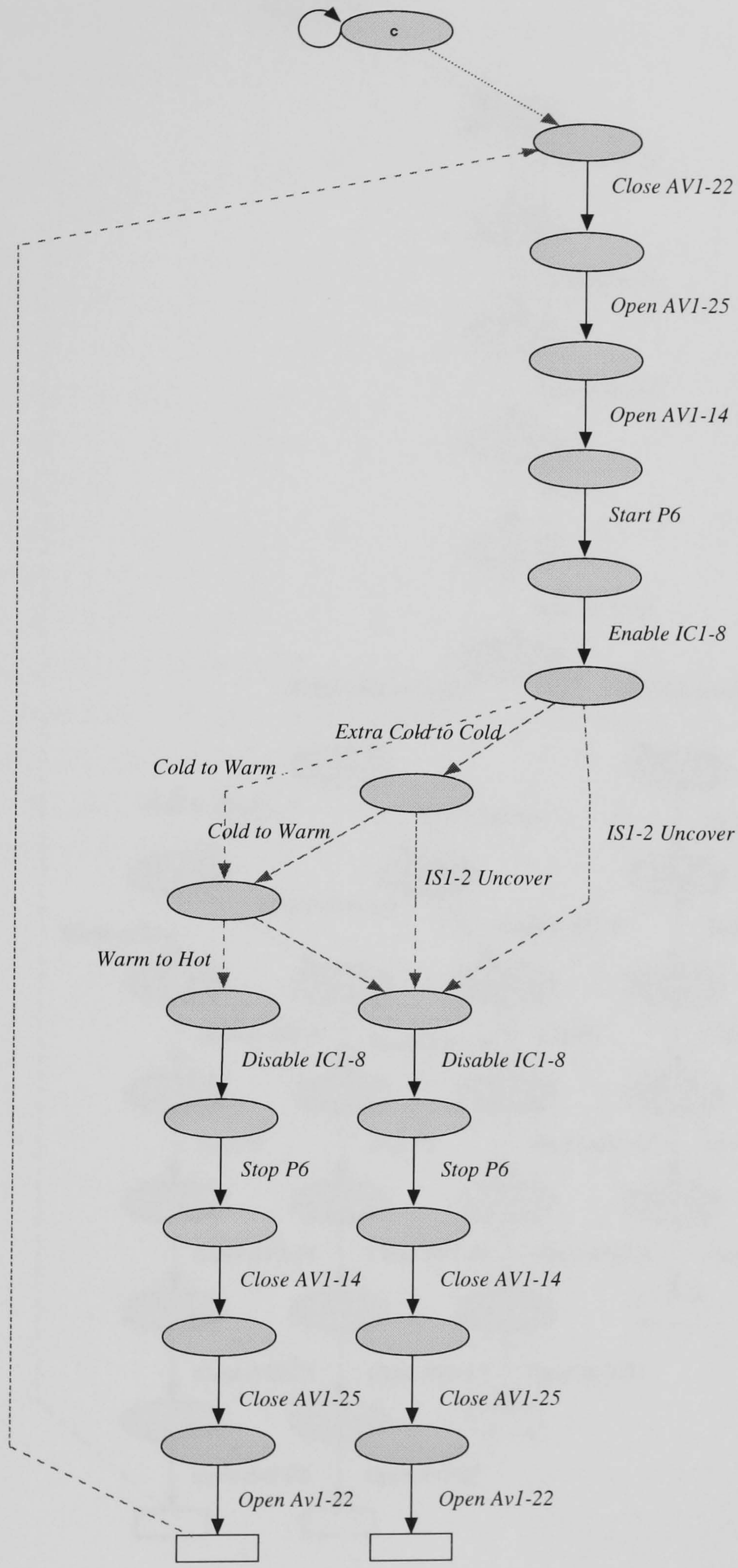


Figure 6.6: CIP Heating Controller with Partial-Knowledge Initial States

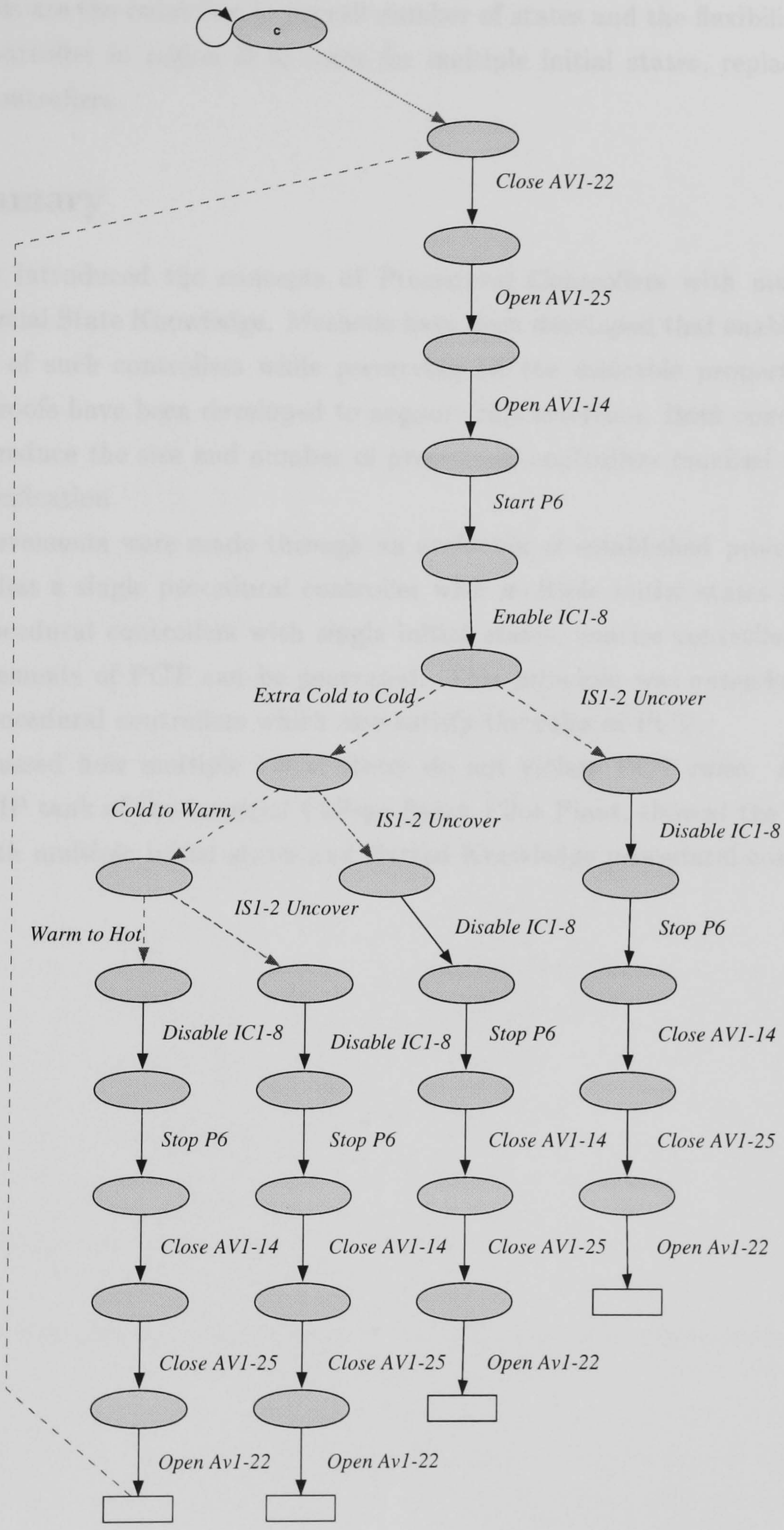


Figure 6.7: The implementable CIP Heating Controller with covered Initial States but no compression after an uncontrollable transition has occurred.

and Region B has eight. The controller is shown in Figure 6.8. Important achievements of the synthesis are the reduction in overall number of states and the flexibility of a single procedural controller in region B to cater for multiple initial states, replacing 6 single initial state controllers.

6.6 Summary

This Chapter introduced the concepts of Procedural Controllers with multiple initial states and Partial State Knowledge. Methods have been developed that enable the proper consideration of such controllers while preserving all the desirable properties of PCT. Theoretical proofs have been developed to support this assertion. Both concepts help to dramatically reduce the size and number of procedural controllers required to satisfy an operating specification.

These achievements were made through an evolution of established pre-check theory. By showing that a single procedural controller with multiple initial states is equivalent to several procedural controllers with single initial states, concise controllers that have all the requirements of PCT can be generated. This principle was extended to Partial Knowledge procedural controllers which also satisfy the rules of PCT.

It was discussed how multiple initial states do not violate PCT rules. An example, heating the CIP tank of the Imperial College Batch Pilot Plant, showed the validity and benefits of both multiple initial states and Partial Knowledge procedural controllers.

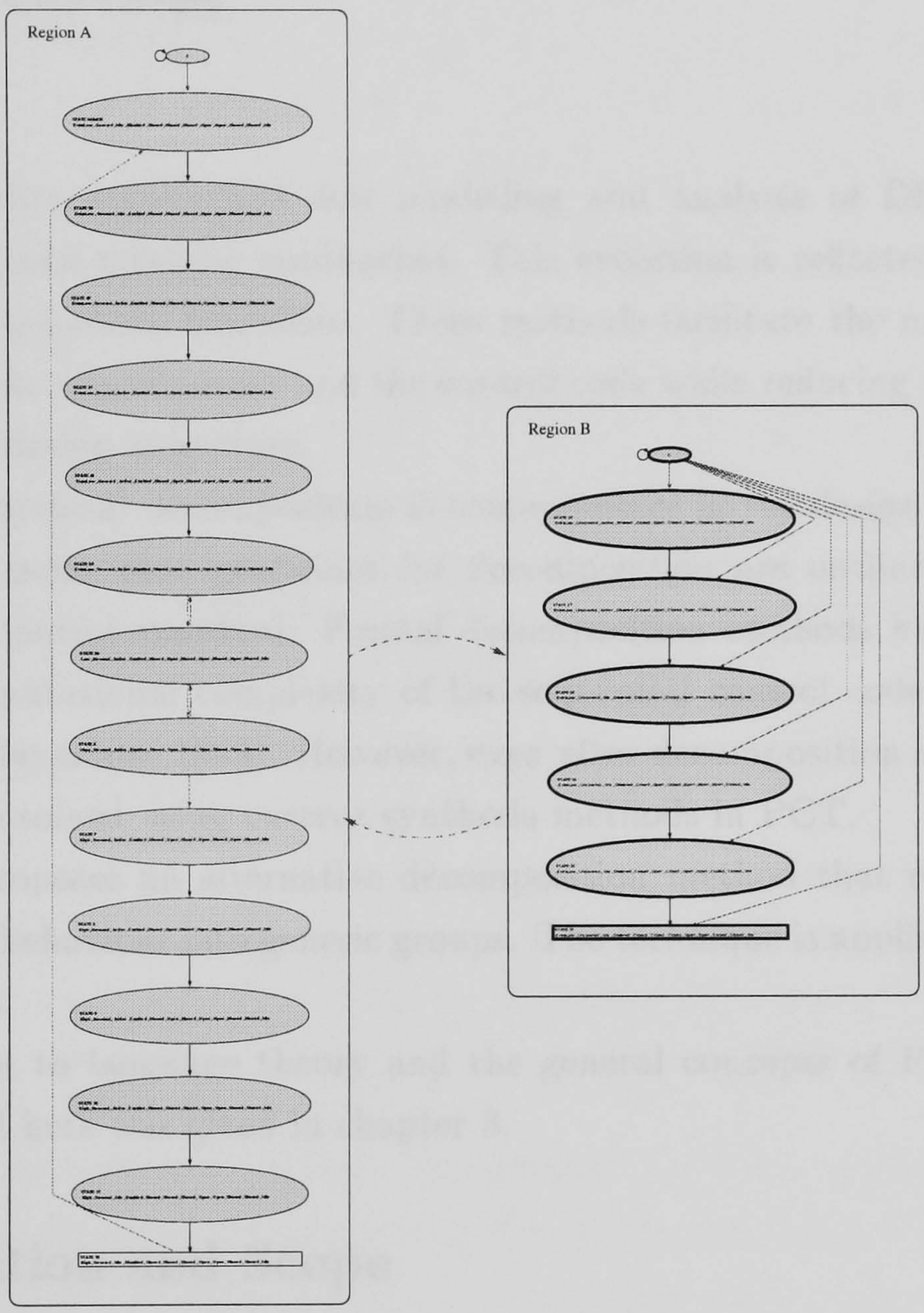


Figure 6.8: CIP Heating Controller with Partitioning and Multiple Initial States

Chapter 7

Partitioning

The literature review emphasised how modelling and analysis of DESs is moving towards hierarchical and modular approaches. This evolution is reflected in the industrial software used for sequential processes. These methods facilitate the modelling task, the understanding of the specifications and the control code while reducing the computational complexity of the design procedure.

Nowadays, hierarchical decomposition is commonplace in the design of sequential control code, to the point that guidelines for decomposition are outlined in the ISA-S88 batch processing control standard. Formal decomposition methods have been proposed to tackle the computational complexity of the sequential control code synthesis (Alsop, 1996; Brave and Heymann, 1993). However, even after decomposition some problems are too complex to be solved using current synthesis methods in PCT.

This chapter proposes an alternative decomposition method that relies on the partitioning of process behaviour into generic groups. The technique is applicable to the widest range of problems.

An introduction to language theory and the general concepts of Procedural Control Theory to be used here was given in chapter 3.

7.1 Motivation and Scope

In PCT, a system model is constructed from elementary models of its sensors and actuators. The system state is described by the combination of the elementary component states. In order to deal with the common component malfunctions and other abnormal situations that arise in industrial process plants, elementary component models incorporate failure events. As a result, their size grows and the overall process model risks state explosion. A small number of elementary components combine to create a process model too large and complex to be calculated using current methods. Moreover, in PCT, failures cannot be preempted by controllable transitions like normal uncontrollable transitions thus creating larger controllers and aggravating the state explosion problem. The resulting procedural controllers also cannot be directly translated into control code due

states being permitted to have both controllable and failure transitions exiting from them.

However, component failures are unique in the sense that when they occur, they change the potential behaviour of a process by restricting the behaviour of the failed component. For example, a valve normally modelled in a process with the freedom to move between open and closed could fail in a closed position. The system behaviour is then restricted to states where the valve is closed, half those prior to the failure.

The main insight driving the new decomposition approach proposed here is the recognition of the important significance of failures in the behaviour of a process and the role they play in the design of procedural controllers. Rather than mixing all normal and abnormal behaviour in a single model, one can attempt to organise the model into separate regions of normal and abnormal behaviour. This approach has been fully recognised in engineering practice. Moreover, industrial experience suggests that responding to failures involves 80% (on average) of the total control code generated.

7.1.1 Definitions

In order to formalise the approach and define its scope, we introduce the following definitions regarding different types of failures.

Definition 7.1 Failure Transition: $\Sigma_f \subseteq \Sigma_u$. *An uncontrollable transition that forces the system to move from normal to abnormal operation.*

Definition 7.2 Fatal Failure: $\Sigma_{ff} \subseteq \Sigma_f$. *A failure transition that causes the system to cease operation.*

Definition 7.3 Non-Fatal Failure: $\Sigma_{nf} \subseteq \Sigma_f$. *A failure transition which does not cause the system to cease operation. That is, measures can be taken locally that allow the system to continue operation.*

Definition 7.4 Independent Failure Transition: $\Sigma_{if} \subseteq \Sigma_f$. *A failure transition that can occur at any stage of the process, irrespective of the state the process.*

Definition 7.5 State Dependent Failure Transition: $\Sigma_{df} \subseteq \Sigma_f$. *An uncontrollable transition which is a failure only when it occurs from certain system states.*

Definition 7.6 Recovery Transition: $\Sigma_r \subseteq \Sigma_u$. *An uncontrollable transition that causes the system to move from an abnormal operation to normal operation.*

In this thesis, we focus on *Independent* and *Fatal Failures*. Development of design methods capable of dealing with *Non-Fatal Failures*, *State Dependent Failures* and *Recovery Transitions* are suggested as subjects for future research.

7.1.2 Hierarchical Model Development

As discussed in chapter 2, academic research and engineering practice agree that there exists an underlying structure of *normal*, *abnormal* and *forbidden operations* within process systems models. However, previous theoretical work has not exploited this underlying structure. For instance, uncontrollable events such as a sensor detecting a *normal* level in a tank, are not distinguished from failure events such as the indication of a valve getting stuck. Similarly, although failures are specifically distinguished in engineering practice, no formal mechanism is provided for their analysis or design.

We shall show in the following, how this underlying structure can be formally defined, captured and analysed by combining the statecharts modelling approach and the PCT analysis framework.

7.2 Applying Statecharts to PCT

PCT applications, in general, have followed a hierarchical, modular approach. Decomposition methods have been suggested which ensure that, when permitted, two controllers operate in parallel without hindering each other. Also, guidelines for serial decomposition and controller interlocking have been provided. The concurrent operation of two procedural controllers can be captured using a separate AND superstate for each of them. The serial decomposition can be formalised using OR superstates. Statecharts can easily represent both decomposition cases while offering the opportunity for a more concise representation. Moreover, they can make the modelling task more intuitive for the engineer. Being a generalisation of FSMs makes their inclusion into PCT an avenue worthwhile exploring.

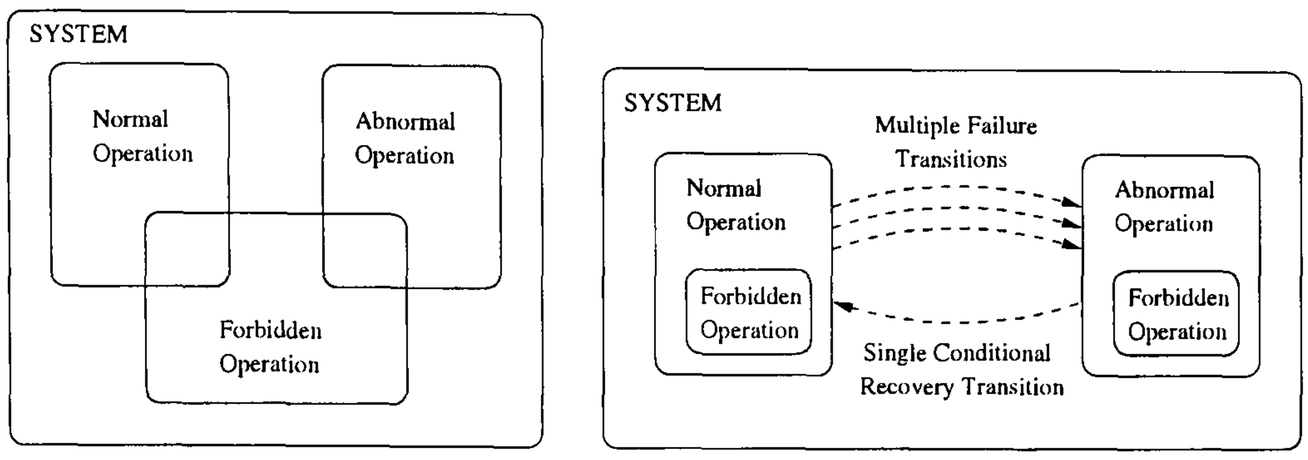
The generic statechart representation for the underlying structure of a process under procedural control is shown in Figure 7.1(a). All levels of the ISA-88 standard's hierarchy can be modelled using this underlying structure, from unit procedures down to the individual elementary components. In the figure, the *normal* and *abnormal operating range superstates* are OR-states, emphasising that the system cannot be in both normal and abnormal operation concurrently. Overlapping both states is the *forbidden range* superstate.

Formally, we define the above mentioned three states and operating ranges as:

Definition 7.7 Normal State: *A state of a process or elementary component that is expected under normal operation and is considered acceptable under all circumstances.*

Definition 7.8 Abnormal State: *An undesired state of a process or elementary component that is reached following some exceptional event. It may be entered but signifies that a failure has occurred and action should be taken to return to a normal state.*

Definition 7.9 Forbidden State: *A state of a process or elementary component that is physically realizable but is never to be entered.*



(a) (b)

Figure 7.1: Statecharts of common process structure.

Definition 7.10 Normal Operating Range: *The superstate of all states where all components are in a normal state and the combinations of states are expected in normal operation.*

Definition 7.11 Abnormal Range: *The superstate of all states where at least one component is in an abnormal state or a combination of state variables creates a uniquely abnormal state.*

Definition 7.12 Forbidden Range: *The superstate of all states where either at least one component is in a forbidden state or a combination of state variables creates a unique forbidden state.*

Note that overlapping states do not exist within statecharts, so forbidden states are removed independently from the process model and specification, as shown by Figure 7.1(b).

7.3 Partitioned Elementary Components

PCT is based on building process models and procedural controllers from elementary components. To preserve this property, the partitioning of the system starts with the partitioning of the elementary components. Each component can, like the overall process, be modelled with normal, abnormal and forbidden ranges (components do not necessarily have states in all ranges). A statechart is produced with superstates containing FSMs which describe the behaviour of the component in each of the ranges. These FSMs are then used to synthesise procedural controllers for the normal, abnormal and forbidden regions of the overall process.

The decision regarding normal, abnormal and forbidden behaviour is very much dependent on the specification and thus varies from specification to specification. In an attempt to maintain formality and a generic approach, the choice to partition a specification into normal, abnormal and forbidden behaviour was made as they are common to

all processes. The definition of elementary components and their partitioning has been formalised in a method outlined by Procedure 7.1.

Examples of some elementary components are shown in Figure 7.2. Figure 7.2(a) shows the synthesis of a valve with failures, Figure 7.2(b) being its equivalent FSM as modelled by Sanchez (1994). A level sensor is modelled also in Figures 7.2(c) and (d).

The valve statechart in Figure 7.2(a) is a clearer representation than the corresponding FSM in Figure 7.2(b). It enables continuous monitoring of failures by modelling them in a higher level of the state hierarchy. Moreover, the normal superstate is the same FSM as the one that would model the valve if it had no failures. This aids in overall process design as the synthesis can be a two step process. First, an ideal elementary component is defined and used to synthesise a procedural controller without failures. Then failures are added and a procedural controller synthesised without having to revise the whole design.

The level sensor statechart is a larger and seemingly more complex model than its equivalent FSM, however each superstate is smaller, leading to smaller procedural controllers. It is also possible for states with the same physical state to be present in both superstates. For example, the level sensor has two empty states. The initial state is empty and is part of normal operation. However, if the level returns to empty unexpectedly, a failure has occurred and the model recognises this by creating a second empty state in the abnormal superstate.

7.3.1 Synthesis of Procedural Controllers within Each Superstate

The partitioning of elementary components is done with the aim to reducing the size of the procedural controllers synthesised. Procedural controllers are synthesised for each combination of the superstates of the elementary components. That is, during normal operation all elementary components are in their normal state. To synthesise a procedural controller for normal operation, only the FSMs within the normal superstates of the elementary components are required. For example, a process consisting of five valves modelled as in Figure 7.2(a), as opposed to Figure 7.2(b), would see a size reduction of the potential model from $7^5 = 16807$ to $2^5 = 32$.

The lowest level in the ISA-S88.01 standard's control hierarchy is a phase, such as the CIP-Heating phase presented in the previous chapters. The controllers created by partitioning theory sit one level below this and will be termed *sub-phases*.

In addition to the benefits of not including part of each elementary component model, the size of each sub-phase procedural controller is reduced when not all elementary component instances are required. Recall the *CIP-Heat* procedure presented in chapter 5. During normal operation, the normal values of the temperature sensor are important. However, if a failure occurs, the instructions are to shut all valves, stop P6 and disable IC1-8. The value of IT1-17 is of no concern to the shutdown procedure and can be omitted from its synthesis. Table 7.1 lists the sub-phases of the *CIP-Heat* phase, the number of component instances and potential model size of each sub-phase together with statistics

Procedure 7.1 (Elementary Component Synthesis)

Beginning with a knowledge of the process component and the operating specification
1 Model *Normal* Superstate
  1.1 Construct FSM describing th desired normal behaviour
  1.2 Put it in a single OR superstate (the *Normal* superstate)
2 Model Failure Transitions
  2.1 DO For all states in the *Normal* superstate
    2.1.1 IF a failure can occur in this state
      2.1.1.1 Attach a failure transition beginning at the edge of the normal superstate.
      2.1.1.2 IF the destination state exists BUT NOT within the normal superstate
        2.1.1.2.1 Terminate transition at destination state
      2.1.1.3 ELSE Create the new basic destination state
        2.1.1.3.1 Terminate transition at the new state
      2.1.1.4 ENDIF
    2.1.2 ENDIF
  2.2 ENDDO
3 Model *Abnormal* Superstate
  3.1 Group all new states within a single *Abnormal* Superstate
  3.2 IF all Failure transitions terminate at a single state
    3.2.1 Add a default state transition to this state.
    3.2.2 Make all failure transitions terminate at the edge of the *Abnormal* Superstate
  3.3 ENDIF
  3.4 DO For all states in *Abnormal* Superstate
    3.4.1 DO For each transition
      3.4.1.3 IF it is NOT a recovery transition
          AND the destination state DOES NOT exist
          3.4.1.3.1 Create new state within Abnormal Superstate
              and terminate transition
      3.4.1.4 ELSE terminate transition at destination state
      3.4.1.5 ENDIF
    3.4.2 ENDDO
  3.5 ENDDO
4 Model *Recovery* Transitions
  4.1 DO For each state in the *Abnormal* Superstate.
    4.1.1 IF a recovery transition can occur
      4.1.1.1 Add it with the *Abnormal* Superstate as the originating state.
      4.1.1.2 Connect transition to destination state in *Normal* Superstate.
    4.1.2 ENDIF
  4.1 ENDDO
  4.2 IF all recovery transitions terminate at the initial state of the *Normal* Superstate.
    4.2.2 Make all recovery transitions terminate at the *Normal* Superstate.
  4.3 ENDIF

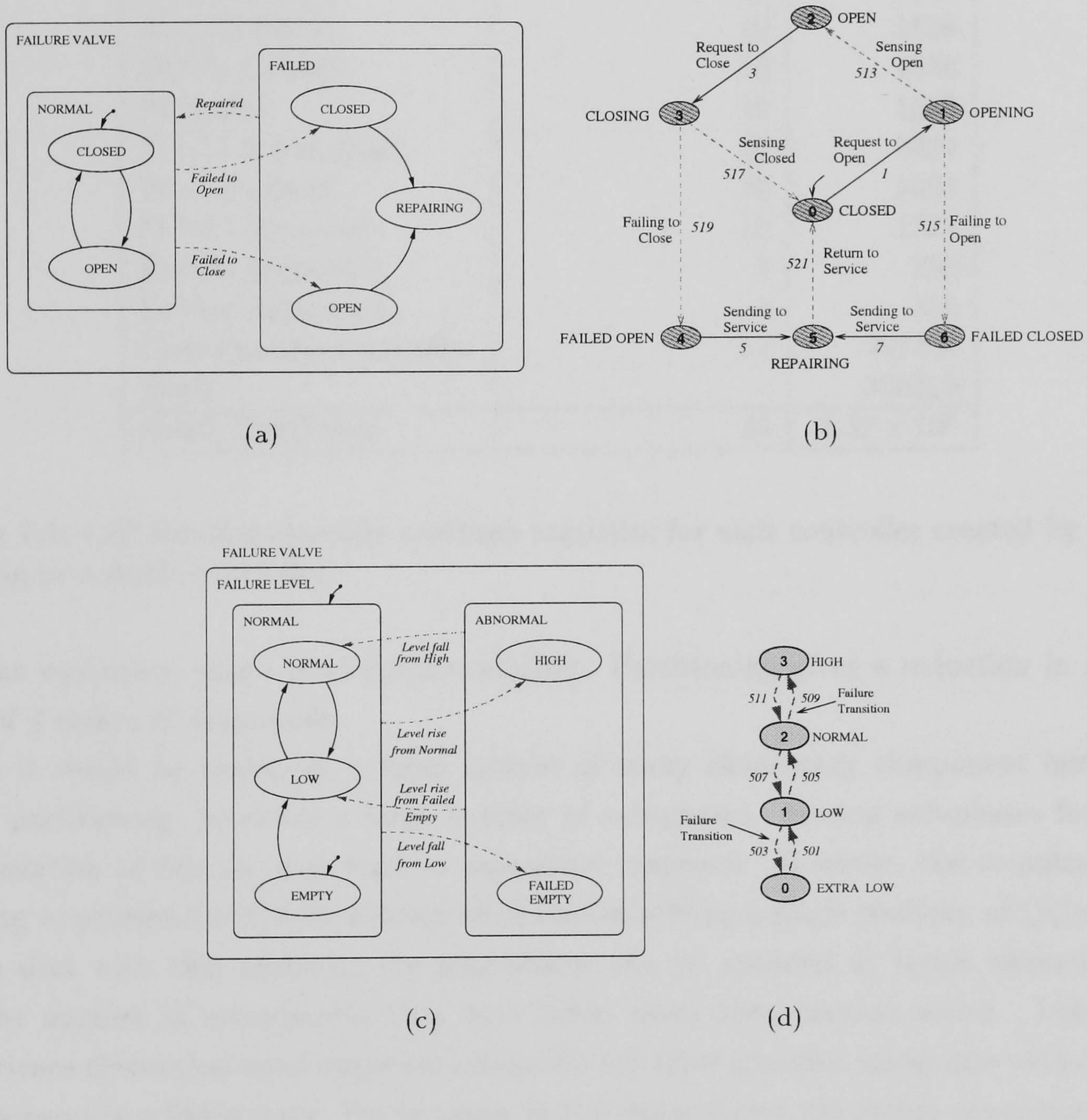


Figure 7.2: Examples of Partitioned Components (a,c) and their non-partitioned equivalents (b,d).

Controller	No. of Elementary Components	Potential Model Size
Normal Operation	11	3072
AV1-10 Failed	10	1536
AV1-14 Failed	10	1536
AV1-15 Failed	10	1536
AV1-16 Failed	10	1536
AV1-20 Failed	10	1536
AV1-22 Failed	10	1536
AV1-24 Failed	10	1536
AV1-25 Failed	10	1536
DDV1-8 Failed	10	1536
P6 Failed	10	1536
IT1-17 Extra High	10	1024
IS1-1 Covered	10	1024
IT1-2 Uncovered	10	1024
Failure Monitor 1	8	256
Failure Monitor 2	8	256
Coordinating Controller	12	354294
Total		376310
Single Controller	13	4.52×10^9

Table 7.1: CIP Heating example synthesis statistics for each controller created by partitioning or a single controller.

for the equivalent single procedural controller. Partitioning gives a reduction in model size of 4 orders of magnitude.

As it would be expected, a large system of many elementary component instances after partitioning, produces a large number of sub-phases. Having sub-phases for each combination of failures may lead to *superstate explosion*. However, the complexity of solving m problems of $\bigcirc(n^k)$ is lower than that of solving a single problem of $\bigcirc((mn)^k)$.

To deal with this problem, the superstates can be grouped in layers characterised by the number of components that have failed when they become active. Industrial experience shows that most implementations do not allow a system to operate with several components in a failed state. For instance, in low risk systems, the design procedure could stop at the single failure level, while in safety critical systems, several layers would have to be considered.

More often, several failure situations will require the same evasive action. Then, the same sub-phase procedural controller can be used and multiple controllers need not to be synthesised. Using the theory in section 6.5, sub-phase procedural controllers which conform to the specification of several failure situations can be synthesised, thus reducing the overall controller size.

7.4 Sub-Phase Coordination and Initiation

To monitor failures and initiate the correct sub-phase procedural controllers, a procedural controller is synthesised at the phase level. This procedural controller contains procedural component instances representing each sub-phase plus FSMs representing each elementary component instance containing failures. The implementation is the same as for any hierarchical procedural controller, as described in Chapter 5. To maintain the generic nature of the approach, there are a few generic operating specifications which can be applied.

1. Only one sub-phase procedural controller operates at any time.

2. A single failure causes the system to become abnormal. The normal procedure should be stopped and relevant emergency sub-phase procedural controller started.

3. The system does not recover to normal operation until all elementary component instances have returned to normal. That is the normal operation sub-phase procedural controller should never be started when an elementary component instance is in its abnormal state.

To implement the partitioned system as a procedural controller, the actual superstate is an FSM as shown in Figure 7.3. From the initial state, a controllable transition initiates the normal operation sub-phase. From the normal state, each failure transition goes to its unique failure state from which a controllable transition initiates the correct emergency sub-phase procedural controller. The controllable transition to restart the normal sub-phase procedural controller only occurs from a state where all elementary component instances are in their normal state.

7.5 Theoretical Specification Decomposition

As stated in Chapter 6, the process model no longer need to be decomposed. The determination of the normal, abnormal and forbidden operating ranges is specification dependent. Therefore, only the specification is decomposed. The normal and abnormal operating range specification written as 7-tuples, S_n and S_{ab}:

$$S_n = \{Q, V^{n_v}, \Sigma - \Sigma_f, \delta, \gamma, Q_{0_n}, Q_{m_n}\} \quad (7.1)$$

$$S_{ab} = \{Q, V^{n_v}, \Sigma - \Sigma_f, \delta, \gamma, Q_{0_{ab}}, Q_{m_{ab}}\} \quad (7.2)$$

where

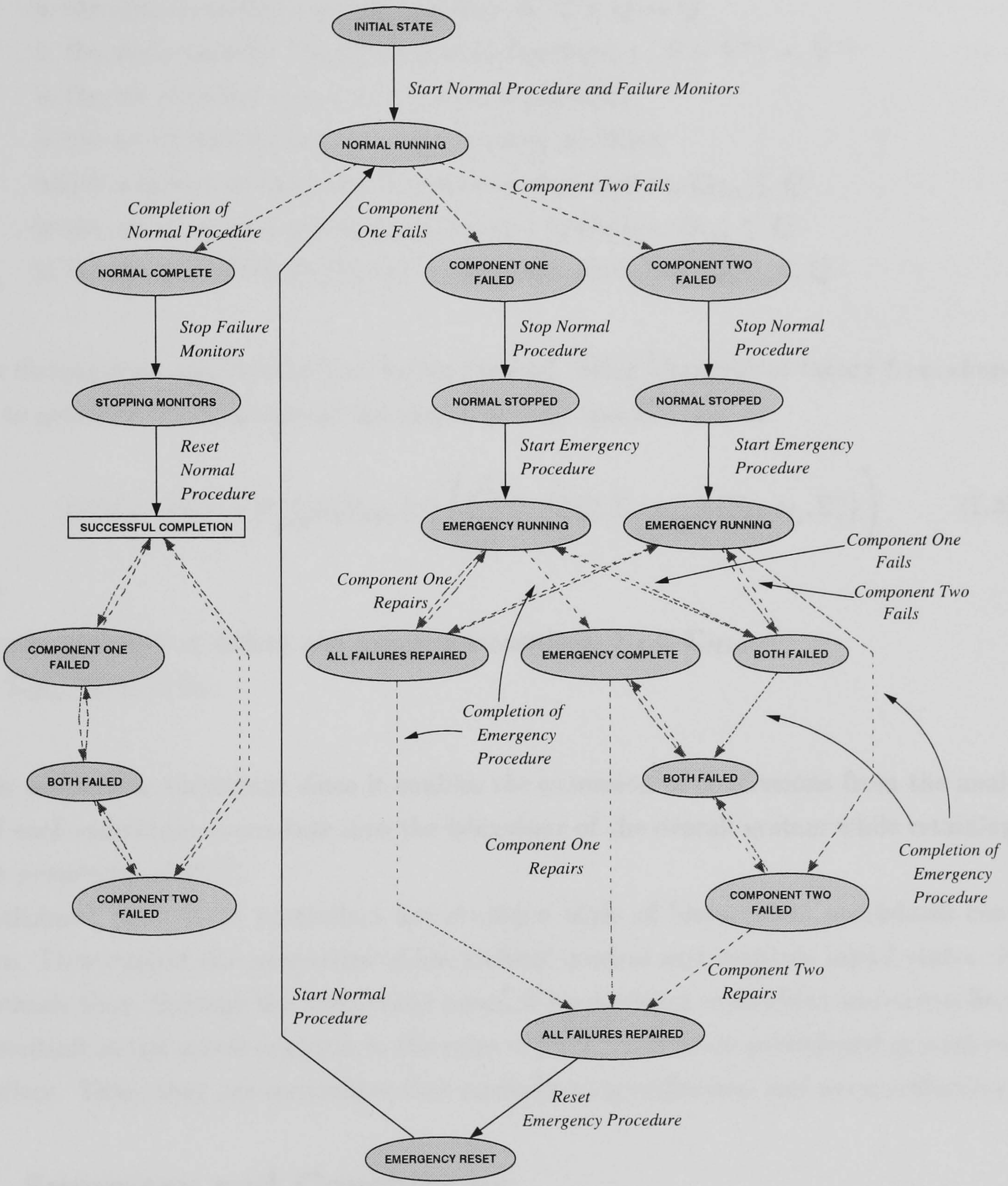


Figure 7.3: An example of a higher level procedural controller.

Q	is the set of substates, $q \in Q$
V^{n_v}	is the set of state variables, $\{(v_j)_q,\ j = 1, 2 \ldots, n_v\}$
n_v	is the number of state variables defining state q
Σ	is the set of all transitions
Σ_f	is the set of failure and recovery transitions
δ	is the state transition partial function, $\delta : \Sigma \times Q \rightarrow Q$
γ	is the state variable transition partial function, $\gamma : \Sigma \times V^{n_v} \rightarrow V^{n_v}$
Q_{0_n}	is the set of initial states in the normal partition
$Q_{0_{ab}}$	is the set of initial states in the abnormal partition which a failure under normal operation may lead to, $Q_{0_{ab}} \subseteq Q$
Q_{m_n}	is the set of marked states under normal operation, $Q_{m_n} \subseteq Q$
$Q_{m_{ab}}$	is the set of marked states under abnormal operation, $Q_{m_{ab}} \subseteq Q$

The decomposed specification can be recombined, using hierarchical theory from chapter 5, to generate the language of the overall system specification as:

$$L(S_{hier} \triangle S_i) = \mathcal{P}_{hier}^{-1} L(S_{hier}) \cap \left(\bigcap_{i=1}^{n} \mathcal{P}_i^{-1} \{\Sigma_i^*.\Sigma_f.\sigma_{i_s}.L(S_i).\sigma_{i_c}.\Sigma_i^*\} \right) \tag{7.3}$$

where

Σ_f is the alphabet of failure and recovery transitions, $\Sigma_f \subseteq \Sigma_{Hier}$

$\Sigma_i \cap \Sigma_{Hier} \subseteq \Sigma_u \cup \Sigma_{ic}$

This property is important since it enables the extension of conclusions from the analysis of each individual superstate into the behaviour of the overall system while retaining all the properties of PCT.

Partitioned procedural controllers are a unique style of hierarchical procedural controllers. They exploit the properties of hierarchical control and multiple initial states. It then stands that, through the theoretical proof, if hierarchical controllers and controllers with multiple initial states conform to the rules of PCT, then so do partitioned procedural controllers. Thus, they are controllers that conform to specification and are nonblocking.

7.6 Summary and Conclusions

A new modelling and specification method, Partitioning Theory, was introduced to synthesise procedural controllers for large systems containing component instances that can fail. The method also allows the continuous monitoring of process equipment failures while satisfying the requirements of PCT in order to produce a provably correct procedural controller.

This is achieved by recognising that a process and its elementary components can be partitioned into the behavioural patterns of *normal*, *abnormal* and *forbidden* operation.

Transitions related to failures or the recovery from failure are given higher priority in the component models. The behaviour of a component within each region is represented by a reduced FSM. From these reduced FSMs, procedural controllers for each type of behaviour can be synthesised, which are smaller and more manageable than a single procedural controller. The synthesis is performed using traditional methods of PCT.

The failure and recovery transitions are used to synthesise a higher level procedural controller (phases) that monitors components for failures and coordinates the execution of the lower level procedural controllers (sub-phases). This controller is also synthesised using traditional techniques. Using hierarchical theory, controllers are combined to create a procedural controller for the overall specification which is provably correct and has a much more manageable size and structure.

Partitioned controllers are a unique type of hierarchical controllers which exploit the properties of hierarchical control and multiple initial state controllers. Through this, it was shown that they conform to specification and are nonblocking.

Chapter 8

Structured Implementation of Procedural Controllers

8.1 Introduction

The acceptance of formal methods has been hindered by difficulties associated with the transferral of synthesised and verified procedures (typically developed in a formalism that supports the modelling and analysing step an an FSM) to implementable control code (typically in a structured computer language). This is largely because few translation algorithms exist and those that do produce code that is significantly larger than manually written code. The absence of a formal translation algorithm or structure also defeats the very purpose of formal synthesis as once manual translation is used, errors can easily occur. Moreover, errors at this stage of the design procedure are often 10 to 100 times more expensive to correct or potentially fatal.

For the rapid, clear and concise modelling of processes, structured modelling methods are required. Concepts introduced by Sanchez *et al.* (1996) lay the foundations for this work and show that a formal structure yields the following benefits:

- a formal interface is established between the discrete controller and the discrete/continuous plant or plant model (from here on, the term process will be used to reference a physical plant or simulation model unless specifically stated),
- implementation errors are minimised, and
- models can be reused for further rapid development of process models and controllers.

8.1.1 Aim

This chapter presents work which aims to extend this implementation structure to enable the translation of procedural controllers into implementable control code. To do so, the structure will be required to have the following properties:

- it must be generic so as to enable the implementation of procedural controllers in a variety of syntaxes such as those used in a range of industrial control systems, simulation packages, and for structured documentation,
- it must reflect the structure of the procedural controller and elementary components it is implementing so as to enable automatic code generation, and
- it should be easily implemented and automated so that most of it is *invisible* to the user, making it more attractive to industry and those not intimate with PCT.

By achieving these aims, PCT will gain acceptance within both the industrial and academic community, producing factory implementable code while maintaining formal methods. The structure will satisfy academia's need for formality and being very easy to translate, satisfy industry.

8.2 Control System Architecture

The main task involved in implementing the *provable correct* procedural controllers of the previous chapters lies in the communication between the discrete event procedural controllers and the discrete/continuous process. We now propose a control system architecture which extends the previous work (Sanchez *et al.*,1996) and is widely applicable, therefore creating a basis for translation algorithms.

Figure 8.1 shows a three tiered architecture which is an extension of the one previously presented (Sanchez *et al.*,1996). At the top of the figure, the operator (or another software application) interacts with the system through discrete-event procedural controllers, requesting them to start, stop and so on. It is assumed all controllers have been designed using the formal synthesis methods described in the previous chapters. At the other end lies the process. While the procedural controllers are purely discrete events systems, the process is a combined discrete/continuous system. Marrying the two components of the hierarchy requires a third, intermediate level of interpretors to communicate the discrete controller requests and observe discrete/continuous process behaviour.

8.2.1 Overview of *gPROMS*

To demonstrate the principles presented in this chapter, illustrations are given in PARACODE (APV, 1994), a typical industrial control code language, and *gPROMS* (Process Systems Enterprise Ltd, 1997), a dynamic simulation language. For more detail on the PARACODE language refer to Chapter 11. The general PROcess Modelling System (*gPROMS*) is proprietary dynamic simulation software developed in the Centre for Process Systems Engineering, Imperial College. Among its capabilities it includes the dynamic simulation of processes with discrete properties. Process models are defined in a hierarchical fashion so that they can be combined to create other models while conforming to the ISA-S88.01 standard. Likewise, sequences known as TASKS are defined to act on the

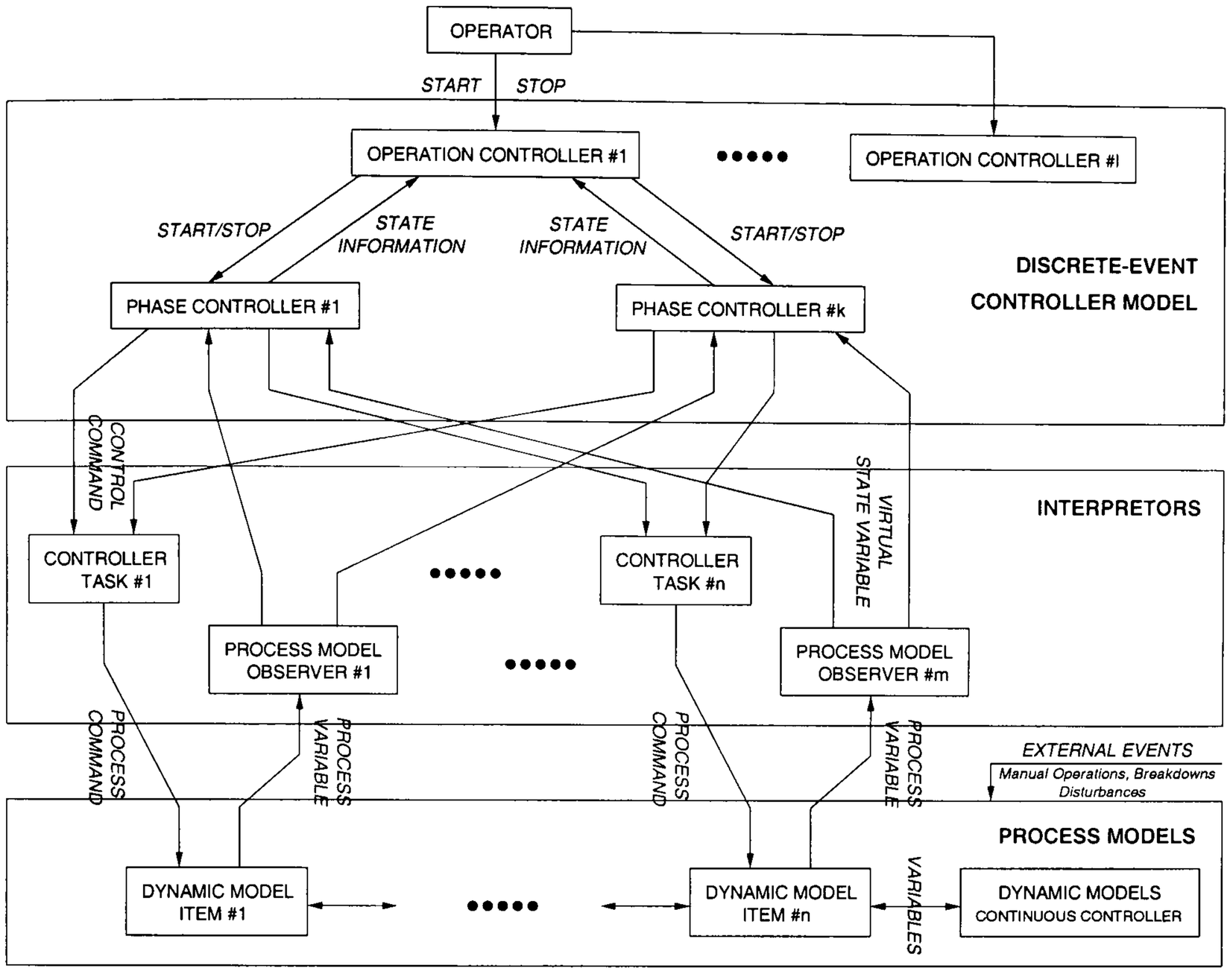


Figure 8.1: Modelling Architecture

model. These also have hierarchical capabilities allowing sequences to be implemented in a manner that also conforms to ISA-S88.01 standard. These features provide the opportunity to show the benefits of using dynamic simulation to verify that procedural controllers (that were provably synthesised in a discrete environment) achieve the desired results in a combined discrete/continuous environment.

Simulation results can be depicted and recorded for analysis using *gRMS* which complements the simulation solver.

8.2.2 Process Models/Items

A procedural controller can be tested using a simulation of the uncontrolled process behaviour rather than by running a real plant. To this purpose, models that are representative of the process itself must be defined in some suitable simulation language. Of course, the simulation response of the uncontrolled plant must be an adequate reflection of the behaviour of the real plant, including normal and abnormal operation. Simulation models must contain only the states and transitions that physically occur and contain methods to determine the possibility of a failure. States and transitions inferred from physical occurrences should not be in the process models.

Each process model must be treated like a black box with only the information seen on the reall process made available to the outside. This includes sensor signals, analogue and discrete. In turn, controllers communicate with the uncontrolled process (or simulation model) though well defined control commands (e.g open/close, signals and setpoints).

In developing the models two guidelines should be considered:

1. only the physical behaviour of the process item is included, and
2. algorithms and conditions determining the occurrence of failures should be contained within the model.

These properties are best illustrated by an example. Consider a digital valve. Figure 8.2 shows on the far left a representation of what of a process component and the signals it receives and sends out. Immediately to its right is an FSM of a digital valve representing the physical behaviour of the valve. Using the signals from the process component, the physical state of the valve can be established.

Likewise Figure 8.3 shows on the far left a *gPROMS* model listing and immediately to its right, an FSM representing the physical states of the digital valve. In the model listing, the states are defined within the `EQUATION` section. Note that the variable STATE can only take on the values *Closed* and *Open* just like in the physical model. Within each of the states there is a `SWITCH` statement to model the transitions. It contains three variables: *command_Action*, *failing* and *external_Action*. *Command_Action* signals a request to open (1) or to close (2). *Failing* and *external_Action* are associated with the failure state of the valve.

Because the simulation language does not have any automatic failure monitoring facilities like the industrial control system, the failure calculations and assessment must be modelled. Therefore, *failing* is the probability of the valve failing to do so (the higher the value, the higher the chance of failure) and *external_Action* emulates external intervention such as an operator manually manipulating a valve. If failures are included in a model, a grace period or *CheckTime* must be included(as it is done in industrial control systems) to allow for the finite time taken for process components to change a physical state. In this example, a discrete selector *Check* is used and has associated variables *CheckTime* and *CheckTimeExp*. *Check* has 2 states, *Waiting* and *Expired*.

Similarly, Figures 8.4 and 8.5 show the mapping between a continuous level sensor and its model. Note how there is only a single signal from both the uncontrolled process item on the plant and the uncontrolled process model in the *gPROMS* model. There is no discretisation at this stage. Nor is there any failure assessment since deciding whether the value of the level is a failure is part of the operating specifications.

8.2.3 Interpretors

Providing the link between the process models and the discrete model of the component, the interpretors are divided into two subclasses : *Process Model Observers* and *Controller Tasks*.

8.2.3.1 Process Model Observers

Process Model Observers monitor information from the process and from the phase controllers to determine the state of the component's discrete model. Importantly, as opposed to previous work, the Process Model Observer does not determine whether an event occurs in the process but merely observes that it occurs. In other words, Process Model Observers contain no commands to drive or control the process they observe. They use the information from the process and the procedural controller to determine whether events have occurred. In particular, by using the operating specification, they determine whether a failure (in the context of the respective operating specification) has occurred. All elementary component instances require a process model observer.

Referring again to Figure 8.2, it shows the mapping between a digital valve, its Statechart as defined for the discrete controller synthesis and the process model observer written in PARACODE. The state-chart has normal and abnormal superstates, in accordance with the Partitioning Theory discussed in Chapter 7. The Process Model Observer ascertains which of these states the process item is in. To do this, it contains only condition `IF` statements to query the value of signal given. Using the generally accepted specification of a valve failing (being stuck between states), the discrete-controller state (regarding whether a request has been made to open or close the valve) and the feedback signals, the process model observer establishes which state of the discrete model the valve is in.

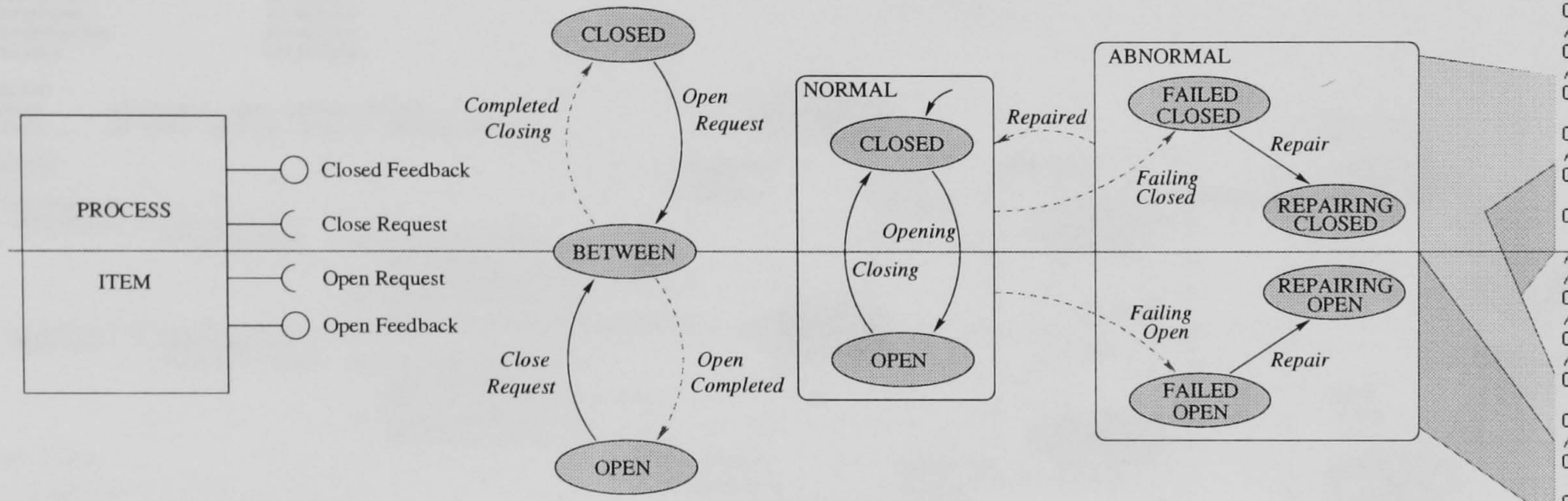
In the corresponding *gPROMS* example, its physical position is determined by querying the variable `VALVE.State`. To determine whether a failure has occurred, information from the discrete-event controller is also used. The actual failure assessment must be explicitly defined in simulation languages and thus the variables *command_Action* and *CheckTime* (set by requests from the Controller) are used to communicate which state the valve should be in. If this does not match with `VALVE.State` , a failure has occurred and the observer moves to the virtual state *Abnormal.*

Similarly, in Figures 8.4 and 8.5, a level's statechart and process model observer are mapped to the physical sensor. The figures show a state-chart which has discretised the continuous signal. The process model observers use the continuous signal to determine which discrete state the level sensor statechart is in.

8.2.3.2 Controller Tasks

Controller Tasks are used for implementing controllable transitions. They are called by the phase controllers and act on the uncontrolled process (models) to cause a change of state. Most useful when implementing simulations, they provide a means to collect several instructions that may culminate in producing the desired controllable transition. There are no corresponding Controller Tasks for uncontrollable transitions.

In most industrial control systems, failure monitoring of components is automatic. Thus, resetting failure monitoring values is not required and the controller tasks are




```
/
.SEQ 2.330
/
/ PARACODE Observer File for
/ Component Instance : AV1-22
/
0330A1  INIA    AV1-22
/
0330TS1 IFIF    AV1-22, 0330SS2
/
0330SS1 MOVN    0, R2.331               / AV1-22 IS Normal
/
0330T0  IFIE    AV1-22, 0330S0          / TRANSITION Close OF AV1-22
        GOTO    0330T1                  / LOOP IN STATE Open
0330S0  MOVN    0, R2.330               / AV1-22 IS Closed
/
0330T1  IFID    AV1-22, 0330S1          / TRANSITION Open OF AV1-22
        GOTO    0330TS1                 / LOOP IN STATE Normal
0330S1  MOVN    1, R2.330               / AV1-22 IS Open
        GOTO    0330TS1                 / LOOP IN STATE Normal
/
/
0330TS2 IFINF   AV1-22, 0330SS1
/
0330SS2 MOVN    0, R2.331               / Valve IS Abnormal
/
0330T2  IFOD    AV1-22, 0330S2
        GOTO    0330T3
0330S2  MOVN    2, R2.330               / AV1-22 IS Failed trying to Close
/
0330T3  IFOE    AV1-22, 0330S3
        GOTO    0330TS2
0330S3  MOVN    3, R2.330               / AV1-22 IS Failed trying Open
        GOTO    0330TS2
/
0330Z   STOP
/
```

Figure 8.2: Mapping between an uncontrolled process item, discrete physical model, intelligent discrete model and a PARACODE observer for a digital valve.

```
MODEL VALVE

VARIABLE
  position                  AS Open_fraction
  flow                      AS flow_rate
  command_Action            AS Control_trans
  external_Action           AS Control_trans
  valvaeIndicator           AS NoType
  superStateIndicator       AS NoType
  Repair_Time               AS NoType
  CheckTime                 AS NoType
  CheckTimeExp              AS NoType
  failing                   AS NoType

SELECTOR
  State       AS (Closed, Open) DEFAULT Closed
  Check       AS (Idle, Waiting, Expired) DEFAULT Idle

EQUATION

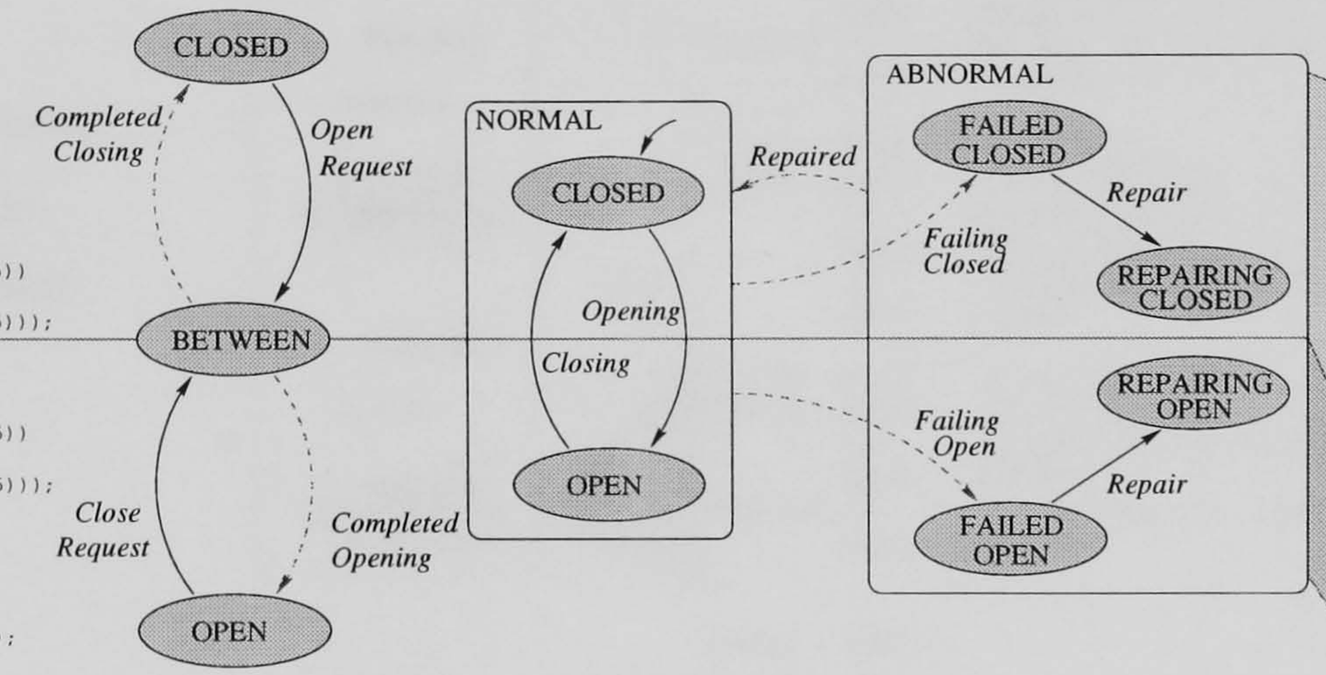
  CASE State OF
    WHEN Closed  : position = 0.0;
                   SWITCH TO Open     IF (((command_Action = 1)
                                          AND (failing > 0.1)
                                          AND (external_Action < 0.95))
                                       OR ((command_Action = 3)
                                          AND (external_Action > 0.95)));

    WHEN Open    : position = 1.0;
                   SWITCH TO Closed   IF (((command_Action = 3)
                                          AND (failing > 0.1)
                                          AND (external_Action < 0.95))
                                       OR ((command_Action = 1)
                                          AND (external_Action > 0.95)));

  END # Case

  CASE Check OF
    WHEN Idle    : $CheckTime = 0.0;
    WHEN Waiting : $CheckTime = 1.0;
                   SWITCH TO Expired    IF (CheckTime > CheckTimeExp);
    WHEN Expired : $CheckTime = 0.0;
  END # Case

END # Model
```




```
MODEL VALVE_OBSERVER

  PARAMETER
    PLANT         AS MODEL Flowsheet
    VALVE          AS MODEL VALVE

  SCHEDULE
   WHILE (PLANT>Endd <> 1) DO
    SEQUENCE
     PARALLEL
      IF (VALVE.superStateIndicator = 700) THEN # Valve State IS Normal
       PARALLEL
        IF (VALVE.Check = VALVE.Expired) AND (VALVE.command_Action = 1) THEN
          MESSAGE "###############VALVE FAILED CLOSED ###############"
          RESET  VALVE.superStateIndicator := 701; END # RESET
        END # IF
        IF (VALVE.Check = VALVE.Expired) AND (VALVE.command_Action = 3) THEN
          MESSAGE "################ VALVE FAILED OPEN ################"
          RESET  VALVE.superStateIndicator := 701;
        END # IF
        IF (VALVE.valveIndicator = 3) AND (VALVE.Check <> VALVE.Expired) THEN
          IF (VALVE.State = VALVE.Closed) THEN
            MESSAGE "############## VALVE CLOSED ###########"
            RESET  VALVE.valveIndicator := 0;
          END # IF
        END # IF
        IF (VALVE.valveIndicator = 0) AND (VALVE.Chack <> VALVE.Expired) THEN
          IF (VALVE.State = VALVE.Open) THEN
            MESSAGE " #########  VALVE OPEN ##########"
            RESET VALVE.valveIndicator := 3;
          END # IF
        END # IF
       END # PARALLEL
      END # IF
      IF (VALVE.superStateIndicator = 701) THEN # Valve State is Abnormal
       PARALLEL
        IF (VALVE.RepairTime > 60) THEN
          MESSAGE "############ VALVE REPAIRED ############"
          RESET   VALVE.superStateIndicator := 700;  END # RESET
        END # IF
        IF (VALVE.valveIndicator = 0) AND (VALVE.command_Action = 99) THEN
          MESSAGE  "########### REPAIRING CLOSED VALVE  ###########"
          RESET  VALVE.valveIndicator := 5; END # RESET
        END # IF
        IF (VALVE.valveIndicator = 3) AND (VALVE.command_Action = 99) THEN
          MESSAGE "########### REPAIRING OPEN VALVE ###########"
          RESET  VALVE.valveIndicator := 6; END # RESET
        END # IF
       END # PARALLEL
      END # IF
     END # PARALLEL
    END # SEQUENCE
   END # WHILE
  END # SCHEDULE
```

gPROMS Model | Physical Representation | Discrete Model (in StateChart form) | *gPROMS* Process Model Observer

Figure 8.3: Mapping between a process model, discrete physical model, intelligent discrete model and a *gPROMS* observer for a digital valve.

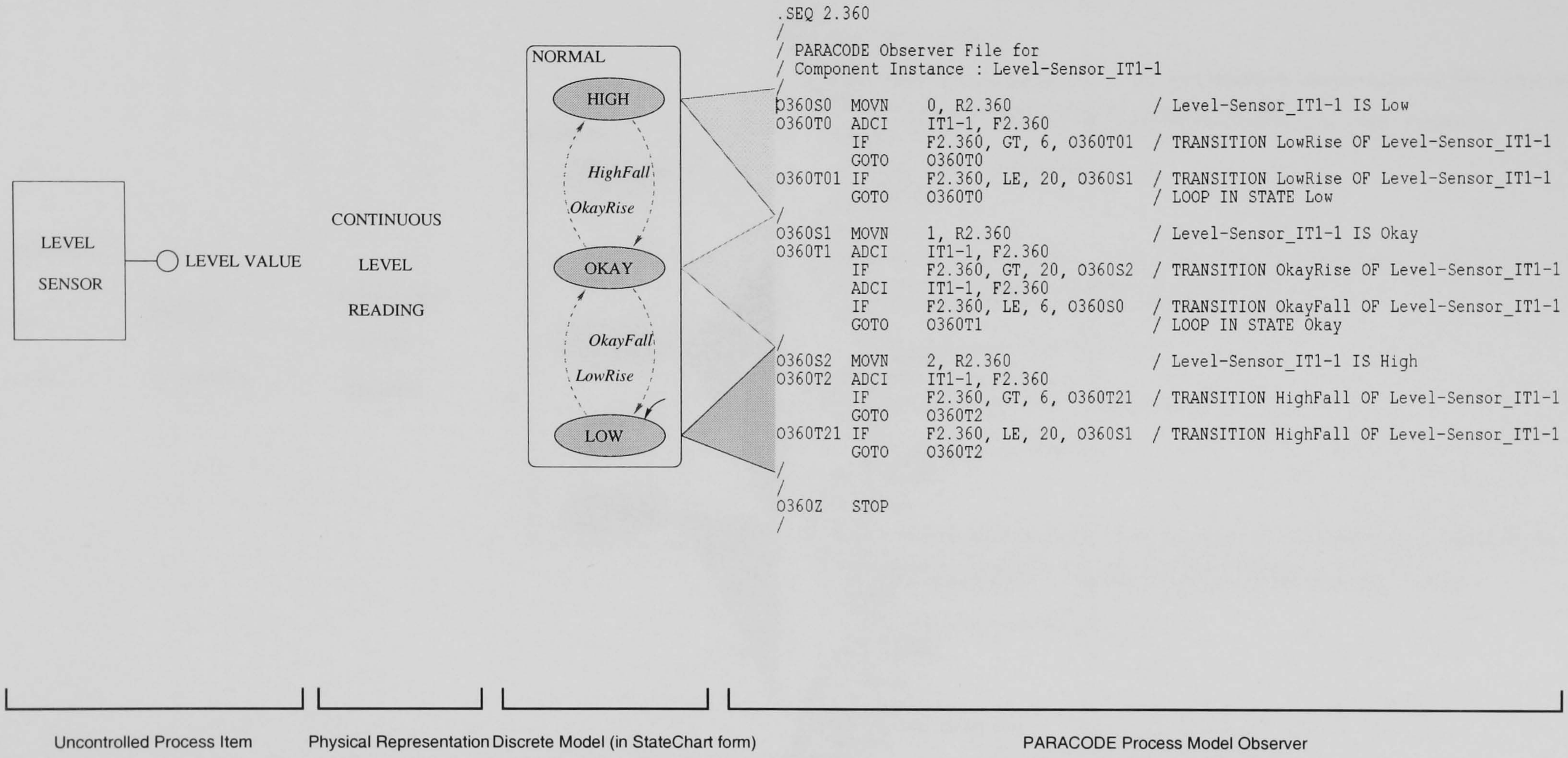


Figure 8.4: Mapping between a process model, discrete physical model, intelligent discrete model and an observer for a continuous level sensor.

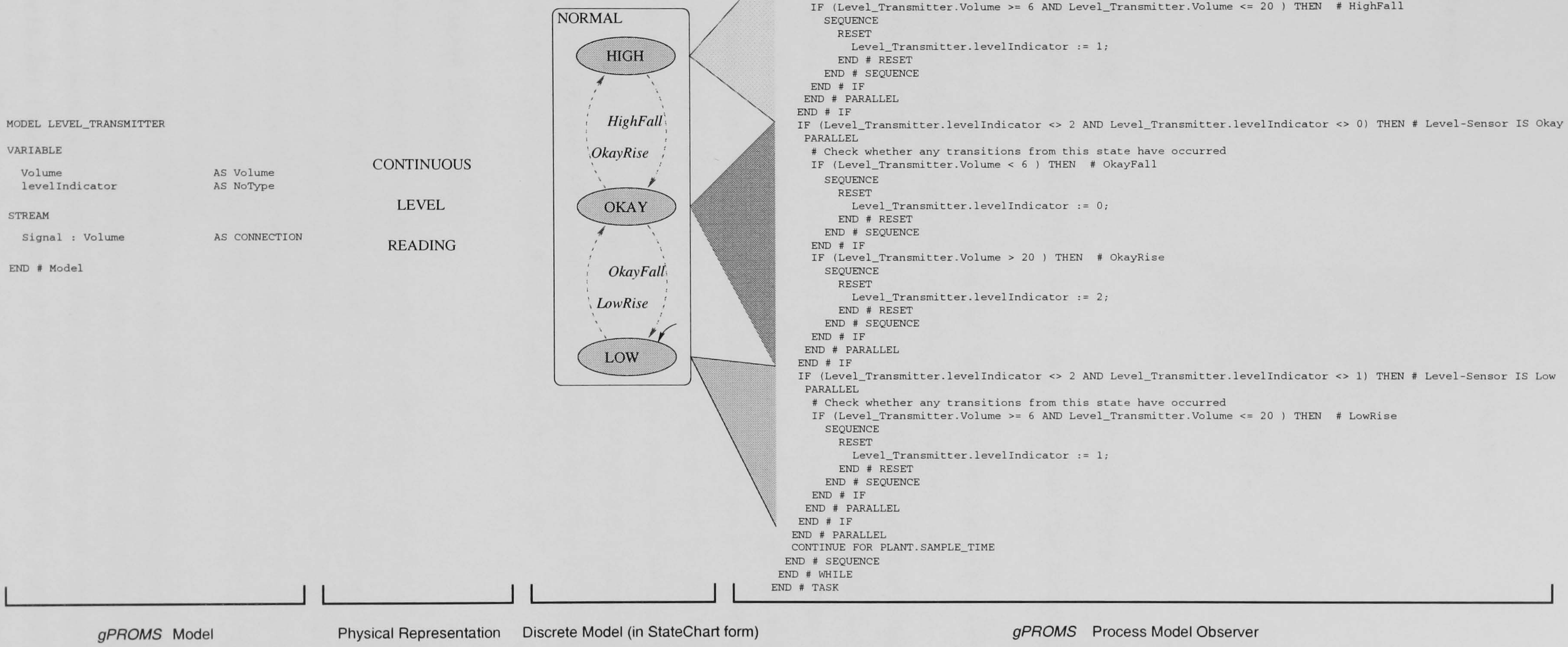


Figure 8.5: Mapping between a process model, discrete physical model, intelligent discrete model and an observer for a continuous level sensor.

```
/
/
.SEQ 2.100
/
/
S100S0  ENGE  AV1-22     / OPEN VALVE AV1-22
/
S100Z   STOP
/
/
```

(a) PARACODE

```
TASK OPEN_VALVE

  PARAHETER
    VALVE                AS HODEL VALVE

  SCHEDULE
    SEQUENCE
      REINITIAL
        VALVE.CheckTime
      WITH
        VALVE.CheckTime = 0.0 ;
      END # REINITIAL
      RESET
        VALVE.failing := UNIFORH(0,1);
        VALVE.command_Action := 1;
      END # RESET
      SWITCH
        VALVE.Check := VALVE.Waiting;
      END # SWITCH
      HESSAGE "OPENING VALVE"
    END # SEQ

END # TASK
```

(b) *gPROMS*

Figure 8.6: PARACODE and *gPROMS* code for the controller task to open a valve.

compact, as shown by Figure 8.6(a). The controller task however, provided the important link between the discrete controller and the uncontrolled process.

In simulations that do not have automatic failure monitoring procedure and therefore require failure monitoring to be simulated, controller tasks are very useful. For the valve, the controllable transitions of *Open*, *Close* and *Repair* all require Controller Tasks. The task to open the valve is shown in Figure 8.6(b). Its main task is to request that the valve be opened. However, it also must reset the failure monitoring variables inside the process models. It must be emphasised that this is done here due to the format of simulation packages and that although the variables are reset here, the actual failure calculations are performed in the process model as required. By grouping these instructions together, the task becomes simple to understand when called from within the discrete-event controller. The overall code produced is also reduced as the task can be reused.

8.2.4 Discrete-Event Controller

The term Discrete-Event Controller is given to the collection of all the procedural controllers acting on the process. Being purely discrete, it cannot communicate directly with the discrete/continuous process models. Therefore, the discrete-event controller receives uncontrollable transition information from the process model observers regarding the discrete states of the process items. It then issues commands to controller tasks which act on the process to bring about a change.

Sections of code, in the PARACODE language and *gPROMS* language respectively, from a procedural controller in the discrete-event controller are shown in Figures 8.7(a) and 8.7(b). It can be seen how each piece of code communicates with the process model observers and the controller tasks. In Figure 8.7(a) (PARACODE), controller tasks to open and close valves are called on lines 8 and 12. At line 17, a wait state is reached

```
1   .SEQ 2.538
2   /
3   .
4   .
5   / BEGIN OPERATION
6   /
7   S538S1  HOVN    1, R2.538    / SET STATE VARIABLE
8           STSQ    2.101        / Close InvValve_AV1-22
9   S538SB1 WAIT    5            / WAIT FOR GRACE TIME
10  /
11  S538S2  HOVN    2, R2.538    / SET STATE VARIABLE
12          STSQ    2.110        / Open Valve_AV1-25
13  S538SB2 WAIT    5            / WAIT FOR GRACE TIME
14  .
15  .
16  .
17  S538S6  HOVN    6, R2.538    / SET STATE VARIABLE
18  S538T6  IF      R2.364, EQ, 0, S538S34
19                               / InvLevel-Switch_IS1-2 Uncover
20          IF      R2.361, EQ, 1, S538S7
21                               / Conc-Sensor_CS LowRise
22          GOTO    S538T6       / LOOP IN STATE 6
23  /
```

(a) PARACODE

```
1   TASK CONTROLLER_NORMAL
2     PARAMETER
3       PLANT                AS MODEL PLANT
4
5     VARIABLE
6       state                AS NoType
7
8     SCHEDULE
9       SEQUENCE
10        state := 1; # Initialisation
11        WHILE (PLANT.Endd <> 1) AND (marked <> 1) DO
12          SEQUENCE
13            PARALLEL
14              IF (state = 1) THEN  # STATE 1
15                PARALLEL
16                  IF (PLANT.ButtonIndicator=3) THEN
17                    state := 16;
18                  END # IF
19                  .
20                  Scan other uncontrollable observers
21                  .
22                PARALLEL
23              END # IF
24              .
25              .
26              IF (state = 16) THEN  # STATE 16
27                SEQUENCE
28                  OPEN_VALVE(VALVE IS PLANT.VALVE_101);
29                  state := 17;
30                SEQUENCE
31              END # IF
32              .
33              .
34  END # TASK
```

(b) *gPROMS*

Figure 8.7: PARACODE and *gPROMS* code for a discret-event controller.

where the controller scans the values of variables indicating component state and that are set by process model observers. Likewise in Figure 8.7(b) (*gPROMS*), an uncontrollable transition is registered on line 16 by the conditional statement, testing the value of a variable set by a process model observer. On line 28 a controller task is initiated.

Using the implementation architecture introduced in section 8.2, a procedural controller, once verified by simulation can be quickly translated into implementable industrial control code. Automatic translation of procedural controllers into simulation code and industrial control code is done using the software documented in Chapter 10. Having a rigorous structure for the implementation enables easy translation into a variety of languages. In this work, a simulation language *gPROMS*, an industrial control systems language PARACODE plus documentation in structured English were produced. This is illustrated in chapter 11 with an implementation on the Batch Pilot Plant at Imperial College.

8.3 Summary and Conclusions

In this Chapter, a method for the verification and implementation of procedural controllers in a discrete/continuous environment was presented. It showed that a structured implementation architecture has the following benefits:

1. facilitates the automatic generation of control code for simulation and implementation,

2. reduces the possibilities of introducing designer errors,

3. modularises and structures the code for easier understanding, and

4. removes the need to access and edit actual code when combined with formal synthesis methods.

These benefits were achieved by:

1. defining a structure where the discrete/continuous plant/simulation and the discrete controllers are isolated and communicate through interpretors,

2. defining rules for the definition of models in process simulations, and

3. modelling process components in the discrete environment with the knowledge of the physical nature of the component.

Throughout the chapter, two examples showed how the structured method simplifies and formalises the process of procedural controller synthesis and implementation. The complete case study is shown in Chapter 11.

Chapter 9

Computational Aspects in the Synthesis of Procedural Controllers

9.1 Introduction

The concepts introduced in the previous four chapter are mathematically intensive and thus lend themselves to being implemented in a computational framework. Even the smallest of process models soon become unmanageable when manually synthesising procedural controllers.

As mentioned in the literature review, software for the formal synthesis of operating procedures has been developed. Supervisory Control Theory uses the program *TCT* that uses only FSMs to depict process models and behaviour. This enables all varieties of specifications to be implemented but the program is hindered by a difficult input language and the relatively small size of controllers that can be synthesised.

Within Procedural Control Theory, software has evolved in two stages. Initially, a piece of software called *PLATEST* was written, using FSMs to define models of elementary components which are then combined to produce the process model. Operating specification is done using temporal logic, making synthesis easier. However, the program also struggles with size limitations and has a primitive input language with many files to be setup, making it difficult to use.

A second program, *fsm*, was then written to replace *PLATEST*. Its contribution was the introduction of a structured text input language and a new approach to handle difficulties related to size. Using a state based approach, controllers for larger processes were synthesised. However, *fsm* lost some of PLATEST's power such as that of forcing sequences of transitions or decisions based on previous transitions. The user was limited to forcing or prohibiting the next transitions based only on the current state of the process.

This chapter presents a new program, called Sequence Solver, that embraces the positive

sides of the previous methods to produce a stronger problem foundation and solving engine. It also outlines the additional development required to implement some of the theory introduced in the previous chapters.

The chapter is organised as follows. It begins by presenting the approaches currently taken when synthesising controllers. A table summarises the different approaches. These approaches a then discussed in subsequent sections. They begin in section 9.3 with a discussion regarding the way processes are modelled. In particular, the question of whether knowledge of a component's state variable value is required during synthesis. Section 9.4 then considers specification techniques and proposes that a method which uses elementary component models and temporal logic is better than using a purely FSM based specification method. This is then followed by section 9.5 which discusses techniques for the actual synthesis procedure itself. It discusses previous approaches which generate a separate process model and operating specification before intersecting the two, to produce the final controller. It then considers a new approach whereby the process model and operating specifications are synthesised together. A section on the incorporation of multiple and covered initial states follows before sections 9.7 and 9.8 present the new synthesis method that is proposed in this work. The new method endeavours to incorporate the benefits outlined by the previous sections to produce a more powerful synthesis tool.

9.2 Current Synthesis Approaches

Synthesis approaches were discussed in Chapter 2 and summarised by Figure 2.1. The general approach as shown by the figure is that, starting with the equipment items, elementary component models are built first. These models, be they state or language based are then combined with the process behaviour to create a process model or combined with the desired control specifications to produce an operating specification. The definition of the modelling constraints and the operating specification has been done by FSMs alone or by combining the elementary component FSMs with temporal logic. The process model and operating specifications are then combined (intersected) to synthesise the controller. Synthesis methods are either state based and contain no knowledge of previous events in the process or transition based do contain knowledge of previous transitions.

The following sections discuss the benefits and costs of each approach. Table 9.1 summarises the synthesis techniques considered and the methods they use.

9.3 State Based versus Language Based

TCT is a purely language or transition based program. It has no knowledge of the state variable values in each state of a process and only tracks the sequence of transitions which have occurred. To synthesise a supervisory controller, sequences of transitions from the initial states of process models and specifications are compared and the common sequences are maintained as the supervisory controller.

Task	Approach	*TCT*	PLATEST	*fsm*	Sequence Solver
Process Modelling	Language Based	√			
	Combined State & Language		√		√
	State Based			√	
Constraint Definition	Solely FSM	√			
	Temporal Logic & FSM		√	√	√
Synthesis Method	Level (state based)			√	
	Path (transition based)	√			
	Combined		√		√

Table 9.1: Summary of approaches to the aspects of controller synthesis

On the other hand, *fsm* is a state based program which assesses the validity of a transition based on the state which it originates from. The algorithm implemented in *fsm* is shown in algorithm 9.1. A temporal logic specification contains a condition which describes a state, if this matches the current process state, then the specification is applied and transitions forced or removed. Being solely based on state variables prevents *fsm* from implementing complex specifications such as sequences of forced transitions or specifications based on a history of transitions. Both are possible using a language based approach, however, specification is more difficult when temporal logic is not used.

Problems in *fsm* were also encountered when two states had the same state variable values but had different transition histories. The purely state based approach ignores how a state is reached, which can be critical.

The approach of *PLATEST* was a mixture of the two above. To build complex process models from simple elementary component models, a state based approach was used. Elementary models were combined via the asynchronous product to produce all possible combinations of states and thus all possible sequences of transitions. Operating specifications were defined using temporal logic to incorporate state variable values. These temporal logic formulae were then converted into FSMs, intersected, and an FSM representing the specification was produced. It was then compared to the process model FSM and common sequences were maintained to produce the procedural controller. Reachability, co-reachability and controllability tests were then performed on the controller.

PLATEST's approach was the most comprehensive approach although limited by size. It appears that a combined state/transition based approach will provide a better solution.

9.4 Purely FSMs versus Elementary Components and Temporal Logic

TCT synthesises supervisors using only FSMs. The process model is built from FSMs

Algorithm 9.1 (*fsm* Procedural Controller Synthesis)

```
Beginning with the component models, modelling constraints and operating specifications.
DO For each Initial State
   Create NEW state as Initial State
   DO For each NEW state
      DO For each state variable
         DO For each transition possible from the state variable's value
            IF no dynamic constraint inhibits the transition THEN
               IF no static constraint forbids the system destination state THEN
                  IF destination state exists THEN
                     Add transition and terminate it at destination state.
                  ELSE Create the NEW state
                     Add transition and terminate it at the new state
                  ENDIF
               ENDIF
            ENDIF
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

which model the subsystem behaviour. These models can often be complex and difficult to create. The operating specification is also in the form of an FSM, requiring an intimate knowledge of FSM theory and the process.

In contrast, *PLATEST* and *fsm* require the elementary components of a process to be defined as simple FSMs. These are easier to define as they only model the behaviour of a single component. The complex process model is then created by combining the elementary component models. The operating specifications are written using temporal logic mirroring the definition of the operating procedure in typical design procedures. *PLATEST* then translates the temporal logic automatically into FSMs while *fsm* uses the temporal logic directly.

Building a complex model directly is very difficult. Thus, the concept of constructing a process model from simple elementary component models is appealing. The use of temporal logic also makes defining the operating specification much easier.

9.5 Separate Model and Specification Synthesis versus Combined Synthesis

As was shown by Figure 2.1, in both *TCT* and *PLATEST*, process models and operating specifications are synthesised separately and then intersected to obtain the controller. As

Figure 9.1 shows, this allows the possibility of two large FSMs (and languages) to be generated that, when intersected, produce a very small controller. Synthesis of the large model and specification FSMs limits the size of problem these programs can handle.

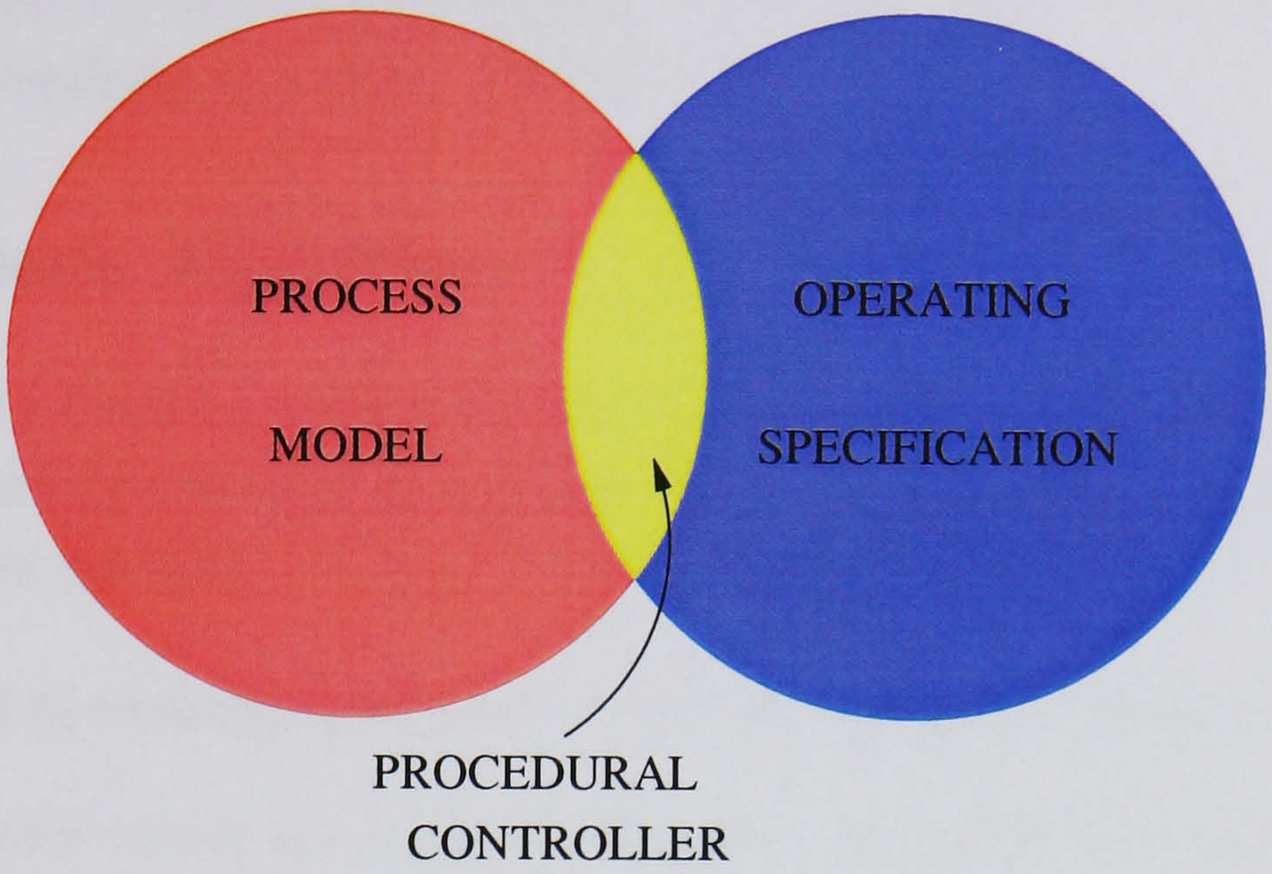


Figure 9.1: How *TCT* and *PLATEST* use the state space

fsm starts at the initial states of a procedure and builds the controller by adding permitted transitions. Using both, the modelling and specification constraints concurrently, it ensures that the smallest FSM is created and the synthesis always stays within the intersection of the process model and operating specification state spaces. In this way, controllers are synthesised that would not have been possible if either their process model or operating specification had to be synthesised completely.

A combined modelling approach is more beneficial as larger processes can be tackled and several controllers synthesised using the previous methods can be condensed into one.

9.6 Multiple and Covered Initial States

Until now, all methods have considered FSMs with a single initial state. However, as the theory expanded to include multiple initial states, so too should the implementation software do. Both, state based and language based approaches can accommodate the use of multiple initial states with very little modification.

9.7 Synthesis Methods: Level versus Path

The exact synthesis algorithms used by *TCT* and *PLATEST* are not clearly documented. However, it is apparent that to maintain a history of transitions, transition sequences must be followed to their termination by way of a tree search or depth first algorithm. This method, from here on, is termed the *Path* method.

Alternatively, *fsm* uses a breadth first approach which will be termed the *Level* method. Each state in a level has the same minimum number of transitions between it and the

initial state. The solving algorithm adds every state in each level before going onto the next level. The algorithm is fast but contains no knowledge of previous transitions and cannot accommodate sequences of transitions in the operating specifications.

The Path method is more computationally intensive but provides many more options to the designer resulting in a more powerful tool.

9.8 Synthesis Algorithm

In an attempt to create a comprehensive synthesis tool that incorporates the positive aspects of all three previous software developments, an algorithm with the following main qualities has been devised:

- Procedural Controllers are built from simple component models,
- State variable values are tracked throughout to enable easy use of temporal logic for operating specification,
- Transition sequences are tracked to implement language based approaches and to enable sequences of transitions,
- Multiple and covered initial states are allowed to be used, and
- Modelling and operating specification constraints are used concurrently as part of an integrated synthesis approach.

The method uses recursion to maintain a history of transitions, tracing a sequence of transitions until the end is reached or a suitable termination state is found. Recursion automatically adds transition sequence history to the algorithm. To complete the synthesis capabilities, three transition sets are used giving the algorithm the ability to achieve qualities of the language based algorithms: *Forced Transition Sequence, Never Transition List and Next Transition List.* These variables provide a way to distinguish one state from another when their state variable values are identical. If any of the three variables do not match, the states are indeed different.

9.8.1 Forced Transition Sequence

When using the Path method, a state's identity involves the transitions that occur from it and, in particular, any forced transitions that occur from it. The Forced Transition Sequence is used to track the forced transitions from a state and is used to compare states. It contains the sequence of transitions that are forced from the state. If a state is not effected by a specification forcing transitions, the Forced Transition Sequence is empty.

9.8.2 Never Transition List

The Never Transition List contains transitions that can never occur from that state or any subsequent state reached from it. It is added when an operating specification states that a transition is never to occur after a certain state is reached or event occurs. The instruction to forbid the transition must be passed to every state reached from that point in order to maintain the assurance that the transition will never exist.

9.8.3 Next Transition List

The next Transition List is used with specifications that are activated based on whether transitions occur or not. It is a temporary variable used only while creating the immediate transitions.

The algorithm synthesises procedural controllers using the *Path* method enabling it to implement constraints involving transitions sequences and state based constraints. It is listed in algorithms 9.2 and 9.3. Algorithm 9.2 is the starting procedure and algorithm 9.3 is the main algorithm. Notice that, when a state is created in algorithm 9.3, it then calls itself with the new state as the argument.

As part of the overall synthesis of the procedural controller, trim structure calculations and tests for controllability are also performed. These algorithms are taken from previous work (Mouille, 1997; Rotstein and Macchietto, 1997) and were included in *fsm*.

Algorithm 9.2 (Start Procedure for new Synthesis Method)

Given the component models, the modelling constraints and the operating specifications
  Create NEW state as Initial State
  Assign Forced Transitions List = NULL
  Assign Never Transitions List = NULL
  CALL ALGORITHM 9.3 (Initial State, Forced Transitions, Never Transitions)

9.9 Summary and Conclusions

This chapter presented a new algorithm for the synthesis of procedural controllers. In doing so, the merits of three previous synthesis algorithms, implemented as computer programs were discussed.

It was found that combined state and language based approach was most beneficial as it enables process models and controllers to be built from simple models of individual elementary component behaviour. When synthesising controllers, process modelling constraints and operating specifications were considered simultaneously. This contributed to creating an efficient synthesis methodology that ensures the smallest structure is built.

Algorithm 9.3 (Recursive Method to Incorporate History into *fsm*)

```
PARSED VARIABLES (State, Forced Transitions, Never Transitions)

Create Forced Transitions List
Create Forbidden Transitions List
Create Never Transitions List
IF Forced Transitions DOES NOT EQUAL NULL THEN
  Add Forced Transitions to Forced Transitions List
ENDIF
IF Never Transitions DOES NOT EQUAL NULL THEN
  Add Never Transitions to Never Transitions List
ENDIF

DO For each Dynamic Constraint
  IF FSM State = Dynamic Constraint State THEN
    Add Temporal Logic information to respective list
  ENDIF
ENDDO
IF Transition IS in First Forced Transition THEN
  IF Dynamic Constraints DO NOT conflict THEN
    IF no static constraint forbids the NEW state THEN
      IF another FORCED transition follows (i.e. sequence) THEN
        Create a NEW state labelled with a sequence flag
        Delete Forbidden Transitions List
        Remove first FORCED transition from Forced Transitions
        CALL ALGORITHM 9.3 (NEW State, Forced Transitions, Never Transitions)
      ELSEIF destination state exists THEN
        Add transition and terminate it at destination state.
      ELSE Create the NEW state and add transition.
        Delete Forbidden Transition List.
        CALL ALGORITHM 9.3 (NEW State, Forced Transitions, Never Transitions)
      ENDIF
    ENDIF
  ENDIF
ELSEIF Transition IS NOT in Forbidden THEN
  DO FOR each state variable
    DO For each transition possible from the state variable's value
      IF no static constraint forbids the system destination state THEN
        IF NEW state exists THEN
          Add transition and terminate it at destination state.
        ELSE Create the NEW state and add transition.
          Delete Forbidden Transition List.
          CALL ALGORITHM 9.3 (NEW State, Forced Transitions, Never Transitions)
        ENDIF
      ENDIF
    ENDDO
  ENDDO
ENDIF
```

The algorithm was also extended to include the new developments for the case of multiple and covered initial states. These improvements were implemented in a software package called Sequence Solver.

Chapter 10

Guided Specification of Procedural Controllers

10.1 Introduction

As many industrialists have been known to comment about academia, no theoretical advances are useful unless they can be implemented. So, there is a need to show that what has been presented in this work can actually be done.

A computing package is required. To gain acceptance, it must meet the following criteria:

- the technical jargon and mathematical formulae must be replaced by a common, easy to understand language, for problem definition and results presentation,
- the presence of thorough error checking and guidance during problem definition must help the designer so that impossible or implausible situations can not be defined, and
- it must be possible to translate the constraints and resulting procedural controllers into a range of results formats with minimal effort from the user.

This chapter outlines the development of an easy to use computing environment which aims to satisfy the above mentioned needs by being capable of the following :

- guiding the user through the process of model definition and operating specification in comfortable way with error checking,
- interacting with the technical solver to produce results (synthesise procedural controllers) in a logical and understandable presentation, and
- automatically generating simulation and control code plus documentation for the generated procedural controllers.

Part of this thesis' objective was to design a prototype package enabling the use of this technology by those not intimate with the theory. For this purpose, a software package, named Sequence Developer, has been developed with an easy to use environment.

10.2 Sequence Developer

A program for the management of specification, synthesis and translation of procedural controllers (called Sequence Developer) has been developed using the JAVA programming language. It has a user friendly GUI as shown in Figure 10.1 with standard features expected in today's software packages such as menus, toolbars and thorough file handling facilities. The purpose of Sequence Developer is to provide a user friendly interface to the PCT synthesis software, Sequence Solver, described in chapter 9. Figure 10.2 shows how the two software packages interact.

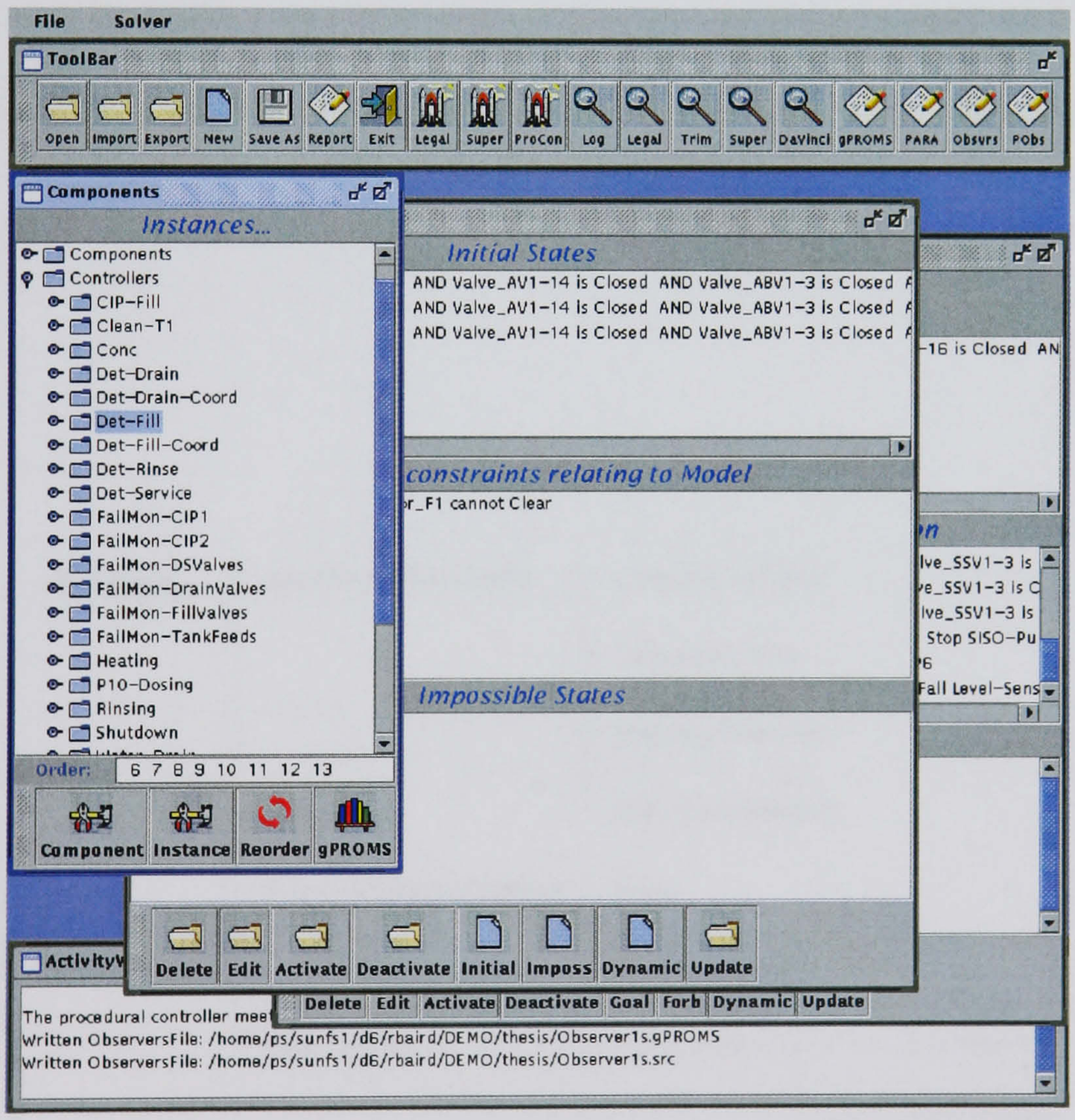


Figure 10.1: Main Window of the *Sequence Developer* GUI.

Figure 10.3 shows the hierarchy of the features present in Sequence Developer. There are three main categories that meet the current requirements: guided modelling and operating specification, Procedural Controller Synthesis and Results Handling.

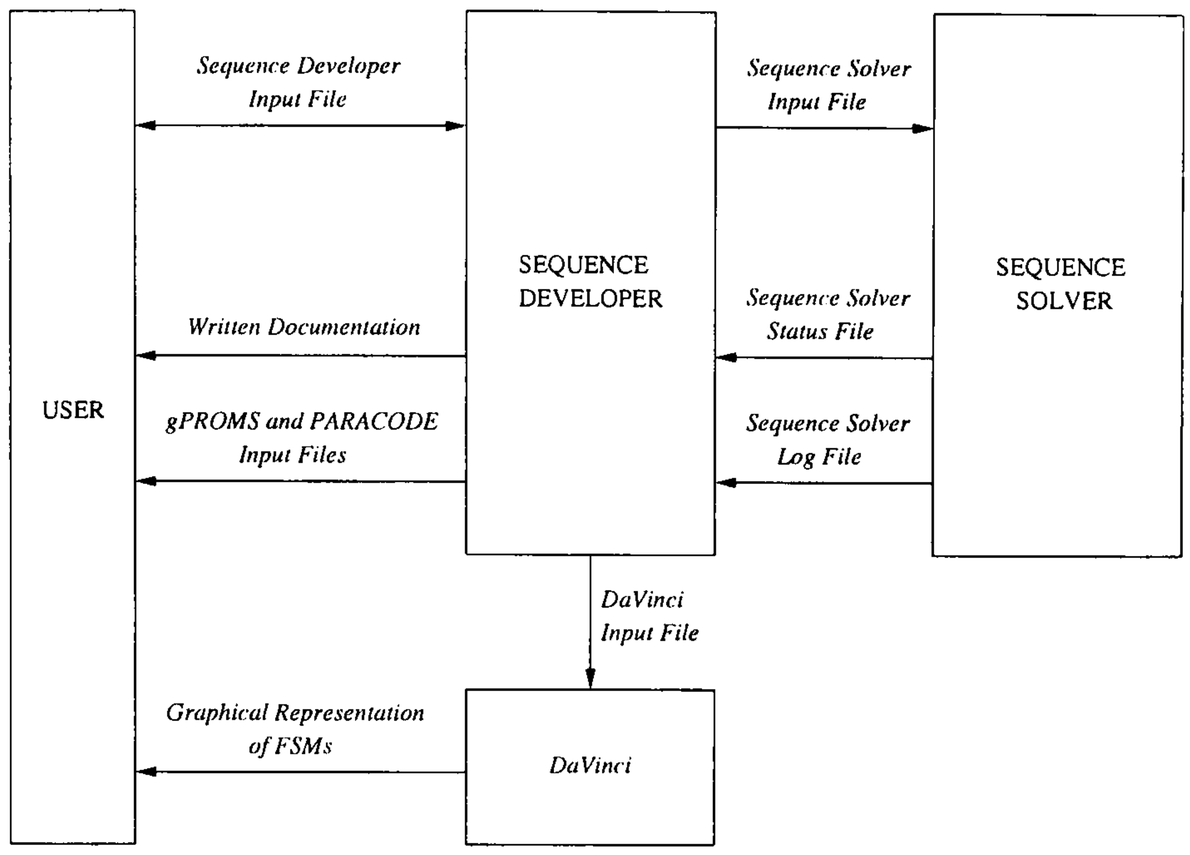


Figure 10.2: How *Sequence Developer* interacts with the user, *Sequence Solver* and *DaVinci*.

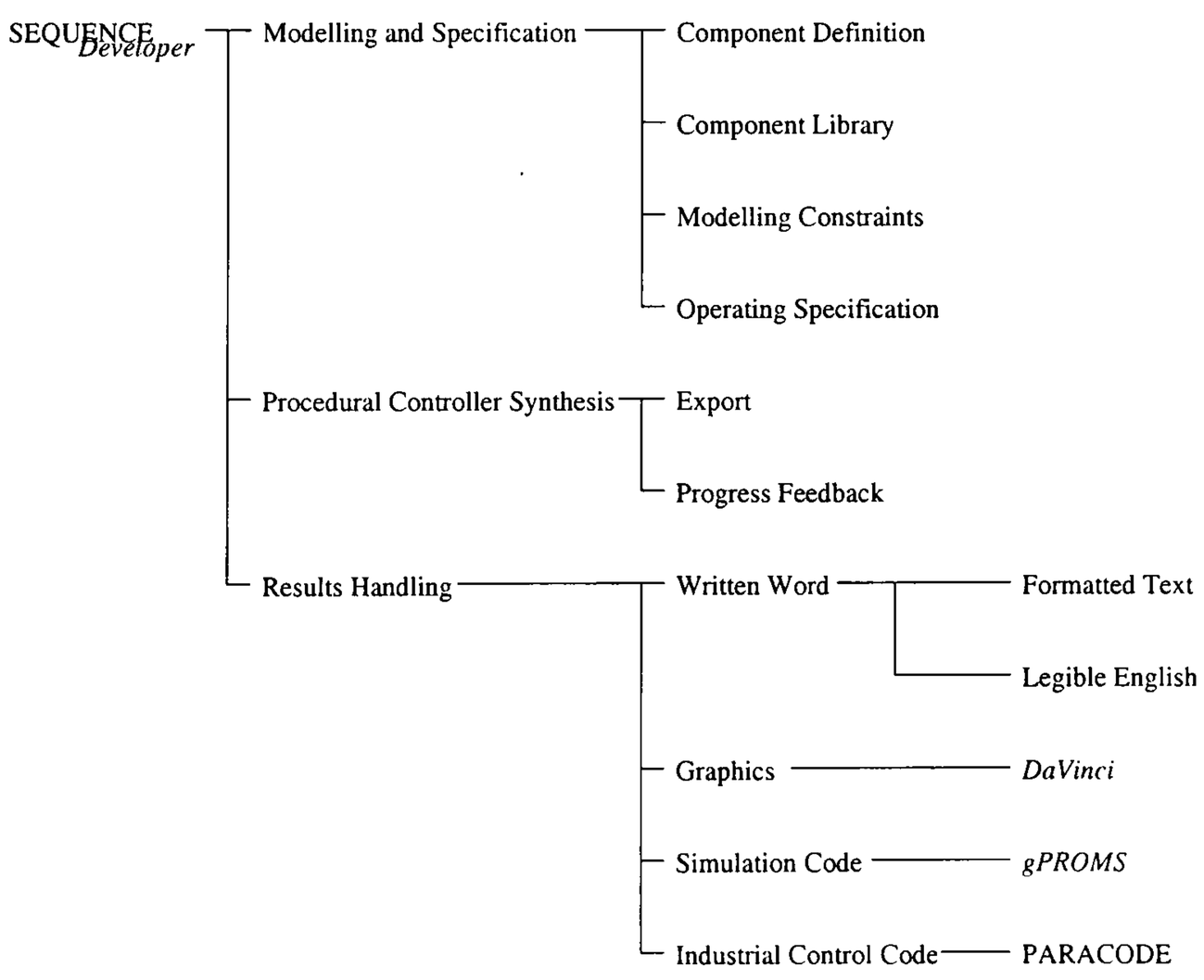


Figure 10.3: Sequence Developer's Features Hierarchy.

10.2.1 Guided Modelling and Operation Specification

The main motivation behind the development of Sequence Developer was to establish an easier way to define process models and to specify operating objectives. This is done using four main categories: Component Definition and Instantiation, Component Libraries. Modelling Constraints and Operating Specifications.

10.2.1.1 Component Definition and Instantiation

PCT builds process models from elementary component models. Definition of these models is done through a Component Builder. Using the Component Builder eliminates

- the use of illegal characters (such as spaces in names),
- spelling errors,
- semantic errors (such as transitions ending at states that do not exist), and
- duplication of names.

This is achieved by scanning all the text entered and by limiting the text input. When defining transitions, users are forced to select originating and destination states from lists of valid options. Figure 10.4 shows the definition of a component, *Valve*. In the top half of the figure is the display of current states (on the left) and transitions (right). States are added using the lower left section and transitions, the lower right. The figure shows the act of adding a transition, *Close*. The originating (or *From*) state, *Open*, is selected in the left list of the *Adding a Transition* section and the destination (or *To*) state is selected in the right hand list. A toggle below the lists sets the transition to controllable or to uncontrollable.

Generally, component definition is done at the beginning of a project. However, components can be added, edited or removed at any stage of a project and the changes propagate through the project.

Once components have been defined, instances of it can be created. To explain, a component is an entity that can occur several times in a process. For example, a process can have many valves. Each occurrence is an instance of the component *Valve*. The Object Orientation requires that component details need only be defined once for several instances. The whole program's object structure is shown in Figure 10.5. It shows how components can contain instances as well as states.

The list of components and associated instances are shown in the Components Internal Window of the Main Window of the program. Figure 10.6 shows the components and instances for the CIP Heating example.

10.2.1.2 Component Libraries

Flexible batch plants, by design, have a large number of components and operating procedures to match their multiple uses. Defining, designing and synthesising these procedures

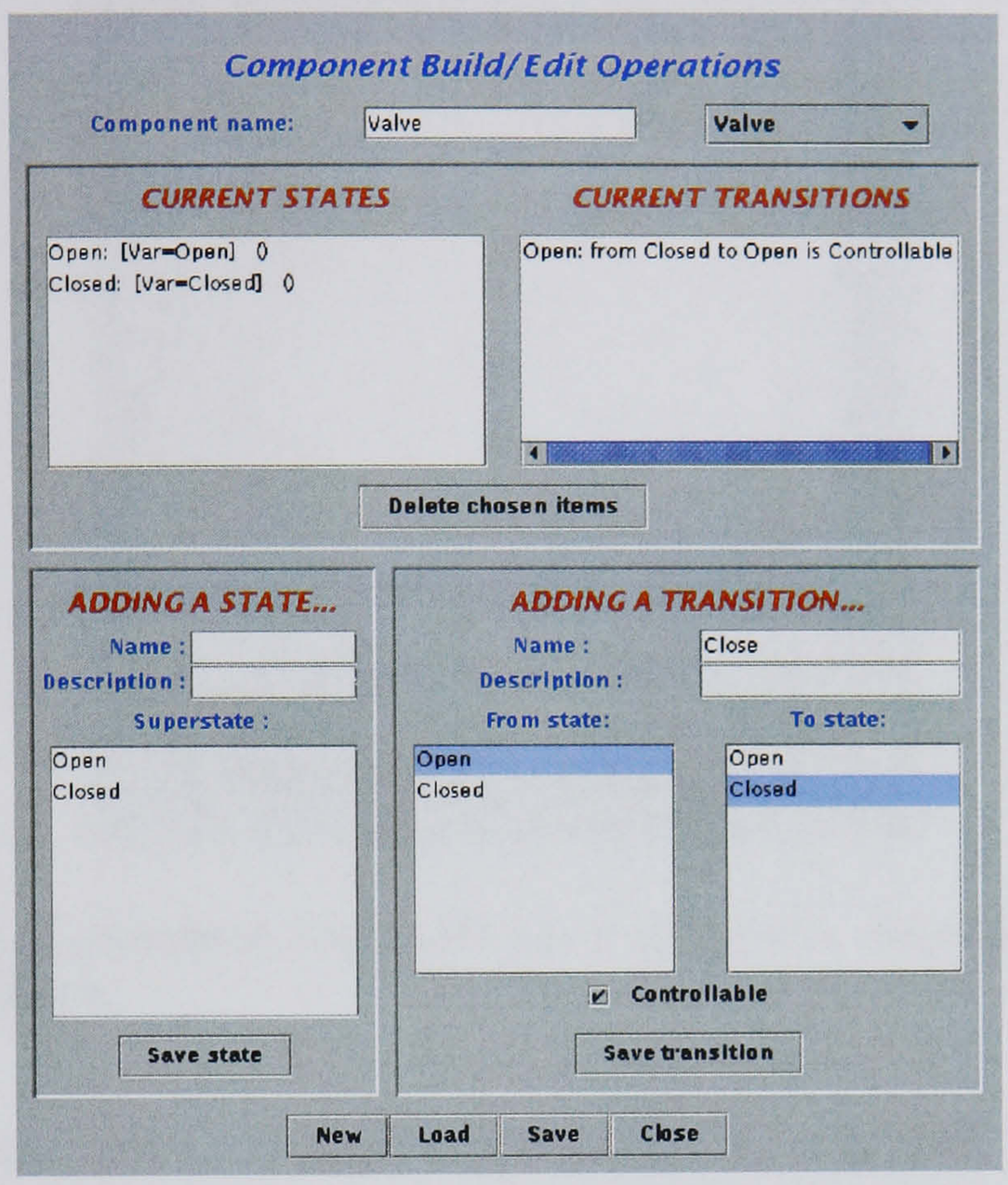


Figure 10.4: Sequence Developer's Component Definition.

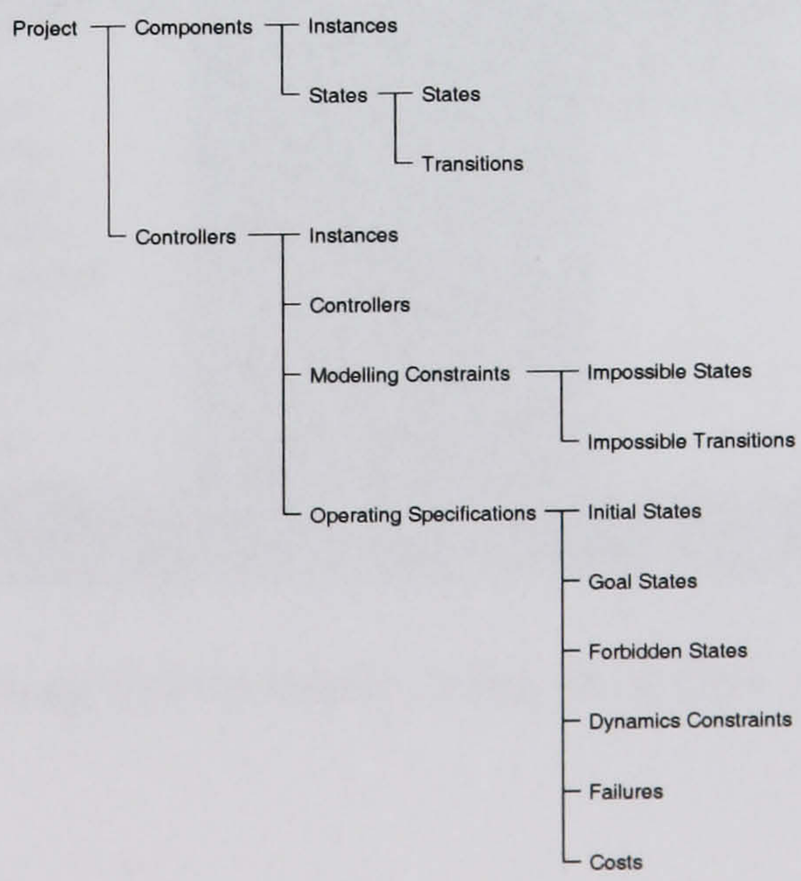


Figure 10.5: Sequence Developer's Object Oriented Hierarchy.

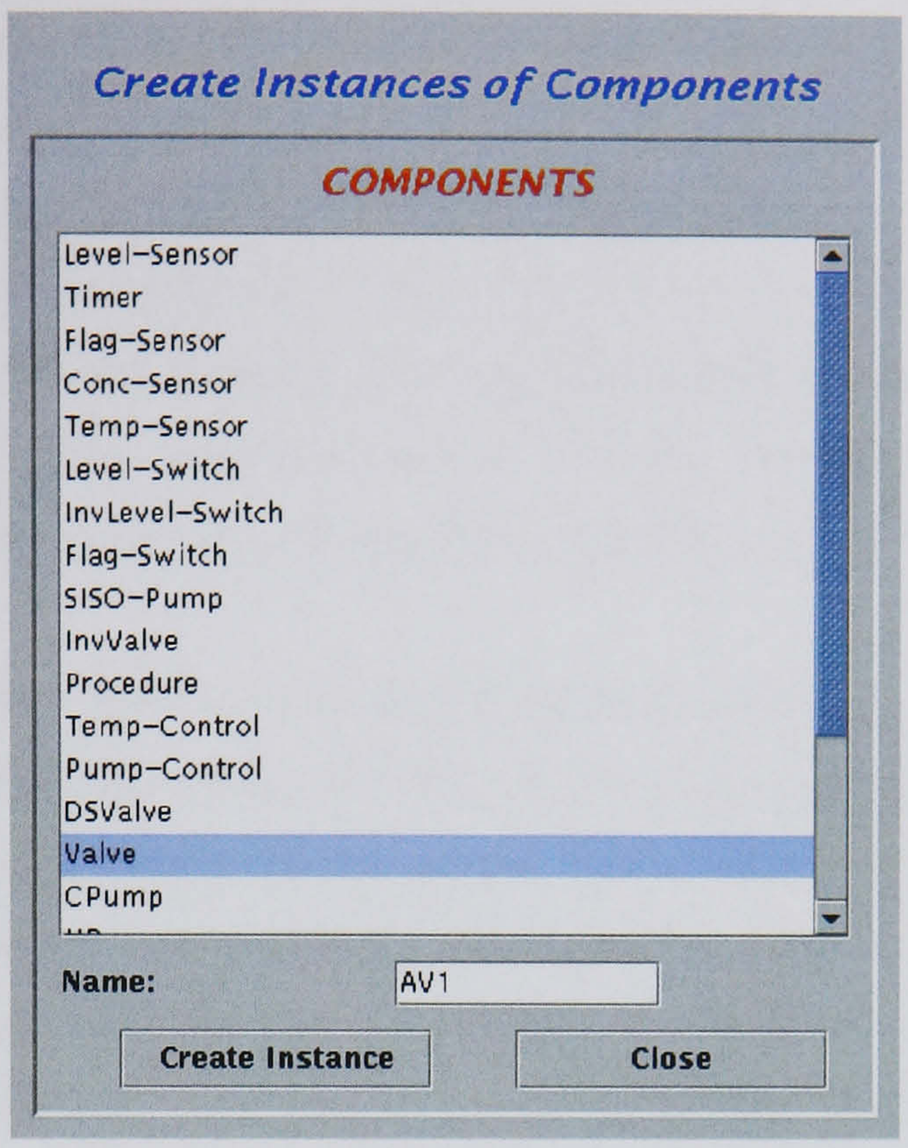


Figure 10.6: Current Component Instances are displayed in Sequence Developer's Component Instances Window.

can be time consuming and error-prone mainly due to repetition. To avoid this fact, a component library facility has been included in Sequence Developer. It enables designers to Save, Load, Delete and View (both, as text and graphics) components in the current project or an external library. Figure 10.7 shows the library window.

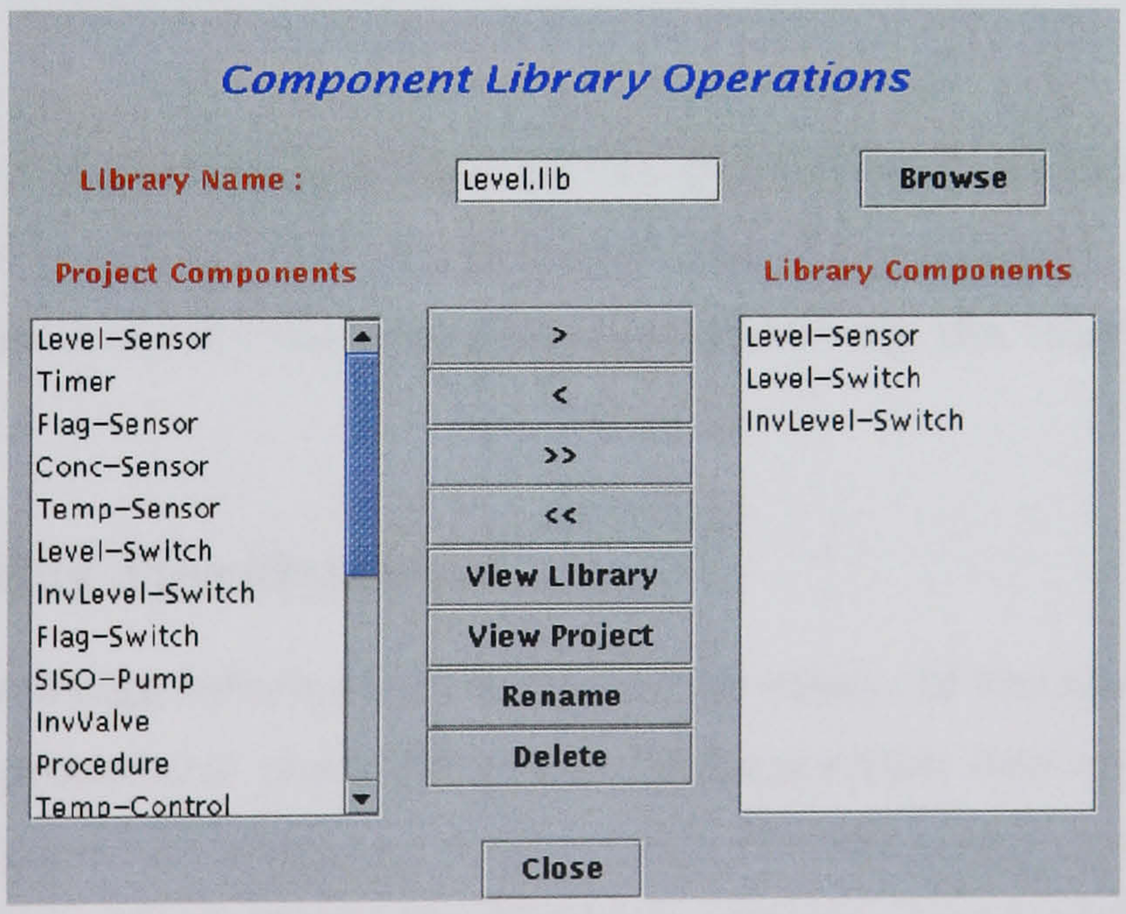


Figure 10.7: Load and saving Components using Sequence Developer's Component Library.

10.2.1.3 Process Modelling

As in theoretical developments, the tasks of process modelling and operating specification are separated within Sequence Developer. When modelling the process, the whole process

is considered as a decomposition only to simplify definition of the operating specifications. This also allows Sequence Developer to establish the relationships between components.

Both Static Constraints (referred to as Impossible States) and Dynamic Constraints (Impossible Transitions) are defined. They are presented in the Modelling Constraints internal window in structured English for easy understanding, as shown in Figure 10.1. Constraint definition restricts keyboard input to solely the name of the constraint. Otherwise, all the information can be provided by selecting it from from filtered option lists using the mouse.

As an example, consider the Static Constraint Definition window shown in Figure 10.8. The designer selects an instance from the list on the upper left of the figure and the states within it are automatically listed on the right. Thus, the risk of defining a non-existent state is removed. Likewise, the definition of conditions within a dynamic constraint is done by following the same process. When specifying the forced or forbidden transitions of a dynamic constraint, filtering also occurs based on the dynamic constraint's conditions. Only valid transitions exiting a state are shown in the option list. The formula defining the set of valid transitions is given by:

$$\Sigma_{option} = \{t \in \Sigma | t \in \Sigma_u \wedge (t, state(\mathcal{I}))!\} \tag{10.1}$$

where,

$\mathcal{I}$	is the selected instance, and
$state(\mathcal{I})$	is the state of the instance $\mathcal{I}$ in the current dynamic constraint's conditions.

In other words, a transition is a valid option if it is uncontrollable and comes from a state of the instance which is listed in the conditions of the dynamic constraint. In Figure 10.9, transitions from a level sensor are filtered so that only the transitions for the *normal* state are shown.

10.2.1.4 Operating Specification

The majority of the design information is entered by means of the operating specifications. It is here that the procedural controllers, the decomposition and controller hierarchy are defined and synthesised. As shown by Figure 10.5, the operating specification also entails defining Initial States, Marked (or Goal) States, Static Constraints (Forbidden States) and Dynamic Constraints (Operating Commands). The specification of initial states, marked states and static constraints is done in the same way as the specification of static constraints for the process model. Dynamic constraints have the same filtering techniques as for process modelling except that the transition filtering removes uncontrollable transitions instead of controllable transitions. Thus, equation 10.1 becomes:

$$\Sigma_{option} = \{t \in \Sigma | t \in \Sigma_c \wedge (t, state(\mathcal{I}))!\}. \tag{10.2}$$

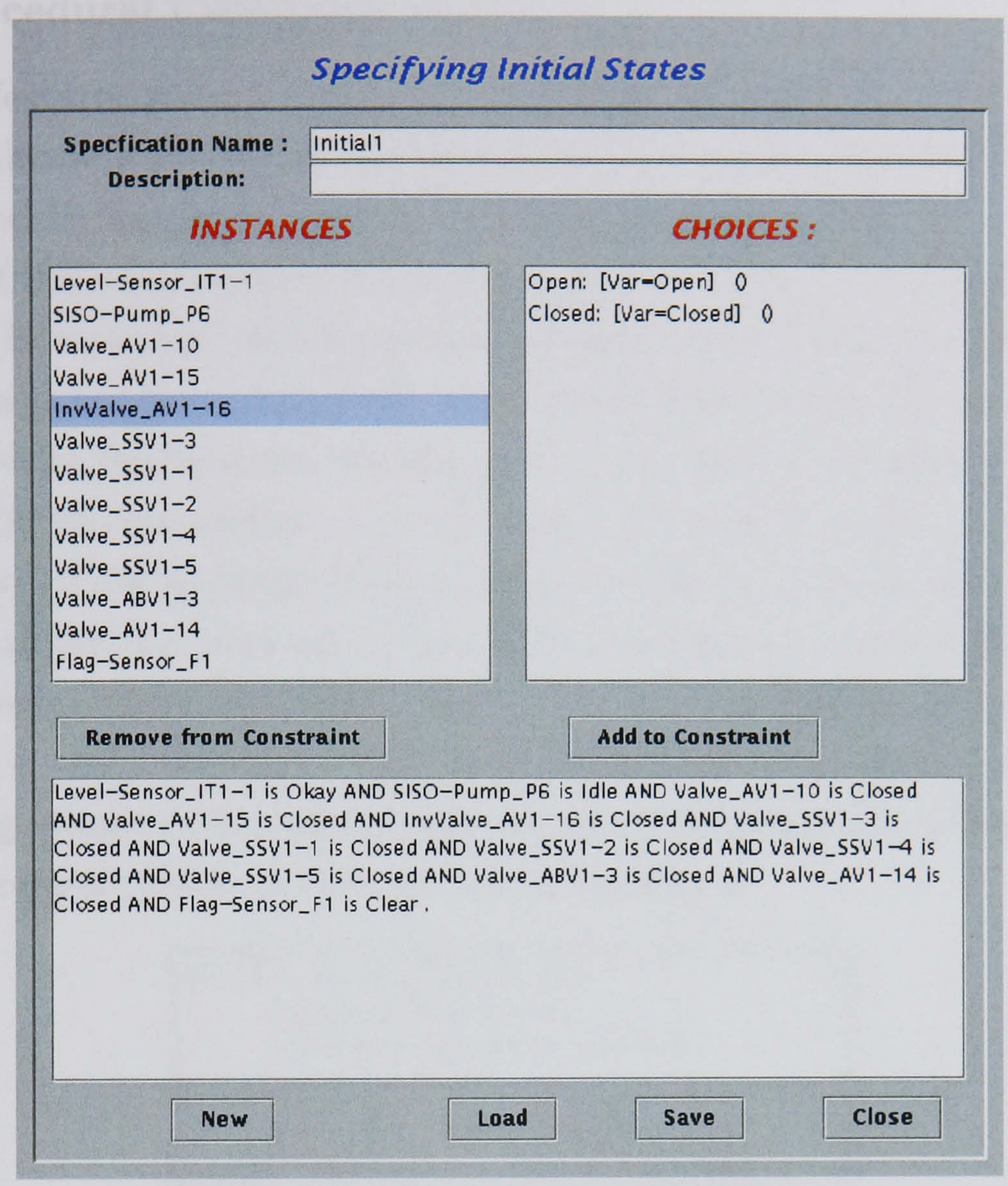


Figure 10.8: Defining Static Constraints using Sequence Developer.

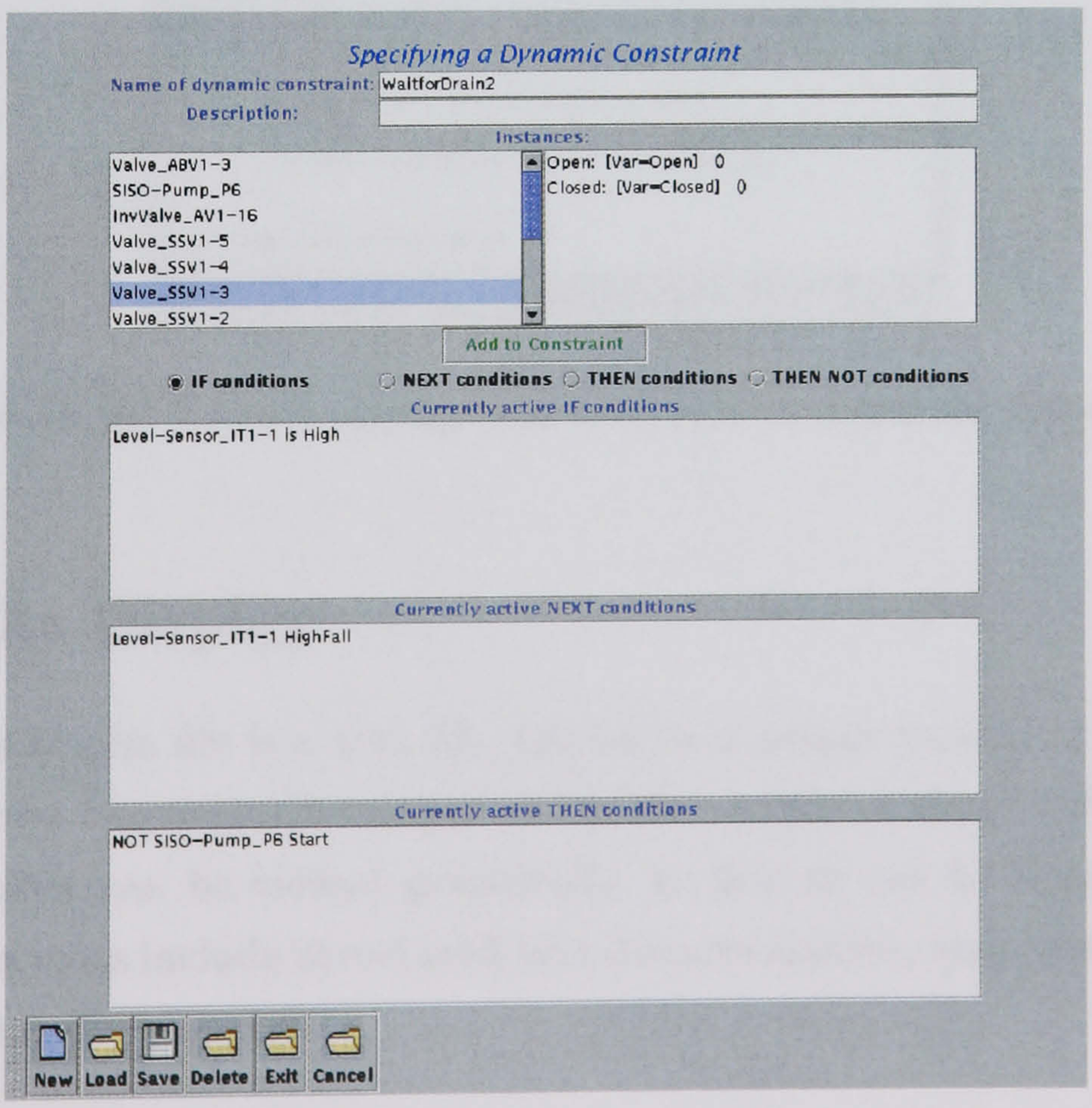


Figure 10.9: Defining Dynamic Constraints using Sequence Developer.

10.2.2 Procedural Controller Synthesis

User see two features when a procedural controller is synthesised. First, they make a request to synthesise a procedural controller and Sequence Developer generates an export file which is used by Sequence Solver to synthesise a procedural controller. The significant contribution of this work however is the feedback that is provided when Sequence Solver has completed its execution. While previous programs simply provided a resultant procedural controller in the form of an FSM (which could often be a null structure), Sequence Developer provides feedback for the designer to aid trouble shooting and thus reduce design time. On the completion of an execution of Sequence Solver, a window appears with a summary of the synthesis (Figure 10.10). Notice how at the bottom the designer is informed that the synthesis was unsuccessful with regard to a procedural controller. Warnings at the top of the figure point the designer to the possible causes of the problems. A detailed log is also produced. Figure 10.11 shows part of the corresponding synthesis log. A warning related to a conflict regarding the specifications is shown and the three lines above it suggest which constraints may be causing it.

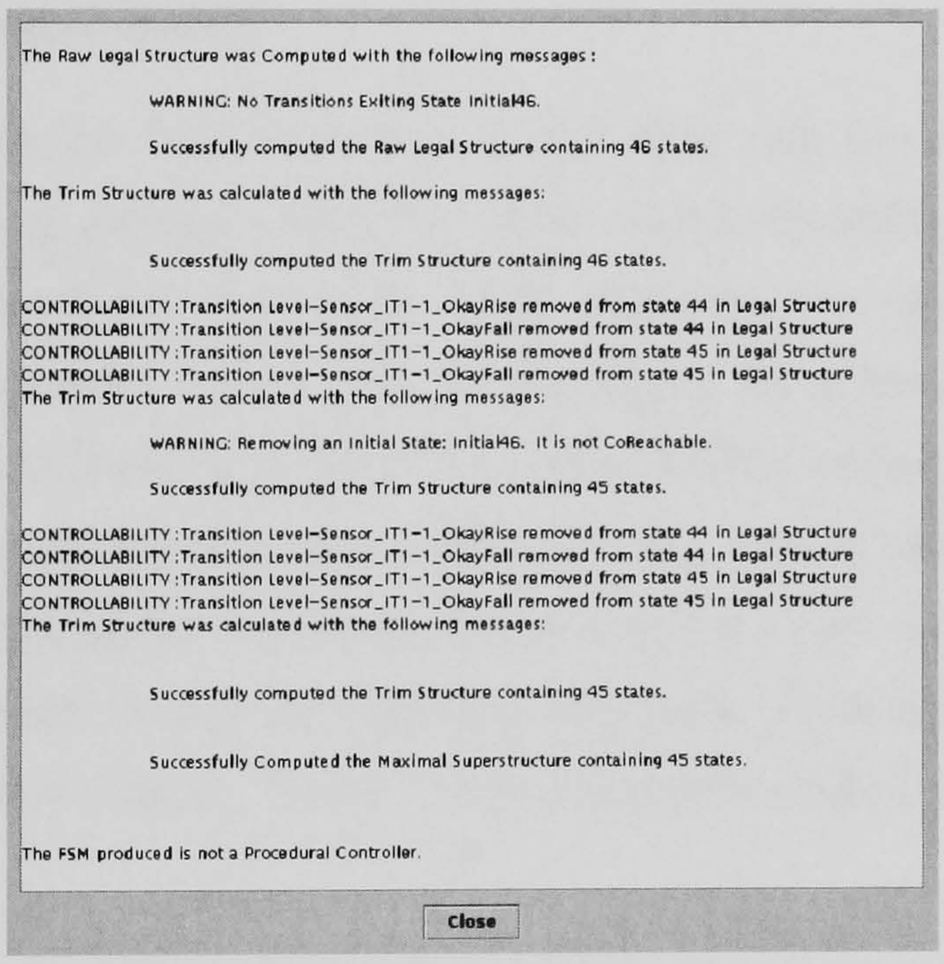


Figure 10.10: Results display and feedback in *Sequence Developer*

10.3 Results Display

Sequence Solver's results file is a text file written in a unique format that, together with the input file allows Sequence Developer to handle results in three different ways. The procedural controller can be viewed graphically, as text or can be translated into other formats. These formats include structured text documentation, simulation languages and industrial control code languages.

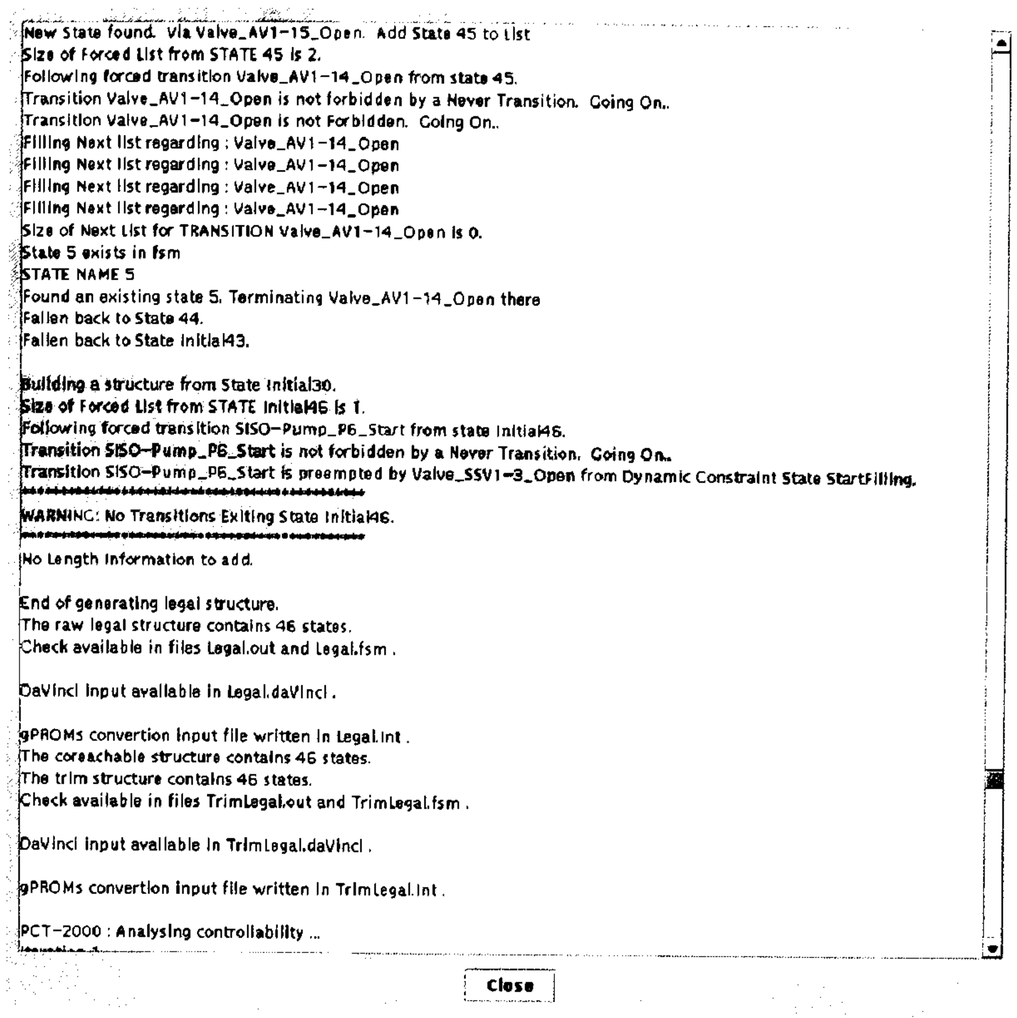


Figure 10.11: Viewing *Sequence Developer*'s synthesis log.

10.3.1 Viewing Graphically

Every execution of Sequence Solver creates a file that can be used as an input file for the graphical flowcharting package *DaVinci* from the University of Bremen (University of Bremen, 1998). Procedural controllers and elementary components can be viewed graphically using this package which can be launched from Sequence Developer's main window. Most suitable for drawing small FSMs, this facility enhances the designer's ability to quickly understand the design of components and the procedural controllers they belong to. It is accepted that graphical representations are the most easily understood formats and, thus, they are essential to any world class software. Typical products from *DaVinci* are shown throughout this thesis. Some other examples were shown in Figures 6.6 and 6.8.

10.3.2 Structured Text Documentation

A recurrent problem in all areas of engineering is the lack of documentation, particularly if computers are involved. Therefore, the automatic generation of documentation to accompany procedural controllers is a significant step forward. The translation algorithm, although simple, is still in its early stages of development. If the rule below is followed, quite legible documentation can be produced.

Rule 10.1 *For sensible documentation to be produced from Sequence Developer in English, all transition names should be labelled using the infinitive verb describing the action being performed.*

For example, a valve's transition should be called *Open* and *Close* as these can then be used to automatically produce documentation like:

```
If Valve AV1-10 is Open
  and the next transition is Valve AV1-40 Closing
  then Close AV1-10.
```

In addition to the English documentation, mathematical formulae corresponding to the specification can also be produced for more technical purposes. This involves direct translation to a rigorous structure without the need to consider sentence structure. The above example also generates the following mathematical expression:

$$(Open,\ \infty, \infty, \infty) \wedge (\tau = \texttt{AV1} - \texttt{40 Closing}) \rightarrow \bigcirc[\tau = \texttt{AV1} - \texttt{10 Close}]$$

Both styles of documentation are produced as LaTeX input files, a shareware type setting package. A plain text version of the English documentation can also be produced.

10.4 Automatic Control Code Generation

As mentioned in Chapter 8, automatic code generation is essential to the integrity of formal methods applications. The objective of translation techniques in this thesis is to make them as automated as possible with the minimum amount of effort from the user as possible.

Previous translation methods (Alsop, 1996) have merely provided a mapping between transitions and lines of control code supplied in an external text based driver file. However, the more interaction and input that the user is allowed, the higher the risk of errors penetrating the code. Thus, Sequence Developer guides and limits the user as to what they can input. All the required information is entered through a Control Code Definition Window, launched from the Component Instances Window. Figure 10.12 shows the window used to enter information for the generation of PARACODE, a proprietary language from APV (APV, 1994). Of importance is the limited amount of text which can be entered. There is a limit to the amount of text to enter, thus reducing errors but, component names and instance references provided by the user. The PARACODE commands are a specific instruction set and thus the user input can be restricted to selection from a list, as shown by the drop-down combo-boxes.

An almost identical window appears for the definition of the simulation language *gPROMS*. Other languages can easily be added to these. Once all translation information is entered and a procedural controller is synthesised, the *gPROMS* code for observers, controllers tasks and procedural controllers can be automatically generated from the main toolbar of Sequence Developer. After testing the control logic with the discrete/continuous process model, the same controllers for PARACODE can be automatically generated in the same fashion and implemented on the real plant.

The generated code and its structure adhere to the structure proposed in chapter 8. Therefore, control code is generated for Process Interpretors (Process Model Observers and Controller Tasks) plus the Procedural Controllers.

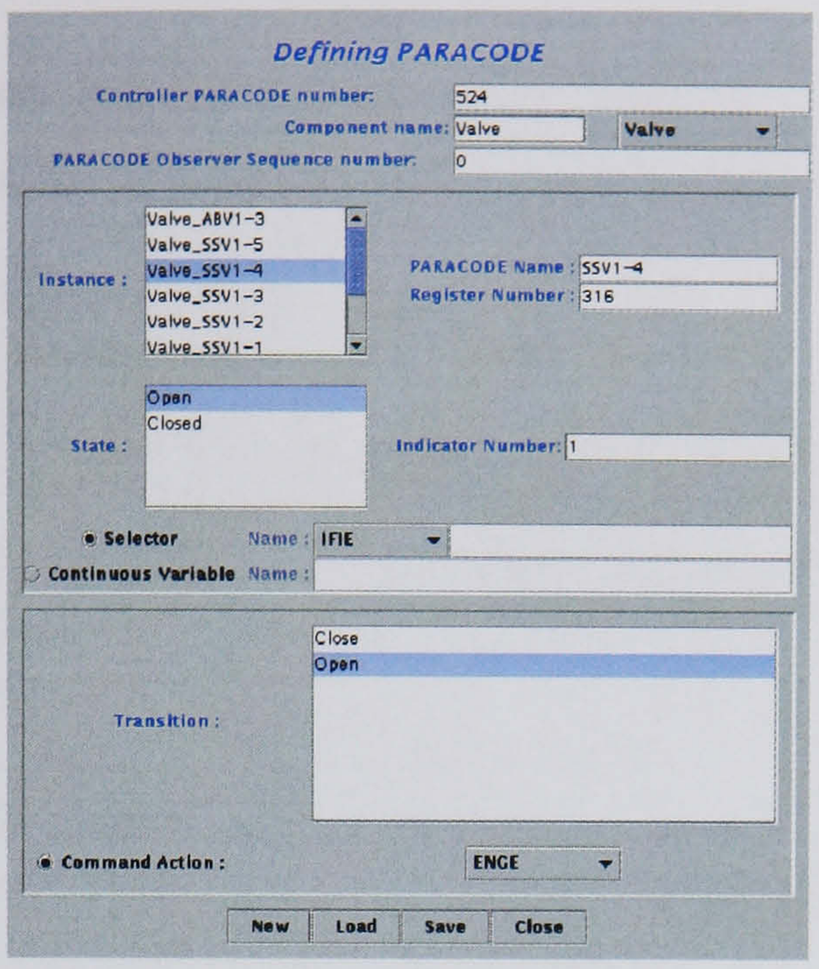


Figure 10.12: *Sequence Developer*'s PARACODE translation input window.

10.4.1 Generation of Process Interpretors

As explained in chapter 8, process interpretors lie between the controllers and the process to enable communication between the purely discrete and the discrete/continuous domains. They enforce structure and formality on the controller and make automatic code generation easier.

Process Model Observers Process Model Observers scan the signals from the process and inform the controller of the discrete state of each elementary component instances. This is done with conditional statements to assess the elementary component instance's state. The specific conditional statements are entered as part of the state specification in the Control Code Definition Window. An integer variable is used to convey active state information to the procedural controller. Values are chosen by the *Indicator Number* in the Control Code Definition Window. With the knowledge of the target language, code for the process model observers is generated using algorithm 10.1.

An example of a process model observer for a valve in PARACODE is shown in Figure 10.13. The figure shows how the observer reflects the component's statechart with unique line labels for each state and the assignment of the variable R2.320 for the Indicator Number. Transitions between states are observed by the conditional statements and `GOTO` directives issued when the conditions are true. Examples of process model observers for *gPROMS* implementations were given in Figures 8.5 and 8.3.

Controller Tasks Controller Tasks are used to drive process items. They are especially useful in simulation code as they collect the failure monitoring information which is automatically done as part of industrial control code. Controller tasks use the command actions entered into Sequence Developer in the transitions definition section of the Code Definition Window. With this, the tasks are generated by algorithm 10.2. Examples of

Algorithm 10.1 (Process Model Observer Generation)

1 Get list of Elementary Components
2 DO For all Elementary Components
  2.1 Append Observer File Header (e.g. Component Name)
  2.2 DO For all states in the Elementary Component
    2.2.1 Append New Line Label and/or assign Indicator Number to indicator variable
    2.2.2 DO For all transitions from current state
      2.1.2.1 Get Destination state
      2.1.2.2 Get Destination state control code
      2.1.2.3 Append conditional statement:
        "IF state variable = Destination state control code
        GOTO Destination State Line Label"
      2.1.2.4 Append statement to loop in current state
    2.2.3 ENDDO
    2.2.4 DO For all substates of current state
      GOTO Step 2.2.1
    2.2.5 ENDDO
  2.3 ENDDO
  2.4 Append File or Task end comments
3 ENDDO
5 Output Observer Files

gPROMS controller tasks were shown in Figures 8.5 and 8.3.

10.4.2 Procedural Controller Translation

Procedural controllers are dynamic models of process operating sequences. To translate them into usable control code in accordance with general industrial practice, the code is divided into three main regions: *Inhibits, Pre-checks* and *General Logic.*

10.4.2.1 Inhibits

Placed at the start of control code, sequence inhibits ensure that no conflicting controllers operate in parallel. Techniques for generation of inhibit control logic have already been devised in Alsop (1996). Note that, controller inhibits are redundant in hierarchical PCT because higher level controllers ensure that no conflicting controllers operate in parallel.

10.4.2.2 Pre-Checks

Controller pre-checks ensure that the controller starts in the correct initial state by checking the state variable values of each elementary component instance in the controller. Then, the appropriate initial state (if one exists) is selected and the controller allowed to

```
.SEQ 2.320
/
/ PARACODE Observer File for
/ Component Instance : AV1-16
/
0320TS1 IFIF    AV1-16, 0320SS2         / Test for AV1-16 Normal
/
0320SS1 HOVN    0, R2.321               / AV1-16 IS Normal
/
0320T0  IFIE    AV1-16, 0320S0          / Test for AV1-16 Closed
        GOTO    0320T1                  / STEP TO Open TEST
0320S0  HOVN    0, R2.320               / AV1-16 IS Closed
/
0320T1  IFID    AV1-16, 0320S1          / Test for AV1-16 Open
        GOTO    0320TS1                 / STEP TO Normal TEST
0320S1  HOVN    1, R2.320               / AV1-16 IS Open
        GOTO    0320TS1                 / STEP TO Normal TEST
/
/
0320TS2 IFINF   AV1-16, 0320SS1         / Test for AV1-16 Abnormal
/
0320SS2 HOVN    1, R2.321               / Valve IS Abnormal
/
0320T2  IFOE    AV1-16, 0320S2          / Test for AV1-16 Closed
        GOTO    0320T3                  / STEP TO Open TEST
0320S2  HOVN    0, R2.320               / AV1-16 IS Failed Closed
/
0320T3  IFOD    AV1-16, 0320S3          / Test for AV1-16 Open
        GOTO    0320TS2                 / STEP TO Abnormal TEST
0320S3  HOVN    1, R2.320               / AV1-16 IS FAILED_OPEN
        GOTO    0320TS2                 / STEP TO Abnormal TEST
/
0320Z   STOP
/
```

Figure 10.13: An example of an automatically generated PARACODE Observer.

start.

For each initial state, a set of conditional statements, usually `IF-THEN` statements, are used to assess state variable values. If all conditions are true for an initial state, a directional statement (`GOTO`) that follows is executed and the controller starts executing from that initial state. Otherwise, the pre-checks continue with the next initial state's conditional statements until the active state is found. If no initial state is suitable, an error message is given. The algorithm for the generation of the pre-check code is given in algorithm 10.3.

The use of algorithm 10.3 is shown on the CIP-Heating procedure introduced in chapter 6. Consider a sub-phase required to circulate fluid from the CIP tank through HE3 using P6. The procedural controller contains five elementary component instances: *P6*, *IS1-1*, *IS1-2*, *AV1-25* and *AV1-14*. It then defines the state of the system (in that order). For brevity, interlocked components have been omitted. Two initial states are allowed: *(Idle, Covered, Covered, Closed, Closed)* and *(Idle, Uncovered, Covered, Closed, Closed)*. The pre-check code generated is shown in Figures 10.14(a) (*gPROMS*) and 10.14(b) (PARACODE).

10.4.3 Generation of General Logic

Generation of the general logic for procedural controllers follows the same path presented in Alsop (1996). Each state in the procedural controller is considered as a milestone in the control code and signified by a unique line label. Then, code for that state is generated. Depending on whether the state is marked or not, the added code is different. Finally,

Algorithm 10.2 (Controller Task Generation)

```
1 Get list of Elementary Components
2 DO For all Elementary Components
  2.1 Append Controller Task Header (e.g. Component Name)
  2.2 DO For all states in the Elementary Component
      2.2.1 DO For all transitions from current state
            2.2.1.1 IF transition is controllable THEN
                  2.2.1.1.1 Append Transition Command from
                        from Control Code Definition Window
                  2.2.1.1.2 Append failure assessment information
            2.2.1.2 ENDIF
      2.2.2 ENDDO
  2.3 ENDDO
  2.4 Append File or Task end comments
3 ENDDO
```

code regarding each transition from that state is added. General logic control code is generated using algorithm 10.4.

Marked States Procedural controllers are either self-terminating or externally terminated. If a controller is self-terminating, code is generated (with an appropriate operator message) to stop the controller. Otherwise, a message is generated, a flag is set and the code is generated for the transitions leaving the state.

Transitions For the purpose of code generation, transitions are classified as controllable or uncontrollable. Controllable transitions generate code that requests controller tasks to force actions on the plant. Request code is generated with the knowledge of the controller task name and the elementary component instance being effected. Uncontrollable transitions generate conditional statements that scan the Process Model Observers to assess what discrete state each elementary component instance is in and whether a transition has occurred or not. After each transition, controllable or uncontrollable, a directive statement is present to take the controller to the correct state after the transition has occurred.

Example In this section, algorithm 10.4 is demonstrated for the CIP-Heating controller introduced in chapter 6. Figures 10.15(a) and 10.15(b) show the code produced for a *gPROMS* simulation and PARACODE implementation, respectively. These were chosen for testing convenience. However, the algorithm can be applied to the generation of any format.

It is observed from Figure 10.15. that each state has a portion of code with a de-

Algorithm 10.3 (Pre–Check Generation)

1 Get FSM
2 DO For all states in FSM
  2.1 IF state and initial state THEN
    2.1.1 Append New Line Label
    2.1.2 DO For all state variables is initial state
      2.1.2.1 IF state variable is NOT ∞ THEN
        2.1.2.1.1 Get state variable Indicator Number
        2.1.2.1.2 Append conditional statement:
          "IF state variable $\neq$ Indicator Number"
        2.1.2.1.3 Append `GOTO` next Initial State Check Line Label
      2.1.2.2 ENDIF
    2.1.3 ENDDO
    2.1.4 Append `GOTO` Initial State Line Label
  2.2 ENDIF
3 ENDDO
4 Append error message informing operator now active initial state
5 Output program

fined structure. A marked state, number 5, shows how the controller is terminated while others show how controllable and uncontrollable transitions are implemented. State 1, implements the controllable transition requesting that valve AV1-25 open (or become energised). State 6 is an example of a wait state with uncontrollable transitions, implemented by conditional statements, assessing the state of elementary components as determined by the process model observers.

10.5 Summary and Conclusions

This chapter introduced the design and implementation of a software program including a graphical user interface, Sequence Developer, which aims to make the theoretical advances in PCT more accessible and accepted by providing a familiar and easy to use style of software package. Sequence Developer achieves this in the following three ways:

1. By guiding the user through component definition, discrete process modelling and operating specification. This has been simplified by the presence of error checking in the component definition and, then, by limiting the user input to mainly selection from lists rather than text input. Furthermore, a component library alleviates the repetition of component definition tasks.

2. By providing an interface to the solving engine Sequence Solver, described in Chapter 9, so that the user receives feedback regarding the synthesis of procedural con-

```
IF (state = 0) THEN
  SEQUENCE
    IF ((Plant.CIP_Unit.P6.Indicator0 = 0) AND
        (Plant.CIP_Unit.CIP.IS1_1.Indicator0 = 1) AND
        (Plant.CIP_Unit.CIP.IS1_2.Indicator0 = 1) AND
        (Plant.CIP_Unit.AV1_14.Indicator0 = 0) AND
        (Plant.CIP_Unit.AV1_25.Indicator0 = 0)) THEN
      state := 1;
    END # If
#
    IF ((Plant.CIP_Unit.P6.Indicator0 = 0) AND
        (Plant.CIP_Unit.CIP.IS1_1.Indicator0 = 0) AND
        (Plant.CIP_Unit.CIP.IS1_2.Indicator0 = 1) AND
        (Plant.CIP_Unit.AV1_14.Indicator0 = 0) AND
        (Plant.CIP_Unit.AV1_25.Indicator0 = 0)) THEN
      state := 4;
    END # If
#
    IF (state = 0) THEN
      SEQUENCE
        HESSAGE "CIP Heating IS NOT in
                 A CORRECT INITIAL STATE!!"
          state := 999;
      END # SEQUENCE
    END # IF
  END # SEQUENCE
END #IF
```

(a) *gPROMS*

```
/
/ PRECHECKS
/
S542A1  IFN     R2.364, EQ,1, S542A2    / CHECK IS1-1 IS Covered
        IFN     R2.364, EQ,1, S542A2    / CHECK IS1-2 IS Covered
        IFN     R2.350, EQ,0, S542A2    / CHECK P6 IS Idle
        IFN     R2.302, EQ,0, S542A2    / CHECK AV1-14 IS Closed
        IFN     R2.306, EQ,0, S542A2    / CHECK AV1-25 IS Closed
        GOTO    S542S1
/
S542A1  IFN     R2.364, EQ,0, S542A2    / CHECK IS1-1 IS Uncovered
        IFN     R2.364, EQ,1, S542A2    / CHECK IS1-2 IS Covered
        IFN     R2.350, EQ,0, S542A2    / CHECK P6 IS Idle
        IFN     R2.302, EQ,0, S542A2    / CHECK AV1-14 IS Closed
        IFN     R2.306, EQ,0, S542A2    / CHECK AV1-25 IS Closed
        GOTO    S542S4
/
S542A2  HESS    1.200, 1.1              / HESSAGE 'CONTROLLER NOT IN
                                        / CORRECT INITIAL STATE'
        ALSQ    2.542                   / ALARH SEQUENCE
        GOTO    S542A1
/
/ BEGIN OPERATION
/
S542S1  .... / INITIAL STATE 1
/
S542S4  .... / INITIAL STATE 4
```

(b) PARACODE

Figure 10.14: Automatically generated procedural controller pre-checks.

trollers. This enables faster troubleshooting and easy alteration of operating specifications or progression to code and documentation generation.

3. By adding extensive results handling capabilities that enable procedural controllers to be viewed quickly in text of graphical form. In this respect, two significant contributions are automatic documentation generation and automatic generation of simulation or industrial control code. Controllers and documentation in various forms are consistent with each other and provably correct.

Algorithm 10.4 (General Logic Generation)

1 Get Procedural Controller FSM
2 Append Controller File Header (e.g. Controller Name)
3 DO For all states on the FSM
  3.1 Append New Line Label and/or assign State Number to state variable
  3.2 IF state is MARKED THEN
    3.2.1 Append Code to Set Sequence Flag to Complete
    3.2.2 IF self-terminating THEN
      3.2.2.1 Append Termination code
    3.2.3 ENDIF
  3.3 ELSE
    3.3.1 DO For all transitions leaving the state
      3.3.1.1 IF transition is controllable THEN
        3.3.1.1.1 Append Controller Task Request Command
        3.3.1.1.2 Append grace period `WAIT` statement
        3.3.1.1.3 Append directional `GOTO` statement to destination state
      3.3.1.2 ELSE
        3.3.1.2.1 Get Destination State
        3.3.1.2.2 Append conditional statement:
          "IF state variable indicator = Destination state indicator value
          `GOTO` Destination State Line Label"
        3.3.1.2.3 Set contains_uncontrollables = true
      3.3.1.3 ENDIF
    3.3.2 ENDDO
    3.3.3 IF contains_uncontrollables = true THEN
      3.3.3.1 Append statement to loop in current state
    3.3.4 ENDIF
  3.4 ENDIF
4 ENDDO
5 Append File or Controller end comments
6 Output Controller File

```
IF (state = 1) THEN
  SEQUENCE
    Valve_Open(Valve IS Plant.CIP_Unit.AV1_25);
    state := 2;
  END # Sequence
END # IF
#
IF (state = 2) THEN
  SEQUENCE
    Valve_Open(Valve IS Plant.CIP_Unit.AV1_14);
    state := 3;
  END # Sequence
END # IF
#
IF (state = 3) THEN
  SEQUENCE
    Pump_Start(Pump IS Plant.CIP_Unit.P6);
    state := 4;
  END # Sequence
END # IF
#
IF (state = 4) THEN
  SEQUENCE
    IF (Plant.CIP_Unit.CIP.IS1_2.Indicator0 = 0) THEN
      # IS1-2 IS Uncovered
      SEQUENCE
        state := 13;
      END # SEQUENCE
    END # IF
    IF (Plant.CIP_Unit.CIP.IS1_1.Indicator0 = 1) THEN
      # IS1-1 IS Covered
      SEQUENCE
        state := 5;
      END # SEQUENCE
    END # IF
  END # Sequence
END # IF
IF (state = 8) THEN
  SEQUENCE
    Valve_Open(Valve IS Plant.CIP_Unit.AV1_25);
    state := 9;
  END # Sequence
END # IF
#
IF (state = 9) THEN
  SEQUENCE
    Valve_Open(Valve IS Plant.CIP_Unit.AV1_14);
    state := 10;
  END # Sequence
END # IF
#
IF (state = 10) THEN
  SEQUENCE
    Pump_Start(Pump IS Plant.CIP_Unit.P6);
    state := 11;
  END # Sequence
END # IF
#
IF (state = 11) THEN
  SEQUENCE
    IF (Plant.CIP_Unit.CIP.IS1_2.Indicator0 = 0) THEN
      # IS1-2 IS Uncovered
      SEQUENCE
        state := 16;
      END # SEQUENCE
    END # IF
    IF (Plant.CIP_Unit.CIP.IS1_1.Indicator0 = 0) THEN
      # IS1-1 IS Uncovered
      SEQUENCE
        state := 19;
      END # SEQUENCE
    END # IF
  END # Sequence
END # IF
IF (state = 32) THEN
  SEQUENCE
    MESSAGE "MARKED STATE 12 REACHED OF Heating"
    RESET Procedure.external_command := 3; END # RESET
    CONTINUE FOR 1
    RESET Procedure.external_command := 0; END # RESET
    state := 0;
  END # Sequence
END # IF
```

(a) *gPROMS*

```
S542S1   MOVN   1, R2.542              / SET STATE VARIABLE
         ENGE   AV1-25                 / Open Valve_AV1-25
S542SB1  WAIT   5                      / WAIT FOR GRACE TIME
/
S542S2   MOVN   2, R2.542              / SET STATE VARIABLE
         ENGE   AV1-14                 / Open Valve_AV1-14
S542SB2  WAIT   5                      / WAIT FOR GRACE TIME
/
S542S3   MOVN   3, R2.542              / SET STATE VARIABLE
         ENGE   P6                     / Start SISO-Pump_P6
S542SB3  WAIT   5                      / WAIT FOR GRACE TIME
/
S542S4   MOVN   4, R2.542              / SET STATE VARIABLE
S542T4   IF     R2.364, EQ, 0, S542S13 / Level-Switch_IS1-2 Uncover
         IF     R2.362, EQ, 1, S542S7  / Level-Switch_IS1-1 Cover
         GOTO   S542T4                 / LOOP IN STATE 4

S542S8   MOVN   8, R2.542              / SET STATE VARIABLE
         ENGE   AV1-25                 / Open Valve_AV1-25
S542SB8  WAIT   5                      / WAIT FOR GRACE TIME
/
S542S9   MOVN   9, R2.542              / SET STATE VARIABLE
         ENGE   AV1-14                 / Open Valve_AV1-14
S542SB9  WAIT   5                      / WAIT FOR GRACE TIME
/
S542S10  MOVN   10, R2.542             / SET STATE VARIABLE
         ENGE   P6                     / Start SISO-Pump_P6
S542SB10 WAIT   5                      / WAIT FOR GRACE TIME
/
S542S11  MOVN   4, R2.542              / SET STATE VARIABLE
S542T11  IF     R2.364, EQ, 0, S542S16 / Level-Switch_IS1-2 Uncover
         IF     R2.362, EQ, 0, S542S19 / Level-Switch_IS1-1 Uncover
         GOTO   S542T11                / LOOP IN STATE 11

S542S12  MOVN   12, R2.542             / SET STATE VARIABLE
         MOVN   2, R2.543              / SET SEQUENCE FLAG REGISTER
                                       / TO 2 FOR COMPLETE
/
S542S12A IF     R2.543, EQ, 0, S542A1  / CHECK FOR RESTART REQUEST
         GOTO   S542S12A               / LOOP IN STATE 12
```

(b) PARACODE

Figure 10.15: Automatically generated procedural controller logic.

Chapter 11

Case Studies

11.1 Introduction

This chapter presents four industrial scale case studies that demonstrate the techniques introduced in chapters 5, 6, 7 and 8, and the software discussed in chapters 9 and 10. The first case study deals with the synthesis of procedural controllers for a multipurpose, multiproduct batch pilot plant typical of a small scale food, fine chemicals or pharmaceuticals manufacturing plant. The plant provides an ideal opportunity to test the principles of Partitioning and Hierarchical Procedural Control Theories through its high complexity and connectivity. It is also fully automated, enabling comprehensive failure monitoring and the automatic generation of control code. In particular, simulation and actual implementation of the task of cleaning one of the process units is considered.

The second case study is performed on a minerals processing screw conveyor control system from literature (Probst *et al.*,1997). It demonstrates the use of Partitioning Theory and the synthesis of procedures for abnormal operation. The third and fourth case studies show that the software tools of chapters 9 and 10 enable PCT to be used and implemented by non-specialists. The third case study was a project conducted by an undergraduate student over a period of 3 months. Procedural controllers were synthesised for a milk pasteurisation plant. The full report is contained in Parra-Mora (1998). The fourth case study used the software packages as part of a 5 week Masters of Science project, as part of a Flexible Plant Operations ourse. In the project, 33 students synthesised procedural controllers for the normal operation of the batch pilot plant used in the first case study.

In the first, third and fourth case studies, the dynamic simulation environment *gPROMS* was used to test whether the operating specifications were met in a combined discrete/continuous environment. For a description of the simulation environment refer to Chapter 8.

Table 11.1 summarises the techniques and tools from previous chapters that are employed in each case study. A tick ($\surd$) indicates that the technique or tool is used in that case study.

Technique	Reference Section	I – Batch Pilot Plant	II – Screw Conveyor	III – PlantX	IV – MSc Course
Hierarchical Synthesis	5.4	√	√	√	
Hierarchical Control	5.4.2	√	√	√	
Reuse of Procedural Controllers	5.5	√	√	√	
Multiple Initial States	6.2	√	√	√	
Covered Initial States	6.5	√		√	
Partitioned Component Modelling	7.3	√	√	√	
Process Partitioning	7.1.2	√	√	√	
Continuous Failure Monitoring	7.4	√	√	√	
Structured Modelling Architecture	8.2	√		√	√
Simulation Verification	8.2.4	√		√	√
Synthesis of models $> 2^{10}$ states	9.5	√	√	√	√
Use of extended specifications	9.6	√	√	√	√
Guided user specification	10.2			√	√
FSM Translation to *gPROMS*	10.4	√	√	√	√
FSM Translation to PARACODE	10.4	√	√	√	√

Table 11.1: Summary of Techniques Demonstrated in the Case Studies

11.2 Case Study 1 - The Batch Pilot Plant

11.2.1 Overview of the Batch Pilot Plant

The Batch Pilot Plant in the Centre for Process Systems Engineering at Imperial College, is a multipurpose flexible manufacturing facility. It is representative of a small food processing, pharmaceutical or fine chemicals plant. Being a batch processing plant, the process has a large number of sequential operations, thus making it an ideal test bed for procedural control techniques.

The plant (a schematic is shown in Figure 11.1) centres around a 100 litre jacketed reactor. The reactor sits on a load cell and is equipped with a stirrer, a pH meter and a viscometer. It can be heated by direct injection of steam through a sparge. The jacket can use steam or cooling water to control the temperature of the reactor. Two feed preparation tanks and two product tanks (all of 100 litres capacity) with continuous level sensors and removable lids, complement the reactor. A Clean-In-Place tank of 100 litres with a reciprocating dosing pump, high and low level switches, concentration sensor and temperature sensor is also included. Three plate heat exchangers are also present. All these items are linked by an elaborate and automatically piping network including 49 single and double-seat valves and 8 pumps. Process fluid can be pumped between any of the tanks and heat exchangers. Tasks can be performed concurrently as long as there is no conflict between items required by the tasks. The operation of the plant is achieved through over 100 modular control sequences.

The plant is equipped with electronic instrumentation, including proximity switches

on tank lids and most valves having positional feedback sensors for open and closed. All these signals feed to an ACCOS 30 distributed control system (DCS).

11.2.1.1 Batch Pilot Plant Computer Control System

The Batch Pilot Plant is controlled by an APV ACCOS 30 DCS. Procedures are written in the proprietary sequential control language PARACODE, compiled and implemented in the ACCOS 30. The ACCOS 30 is housed in a control room which is representative of a modern industrial control facility. It communicates with the plant via digital and analogue I/O cards.

Linked to the ACCOS 30 are two PCs, which act as operator terminals for sequence initiation and general plant monitoring, a message logging printer and an engineering terminal for technical analysis and support. The PCs are also used to compile and download the PARACODE sequences to the ACCOS 30.

In addition, an ACCOS 300 UNIX machine implements supervisory control and scheduling functions, and provides a link between the ACCOS 30 and the department network. Using the proprietary software, CONTROLLINK and ACCLOG, process information can be recorded for later analysis.

For a detailed description of the electronic configuration of the plant refer to Liu (1995).

For the purpose of this case study, plant component and sequence failures are simulated by overriding digital outputs on the digital output cards on the ACCOS 30 and overriding inputs using the engineering terminal.

11.2.2 Clean-In-Place of Tank T1

11.2.2.1 Operating Procedure Overview

Cleaning of Tank T1 has been chosen as a suitable case study to test and demonstrate the theoretical developments of this thesis as it involves a large number of process items. It is one of the most complex procedures for this plant, and an excellent example of the complexity of industrial installations. It was also previously used to demonstrate the use of PCT (Alsop, 1996). By choosing the same procedure, the progress and benefits of the new developments from this work can be seen when the two implementations are compared.

Cleaning Tank T1 is termed as an *Unit Procedure* within the ISA-S88.01 standard and contains three operations:

- *Water-Rinse.* Intermittently flushing T1 with fresh water for 10 minutes to remove any debris or residue which may remain in T1 after processing. Water is discharged from T1 to the drain.
- *Detergent-Service.* Preparation of hot caustic cleaning solution in the CIP tank.

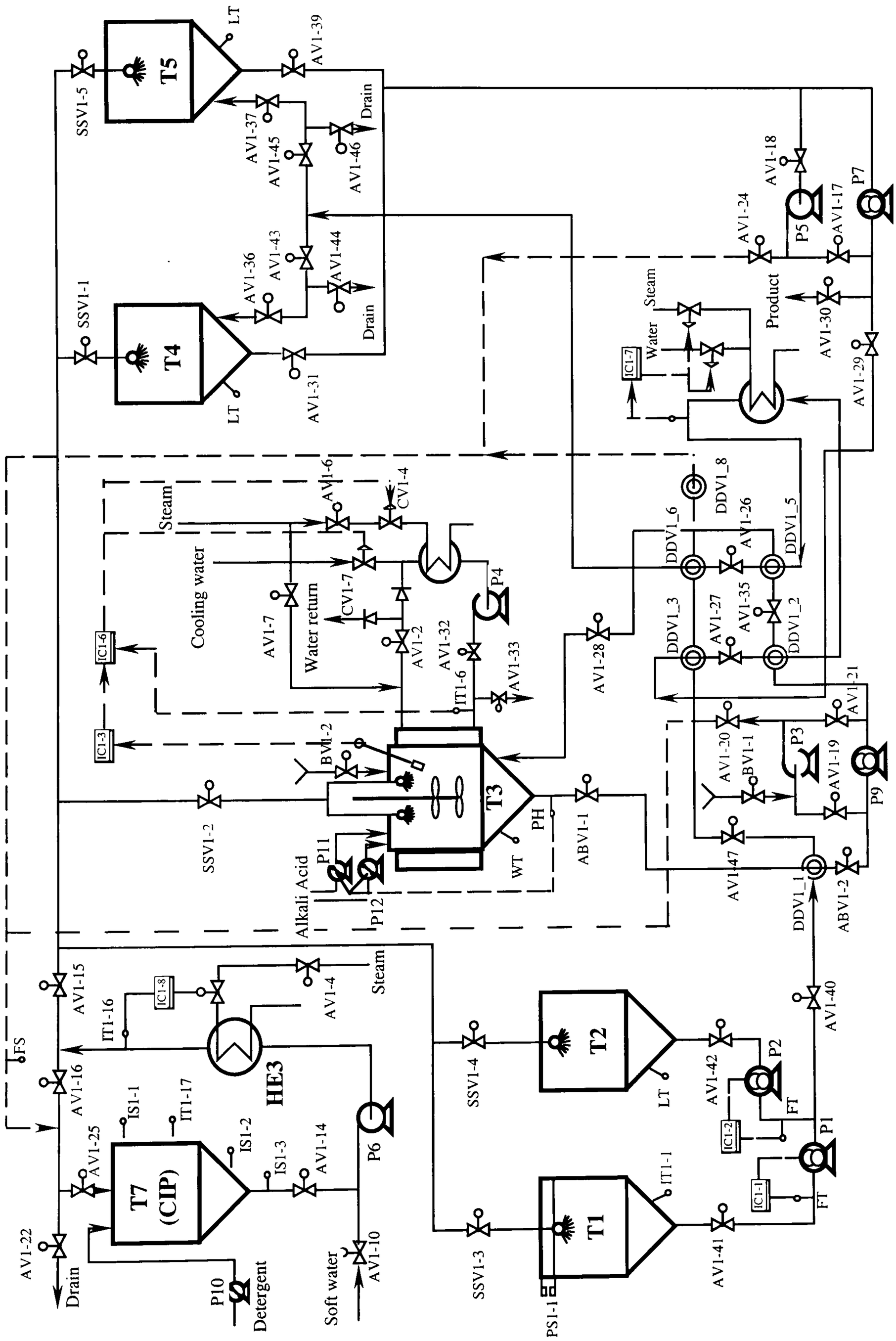


Figure 11.1: The Batch Pilot Plant

- *Detergent-Rinse.* Intermittently flushing T1 with hot caustic solution over a 10 minute period to sterilise the tank and associated equipment. The caustic solution is drained from T1 and returned to the CIP tank.

The recipe calls the three operations in order and then repeat the Water-Rinse procedure to clear any caustic residue from T1. More detailed descriptions of each operation and their subsequent phases are given in the sub-sections that follow.

11.2.2.2 Equipment Overview

Figure 11.2 shows a schematic of the CIP tank, tank T1 and associated equipment. The CIP tank is equipped with a high level switch (IS1-1), a low level switch (IS1-2) and a continuous temperature sensor (IT1-17). Note, that IS1-1 is de-energised when uncovered by the fluid in the tank and energised when covered, while IS1-2 is de-energised when covered and energised when uncovered. In the piping leaving the bottom of the CIP tank there is a conductivity switch (IS1-3) (energised when a specified conductivity threshold is satisfied). The heat exchanger HE3 has a PI Controller (IC1-8) that manipulates a continuous valve (CV1-2), adjusting the steam flow rate to control the temperature, which is measured by a continuous temperature sensor (IT1-16). A positive displacement pump (P10) doses the tank with concentrated caustic solution.

Tank T1 is equipped with a continuous level sensor (IT1-1) and a proximity switch (PS1-1) which monitors the position of the lid (energised when closed). A variable speed pump (P1), manipulated by a PI Controller (IC1-1) to control flow rate, drains T1. Pumps P10 and P6 are fixed speed pumps.

Other than CV1-2, all valves are digital solenoid valves. Valves DDV1-1, DDV1-3 and DDV1-6 are hygenic specification double-seat valves that, when deenergised, link the pipes above and below the valves in Figure 11.2. DDV1-8 is also a double seated valve but it links the pipes above and below it when energised. All other valves are single seated valves. They are all closed when deenergised except for AV1-16 and AV1-22 which are open when de-energised.

Several valves link the two units to other units in the plant. However, the units can be isolated from other units by interlocking the relevant valves during operation.

Table 11.2 lists all the component types and instances used in the Clean-T1 unit procedure. It also references the sub-figure (of Figure 11.3) that is the statechart representation of the each component. On the statecharts in Figure 11.3 controllable transitions are depicted by solid lines and uncontrollable transitions by broken lines.

11.2.3 Procedural Controller Synthesis

To clean Tank T1 required 26 procedural controllers assembled using Partitioning and Hierarchical Procedural Control Theory. In order to aid the reader in understanding the resulting hierarchy of controllers, it is presented now in Figure 11.4. The synthesise the

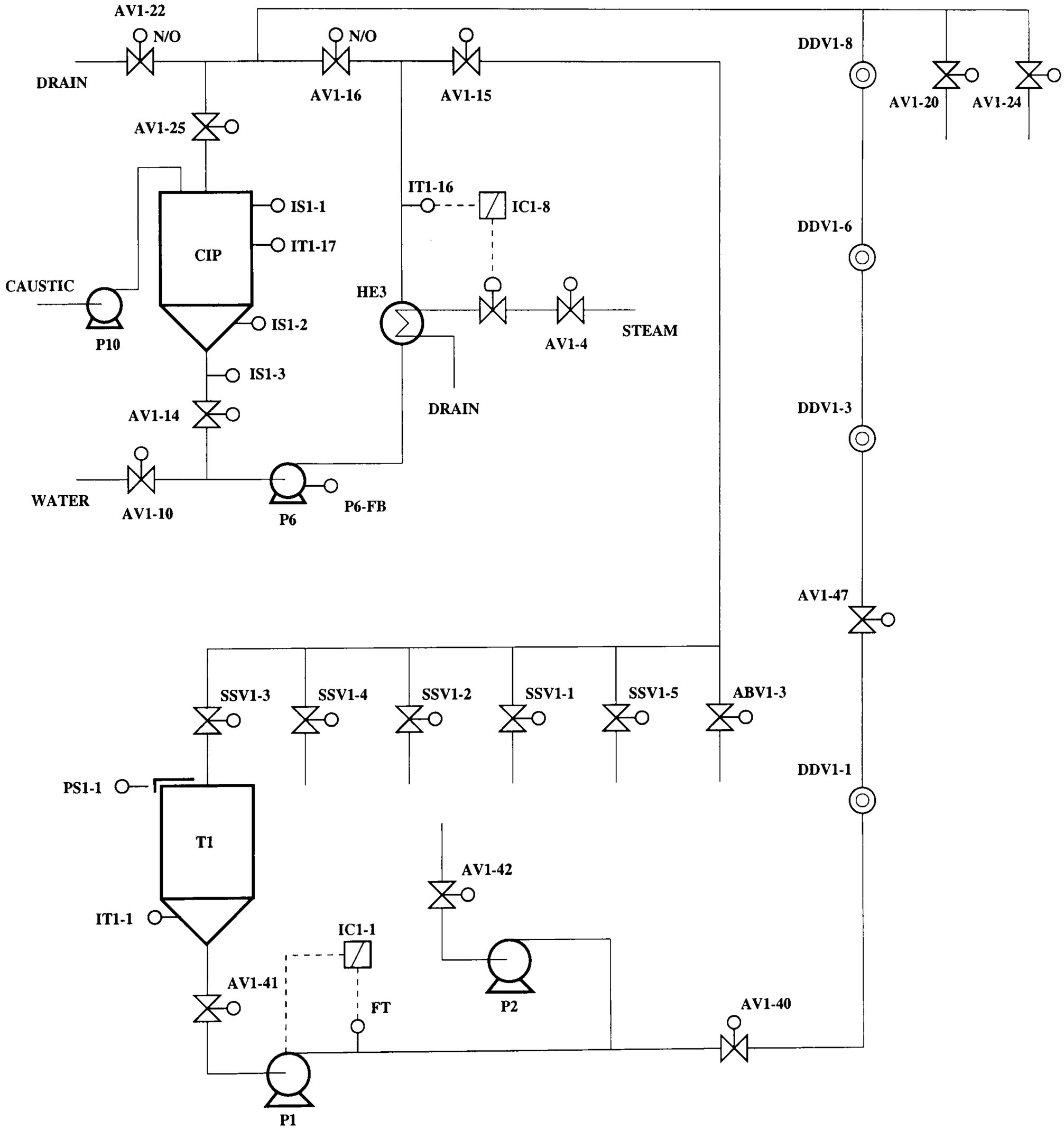
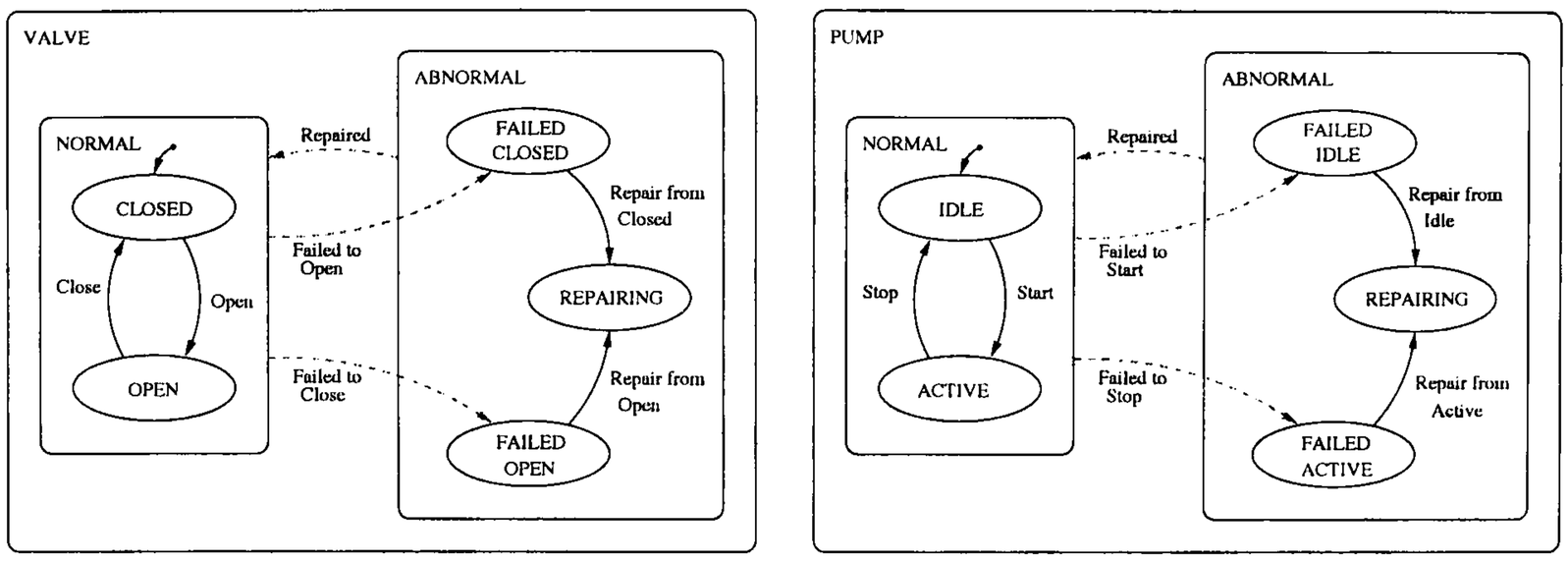


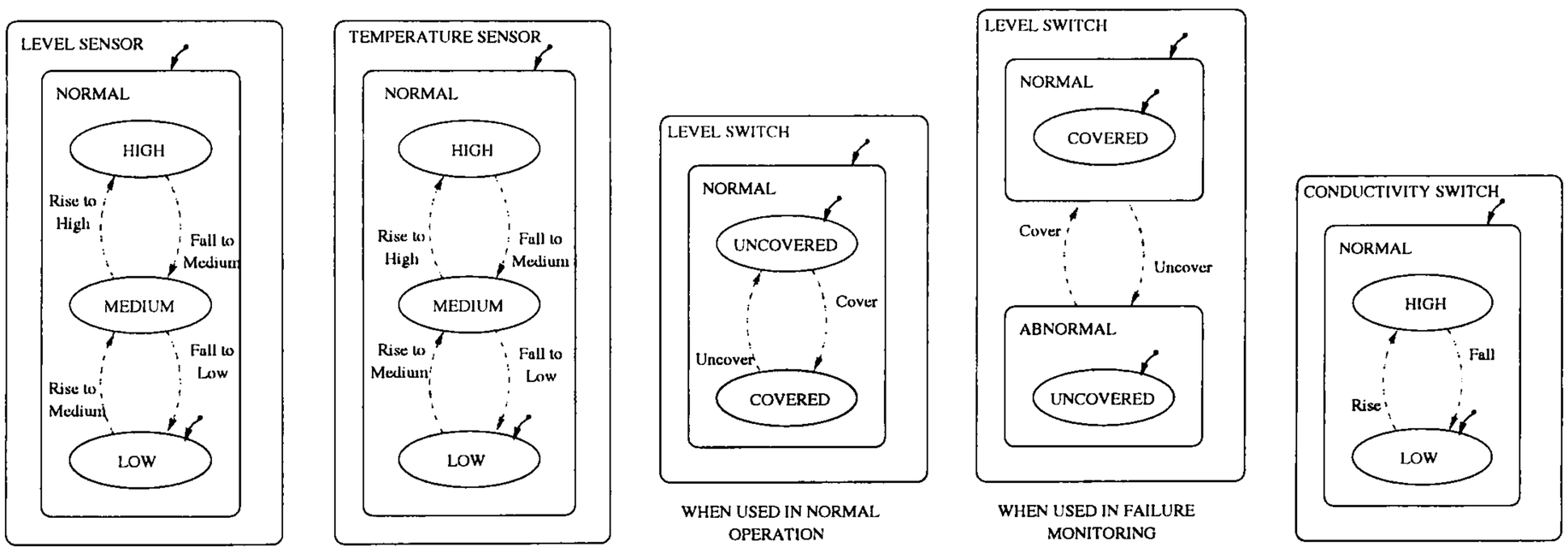
Figure 11.2: Schematic of CIP and Tank T1

Component Type	Corresponding StateChart	Instances
Valve	11.3(a)	AV1-22 AV1-4 AV1-10 AV1-14 AV1-15 AV1-16 AV1-20 AV1-24 AV1-25 AV1-40 AV1-41 AV1-42 AV1-47 ABV1-3 DDV1-1 DDV1-3 DDV1-6 DDV1-8 SSV1-1 SSV1-2 SSV1-3 SSV1-4 SSV1-5
Pump	11.3(b)	P1 P2 P6 P10
Level Sensor	11.3(c)	IT1-1
Temperature Sensor	11.3(d)	IT1-17
Level Switch	11.3(e)	IS1-1 IS1-2
Conductivity Switch	11.3(f)	IS1-3
Proximity Switch	11.3(g)	PS1-1
PI Controller	11.3(h)	IC1-1 IC1-8
Timer	11.3(i)	T10
Procedure	11.3(j)	Refer Table 11.3

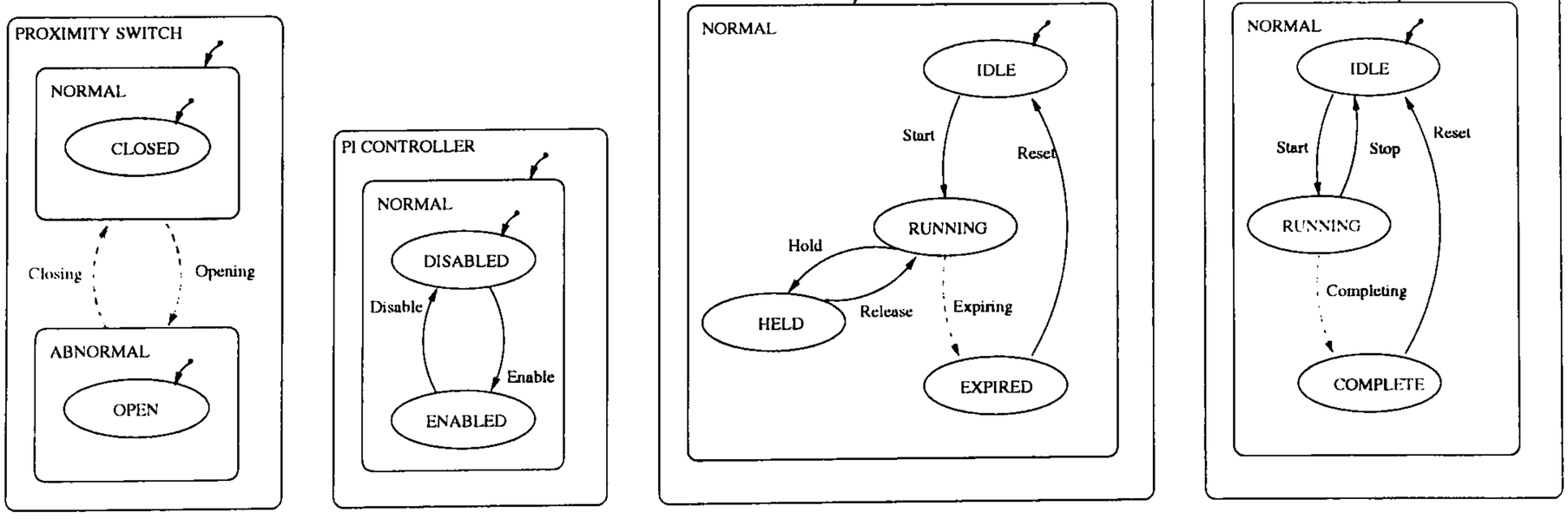
Table 11.2: Summary of Instances used in the Batch Pilot Plant Case Study.



(a) (b)



(c) (d) (e) (f)



(g) (h) (i) (j)

Figure 11.3: Elementary Components of the section of the Batch Pilot Plant studied.

hierarchical controller, first the operating specification was decomposed in a top-down fashion. Then to procedural controllers where synthesised from component models in a bottom-up fashion. For ease of understanding the overall synthesis procedure is described here in a top-down fashion.

11.2.3.1 Clean-T1

As the process consists of approximately 35 components, a single discrete model would consist of at least 10^{10} states which is unmanageable. Therefore, the Clean-T1 unit procedure must be decomposed. Using series decomposition theory and hierarchical theory, the unit procedure is decomposed (manually) into the three operations described in section 11.2.2.1. This decomposition satisfies the ISA-S88.01 standard, conforms to the series decomposition rules and allows the reuse of the *Water-Rinse* operation as described in hierarchical theory.

The language produced by the hierarchical implementation of the *Clean-T1* controller, C_{ClnT1}, and its operation controllers, C_{wr}, C_{ds} and C_{dr} is defined as follows.

$$\begin{aligned} L(C_{ClnT1}\triangle(C_{wr},C_{ds},C_{dr})/M) &= \mathcal{P}^{-1}_{C_{ClnT1}} L(C_{ClnT1}) \\ &\cap \mathcal{P}^{-1}_{wr}\{\Sigma^*_{wr}.\sigma_{s_{wr}}.L(C_{wr}).\sigma_{c_{wr}}.\Sigma^*_{wr}\} \\ &\cap \mathcal{P}^{-1}_{ds}\{\Sigma^*_{ds}.\sigma_{s_{ds}}.L(C_{ds}).\sigma_{c_{ds}}.\Sigma^*_{ds}\} \\ &\cap \mathcal{P}^{-1}_{dr}\{\Sigma^*_{dr}.\sigma_{s_{dr}}.L(C_{dr}).\sigma_{c_{dr}}.\Sigma^*_{dr}\} \\ &\cap L(\mathcal{S}^{wr}_{\sigma_{s_{wr}}}\mathcal{S}^{ds}_{\sigma_{s_{ds}}}\mathcal{S}^{dr}_{\sigma_{s_{dr}}}\mathcal{S}^{wr}_{\sigma_{c_{wr}}}\mathcal{S}^{ds}_{\sigma_{c_{ds}}}\mathcal{S}^{dr}_{\sigma_{c_{dr}}} M) \end{aligned}$$

Assuming that C_{wr}, C_{ds} and C_{dr} are nonblocking on M and conform to specification (as it was proven in subsequent sections and since C_{ClnT1} conforms to specification S_{ClnT1}), $C_{ClnT1}\triangle(C_{wr},C_{ds},C_{dr})$ conforms to specification. Considering theorem 5.2, the first condition is satisfied as a procedural controller was synthesised for C_{ClnT1}. The alphabets are disjoint, thus, the third condition is satisfied. Trivially, $\mathcal{P}^{-1}_{C_{ClnT1}} L(C_{ClnT1})$, $\mathcal{P}^{-1}_{wr}\{\Sigma^*_{wr}.\sigma_{s_{wr}}.L(C_{wr}).\sigma_{c_{wr}}.\Sigma^*_{wr}\}$, $\mathcal{P}^{-1}_{ds}\{\Sigma^*_{ds}.\sigma_{s_{ds}}.L(C_{ds}).\sigma_{c_{ds}}.\Sigma^*_{ds}\}$ and $\mathcal{P}^{-1}_{dr}\{\Sigma^*_{dr}.\sigma_{s_{dr}}.L(C_{dr}).\sigma_{c_{dr}}.\Sigma^*_{dr}\}$ are nonconflicting. By construction, each controller is nonconflicting with M. The presence of the inverse projection operator ensures no conflict between the controller terms and selflooped M as all transitions outside each controllers alphabets are permitted and each controller was synthesised by intersection with M.

11.2.3.1.1 Water Rinse Operation The Water Rinse Operation (*Water-Rinse*) flushes T1 with fresh water over a 10 minute period. This is done by filling the tank from the fresh water supply and then discharging it to the drain. The task involves 29 items which, when considering failures, create a potential model size of approximately 10^8 states. Once again the model is too large to be realistically dealt with by a single procedural controller.

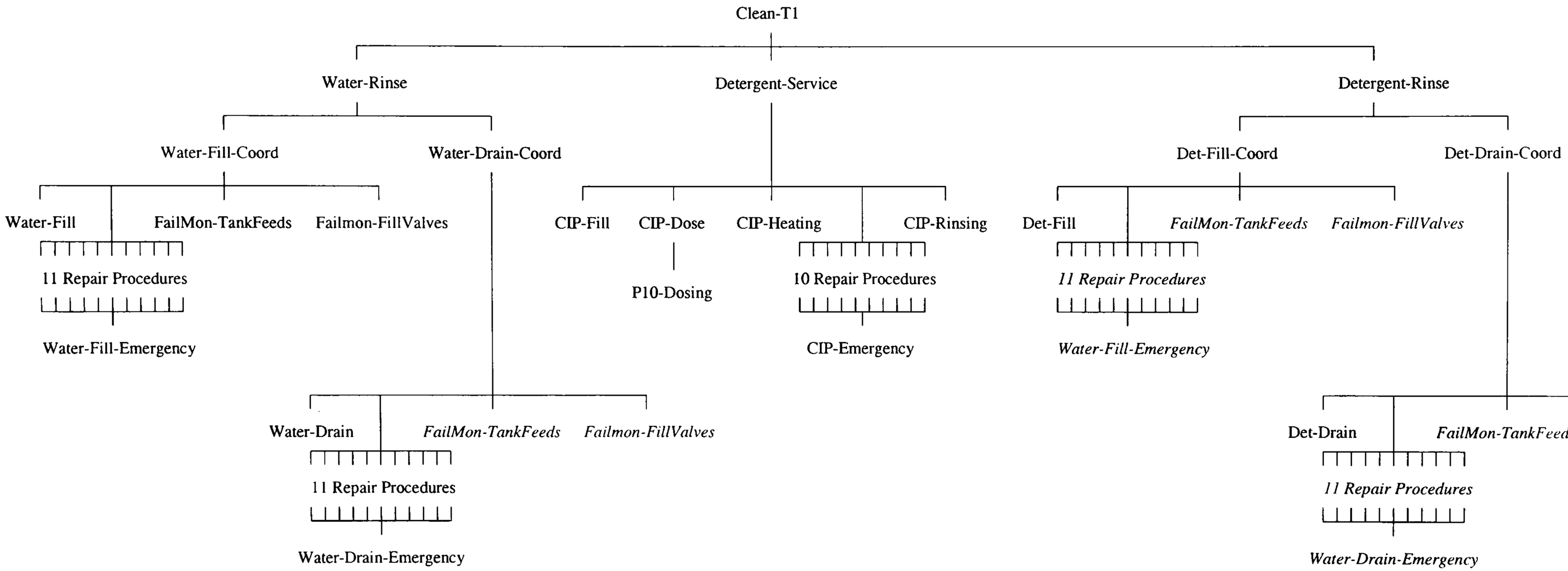


Figure 11.4: Clean-T1 Sequence Hierarchy

The task is decomposed into two parallel phases: *Water-Fill-Coord* and *Water-Drain-Coord*. These tasks and their subsequent phase controllers share only two items. They also satisfy the rules associated with concurrent cooperating controllers. Therefore, they are able to operate in parallel. The common components are:

- **IT1-1**, that contains only uncontrollable transitions, and
- **AV1-16**, that contains controllable transitions but is interlocked closed for the duration of the parallel operation of both controllers.

The *Water-Rinse* procedural controller coordinates the operation of lower controllers using hierarchical theory. It exploits the ability to use both procedural components and elementary components within the same procedural controller. *Water-Rinse* initiates and monitors the operation of the two phase controllers. It also includes a timer to measure the 10 minute rinsing time.

The language produced by the hierarchical implementation of the Water-Rinse controller, C_{wr}, and its operation controllers, C_{wfc} and C_{wdc} is defined as follows:

$$\begin{aligned} L(C_{wr}\triangle(C_{wfc},C_{wdc})/M) &= \mathcal{P}^{-1}_{C_{wr}}L(C_{wr}) \\ &\cap \mathcal{P}^{-1}_{wfc}\{\Sigma^*_{wfc}.\sigma_{s_{wfc}}.L(C_{wfc}).\sigma_{c_{wfc}}.\Sigma^*_{wfc}\} \\ &\cap \mathcal{P}^{-1}_{wdc}\{\Sigma^*_{wdc}.\sigma_{s_{wdc}}.L(C_{wdc}).\sigma_{c_{wdc}}.\Sigma^*_{wdc}\} \\ &\cap L(\mathcal{S}^{wfc}_{\sigma_{s_{wfc}}}\mathcal{S}^{wdc}_{\sigma_{s_{wdc}}}\mathcal{S}^{wfc}_{\sigma_{c_{wfc}}}\mathcal{S}^{wdc}_{\sigma_{c_{wdc}}}M). \end{aligned}$$

As in the case of the Clean-T1 procedural controller, since the subsequent controllers, C_{wfc} and C_{wdc}, conform to specification and are nonblocking (as shownin the following paragraphs). Theorem 5.2 is satisfied because

1. by provable correct synthesis, C_{wr} is nonblocking on H_{wr},
2. since condition 3 is satisfied and the inverse projection operator is present in all terms, the non-conflict requirements hold, and
3. the two shared items are uncontrollable or are interlocked.

Water-Fill-Coord. The *Water-Fill-Coord* task coordinates the phases associated with the filling of T1. In the previous study by Alsop (1996), no failures were considered. Sixteen elementary component instances are affected by the task. If all components are able to fail, the model contains more than 10^{11} states. This is computationally time consuming and becomes highly complex for a designer to tackle. Again, decomposition is required. However, component dependencies make previous methods unsuitable. Partitioning Theory provides a way to decompose the process into more manageable models. The specification is broken into its normal and abnormal operations. Thus, the *Water-Fill-Coord*

procedural controller initiates and monitors a normal operation procedural controller (*Water-Fill*) and multiple abnormal operation procedural controllers while monitoring the components for failures. Due to the large number of components, the monitoring task is broken into two more procedural controllers which signal the *Water-Fill-Coord* controller when a failure occurs. The need for these controllers is discussed in more detail in section 11.6.2.

The specification for the *Water-Fill-Coord* procedural controller is to:

1. starting with the all procedural controllers in their idle state and all component in a normal state,
2. start the two failure monitoring procedural controllers,
3. start the normal operation procedural controller (*Water-Fill*), and
4. wait until either the normal operation controller completes or a failure occurs.
5. If a failure is sensed then,
   (a) stop normal operation controller, start appropriate *Water-Fill-Emergency* controller, and
   (b) wait until *Water-Fill-Emergency* controller has completed and no components are in a failed state then,
   (c) goto step 2.
6. otherwise, stop the two failure monitoring procedural controllers and reset *Water-Fill.*

The language produced by the hierarchical implementation of the Water-Fill-Coord controller, C_{wfc}, and its phase controllers, C_{wf}, C_{wfe}, C_{fm1} and C_{fm2} is defined as follows.

$$
\begin{aligned}
L(C_{wfc}\triangle(C_{wf},C_{wfe},C_{fm1},C_{fm2})/M) \;=\; & \mathcal{P}^{-1}_{C_{wfc}}L(C_{wfc}) \\
\cap\; & \mathcal{P}^{-1}_{wf}\{\Sigma^{*}_{wf}.\sigma_{s_{wf}}.L(C_{wf}).\sigma_{c_{wf}}.\Sigma^{*}_{wf}\} \\
\cap\; & \mathcal{P}^{-1}_{wfe}\{\Sigma^{*}_{wfe}.\sigma_{s_{wfe}}.L(C_{wfe}).\sigma_{c_{wfe}}.\Sigma^{*}_{wfe}\} \\
\cap\; & \mathcal{P}^{-1}_{fm1}\{\Sigma^{*}_{fm1}.\sigma_{s_{fm1}}.L(C_{fm1}).\sigma_{c_{fm1}}.\Sigma^{*}_{fm1}\} \\
\cap\; & \mathcal{P}^{-1}_{fm2}\{\Sigma^{*}_{fm2}.\sigma_{s_{fm2}}.L(C_{fm2}).\sigma_{c_{fm2}}.\Sigma^{*}_{fm2}\} \\
\cap\; & L(\mathcal{S}^{wfc}_{\sigma_{s_{wfc}}}\mathcal{S}^{wfe}_{\sigma_{s_{wfe}}}\mathcal{S}^{fm1}_{\sigma_{s_{fm1}}}\mathcal{S}^{fm2}_{\sigma_{s_{fm2}}}\mathcal{S}^{wf}_{\sigma_{c_{wf}}}\mathcal{S}^{wfe}_{\sigma_{c_{wfe}}}\mathcal{S}^{fm1}_{\sigma_{c_{fm1}}}\mathcal{S}^{fm2}_{\sigma_{c_{fm2}}}M
\end{aligned}
$$

Water-fill-Coord satisfies theorem 5.2 because the *Water-Fill* and *Water-Fill-Emergency* procedural controllers operate in series, thus do not conflict. Moreover, the two failure monitoring procedural controllers contain only uncontrollable transitions common to other procedural controllers. Each procedural controllers was synthesised using sound methods and thus, satisfy definition 3.3, conform to specification and are nonblocking.

Water-Fill. The normal operation or *Water-Fill* procedure contains 13 components, 12 have two states and 1 has three states. Valves AV1-16, AV1-14, SSV1-1, SSV1-2, SSV1-4, SSV1-5 and AV1-25 are interlocked to closed for the whole procedure thus reducing the size of the active model region.
The operating specification is to:

1. starting with AV1-15, SSV1-3 and AV1-10 closed, P6 idle, the flag unset, and IT1-1 Low,
2. open SSV1-3, AV1-15,
3. open AV1-10 and start pump P6 in order, and
4. wait until level sensor IT1-1 reads 20% full or the flag sensor FS1 is set.
5. If the flag is not set then,
   (a) stop P6 and close AV1-10.
   (b) Wait until level sensor IT1-1 reads below 6% full. Then,
   (c) if FS1 is set, go to step 6, otherwise,
   (d) go to step 2.
6. otherwise, stop P6 and close AV1-10, then,
7. close AV1-15 and SSV1-3.

A procedural controller for normal operation with XX states was synthesised using traditional PCT synthesis methods.

Water Fill Abnormal Phase. If a component instance fails, the emergency procedures are called. The general emergency operating procedure for the *Water-Fill-Coord* procedure is to alert the operator of the failed instance and then return all other instances to their deenergised state. Alarming each instance must be done by individual procedural controllers. However, returning all working instances to their deenergised state is common to all emergency procedures and thus, it can be used in all the procedures as shown in Figure 11.4. Each component emergency procedure consists of two components: the abnormal FSM of the failed instance and a procedural component representing the *Water-Fill-Emergency* procedural controller. Each controller's specification is to alert the operator of the failed instance and to call the *Water-Fill-Emergency* procedure. For the *Water-Fill-Coord*, 13 procedural controllers of 5 states are synthesised.

The simplicity of the emergency specification eliminates the need for considering procedures when a combination of instances fail. If additional failures occur during the emergency procedure's operation, it is started again and the operator is informed of the newly failed instance. The emergency procedures do not terminate prematurely if the system returns to normal operation while it is executing.

Water-Fill-Emergency. The *Water-Fill-Emergency* procedure checks each of the 11 components with controllable transitions and ensures that they are returned to their deenergised state. The implementation falls into the theory of section 6.5 to produce a concise controller of 33 states. The initial state has all variables as covered variables. In other words, all combinations of state variable values are valid initial states. This is because independent failures are being considered here, which can occur at any state of the process. Therefore the emergency procedure must be able to start from any state of the process.

The controller then steps through each state variable using uncontrollable assessment transitions (implemented by conditional statements) to establish the state of each instance. If needed, a controllable transition is executed, it forces the instance to a safe state. Otherwise, the controller moves to the next uncontrollable assessment transition.

This procedural controller is the first formal implementation of the more common approach to design emergency procedures where each step contains a conditional test with resultant actions. The resultant procedural controller is equivalent to a procedural controller with $2^{11} = 2048$ states all of which can be initial states. It is clear that the 33 state procedural controller is much more manageable.

Failure Monitoring Controllers : FailMon-FM_TF and FailMon-FM_FV. Eleven components in the *Water-Fill-Coord* phase contained failure transitions. To include failure monitoring in the main *Water-Fill-Coord* procedure would have increased its size by a factor of 2^{11}. Instead, two procedural controllers were used to monitor the failure status of the components. These controllers contained the superstate models of the components which consisted of only uncontrollable transitions. All combinations of state variable values were possible, thus resulting in large controllers. The 11 components were monitored in two procedural controllers (each with 7 components) for easier management. Three additional components (2 in FailMon-FM_TF and 1 in FailMon-FM_FV) were used for the communication between controllers and the *Water-Fill-Coord* controller. As a result, two controllers with 71 states each were synthesised using PCT methods.

Water-Drain-Coord. The *Water-Drain-Coord* procedure is analogous to the *Water-Fill-Coord* procedure but coordinates the phases associated with draining T1. The procedure affects 18 elementary component instances. Like *Water-Fill-Coord*, when failures are included in the *Water-Drain-Coord* procedure, the model becomes too large to be synthesised as a single procedural controller. Partitioning is used once again to decompose the problem into manageable problems. The operating specification is broken into normal (*Water-Drain*) and abnormal operations. The *Water-Drain-Coord* procedure coordinates the operation of these two sub-phases and two failure monitoring procedural controllers.

The operating specification is to:

1. starting with *Water-Drain*, the Water Drain Emergency Procedures and the Water Drain Failure monitors all idle.

2. start the two failure monitoring procedures,
3. call *Water-Drain*, and
4. wait until *Water-Drain* completes or a failure occurs.
5. If a failure occurs,
   (a) stop *Water-Drain* and start the appropriate *Water-Drain-Emergency* procedure, and
   (b) wait until no components are in a failure state and *Water-Drain-Emergency* has completed, then
   (c) go to step 2.
6. otherwise, stop the two failure monitoring procedures and reset *Water-Drain.*

As will be shown in the following paragraphs, *Water-Drain* and the abnormal operation procedures conform to specification and are nonblocking. Therefore, for the same reasons that the *Water-Fill-Coord* procedure conforms to specification and is nonblocking, *Water-Drain-Coord* does too. The language produced by the hierarchical implementation of the Water-Drain-Coord controller, C_{wdc}, and its phase controllers, C_{wd}, C_{wde}, C_{fm3} and C_{fm4} is defined as follows:

$$
\begin{aligned}
L(C_{wdc}\triangle(C_{wd},C_{wde},C_{fm3},C_{fm4})/M) \;=\; & \mathcal{P}^{-1}_{C_{wdc}}L(C_{wdc}) \\
\cap\; & \mathcal{P}^{-1}_{wd}\{\Sigma^*_{wd}.\sigma_{s_{wd}}.L(C_{wd}).\sigma_{c_{wd}}.\Sigma^*_{wd}\} \\
\cap\; & \mathcal{P}^{-1}_{wde}\{\Sigma^*_{wde}.\sigma_{s_{wde}}.L(C_{wde}).\sigma_{c_{wde}}.\Sigma^*_{wde}\} \\
\cap\; & \mathcal{P}^{-1}_{fm3}\{\Sigma^*_{fm3}.\sigma_{s_{fm3}}.L(C_{fm3}).\sigma_{c_{fm3}}.\Sigma^*_{fm3}\} \\
\cap\; & \mathcal{P}^{-1}_{fm4}\{\Sigma^*_{fm4}.\sigma_{s_{fm4}}.L(C_{fm4}).\sigma_{c_{fm4}}.\Sigma^*_{fm4}\} \\
\cap\; & L(\mathcal{S}^{wdc}_{\sigma_{s_{wdc}}}\mathcal{S}^{wde}_{\sigma_{s_{wde}}}\mathcal{S}^{fm3}_{\sigma_{s_{fm3}}}\mathcal{S}^{fm4}_{\sigma_{s_{fm4}}}\mathcal{S}^{wd}_{\sigma_{c_{wd}}}\mathcal{S}^{wde}_{\sigma_{c_{wde}}}\mathcal{S}^{fm3}_{\sigma_{c_{fm3}}}\mathcal{S}^{fm4}_{\sigma_{c_{fm4}}}M)
\end{aligned}
$$

Water-Drain. *Water-Drain* controls the normal operation of the drainage of T1. It contains 15 elementary components (14 two-state and 1 three-state) yielding a potential model size of 50000 states. However, 7 instances are interlocked for the duration of operation reducing the active model size to approximately 192 states.

The operating specification is:

1. starting with the level of T1 at any value,
2. open AV1-40, AV1-47 and DDV1-8.
3. Wait until the level rises above 20 % or flag sensor FS1 is set.
4. If FS1 is not set, then
   (a) open AV1-41, start P1 and enable controller IC1-1.
   (b) Wait until the level falls below 6 %, then

(c) disable controller IC1-1, stop P1 and close AV1-41, and

(d) go to step 3.

5. otherwise, close DDV1-8, AV1-47 and AV1-10.

From these models and constraints, a procedural controllers with 36 states was synthesised that is controllable, nonblocking and conforms to specification.

Water Drain Abnormal Phase. The Water Drain abnormal operation procedural controllers are very similar to those for Water Fill abnormal operation. The operating specification was to alert the operator of the failed component instance and then return all other instances to their deenergised states. Thirteen component instances are monitored and the appropriate emergency procedure called when a failure occurs. Again, covered initial states are used to synthesise a single procedural controller to perform the shutdown of the process once a failure has occurred.

11.2.3.1.2 Detergent Rinse Operation. The Detergent Rinse Operation (*Det-Rinse*) flushes T1 with caustic solution from the CIP tank over a ten minute interval. It is very similar to the Water Rinse Operation and affects the same 28 elementary component instances plus the two level switches in the CIP tank. When failures are considered the potential model size is unmanageably large. Instead, the model is decomposed into two parallel procedures *Det-Fill-Coord* and *Det-Drain-Coord.* For the same reasons as in the Water Rinse Unit Operation, these two phases can operate in parallel (see section 11.2.3.1.1).

As table 11.3 shows, only *Det-Fill-Coord, Det-Fill, Det-Drain-Coord* and *Det-Drain* are synthesised for the Detergent Rinse Unit Operation. This is because the failure monitoring and emergency procedures for the Water Rinse Unit Operation are identical to those for the Detergent Rinse unit Operation and thus can be re-used. This reduces the overall controller size and design effort considerably.

Operating specification differences in *Det-Fill* and *Det-Drain* are only that for *Det-Fill* AV1-10 remains closed and AV1-14 is opened so that caustic solution could be taken from the CIP tank, while for *Det-Drain*, AV1-22 was closed and AV1-25 opened so that used caustic solution could be returned to the CIP tank.

Traditional PCT synthesis was then performed producing four new procedural controllers for the Detergent Rinse Unit Operation. Their sizes and other statistics are shown in Table 11.3.

11.2.3.1.3 Detergent Service Unit Operation. The Detergent Service unit operation supervises the preparation of hot caustic solution in the CIP tank. In total, 16 process items are effected by this unit operation. This yields a potential model of size 2^{16} if all items are considered to only have two states. However, when failures are included, the model size becomes quite unmanageable. The unit operation can be described as a

series of four phases: filling the tank, charging with detergent, heating and rinsing the surrounding pipework. It was decomposed using series decomposition techniques and hierarchical theory. As a result, the unit operation comprised four procedural components representing the four phase operations plus one for an emergency shutdown procedure and two procedural components controlling the operation of two failure monitoring controllers. An elementary component failure sensor was included for communication with the failure monitors.

The operating specification is to:

1. start with all *CIP-Fill, CIP-Dose, CIP-Heat, CIP-Rinse*, the CIP emergency procedures and the CIP failure monitoring procedures all idle, then
2. start the two failure monitoring procedures, then
3. start *CIP-Fill*, and
4. wait until *CIP-Fill* completes or a failure occurs.
5. If a failure occurs, then
   (a) stop *CIP-Fill* and then start the appropriate *CIP-Emergency* procedure, then
   (b) wait until the *CIP-Emergency* procedure has completed and no instances are in a failure state, then
   (c) go to step 2.
6. otherwise, start *CIP-Dose*, and
7. wait until *CIP-Dose* completes or a failure occurs.
8. If a failure occurs, then
   (a) stop *CIP-Dose* and then start the appropriate *CIP-Emergency* procedure, then
   (b) wait until the *CIP-Emergency* procedure has completed and no instances are in a failure state, then
   (c) go to step 5.
9. otherwise, start *CIP-Heat*, and
10. wait until *CIP-Heat* completes or a failure occurs.
11. If a failure occurs, then
    (a) stop *CIP-Heat* and then start the appropriate *CIP-Emergency* procedure, then
    (b) wait until the *CIP-Emergency* procedure has completed and no instances are in a failure state, then
    (c) go to step 8.
12. otherwise, start *CIP-Rinse*, and
13. wait until *CIP-Rinse* completes or a failure occurs.
14. If a failure occurs, then

(a) stop *CIP-Rinse* and then start the appropriate *CIP-Emergency* procedure, then

(b) wait until the *CIP-Emergency* procedure has completed and no instances are in a failure state, then

(c) go to step 11.

As all phases are called in series and no components (and thus transitions) are shared between controllers, Theorem 5.2 is satisfied and the controller is guaranteed to meet specification and be nonblocking (assuming the lower controllers conform and are nonblocking). The five lower controllers contain only elementary components and are synthesised using traditional PCT methods, thus they conform to specification and are nonblocking. Table 11.3 presents the statistics for each controllers synthesis. The operating specification for each controller is presented in the following.

CIP-Fill The *CIP-Fill* phase fills the CIP tank with fresh water until the high level switch (IS1-1) is covered. The phase affects 12 component instances, but 6 (P10, AV1-14, AV1-16, AV1-15, DDV1-8, AV1-24) are interlocked to their deenergised positions for the duration of the phase, thus reducing the active model region to $2^6 = 64$ states.
The operating procedure is to:

1. start with the level of the CIP tank at any value, then
2. close AV1-22, open AV1-25, open and AV1-10, and start P6, in order, then
3. wait until the high level switch is covered (energised). Then
4. stop P6, close AV1-10, close AV1-25, and open AV1-22.

CIP-Dose Dosing the CIP tank with caustic solution is executed by *CIP-Dose*. The phase affects directly 12 component instances of which 4 are interlocked during operation. The operating specification is to:

1. start with the CIP tank full, and
2. open AV1-14, open AV1-25, close AV1-22 and start P6. Then,
3. start *P10-Dosing* which runs P10 intermittently (10 seconds on, 30 seconds off) to dose the CIP tank. (The intermittent running of P10 is to avoid overdosing due to slow mixing dynamics of the CIP tank.)
4. Wait until concentration switch (IS1-3) is energised, then
5. stop *P10-Dosing*, stop P6, open AV1-22, close AV1-14 and close AV1-25.

CIP-Heat After dosing, the contents of the CIP tank are heated to 60 degrees Celsius. *CIP-Heat* does so by recirculating water through HE1 via P6. 12 component instances are directly affected and five of those are interlocked for the procedure. The operating specification is to:

1. start with the CIP tank full, and
2. open AV1-14, open AV1-25, close AV1-22 and start P6. Then,
3. enable controller IC1-8 with a set point of 80 C (the temperature of process fluid leaving the heat exchanger), and
4. wait until temperature observer for IT1-16 registers high ($> 60C$). Then,
5. disable IC1-8, stop P6, open AV1-22, close AV1-14 and close AV1-25.

CIP-Rinse The piping around the CIP tank is flushed once the caustic solution is prepared. Fresh water is pumped by P6 through HE1 and AV1-22 to the drain for 10 minutes. Again 12 instances are affected plus a timer is used. The timer is explicitly modelled as an elementary component with *Idle, running* and *Expired* states. Seven instances are interlocked throughout the procedure, thus the active model size is $2^6 = 64$ states. The operating specification is to:

1. start with all actuated items deenergised, then
2. open AV1-10, start P6 and start a 10 minute timer, and
3. wait until the timer expires. Then,
4. stop P6, close AV1-10 and reset the timer.

CIP Abnormal Operating Procedures Like the previous abnormal operation procedures described, the CIP abnormal operating specifications are to alert the operator of the failed instance and then drive the remaining instances to their safe (deenergised) state. Twelve component instances are monitored and thus twelve 5 state procedural controllers were synthesised plus a 25 state procedural controller that is called by the other twelve.

While synthesising the controllers for the CIP of T1 for the Batch Pilot Plant, the properties of partitioning, covered initial states and procedural controller re-use were all exploited.

Controller	Process Model Construction					Controller Synthesis		
	No. of Instances	No. of Initial States	No. of Goal States	No. of Impossible Transitions	Potential No. of States	No. of Forbidden States	No. of Forbidden Transitions	No. of States
Clean-T1	3	1	1	0	27	3	7	10
Water-Rinse	8	1	1	2	1152	0	22	26
Water-Fill-Coord	6	1	1	1	324	1	14	21
Water-Fill	13	3	1	1	12288	7	15	31
Water-Fill-Emergency	11	1	1	0	2048	0	12	23
FailMon-TankFeeds	7	1	1	0	128	0	9	71
FailMon-FillValves	7	1	1	0	128	0	9	71
Water-Drain-Coord	6	1	1	1	324	1	14	21
Water-Drain	15	3	1	1	49152	7	16	36
Water-Drain-Emergency	13	1	1	0	8192	0	14	27
FailMon-DSValves	7	1	1	0	128	0	9	71
FailMon-DrainValves	7	1	1	0	128	0	9	71
Det-Service	8	1	3	1	4374	10	19	42
CIP-Fill	12	2	1	2	4096	6	5	15
CIP-Dose	12	1	2	2	6144	4	13	38
P10-Dosing	4	1	1	0	36	1	9	14
CIP-Heat	12	1	2	3	4096	5	7	18
CIP-Rinse	10	1	1	0	1536	7	5	7
CIP-Emergency	10	1	1	0	1024	0	11	21
FailMon-CIP1	6	2	1	0	64	0	9	42
FailMon-CIP2	7	1	1	0	128	0	10	76
Det-Rinse	8	1	1	5	1152	0	20	31
Det-Fill-Coord	6	1	1	1	324	1	13	23
Det-Fill	13	3	1	1	12288	7	15	24
Det-Drain-Coord	6	1	1	1	324	1	13	23
Det-Drain	15	3	1	1	49152	5	15	37

Table 11.3: Models and Controller Synthesis Statistics for Cleaning T1.

11.2.4 Translation to *gPROMS* and PARACODE

Using the translation facilities of Sequence Developer, all the procedural controllers were first translated into *gPROMS* simulation code as `TASKS`. They were then inserted into a model of the uncontrolled plant that has been tested and verified that it is a correct depiction of the real process. After testing that the procedural controllers operate the simulated process in a way that conforms to the operating specifications, the controllers were translated into PARACODE for compilation and implementation on the Batch Pilot Plant. As described in Chapter 10, the appropriate translation commands are entered through a window of Sequence Developer that is initiated from the main interface. This information about the mapping of component, instance and transition names is contained within the Sequence Developer input files. Appendix A has a listing of the such an input file. The corresponding *gPROMS* and PARACODE commands for each state and transition can be seen in the component definition section. With the development of a single project file containing all relevant controllers, the definition of components and their corresponding code was only done once.

Some staistics for both translations are shown in Table 11.4. Figures refer to the number of lines of code generated in the respective languages. An example of each translation is shown in Appendix B. For the case of *gPROMS* translation, the files produced were not directly executable. However, they included a header which listed instructions for where to insert the code in the relevant *gPROMS* simulation model file. PARACODE files where directly downloaded into the industrial control system.

11.2.5 Implementation in gPROMS

All *gPROMS* files generated from the procedural controllers were inserted into the batch pilot plant *gPROMS* model and a simulation was performed. Unlike in previous work (Sanchez et al., 1996) on this case study, no manually written tasks were required. The simulations results are discussed in the sections below. Figure 11.8 then follows showing profiles of tank levels and valve positions for a complete simulation of the *Clean-T1* Operation. In each figure the variable names on the y-axis correspond to the components on the schematic diagram shown in in Figure 11.2.

11.2.5.1 Water-Rinse

As Figure 11.5 shows, the *Water-Rinse* procedure starts execution at time zero. The figure shows a delay before there is a change in the level of tank T1 (IT1-1). This is due to the following procedures starting: *Water–Fill–Coord* and *Water–Drain–Coord*, then *FailMon_FM-TF*, *FailMon_FM-FV*, *FailMon_FM-DV* and *FailMon_FM-DR*, before *Water-Fill* and *Water-Drain*. The operation continues until the observer for IT1-1 senses it becoming high (reaching 20 % of tank capacity). Then, *Water-Fill* stops P6 and closes AV1-10 and concurrently, *Water-Drain* starts P1 and drains T1. When IT1-1's level

Controller	No. of Instances	Lines of Code Generated						
		gPROMS Tasks			PARACODE Sequences			
		Pre-Checks	Logic	Total	Pre-Checks	Logic	Comments	Total
Clean-T1	3	14	119	133	15	61	41	117
Water-Rinse	8	14	385	399	25	148	55	228
Water-Fill-Coord	6	14	303	317	21	141	48	210
Water-Fill	13	20	337	357	91	157	60	308
Water-Fill-Emergency	11	14	315	329	31	107	51	189
FailMon-TankFeeds	7	14	3020	3034	23	615	101	739
FailMon-FillValves	7	14	3020	3034	23	615	101	739
Water-Drain-Coord	6	14	303	317	21	141	48	210
Water-Drain	15	20	384	404	103	183	65	291
Water-Drain-Emergency	13	14	365	379	35	125	55	215
FailMon-DSValves	7	14	3020	3034	23	615	101	739
FailMon-DrainValves	7	14	3020	3034	23	615	101	739
Det-Service	8	14	479	493	25	254	67	346
CIP-Fill	12	17	146	163	85	76	45	206
CIP-Dose	12	14	420	434	33	198	68	299
P10-Dosing	4	14	175	189	17	73	42	132
CIP-Heat	12	14	176	190	33	92	43	168
CIP-Rinse	10	14	86	100	29	37	32	98
CIP-Emergency	10	14	304	361	29	98	46	173
FailMon-CIP1	6	14	1896	1910	35	322	67	444
FailMon-CIP2	7	17	3250	3267	23	660	101	784
Det-Rinse	8	14	462	476	25	166	60	251
Det-Fill-Coord	6	14	350	364	21	151	50	222
Det-Fill	13	20	328	348	91	141	57	289
Det-Drain-Coord	6	14	350	364	21	151	50	222
Det-Drain	15	20	402	422	103	188	66	357
Total		394	23401	23795	1004	6130	1626	8755

Table 11.4: Generated Code Statistics for Cleaning T1.

observer changes to Low (below 6 % of tank capacity), *Water-Drain* stops P1 and closes AV1-41, while *Water-Fill* starts P6 and opens AV1-10. The process is repeated for the 10 minute duration.

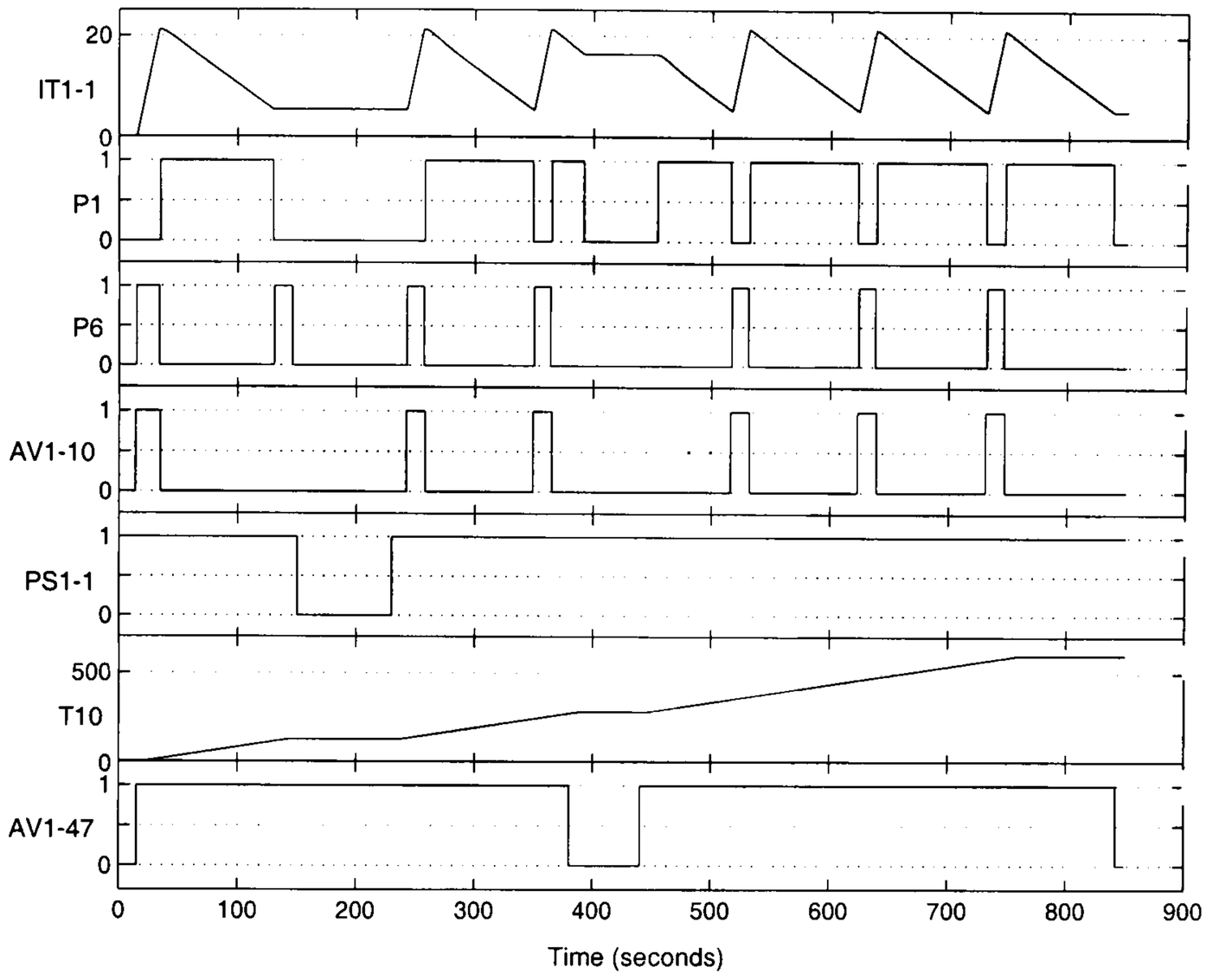


Figure 11.5: Level profile and valve positions for *Water-Rinse* procedure simulation in *gPROMS*.

The slight level overshoots at both the high and low levels are due to delays associated with the scanning procedure. The *gPROMS* simulation model has been set to scan the observers and execute a single action every 1 second. This is slower than a standard control system which scans every 0.25 seconds but was chosen as the most suitable scanning time for optimum simulation accuracy and solving time.

At time t= 130 seconds, *Water-Fill* tries to open AV1-10 but it sticks and fails closed. *Water-Fill-Coord* then stops *Water-Fill* and starts *Water-Fill-Emergency* which shuts all valves involved in filling T1. While *Water-Fill-Emergency* is executing, the lid of T1 is opened also (monitored by PS1-1). So when AV1-10 is fixed (after 60 seconds), the *Water-Fill* procedure cannot be restarted. Once the lid has been closed and the proximity switch observer senses it is closed, the *Water-Fill* procedure restarts, at t = 250 seconds. While *Water-Fill* was not operating, Timer-T10 was held by *Water-Rinse*.

The *Water-Rinse* procedure continues until t = 380 seconds when AV1-47 is closed manually. The valve observer senses this and infers that the valve has failed. *Water-Drain* is stopped and *Water-Drain-Emergency* is started. P1 is stopped and all valves closed until AV1-47 is repaired at t = 440 seconds, and *Water-Drain* is restarted. Again Timer-T10 is held while *Water-Drain* is not active. The procedure continues until the timer expires and the tank is drained.

11.2.5.2 Detergent Service Unit Operation

Figure 11.6 shows the results of the simulation in *gPROMS* for the Detergent Service Unit Operation. It can be seen that from when all CIP controllers are idle, *CIP-Fill* is called and the tank filled to 95 litres, before P6 is stopped and the phase is completed at time 90 seconds. *CIP-Dose* is then called immediately and the concentration is increased. Note that intermittent dosing is performed, allowing for the slow mixing dynamics in the real plant (not modelled by the simulation). On completion of *CIP-Dose* however, at time 370 seconds, valve AV1-25 sticks open and thus it is assumed to have failed at t= 340 seconds. *CIP-Dose* is stopped and *CIP-Shutdown* called. After 60 seconds AV1-25 is repaired and returned to service in the closed position. *CIP-Heat* is then called and the solution is recirculated through HE1 while it is heated to 60 C. However, at t = 450 seconds, P6 fails and stops unexpectedly. This is monitored by the *CIP-FM1* and *CIP-FM2* failure monitoring controllers. They alert the *Det-Service* Unit Operation procedural controller, *CIP-Heat* is stopped and *CIP-Shutdown* called again. After 120 seconds, the pump is repaired and the normal operation is resumed when *CIP-Heat* is called. The operation continues until the tank temperature reaches 60 C when *CIP-Heat* completes its operation at time 975 seconds. *CIP-Rinse* is then called to flush pipework with fresh water.

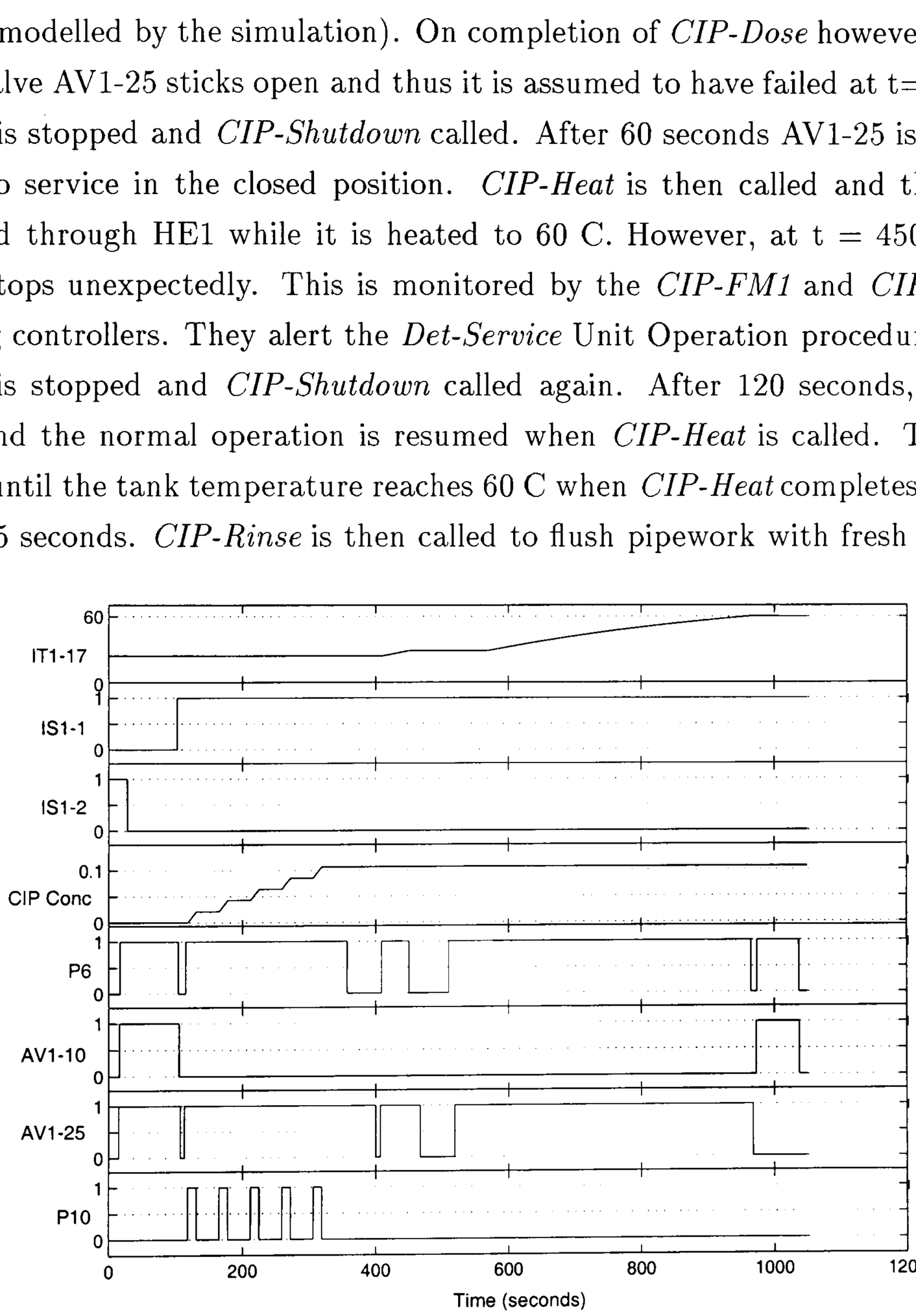


Figure 11.6: CIP Level, Concentration and Temperature profiles with valve and pump states for *Detergent-Service* procedure simulation in *gPROMS*.

11.2.5.3 Detergent Rinse Unit Operation

The results of the simulation in *gPROMS* of the Detergent Rinse Unit Operation are shown in Figure 11.7. The levels of T1 (IT1-1) and the CIP tank show how the caustic solution is drained from the CIP tank to fill T1 and then returned to the CIP tank from T1. Like

the Water Rinse Unit Operation, *Detergent-Fill-Coord* ad *Det-Drain-Coord* are called in parallel. They start their associated failure monitors before starting *Det-Fill* and *Det-Drain*. Rinsing continues until t = 400 when valve AV1-16 is accidentally opened. This is critical to both the filling and draining procedures, so abnormal situations are signalled to both by their respective failure monitoring procedures. *Water-Fill-Emergency* and *Water-Drain-Emergency* are called to close valves and stop pumps. The figure shows that P6 is idle at the time but P1 was stopped due to the emergency situation. After 60 seconds, the valve is returned to service and closed before *Det-Fill* and *Det-Drain* are called again. During the failure, the timer was held and is restarted when normal operation resumed. Once the timer expires, then the procedure is stopped at the next time T1 is empty (t = 770 sec).

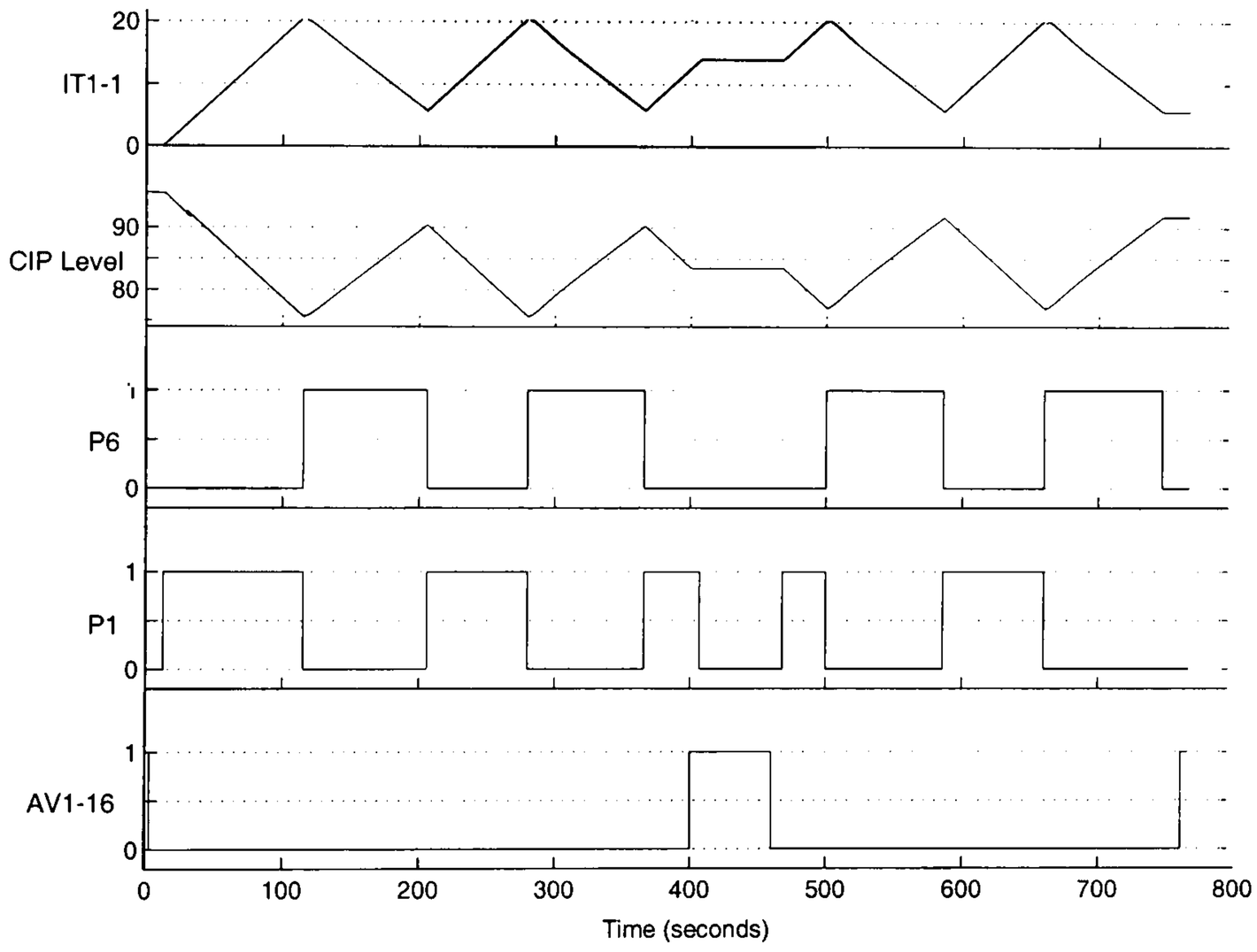


Figure 11.7: CIP and T1 Level profiles with valve and pump states for *Detergent-Rinse* procedure simulation in *gPROMS*.

11.2.6 Implementation on the Batch Pilot Plant

After being verified using *gPROMS*, the same controllers were translated into the PARA-CODE language and executed on the Batch Pilot Plant. Once again, no manually editing of the control code was required. The following three sections detail the performance of each of the three unit operations performed with associated figures. Figure 11.12 shows the profiles for the tank levels and the valve positions for a complete run of the *Clean-T1* Operation.

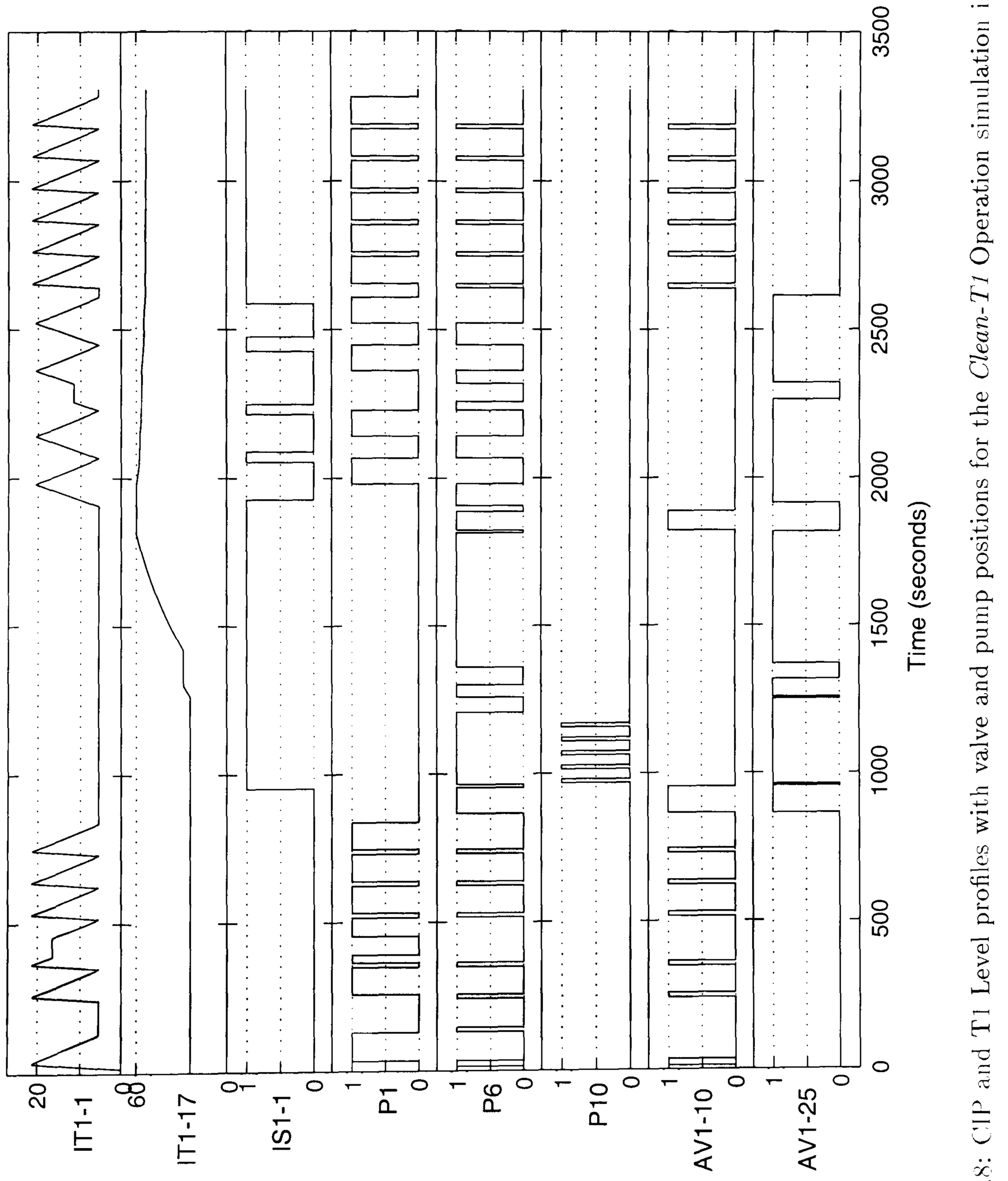


Figure 11.8: CIP and T1 Level profiles with valve and pump positions for the *Clean-T1* Operation simulation in *gPROMS*

11.2.6.1 Water Rinse Unit Operation

Profiles for the Water Rinse Unit Operation are shown in Figure 11.9. The same procedure as in the *gPROMS* simulation was followed with AV1-10 sticking at 150 seconds. Note that, it appears that AV1-10 opened and then closed at time 150 seconds. This is due to the grace time allowed before a valve is termed failed. For 10 seconds, the control system believed that the opening was successful while the grace time was active. After that, the failure was noted. During the repair time for AV1-10, the lid on T1 was opened and PS1-1 sensed this accordingly. Once both the valve was repaired and the lid closed, normal operation was resumed. Then at t = 350 seconds, valve AV1-47 was accidently closed and emergency procedures invoked. Once it was repaired, normal operation was resumed and ran until the timer had expired.

The overshoot in the levels are larger than in the *gPROMS* simulation results. Moreover, when failed items are fixed, the system seems to take a long time before it resumes normal operation. This is due to a precautionary measure placed in the translated code. A grace time of 5 seconds after every controllable transition was inserted automatically to allow for the finite time a component takes to change state. This causes the delays observed. Considering the closing of the lid of T1 at t = 230 seconds, it is not until t= 270 seconds that the level begins to rise again. This is because the communication of the recovery from a failure involves controllable transitions, as does calling the normal operations and the commands to open valves and start pumps. Strategies to reduce this effect are discussed in section 12.3.1.

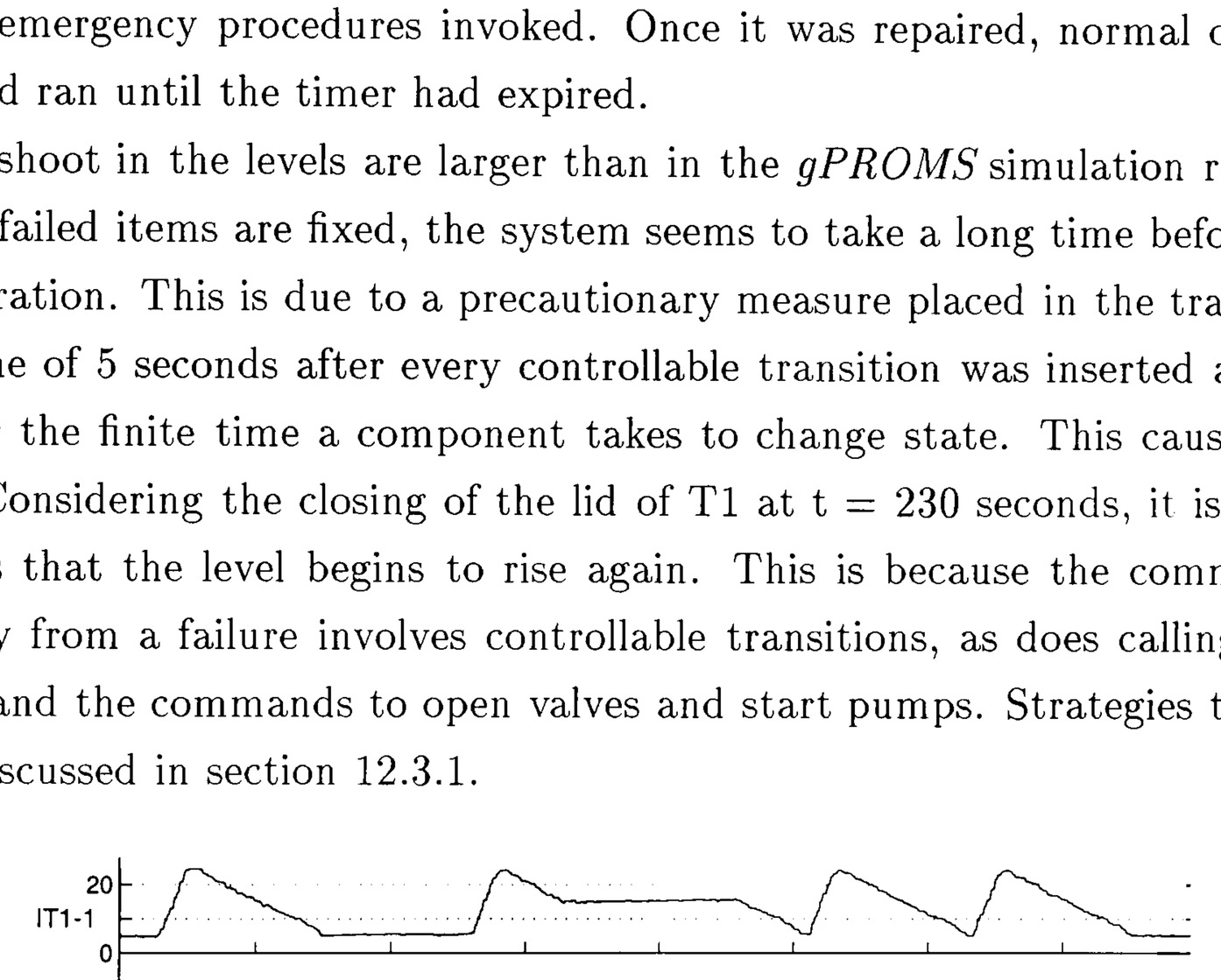


Figure 11.9: Level profile and valve positions for *Water-Rinse* procedure operation on the Batch Pilot Plant.

11.2.6.2 Detergent Service Unit Operation

Procedural controllers for the Detergent Service Unit Operation were also translated into PARACODE and executed on the Batch Pilot Plant. Figure 11.10 shown temperature and level profiles as well as some keys valve and pump positions. The same operating conditions as in the *gPROMS* simulation were applied with the procedural controllers successfully handling alarm situations. The figure shows that the CIP tank is successfully filled by *CIP-Fill* (IS1-1 becomes energised and IS1-2 deenergised) without any component failures. *CIP-Dose* is then executed and after three doses of caustic soda through P10, the concentration sensor (IS1-3) becomes energised. However, when the procedure is stopping AV1-25 becomes stuck open (t=370 seconds). It is then repaired and *CIP-Heating* starts (t=490 seconds). During *CIP-Heating*, P6 stops unexpectedly (t=800 seconds) causing the procedure to stop and *CIP-Shutdown* to be called. After 120 seconds, P6 is repaired and *CIP-Heating* resumed. Once IT1-17 senses that the temperature has reached 60 C (t=1190 seconds), *CIP-Heating* stops and then a final rinse of the surrounding pipework is performed by *CIP-Rinse*.

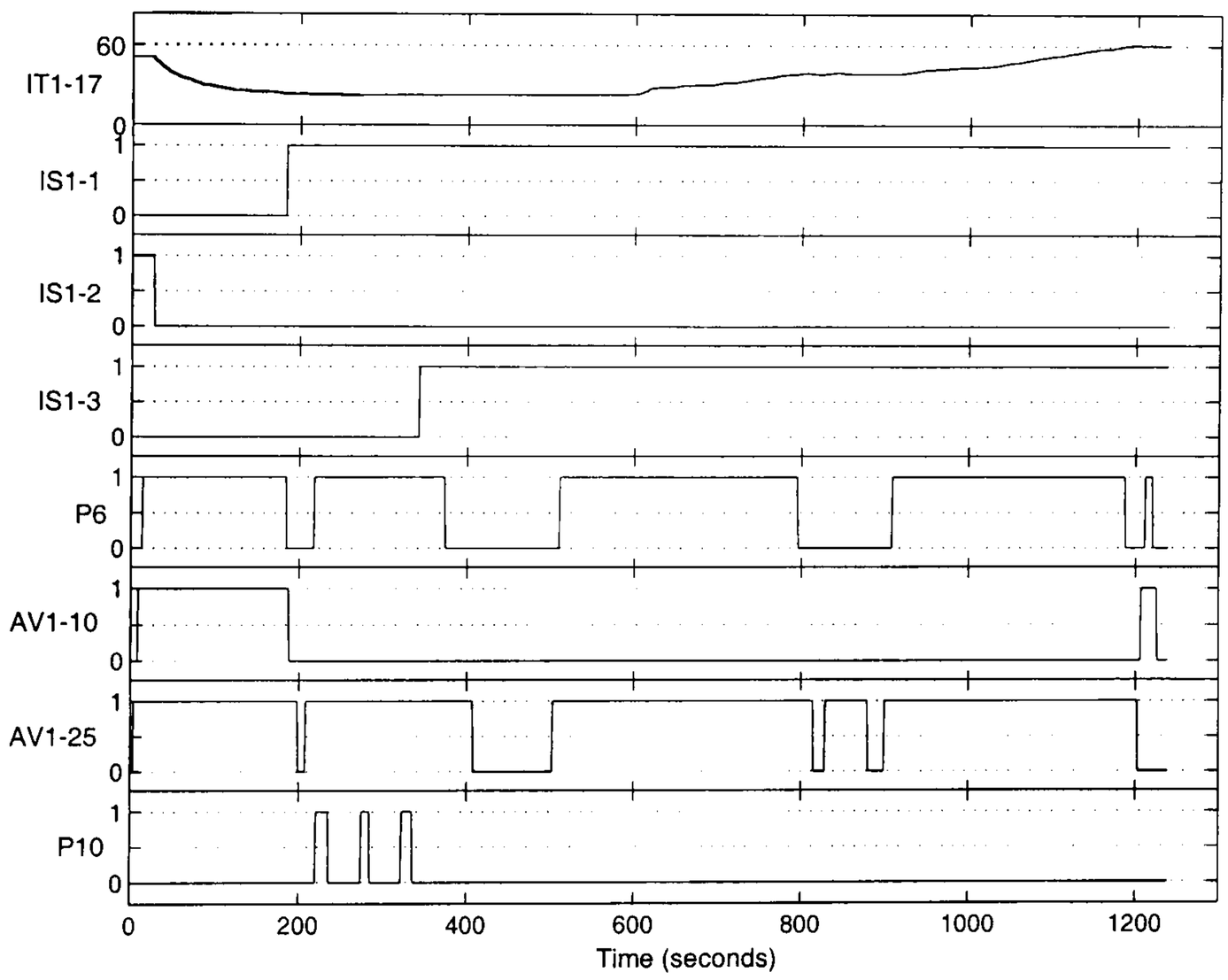


Figure 11.10: Temperature profile and valve positions for the *Detergent-Service* operation on the Batch Pilot Plant.

11.2.6.3 Detergent Rinse Unit Operation

Figure 11.11 shows the profiles for the Detergent Rinse Unit Operation on the Batch Pilot Plant. The same operating circumstances as in the *gPROMS* simulation were reproduced for this run. The figure shows how normal operation occurs until t = 190 seconds when AV1-16 is accidently opened. Both *Det-Fill-Coord* and *Det-Drain-Coord* are affected by the failure and call their emergency procedures. As a result, P6 is stopped and the

associated valves closed. After 60 seconds, AV1-16 is repaired and returned to service. Notice that, once again the delay due to the inclusion of the 5 second grace time in various places causes the system not to resume normal operation for a further 40-50 seconds. Normal operation then continues until the timer expires and the unit operation is stopped at the next safe process state.

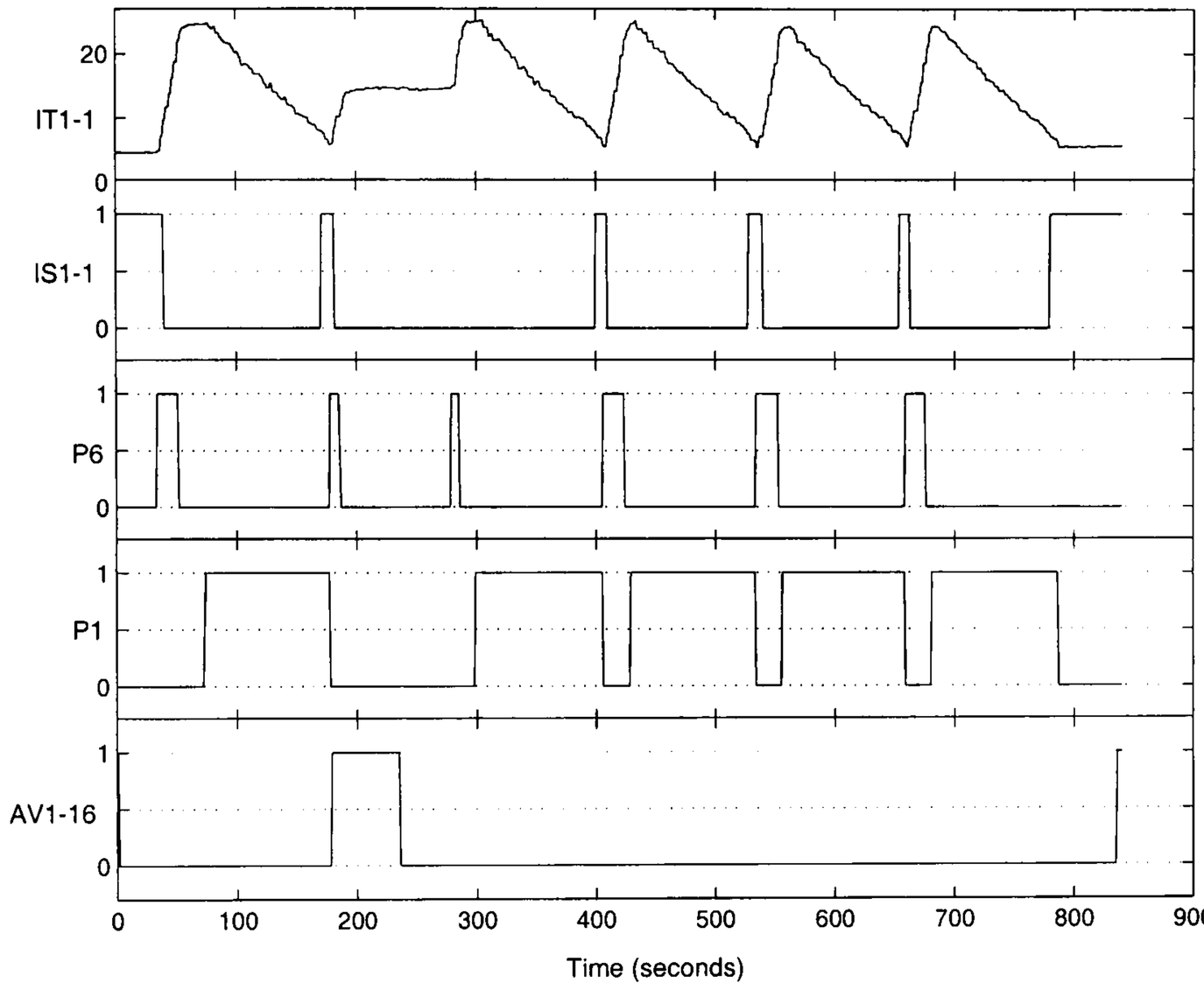


Figure 11.11: CIP and T1 Level profiles with valve and pump states for *Detergent-Rinse* procedure operation on the Batch Pilot Plant.

11.2.6.4 Summary of Case Study 1

The cleaning of Tank T1 accounting for equipment failures was successfully achieved through the synthesis of hierarchical and partitioned procedural controllers. The *Clean-T1* unit procedure was (manually) decomposed and partitioned into 26 procedural controllers that were then recombined using Hierarchical Theory. The controllers combined to form *phases*, *operations* and *unit operations* that conform with the ISA-S88.01 standard. It was shown that this implementation of Hierarchical and Partitioning Theory satisfies the rules of PCT and that the resulting procedure conforms to specification, is nonblocking and is controllable.

The formally synthesised, provably correct, procedural controllers where translated into *gPROMS* TASKS and implemented in a combined discrete/continuous dynamic simulation. Once verified by the simulation, the procedural controllers were translated into PARACODE, compiled and downloaded into the ACCOS control system for execution on the Batch Pilot Plant. The unit procedure was executed on the plant and the results are shown in Figure 11.12. Several disturbances, by way of valve and pump failures or lid openings, occurred during operation and were accounted for by the controller. This

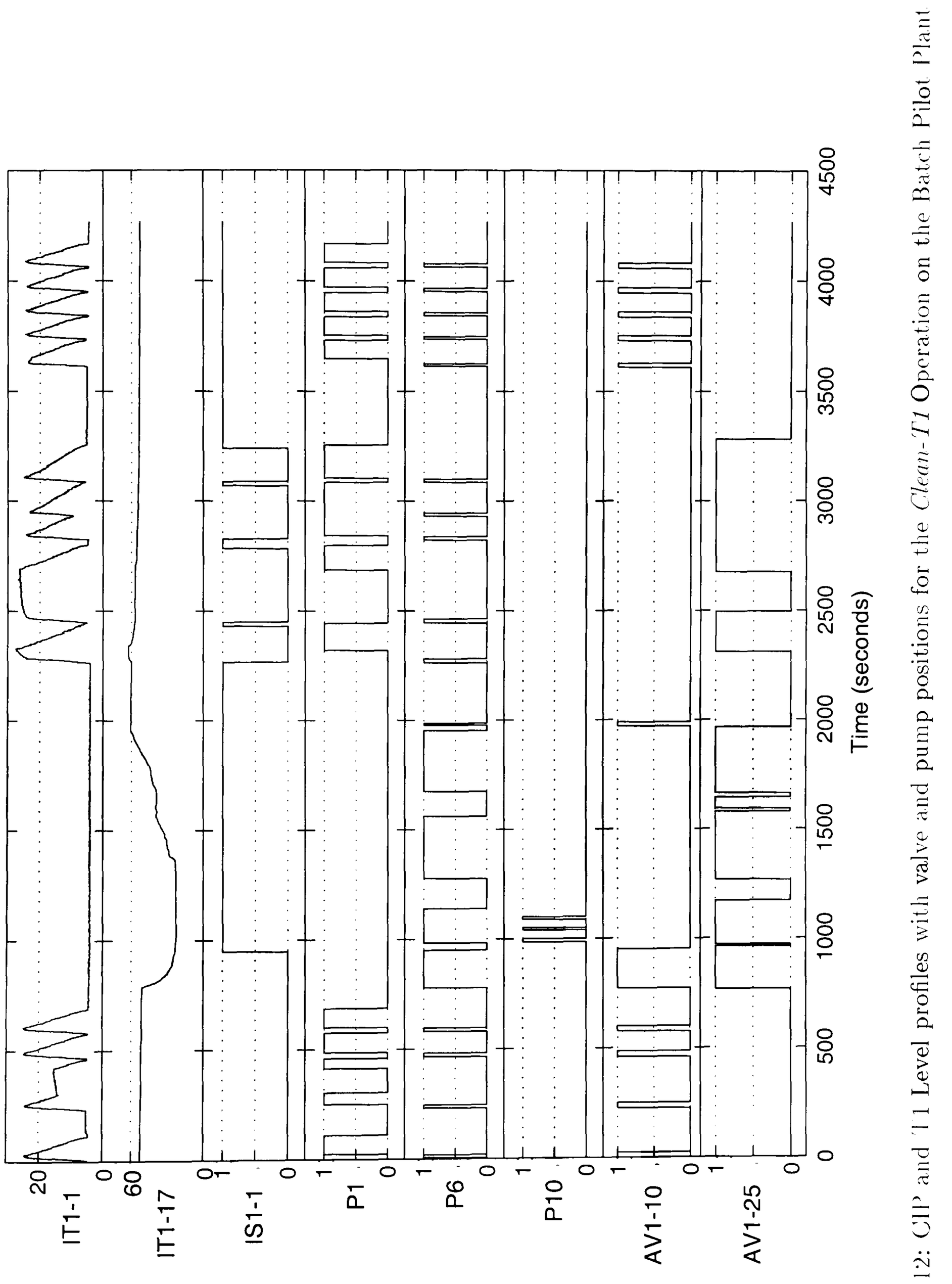


Figure 11.12: CIP and '11 Level profiles with valve and pump positions for the *Clean-T1* Operation on the Batch Pilot Plant

shows that the formally synthesised and translated *Clean-T1* unit procedure conforms to specification and is robust to process disturbances, in particular equipment failures.

11.3 Case study II - Mineral Processing Screw Conveyor

First introduced by Probst and Powers (1994) and revisited by Mouille (1997), a mineral processing transport screw conveyor problem was studied. This example is challenging since it is at the limit of current computational time barriers. The aim of this example is to show that when more complex abnormal operating specifications exist, the synthesis of abnormal procedures may be required to consider the scenarios for several components failures at once.

The process, shown in Figure 11.13, consists of a distributor bin for solid ingredients with two sides, each one with a level sensor and knife gate valve (KGV) to control the feed. The sides have a common feed from two screw conveyors in series. These conveyors have an additional three screw conveyors feeding them from three feed tanks. Two of the feed tanks contain anode cover while the third contains cryolite cover. Each tank has a KGV to control the discharge of material.

In the previous applications on this case, all the components were considered to be free of failures which avoided dealing with a large model size. Here, all components are allowed to fail. The elementary components are modelled using the techniques outlined in chapter 7. Examples of each type of elementary component are shown in Figure 11.14.

The operating specification during normal operation was taken from Probst and Powers (1994) and is given as follows:

- Starting with all KGVs closed, all screws idle and both sides of the distributor bin empty.
- The KGVs above the distribution bucket are opened.
- The transfer screws begin turning (screw 2 starts before screw 1).
- The three anode cover tank screws begin turning.
- The KGVs at each of the anode cover feed tanks are opened.
- Material enters both sides of the bucket.
- One side of the bucket fills first. The KGV above that side of the bucket closes. The other side of the bucket continues to fill.
- The other side of the bucket becomes full.
- The KGVs at each of the anode cover feed tanks close.
- The anode cover tank screws are purged (continue to turn for 30 seconds) and then stop turning.
- The transfer screws stop turning.
- The KGV above the bucket which filled last closes.

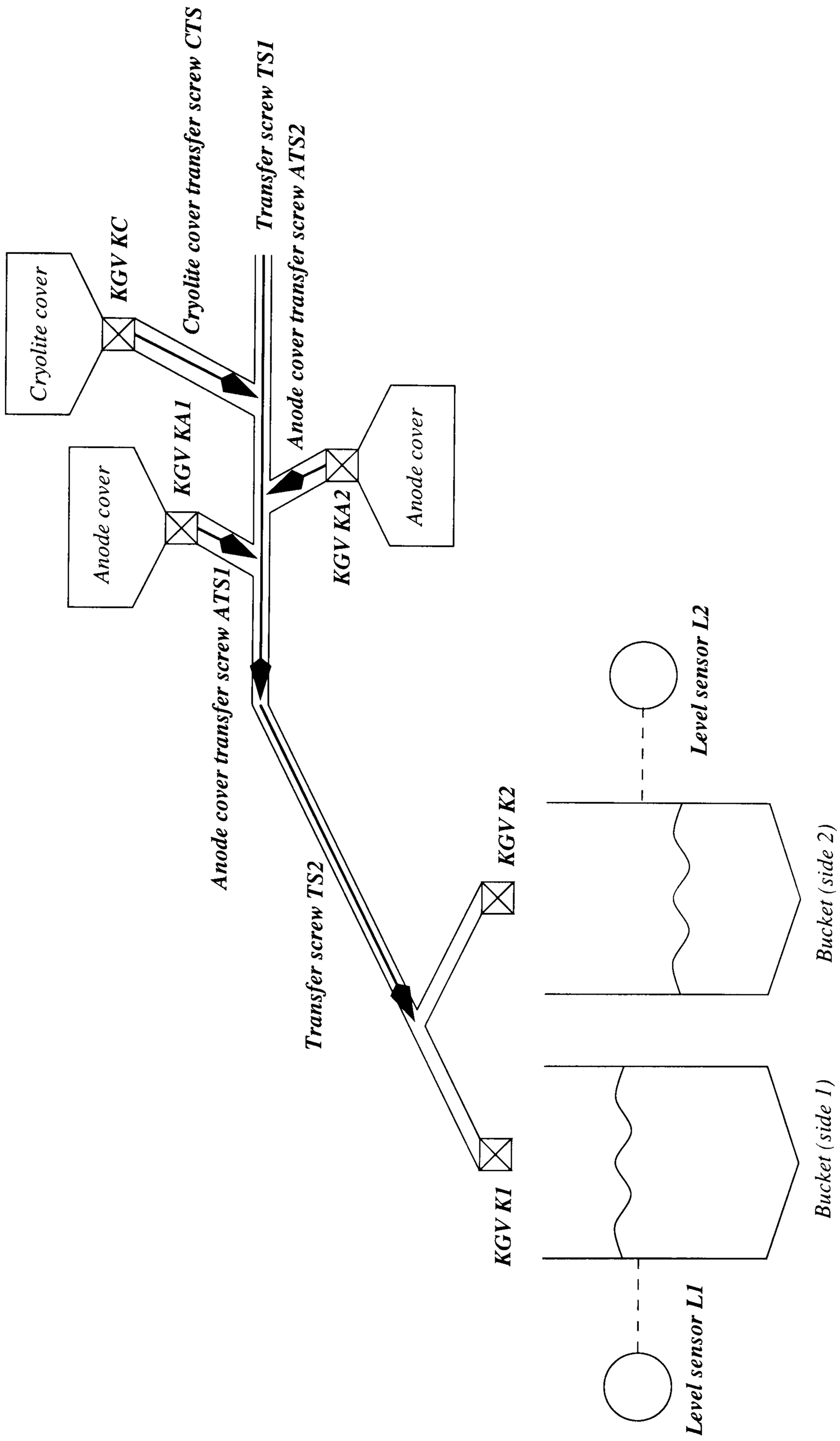


Figure 11.13: Schematic of Screw Conveyor.

- At all times if a screw is idle, then the screw immediately above it should be idle.

These specifications required the introduction of an elementary component describing a timer, taking the total number of elementary components to 13. Since all the components except the timer, are allowed to fail, extra specifications are required for abnormal operation. These are:

- If a transfer screw stops unexpectedly,
  - stop all screws feeding to it,
  - close all KGVs feeding to it,
  - purge all screws below it, and then
  - close KGVs to distributor bucket.
- If a transfer screw fails to stop or starts unexpectedly,
  - stop all screws, and then
  - close all KGVs.
- If an anode cover transfer screw stops unexpectedly,
  - close KGV above it, and then
  - continue to fill distributor bucket with other anode cover feed.
- If an anode cover transfer screw fails to stop or starts unexpectedly,
  - stop all screws, and then
  - close all KGVs.
- If an anode cover KGV fails to open or closes unexpectedly,
  - stop the respective anode cover transfer screw, and then
  - continue to fill the distributor with the remaining anode cover feed.
- If an anode cover KGV fails to close or opens unexpectedly,
  - stop all screws, and then
  - close all KGVs.
- If a distributor bucket KGV fails to open or closes unexpectedly,
  - fill remaining side of distributor bucket as usual.
- If a distributor bucket KGV fails to close or opens unexpectedly,
  - continue as normal.

  - KGV can remain open as long as level does not alarm.

- If a level sensor reads a high level,
  - close the associated KGV, and then
  - if both distributor bucket KGVs are closed shutdown system.

- If a level sensor reads a high level and the KGV has failed open,
  - stop all screws,
  - close anode cover KGVs, and then
  - if open, close other distributor bucket KGV.

- If both anode cover KGVs, both anode cover transfer screws or a combination of one anode cover screw and the other KGV fail in a way that stops all anode cover delivery,
  - close relevant anode cover KGVs,
  - purge transfer screws (continue turning for 30 seconds),
  - stop transfer screws, and then
  - close distributor bucket KGVs.

It can be seen that the abnormal operation specifications are the bulk of the specifications. This is in line with industrial experience where it is recognised that 80% of the control code deals with abnormal situations.

If a single FSM was used to model the screw conveyor example, it would have had 13 elementary components contributing to more than 10^9 finite states. However, the cryolite cover KGV and transfer screw were interlocked to shut and idle, reducing the active model to 11 elementary components, but this still gives over 10^7 states. No decomposition using traditional methods is possible. The model is too large to be computed as a whole and Partitioning is employed to overcome the state explosion.

The normal operation model consists of 4 screw conveyors, 4 KGVs, 2 level sensors (all two-state components) and a timer consisting of three states. The corresponding FSM has 3096 states. The single initial state of all screws idle, all KGV closed, levels low and the timer idle is assigned. There is a single marked state: all screws idle, all KGV closed, levels high and the timer expired. Forbidden states to avoid compaction of the minerals are included in the specification.

In developing the procedural controllers for abnormal operation, the property of internal consistency (Alsop, 1996) is exploited in order to reduce each individual model's size. Only the required elementary components are included in each model. All the required procedural controllers were successfully synthesised and a hierarchical procedural controller produced. Table 11.5 lists all the modelled sub-systems, components, model size and procedural controller size.

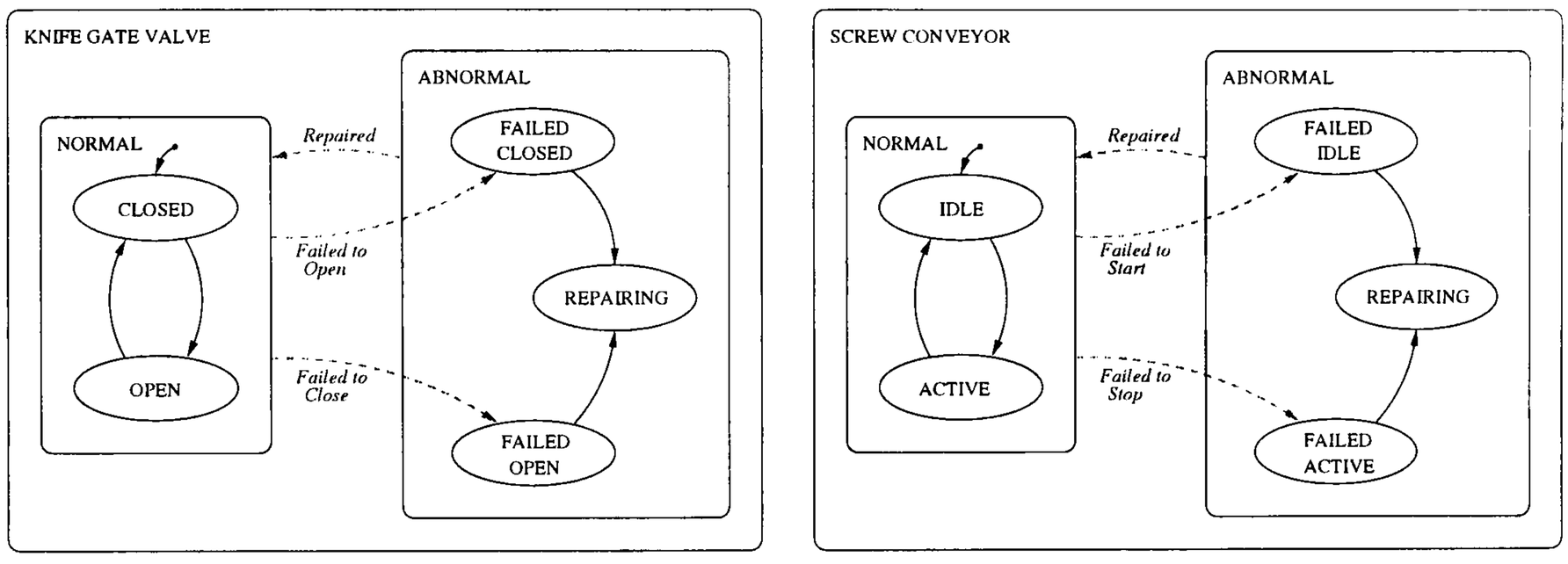


Figure 11.14: Elementary Components of the Screw Conveyor studied.

SUBSYSTEM	ELEMENTARY COMPONENTS	MODEL SIZE	CONTROLLER SIZE
Normal	`TS1, TS2, ATS1, ATS2, K1, K2, KA1, KA2, L1, L2, T1`	3096	32
TS1 Failed Idle	`TS2, ATS1, ATS2, K1, K2, KA1, KA2, T1`	768	65
TS2 Failed Idle	`TS1, ATS1, ATS2, K1, K2, KA1, KA2`	768	48
TS1 Failed Active	`TS2, ATS1, ATS2, K1, K2, KA1, KA2`	256	45
TS2 Failed Active	`TS1, ATS1, ATS2, K1, K2, KA1, KA2`	256	48
ATS1 Failed Idle	`TS1, TS2, ATS2, K1, K2, KA1, KA2, L1, L2, T1`	3096	61
ATS2 Failed Idle	`TS1, TS2, ATS1, K1, K2, KA1, KA2, L1, L2, T1`	3096	61
ATS1 Failed Active	`TS1, TS2, ATS2, K1, K2, KA1, KA2`	256	28
ATS2 Failed Active	`TS1, TS2, ATS1, K1, K2, KA1, KA2`	256	28
KA1 Failed Closed	`TS1, TS2, ATS1, ATS2, K1, K2, KA2, L1, L2, T1`	3096	67
KA2 Failed Closed	`TS1, TS2, ATS1, ATS2, K1, K2, KA1, L1, L2, T1`	3096	67
KA1 Failed Open	`TS1, TS2, ATS1, ATS2, K1, K2, KA2, L1, L2, T1`	3096	70
KA2 Failed Open	`TS1, TS2, ATS1, ATS2, K1, K2, KA1, L1, L2, T1`	3096	70
K1 Failed Closed	`TS1, TS2, ATS1, ATS2, K2, KA1, KA2, L2, T1`	1548	66
K2 Failed Closed	`TS1, TS2, ATS1, ATS2, K1, KA1, KA2, L1, T1`	1548	66
K1 Failed Open	`TS1, TS2, ATS1, ATS2, K2, KA1, KA2, L1, L2, T1`	3096	54
K2 Failed Open	`TS1, TS2, ATS1, ATS2, K1, KA1, KA2, L1, L2, T1`	3096	54
L1 Failed High	`TS1, TS2, ATS1, ATS2, K1, K2 KA1, KA2, L2, T1`	1548	45
L2 Failed High	`TS1, TS2, ATS1, ATS2, K1, K2 KA1, KA2, L1, T1`	1548	45
L1 Failed High and K1 Failed Open	`TS1, TS2, ATS1, ATS2, K2 KA1, KA2`	256	33
L2 Failed High and K2 Failed Open	`TS1, TS2, ATS1, ATS2, K1 KA1, KA2`	256	33
ATS1 Failed Idle and KA2 Failed Closed	`TS1, TS2, ATS2, K1, K2 KA1, T1`	768	76
ATS2 Failed Idle and KA1 Failed Closed	`TS1, TS2, ATS1, K1, K2 KA2, T1`	768	76
ATS1 Failed Idle and ATS2 Failed Idle	`TS1, TS2, K1, K2 KA1, KA2, T1`	768	44
KA1 Failed Closed and KA2 Failed Closed	`TS1, TS2, ATS1, ATS2, K1, K2 T1`	768	80

Table 11.5: Partitioned models of the Screw Conveyor.

Note that 25 problems of order $\bigcirc(10^3)$ are solved rather than a single problem of $\bigcirc(10^7)$. This dramatically reduces the computational load, making the problem tractable.

11.4 Case Study III - Benchmarking with an Industrial Example

A final case study was performed with the assistance of an undergraduate student in which the synthesis techniques of this thesis were applied to a small pasteurisation plant. The plant is taken from a previous project (Presto Project P4, 1996) and is a hypothetical but representative industrial plant used internally by APV for standardising its engineering procedures for the design and implementation of control systems. The plant, known as Plant X, is shown in Figure 11.15.

The plant consists of three main process units, *Raw Milk Tank*, *Pasteuriser* and *Pasteurised Milk Holding Tank*. The operating procedure is described as follows. Raw milk is delivered by truck and stored in the Raw Milk Tank. The pasteuriser is first sterilised and brought to an equilibrated temperature by recirculating hot water. Raw milk from the Raw Milk Tank is then passed through the pasteuriser to the Holding Tank under strict temperature control. Then, product trucks collect the milk from the Holding Tank.

Procedural controllers for the normal operation of the complete plant were synthesised. In addition, procedural controllers including alarm handling capabilities for the Raw Milk Intake Unit Operation were also synthesised. For the latter, four of the ten elementary component instances could fail during operation (the pump, valves and the hose connector). A single normal operation controller and six abnormal operation controllers were synthesised to satisfy the specifications.

To test the resulting procedural controllers, the Raw Milk Intake procedural controllers were translated into the *gPROMS* language and used to run a dynamic *gPROMS* simulation of the uncontrolled plant. The operation scenario includes the arrival and unloading of three raw milk tankers, with a variety of problems occurring (the hose becomes disconnected, a valve fails and a pump fails). The results are shown in Figure 11.16. As it can be seen, the procedural controller manages the unit operation and successfully fills Tank 100 despite several failures occurring.

11.5 Using Sequence Developer in a Teaching Role

Sequence Developer was used as part of a Masters of Science degree course taught within the Centre for Process Systems Engineering at Imperial College in 1999. The objectives of the the project were:

1. to fast-track and test the GUI development by exposing it to a large user base.

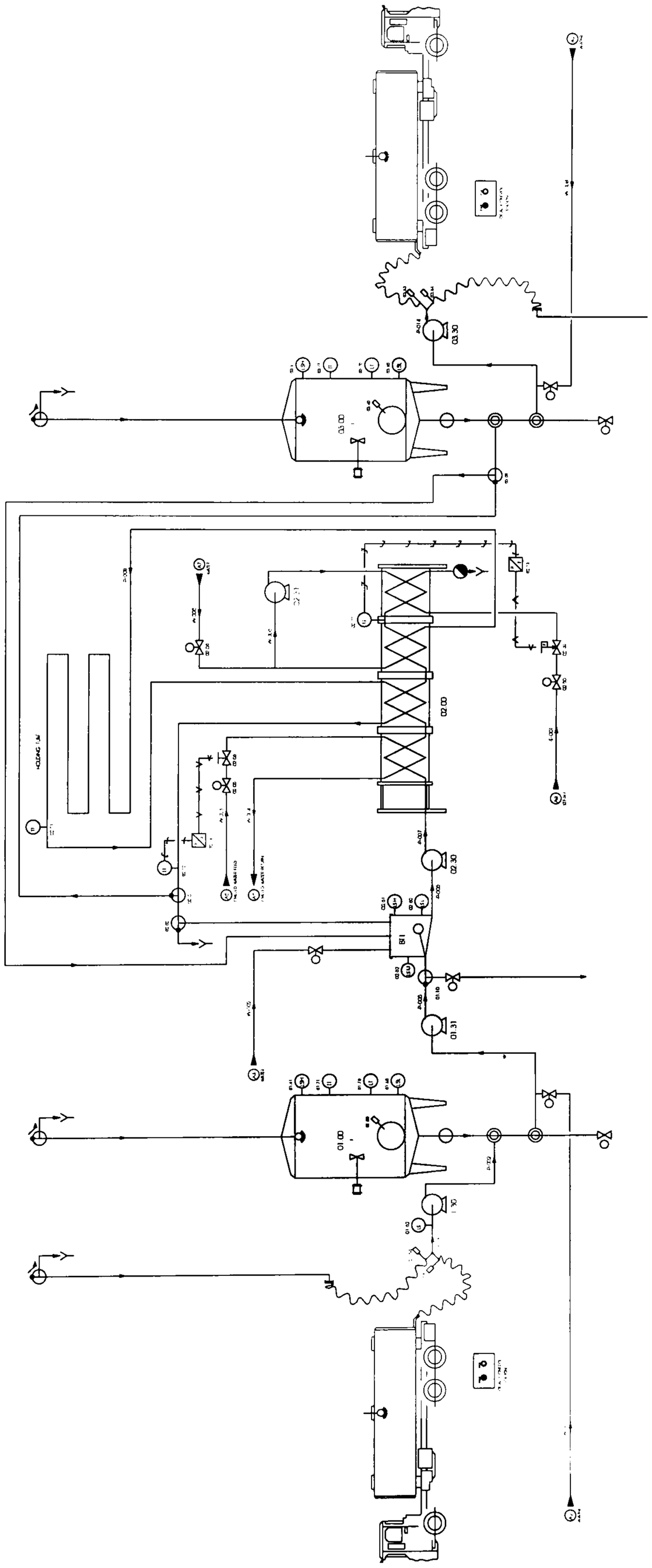


Figure 11.15: PLANT LAYOUT

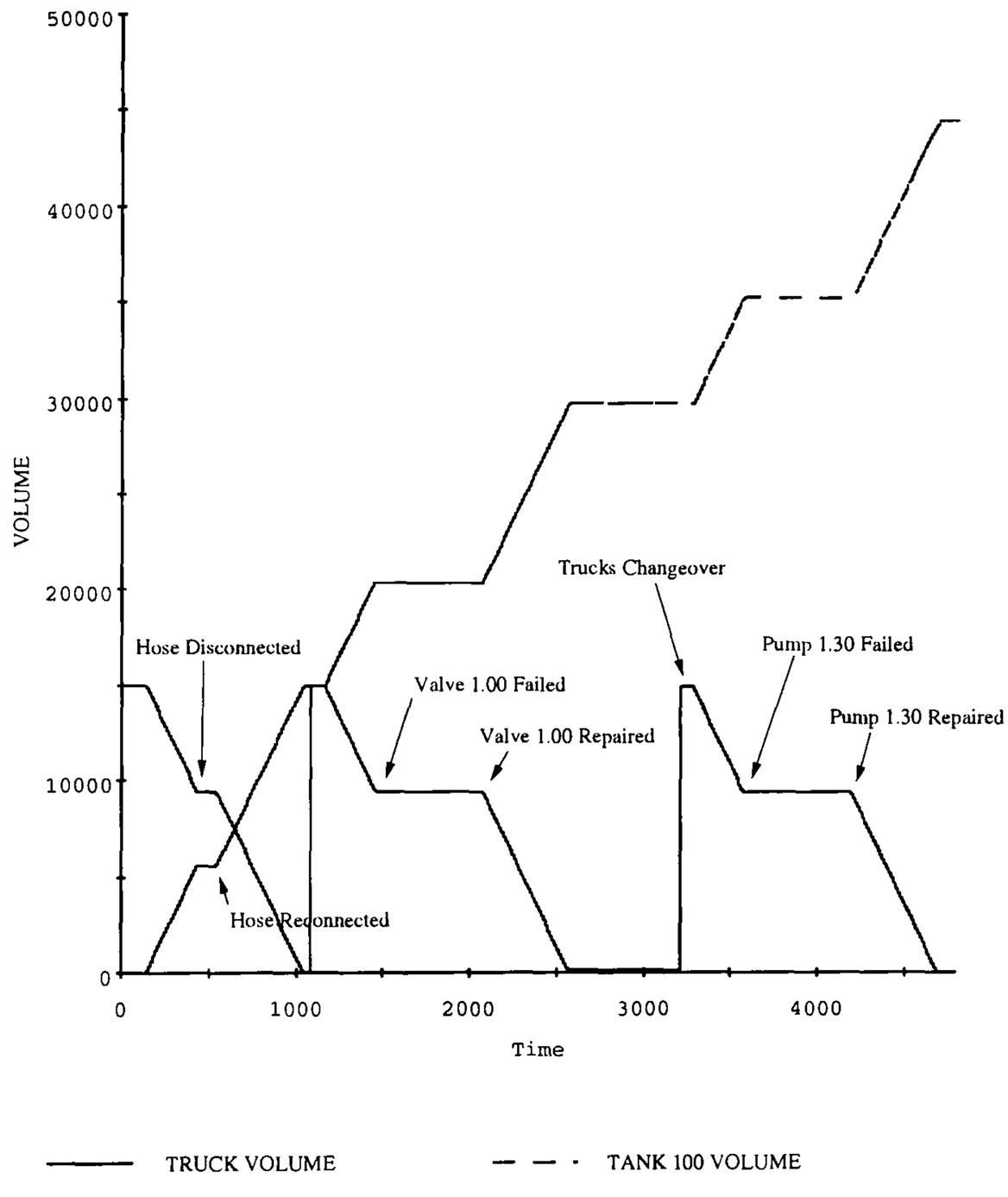


Figure 11.16: Tank 100 and the trucks' volume profiles for the Raw Milk Intake Procedure.

2. to show how PCT could be used by people with little previous experience and minimal training, and

3. to show how the GUI and translation algorithms can help in the rapid development of procedural controllers.

The course was run over five weeks with just five hours of lectures providing a background in the theory. The majority of the time was spent using the software in a project, with approximately 10 hours a weeks allocated to this tool. The objective was to repeat the synthesis of procedural controllers for the normal operation of the batch pilot plant described in section 11.2. The controllers were then translated into gPROMS code and a simulation was run to verify that they achieved the goals and conformed to specification. Thirty-Three students divided into 11 groups were enrolled in the course.

All students completed the tasks within the specified time with variable degrees of ease. It was found that the GUI and its comprehensive error checking provided an easy environment for the students to model and specify the process and operations. Significant refinements in the error-checking, translation algorithms and general information presentation were achieved over the duration of the course. Results suggest that the techniques and software will greatly impact on the rapid development of procedural controllers.

11.6 Conclusions from the Case Studies

The four case studies presented show that new techniques developed in this thesis enable the synthesis of procedural controllers for large processes and those that contain equipment that can fail. The also showed that the hierarchical and modular approach, combined with sophisticated software reduces the burden on the designer aid them in the rapid production and retrofit of procedural controllers.

During the case studies issues were raised that showed the benefits of the new techniques as well as some potential areas for improvement. As they were a direct result of the case studies they are presented below.

11.6.1 Quantitative Dynamics

The modelling framework used in the discrete environment contains no quantitative time steps and more significantly no quantitative dynamic information with regard to transitions that are modelled. An assumption is made that controllable transitions occur instantaneously but the time that it takes for an uncontrollable transition to occur is not considered. This allows the synthesis of controllers that may achieve their goals in the discrete domain, but could fail to do so in the continuous domain due to the dynamics of the process. This was encountered when verifying the first *Water-Rinse*. When the failure of AV1-47 was tested, the procedural controller successfully brought the plant to a safe state. However, when AV1-47 was returned to service, the system remained in the

safe state because both *Water-Fill* and *Water-Drain* were specified to do nothing when the level was *Okay* (between 6 and 20 %). While *Water-Fill* waited for a fall in level, *Water-Drain* waited for a rise. The discrete event model was correct in its own view of the situation in the sense that it allowed the level to rise, fall or not change (as it did). This was clearly a problem with the current specifications and a quick change to the specifications of *Water-Fill* and *Water-Drain* overcame the problem. As a result, the benefits of verification against a combined discrete/continuous model before implementation on a real plant were clearly shown. Also, the ability to quickly retrofit the procedural controllers was made easier by Sequence Solver.

11.6.2 Uncontrollable Transitions and Controller Size

From Tables 11.3 and 11.4, it is seen that the largest procedural controllers are those related to failure monitoring. This happens despite having quite small potential model sizes and is due to a large number of uncontrollable transitions that need to be considered. The nature of PCT requires that all states are explicitly generated. When a large number of uncontrollable transitions are present, the operating specification cannot reduce the model size to produce a small controller.

In this case, there are very few significant transitions. This is to say that for most of the time, when an uncontrollable transition occurs, the controller takes no action. So the question arises as to why every transition must be tracked with the explicit knowledge of each state.

When PCT was first defined by Sanchez (1994), operating specifications were represented by lattice structures such as shown in Figure 11.17. This was then intersected with the asynchronous product and other specifications to produce the procedural controller. It is now suggested that, instead of beginning with the asynchronous product and slowly reducing its size by applying specifications, the synthesis approach should assess whether a lattice structure can represent the desired operation of the controller. If so, the complete process model need not be synthesised. For example, the failure monitoring controller in the first case study could be implemented as the four state procedural controller shown in Figure 11.17. This reduces failure monitoring controller sizes and in fact creates a fixed controllers size that is independent of the number of components that are being monitored. As a result, more efficient procedural controllers and thus control code can be synthesised.

11.6.3 Communication between procedural controllers

As outlined in Chapter 5, the introduction of Hierarchical Theory enabled the communication between procedural controllers without the use of flag variables. A procedural component was introduced to represent those states of operation of a procedural controller (Figure 5.2). For most controllers this is sufficient. However, the only communication available to other controllers is the confirmation that the procedural controller has done

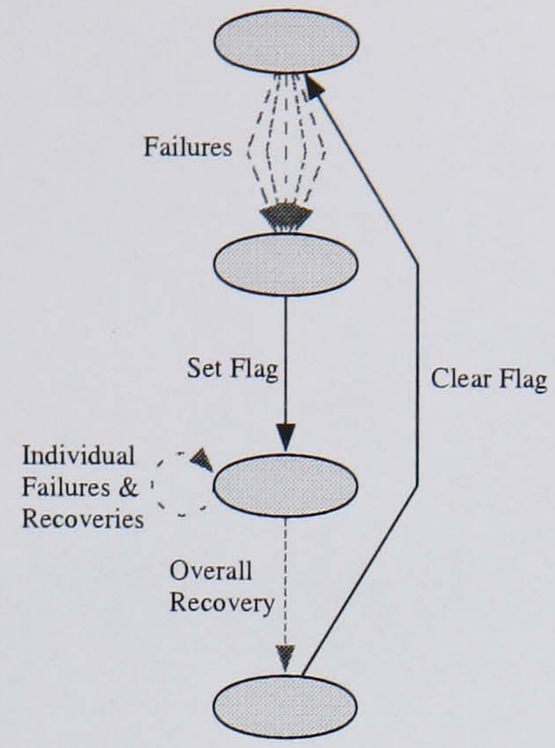


Figure 11.17: Lattice structure representing the operating specification for a failure monitoring procedure.

what has been requested, that is, it has completed normal operation, completed restarting or completed the hold logic, for example. It does not allow for a controller to communicate the state of operation within the controller itself.

Considering the failure monitoring procedural controllers, a flag variable is used to communicate failure information to other procedural controllers. As is shown in Figure 11.18, the procedural controller has an underlying partitioned structure of *Okay* and *Failed* operation. If this structure was captured by the procedural component, then the flag variable could be eliminated.

As a subject for future research, it is proposed to investigate techniques that enable each procedural component to contain a more detailed model of the behaviour of the procedural controller. Such a model is suggested in Figure 11.19. Its practical application is still to be investigated.

11.6.4 Feedback Failure and Soft Sensors

During the implementation of the procedural controllers on the Batch Pilot Plant, a type of failure occurred which was outside the scope of this thesis. During the *Water-Rinse* unit operation, the intention was to manually close AV1-47 (by overiding the control command). It was expected that the feedbacks would not correspond to the driven signal and thus the valve would fail. Instead, when the valve was closed, the feedback signals remained unchanged (signalling that the valve was still open). So the procedural controller continued normal operation unaware of the failure. This situation could also arise if a blockage or leakage occurred in a pipe, or in the casse of failure of the valve position sensor or signal.

However, other variables indicated that a failure had occurred. Namely, the level of T1 stopped falling. The system also has a measurement of the flow rate from P1 which was initially omitted as its value was thought to be unimportant. Now, it is realised that it

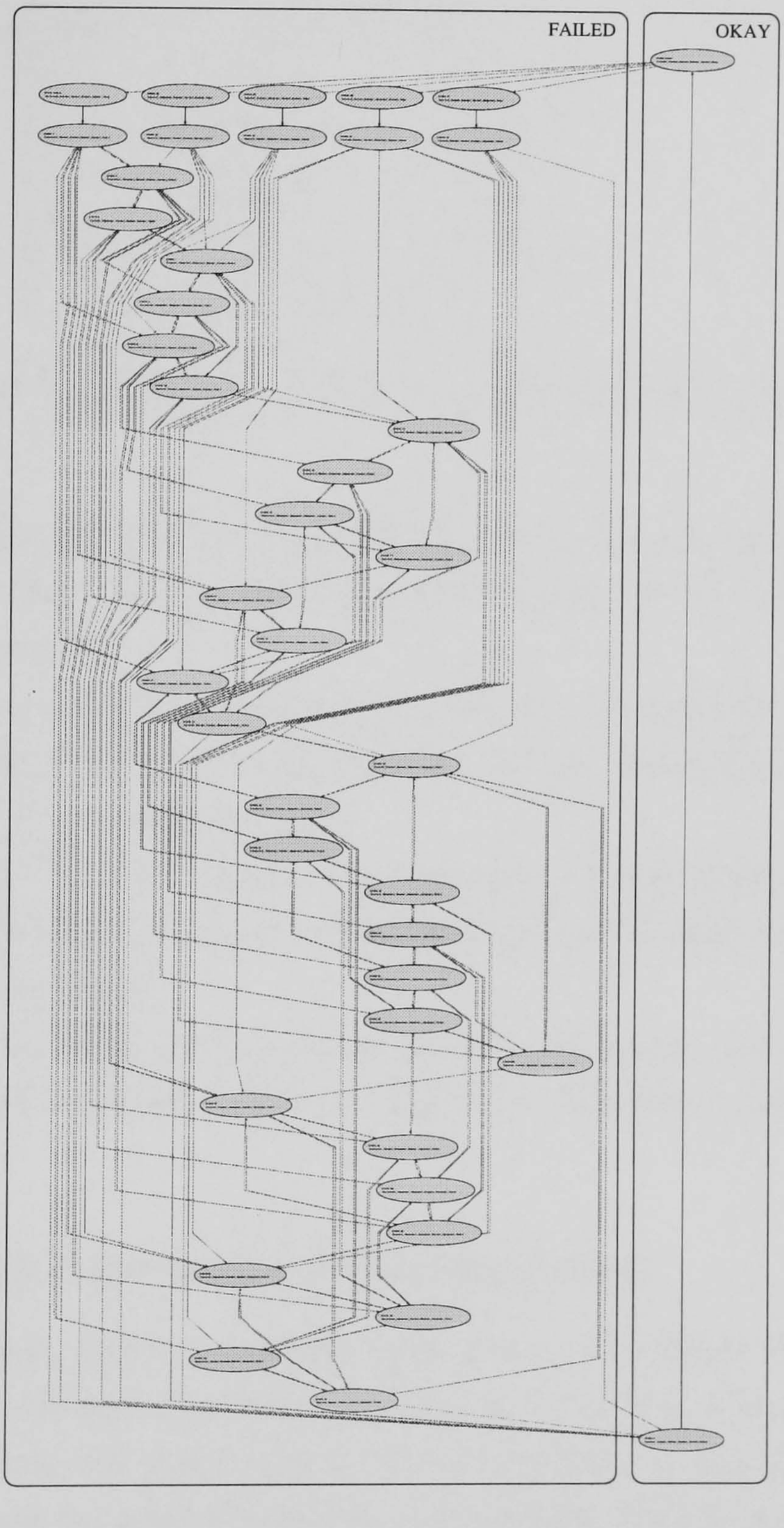


Figure 11.18: CIP and T1 Level profiles with valve and pump states for *Detergent-Rinse* procedure operation on the Batch Pilot Plant.

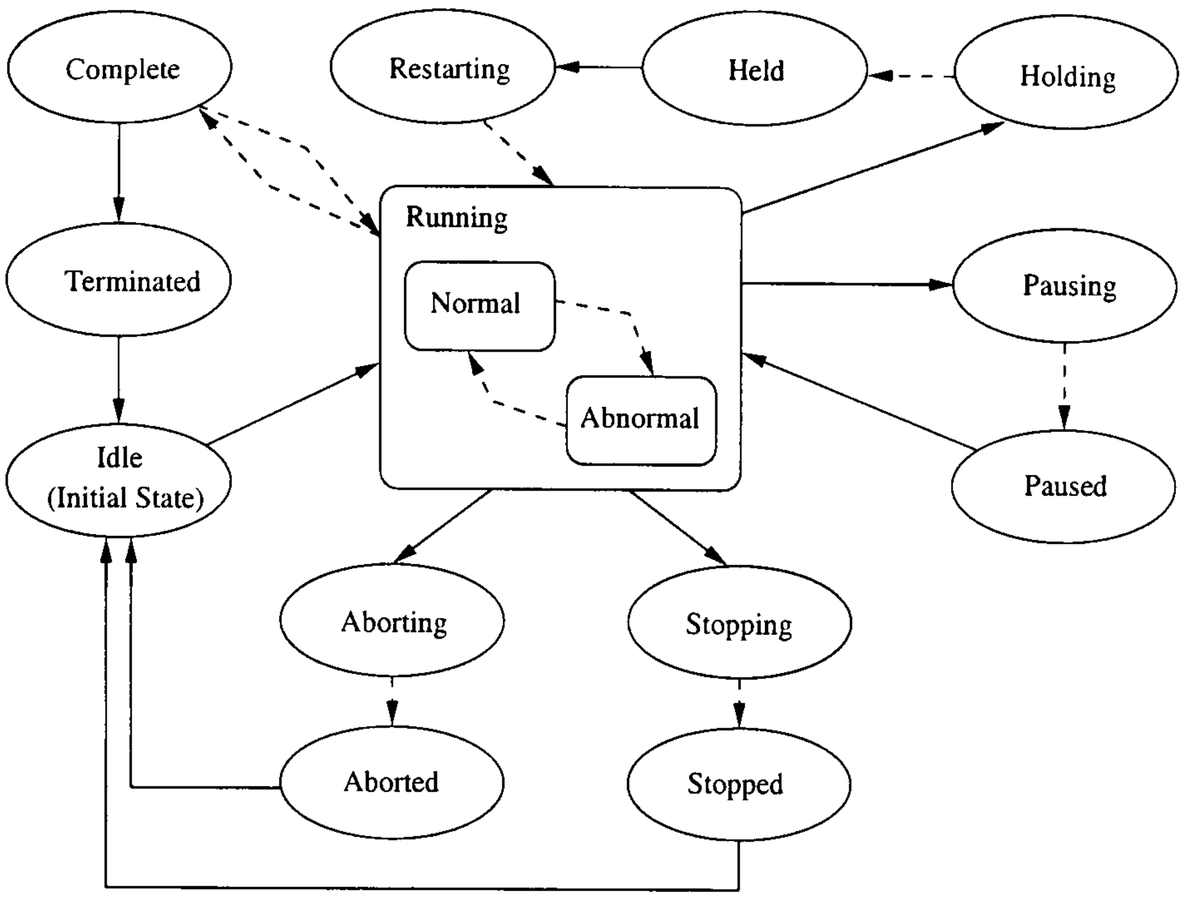


Figure 11.19: Proposed procedural component enabling easier communication between controllers.

is necessary. The error lay both in the modelling and in the specification of the system. It is important that every possible signal to and from the control system is modelled to create the complete model.

Realising that the sensor was required was intuitive (*a posteriori*) and needs to be formally included. The question of observability of systems, especially under signal failure, is also an area that requires further work.

If it is assumed that there was no flow measurement, an alternative method to assess for a failure could use the level sensor. If its value did not change within a certain time limit an alarm could be raised.

More in general, the issue of using the state observer models for inferring mismatch between observed and expected behaviour and fault detection remains an interesting research aspect.

11.6.5 Integrated Controller and Process Design

In the PlantX case study, errors were found in the process' simulation model and therefore possibly in the process design. At the end of filling Tank 100 from a tanker a cyclic process of stopping and starting the pump arose. This was caused by the head of milk in the tank being large enough to cause milk to flow back through the pump and into the tanker by gravity with the pump off. To overcome this, the process required a non-return valve in the feed line to the tank for the desired operation to be achieved. Thus, the benefits of using a simulation tool to verify the design of both the control code and the actual process, were shown.

11.6.6 Forcing Sequences under Partitioning

The aim of the PlantX study was to compare traditional methods of synthesising procedural controllers for systems containing failures with the methods derived from Partitioning Theory. The work was performed by an undergraduate student receiving a brief introduction to the theory before using the software tools. It was found that using traditional methods the models very quickly became too large to understand. Computation of languages and FSMs was possible but understanding them was very difficult. This was accentuated when every component modelled as a failure included uncontrollable transitions in the path from one physical state to another. For example, valves were defined as shown in Figure 7.2(b). To open the valve, a forcing specification had to be written and then a specification forbidding any other controllable transition to happen while the controller waits for the valve to open had to be included. This caused a specification forcing a sequence of controllable transitions to be broken into several specifications forcing each transition. The specifications became more state based and history qualities were lost, which created other problems as discussed in chapter 9. As a result, controllers could not be produced using the traditional synthesis methods.

Using Partitioning enabled the student to first synthesise procedural controllers for the normal operation of PlantX and verify that they conformed to specification and achieved the goals. The abnormal operation procedural controllers were then just added and verification of the alarm handling controllers against a plant with failures was performed. By keeping the failure monitoring separate from the normal operation, sequences of transitions could be forced and transition histories maintained.

Chapter 12

Conclusions and Future Work

This thesis continues the evolution of Procedural Control Theory after its inception by Sanchez (1994) and extension by Alsop (1996). The framework has evolved theoretically to now encompass the use of hierarchical controller architecture, production of concise controllers through the use of multiple initial states and the introduction of partitioning theory which enables the treatment of failures. Implementation issues have also been addressed with the introduction of a structured procedural control architecture, and the development of software to synthesise procedural controllers and to guide the user during process modelling and operating procedure specification.

12.1 Significance

The major contribution of this work to PCT and, in consequence, to the control of discrete event systems can be summarised as follows:

- Unique Alarm Handling
  - Failure handling is a large part of the current procedural control code. To be able to synthesise *provably correct* procedural controllers that handle Safety Critical Systems in their most potentially dangerous state, constitutes a very important step forward.
  - The inclusion of failures in traditional techniques caused state explosion and made synthesis impossible. Now, we can synthesise procedural controllers with alarm handling capabilities for large processes by using Partitioning Theory.
  - Traditional PCT assumed that events are instantaneous which is not always true. Partitioning Theory now allows this assumption to be removed.
- Structured Implementation
  - Industrial standards are increasingly being introduced with the aim to create a general structure. To minimise errors when converting from the FSM for-

mat to the desired presentation format, a formal structure and method for implementation has been proposed.

- To minimise manual intervention between procedural controller synthesis and implementation, an automatic translation algorithm was developed. This is essential to maintain the codes integrity.

• Computer Aided Design

- The techniques used to model discrete event systems mean that even the smallest system quickly becomes complex and synthesis is an arduous task if done by hand. Therefore, a software package to synthesise provable correct procedural controllers has been developed.
- Capturing mistakes at the early stage of a design saves considerable effort, thus a Guided User Interface (GUI) was added to the software package providing thorough error checking and guidance in the beginning.
- To make the technology available to a much wider community, the GUI includes interfaces that enable a very sophisticated theory to be used by engineers in a simple and understandable fashion.

12.2 Summary

The thesis has been arranged in three main parts: background, theoretical developments and practical developments.

The background for the thesis is presented starting with a literature review in Chapter 2. Several approaches to formal techniques for specification, verification and formal synthesis of controllers for discrete event systems were reviewed. PCT was chosen because it could be considered a robust synthesis method that could provide the best opportunities for the application of formal techniques to process industry problems. On the other hand, statecharts were identified as a viable technique for modelling complex behaviour in a concise way. They mapped easily to industrial code structures and to the common way of thinking that most people approach these problems. Summaries of PCT and Statecharts followed in Chapters 3 and 4, highlighting the significant features of both formalisms that are relevant to this work.

Theoretical contributions of this thesis begin in Chapter 5, where hierarchy in PCT is formally introduced. Theory in Chapter 5 builds on Decomposition Theory (Alsop, 1996), which enabled large problems to be tackled, and formalises the hierarchical methods for the control of the decomposed system. Modelling techniques were proposed for the representation of lower level procedural controllers within higher level procedural controllers. Theoretical foundation was given for the formal synthesis of each procedural controller in the hierarchy as a provably correct controller, so that the overall procedural controller

was also provably correct. The method allows a hierarchy of controllers to be developed with any number of levels.

In Chapter 6, methods dealing with multiple and covered initial states were introduced to reduce the number and size of procedural controllers required to meet an operating specification. Here, a new modelling concept was presented whereby a process model does not contain initial states or marked states. This is due to the fact that initial and marked states are context dependent and determined by an operating specification. This increases the reusability and modularity of controllers.

The developments of Chapters 5 and 6 were brought together to develop Partitioning Theory presented in Chapter 7. The problem of state explosion when the models of elementary components include failures using traditional methods was addressed. A new modelling formalism was proposed that recognises the unique way that failures affect a process, and the common practice of separating the operating specifications for a process under normal operation from those for a process when failures have occurred. With this, a method was proposed which models components compactly as statecharts with regions of normal and abnormal operation. Procedural controllers for each region of operation were then synthesised to meet specification. These smaller procedural controllers (some with multiple initial states) can then be combined using hierarchical theory to produce an overall procedural controller which conformed to specification and is nonblocking.

It was recognised that for theoretical developments to gain acceptance, their application to real world problems must be considered. In Chapter 8, a control architecture that enables the rapid implementation of procedural controllers on real plants is proposed. The architecture possesses a three tiered system whereby a procedural controller, synthesised in the purely discrete domain, interfaces with the process (in the discrete/continuous domain) through interpretors. The formalisation of the implementation method helps to minimise semantic errors associated with translation.

In Chapter 9, a new synthesis algorithm is proposed to increase the size of procedural controllers that can be produced using traditional PCT methods. The algorithm is then used as part of the new techniques to design procedural controllers for large scale problems. The achievements of the work include building the procedural controller from the initial states and using the process modelling and operating specification constraints together so as to never synthesise the complete process model. Thus, problems with large models but strict operating constraints can be synthesised while the overall model is never generated.

The need to make the theory available to a wider user base was recognised in Chapter 10. A graphical user interface (GUI) was developed to guide the user through the stages of process model definition and operating specification before the synthesis package is called. The GUI provides thorough error-checking to remove syntax and semantic errors at an early stage. It also presents the problem formulation in plain English and a familiar Windows environment. Translation algorithms have also been incorporated to automatically produce the control logic this developed in a variety of target control languages. In this

case, *gPROMS* simulation code and PARACODE control code for procedural controllers is successfully produced. The automatic generation removes the risk of syntax errors on translation and greatly speeds up field implementation.

All the developments of this thesis were tested in a series of demanding practical case studies described in Chapter 11. The first case study was of a complex CIP operation for a batch pilot plant. Procedural controllers with alarm handling capabilities were synthesised, translated and implemented in a *gPROMS* simulation and on the actual plant. The controllers were found to meet specification and handle all alarm situations. Two other case studies are discussed to show the application of the techniques. Procedural controllers were synthesised for a Minerals Processing Screw Conveyor and a small Milk Pasteurisation plant. Finally, the software and theory developed in this work were *acid tested* by 33 MSc students over a 5 week course with excellent succes.

To summarise, the aims of this thesis as presented in Chapter 1, are achieved by:

- Theoretically,
    1. Clarifying the ability of PCT to address large scale problems, by formalising the hierarchical implementation architecture.
    2. Reducing the number and size of procedural controllers required to satisfy a controller specification, by introducing the concepts of multiple and covered initial states.
    3. Addressing the problem of state explosion within procedural controllers when failures are considered, by developing partitioning theory which divides process behaviour into normal and abnormal operation.
- Practically,
    1. Introducing an structured implementation architecture which reflects the structure of the procedural controllers and elementary components which it implements, accounting for the structure of the simulation or industrial control code languages for actual execution.
    2. Developing a graphical user interface which guides the user through the stages of process model and operation procedure specification, while incorporating thorough error checking and hiding the theoretical developments.
    3. Increasing the power and the use of synthesis software so that it is capable of implementing sequential operating specifications using a text based input language.

12.3 Future Directions

12.3.1 Grace Times

As discussed in the case studies of Chapter 11, grace times were inserted in the translated code after each controllable transition to ensure that the affected item had enough time to complete the transition before the procedural controller moved on. This, however, created significant delays in processing when a sequence of controllable transitions occurred. Grace times after every controllable transition seem to be over cautious. A suggested solution is to place a single grace time after a sequence of controllable transitions. In other words, to place grace times at wait states, before conditions are tested. If the controllable transitions leading to the wait state have varying grace times then the maximum grace time could be used.

12.3.2 Shared Controllable Transitions

PCT does not allow the concurrent operation of procedural controllers that share components with the same controllable transitions. This is done to avoid controllers competing for the use of the component. If it was allowed, situations of deadlock and livelock may arise. However, by imposing this restriction, PCT limits its applicability as many situations do require this facility. An example is related to failures that do not cause normal operation to be abandoned as discussed below in section 12.3.3.

A suggestion to enable the sharing of controllable transitions is made by recognising that controllable transitions could be observed as well as forced and forbidden. In PCT, the ability of a component to execute a controllable transition is inferred based on the initial state of the system and the sequence of transitions that have occurred since then. Meanwhile, uncontrollable transitions are observed to occur using conditional statements. PCT assumes that items with controllable transitions can only be forced and never be observed. This, in many cases, is not true.. Then, observed controllable transitions could be shared between controllers. An external model would then consider a controllable transition that is forbidden, forced or observed.

12.3.3 Non-Fatal Failure Procedures

Quite often, items may fail during process operation that can be repaired while the process continues operating. However, the current developments regarding failure handling require normal operation to be terminated before alarm handling procedures can be invoked. The restricting factor in this case is again given by the fact that controllable transitions cannot be shared between controllers. Using the concepts of observed controllable transitions, suggested in section 12.3.2, techniques could be developed to enable the concurrent operation of the normal operation procedural controller and a non-fatal emergency procedural controller.

12.3.4 Further Hierarchical Architecture Development

In the current development of the hierarchical architecture, communication between different levels requires the definition of additional components, to be used as flags. Especially visible in the failure monitoring procedures is the use of flags to communicate the state of the procedural controller. Essentially, this can be done by describing the superstates of the procedural controller itself. Superstates of the controller, however, are very specification dependent and thus are established at the synthesis stage. However, the procedural components presented in Chapter 5 are fixed generic representations and therfore can be limiting in their application. Further development of the procedural component, possibly towards a structure similar to that suggested by Figure 11.19, is required.

12.3.5 Further Development of the CAD Tool

The GUI presented in Chapter 10 is the first step in providing a viable design environment and there is a potential for more improvement. This includes:

1. A design canvas where the process model is created by drawing a P&I Diagram using a drag and drop facility. Users would be able to select components from a library or design their own and add them to the library. Component design includes creating a suitable graphic plus the statechart representing the behaviour and inserting the associated translation information for target simulation code.

2. Live interaction between the GUI and simulation packages to animate the process specification during verification with a discrete/continuous modelling. This would enable the designer to interact with the simulation causing components to fail and so on.

3. The extension of the range of target languages to which the design results can be translated into, in order to make the package attractive to a wider audience.

Appendix A

Sample Sequence Developer Input File

```
COHPONENT Timer

 STATE Idle  INDICATOR 0  PHYSICAL Idle  PARA IFTZ     #
    VARIABLES  Idle
    TRANSITION Start TO Running  COHHAND 1 PARA RLTH  #

 STATE Expired  INDICATOR 2  PHYSICAL Expired  PARA IFTZ     #
    VARIABLES  Expired
    TRANSITION Reset TO Idle  COHHAND 99 PARA STHN  #

 STATE Running  INDICATOR 1  PHYSICAL Running  PARA IFTR     #
    VARIABLES  Running
    TRANSITION Hold TO Held  COHHAND 3 PARA FZTH  #
    TRANSITION Expire TO Expired  #

 STATE Held  INDICATOR 3  PHYSICAL Held  PARA IFTF     #
    VARIABLES  Held
    TRANSITION Release TO Running  COHHAND 5 PARA RLTH  #

 UNCONTROLLABLES
Expire

 INSTANCES
        Timer_T1 gPROHS Plant.T1 COHPNAHE Timer PARANAHE 1.200 PARANUH 370
        Timer_T10 gPROHS Plant.T10 COHPNAHE Timer PARANAHE 1.202 PARANUH 371
        Timer_T10S gPROHS Plant.T10S COHPNAHE Timer PARANAHE 1.204 PARANUH 372
        Timer_T30S gPROHS Plant.T30S COHPNAHE Timer PARANAHE 1.206 PARANUH 373

END # Component Timer

COHPONENT Level-Sensor

 STATE Okay  INDICATOR 1 PARA GT;6 AND LE;20     #
    VARIABLES  Okay
    TRANSITION OkayRise TO High  gPROHS IF Level > 20  THEN  #
    TRANSITION OkayFall TO Low  gPROHS IF Level < 6  THEN  #

  STATE High  INDICATOR 2 PARA GT;20     #
    VARIABLES  High
    TRANSITION HighFall TO Okay  gPROHS IF Level >= 6 AND Level <= 20  THEN  #

  STATE Low  INDICATOR 0 PARA LE;6     #
    VARIABLES  Low
    TRANSITION LowRise TO Okay  gPROHS IF Level >= 6 AND Level <= 20  THEN  #

 UNCONTROLLABLES
OkayRise , OkayFall , HighFall , LowRise

 INSTANCES
       Level-Sensor_IT1-1 gPROHS Plant.R_T1_2.T1.LT COHPNAHE Level_Sensor PARANAHE IT1-1 PARANUH 360

END # Component Level-Sensor
```

```
COHPONENT Flag-Sensor

 STATE Set  INDICATOR 1  PHYSICAL Set  PARA IFFS     #
    VARIABLES  Set
    TRANSITION Clear TO Clear  #

 STATE Clear  INDICATOR 0  PHYSICAL Clear  PARA IFFC     #
    VARIABLES  Clear
    TRANSITION Set TO Set  #

 UNCONTROLLABLES
Clear , Set

 INSTANCES
        Flag-Sensor_F1 gPROHS Plant.F1 COHPNAHE Flag PARANAHE T2.100 PARANUH 366
        Flag-Sensor_F2 gPROHS Plant.F2 COHPNAHE Flag PARANAHE T2.102 PARANUH 367

END # Component Flag-Sensor

COHPONENT Conc-Sensor

 STATE Okay  INDICATOR 1  PHYSICAL High  PARA IFIE     #
    VARIABLES  Okay
    TRANSITION OkayFall TO Low  #

 STATE Low  INDICATOR 0  PHYSICAL Low  PARA IFINE     #
    VARIABLES  Low
    TRANSITION LowRise TO Okay  #

 UNCONTROLLABLES
OkayFall , LowRise

 INSTANCES
        Conc-Sensor_CS gPROHS Plant.CIP_Unit.CS COHPNAHE Concentration_Switch PARANAHE IS1-3 PARANUH 361

 END # Component Conc-Sensor

 COHPONENT SISO-Pump

  STATE Idle  INDICATOR 0  PHYSICAL Idle  PARA IFINE     #
     VARIABLES  Idle
     TRANSITION Start TO Active  COHHAND 1 PARA ENGE  #

  STATE Active  INDICATOR 1  PHYSICAL Active  PARA IFIE     #
     VARIABLES  Active
     TRANSITION Stop TO Idle  COHHAND 3 PARA DENG  #

  UNCONTROLLABLES


  INSTANCES
        SISO-Pump_P6 gPROHS Plant.CIP_Unit.P6 COHPNAHE C_SISO_Pump PARANAHE P6 PARANUH 350

END # Component SISO-Pump


COHPONENT Valve

 STATE Open  INDICATOR 1  PHYSICAL Open  PARA IFIE     #
    VARIABLES  Open
    TRANSITION Close TO Closed  COHHAND 3 PARA DENG  #

STATE Closed  INDICATOR 0  PHYSICAL Closed  PARA IFINE     #
    VARIABLES  Closed
    TRANSITION Open TO Open  COHHAND 1 PARA ENGE  #

UNCONTROLLABLES


INSTANCES
        Valve_AV1-10 gPROHS Plant.CIP_Unit.AV1_10 COHPNAHE Digital_Valve PARANAHE AV1-10 PARANUH 300
        Valve_AV1-14 gPROHS Plant.CIP_Unit.AV1_14 COHPNAHE Digital_Valve PARANAHE AV1-14 PARANUH 302
        Valve_AV1-15 gPROHS Plant.CIP_Unit.AV1_15 COHPNAHE Digital_Valve PARANAHE AV1-15 PARANUH 304
        Valve_AV1-25 gPROHS Plant.CIP_Unit.AV1_25 COHPNAHE Digital_Valve PARANAHE AV1-25 PARANUH 306
        Valve_AV1-24 gPROHS Plant.Unit_P5_7.AV1_24 COHPNAHE Digital_Valve PARANAHE AV1-24 PARANUH 308
        Valve_SSV1-1 gPROHS Plant.R_T4_5.SSV1_1 COHPNAHE Digital_Valve PARANAHE SSV1-1 PARANUH 310
        Valve_SSV1-2 gPROHS Plant.R_T3.SSV1_2 COHPNAHE Digital_Valve PARANAHE SSV1-2 PARANUH 312
```

```
        Valve_SSV1-3 gPROHS Plant.R_T1_2.SSV1_3 COHPNAHE Digital_Valve PARANAHE SSV1-3 PARANUH 314
        Valve_SSV1-4 gPROHS Plant.R_T1_2.SSV1_4 COHPNAHE Digital_Valve PARANAHE SSV1-4 PARANUH 316
        Valve_SSV1-5 gPROHS Plant.R_T4_5.SSV1_1 COHPNAHE Digital_Valve PARANAHE SSV1-5 PARANUH 318
        Valve_ABV1-3 gPROHS Plant.ABV1_3 COHPNAHE Digital_Valve PARANAHE ABV1-3 PARANUH 320
        Valve_AV1-40 gPROHS Plant.R_T1_2.AV1_40 COHPNAHE Digital_Valve PARANAHE AV1-40 PARANUH 322
        Valve_AV1-41 gPROHS Plant.R_T1_2.AV1_41 COHPNAHE Digital_Valve PARANAHE AV1-41 PARANUH 324
        Valve_AV1-47 gPROHS Plant.AV1_47 COHPNAHE Digital_Valve PARANAHE AV1-47 PARANUH 326

END # Component Valve

CONTROLLER Det-Drain

DESCRIPTION
        The Det-Drain procedure drains Tank T1. The general specification is to set the valves and then empty T1 using P1 until T1
is less than 1 l. The sequence then stops P2 and closes the valves in the route.
  CONTNUH 526
  INSTANCES
    Level-Sensor_IT1-1, CPump_P1, Pump-Control_IC1-1, Valve_AV1-41, Valve_AV1-40, Valve_AV1-47, DSValve_DDV1-1, DSValve_DDV1-3,
    DSValve_DDV1-6, DSValve_DDV1-8, Valve_AV1-24, InvValve_AV1-22, Valve_AV1-25, InvValve_AV1-16, Flag-Sensor_F1
  # initial states of machine to be controlled
  FSH initial_state

   STATE Initial2
    VARIABLES Low , Idle , Disabled , Closed , Closed , Closed , Crossed , Crossed , Crossed , Crossed , Closed , Open ,
              Closed , Closed , Clear
   STATE Initial1
    VARIABLES Okay , Idle , Disabled , Closed , Closed , Closed , Crossed , Crossed , Crossed , Crossed , Closed , Open ,
              Closed , Closed , Clear
    STATE Initial
    VARIABLES High , Idle , Disabled , Closed , Closed , Closed , Crossed , Crossed , Crossed , Crossed , Closed , Open ,
            Closed , Closed , Clear
  END
  # marked states of machine to be controlled
  FSH marked_states

   STATE Goal       HARKED
    VARIABLES Low , Idle , Disabled , Closed , Closed , Closed , 88 , 88 , 88 , 88 , 88 , Open , Closed , 88 , Set

  END
  # static constraints of machine to be controlled
  FSH static_constraints

   STATE AV116      SPEC
    VARIABLES 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , Open , 88

   STATE DDV13      SPEC
    VARIABLES 88 , 88 , 88 , 88 , 88 , 88 , 88 , Uncrossed , 88 , 88 , 88 , 88 , 88 , 88 , 88

   STATE AV124      SPEC
    VARIABLES 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , Open , 88 , 88 , 88 , 88

   STATE DDV11      SPEC
    VARIABLES 88 , 88 , 88 , 88 , 88 , 88 , Uncrossed , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88

   STATE DDV16      SPEC
    VARIABLES 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , Uncrossed , 88 , 88 , 88 , 88 , 88 , 88

  END
  # List of dynamic_constraints
  FSH dynamic_constraints

   STATE LowStart     SPEC
    VARIABLES Low , Idle , Disabled , Closed , Closed , Closed , 88 , 88 , 88 , Crossed , Closed , Open , Closed , Closed , Clear
     TRANSITION NValve_AV1-47_Open
     TRANSITION NPump-Control_IC1-1_Enable
     TRANSITION NValve_AV1-24_Open
     TRANSITION NDSValve_DDV1-8_Uncross
     TRANSITION NValve_AV1-25_Open
     TRANSITION NInvValve_AV1-16_Open
     TRANSITION NCPump_P1_Start
     TRANSITION NValve_AV1-40_Open
     TRANSITION NValve_AV1-41_Open
     TRANSITION NInvValve_AV1-22_Close

   STATE DontOpenValves     SPEC
    VARIABLES 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , Set
```

```
  TRANSITION NValve_AV1-47_Open
  TRANSITION NValve_AV1-40_Open
  TRANSITION NValve_AV1-41_Open
  TRANSITION NDSValve_DDV1-8_Uncross
  TRANSITION NInvValve_AV1-22_Close
  TRANSITION NValve_AV1-25_Open

STATE DontStopDraining1    SPEC
 VARIABLES Okay , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88
  TRANSITION NCPump_P1_Stop
  TRANSITION NPump-Control_IC1-1_Disable

STATE CloseValves    SPEC
 VARIABLES Low , Idle , Disabled , Open , Open , Open , 88 , 88 , 88 , 88 , 88 , Closed , Open , 88 , Set
  TRANSITION Valve_AV1-40_Close
  TRANSITION Valve_AV1-41_Close
  TRANSITION Valve_AV1-47_Close
  TRANSITION DSValve_DDV1-8_Cross
  TRANSITION Valve_AV1-25_Close
  TRANSITION InvValve_AV1-22_Open

STATE StartDraining    SPEC
 VARIABLES High , Idle , Disabled , Closed , Closed , Closed , 88 , 88 , 88 , 88 , 88 , Open , Closed , 88 , Clear
  TRANSITION Valve_AV1-41_Open
  TRANSITION Valve_AV1-40_Open
  TRANSITION Valve_AV1-47_Open
  TRANSITION DSValve_DDV1-8_Uncross
  TRANSITION InvValve_AV1-22_Close
  TRANSITION Valve_AV1-25_Open

STATE DontStopDraining    SPEC
 VARIABLES High , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88
  TRANSITION NPump-Control_IC1-1_Disable
  TRANSITION NCPump_P1_Stop

STATE ValvesClosed    SPEC
 VARIABLES 88 , 88 , 88 , Closed , Closed , Closed , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88
  TRANSITION NPump-Control_IC1-1_Enable
  TRANSITION NCPump_P1_Start

STATE StartDraining2    SPEC
 VARIABLES Okay , Idle , Disabled , Closed , Closed , Closed , 88 , 88 , 88 , 88 , 88 , Open , Closed , 88 , Clear
  TRANSITION Valve_AV1-41_Open
  TRANSITION Valve_AV1-40_Open
  TRANSITION Valve_AV1-47_Open
  TRANSITION DSValve_DDV1-8_Uncross
  TRANSITION InvValve_AV1-22_Close
  TRANSITION Valve_AV1-25_Open

STATE DontCloseValvesOkay    SPEC
 VARIABLES Okay , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88
  TRANSITION NValve_AV1-41_Close
  TRANSITION NValve_AV1-40_Close
  TRANSITION NValve_AV1-47_Close
  TRANSITION NDSValve_DDV1-8_Cross
  TRANSITION NValve_AV1-25_Close
  TRANSITION NInvValve_AV1-22_Open

STATE DontCloseValvesPump    SPEC
 VARIABLES 88 , Active , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88
  TRANSITION NValve_AV1-47_Close
  TRANSITION NValve_AV1-40_Close
  TRANSITION NValve_AV1-41_Close
  TRANSITION NValve_AV1-25_Close
  TRANSITION NInvValve_AV1-22_Open
  TRANSITION NDSValve_DDV1-8_Cross

STATE NoFlagClear
 VARIABLES 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , Set
  TRANSITION NFlag-Sensor_F1_Clear

STATE DontCloseValves    SPEC
 VARIABLES 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , Clear
  TRANSITION NValve_AV1-41_Close
  TRANSITION NValve_AV1-40_Close
  TRANSITION NValve_AV1-47_Close
  TRANSITION NValve_AV1-25_Close
  TRANSITION NDSValve_DDV1-8_Cross
```

```
    TRANSITION NInvValve_AV1-22_Open

  STATE DontReStart1     SPEC
   VARIABLES Okay , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88
    TRANSITION NCPump_P1_Start
    TRANSITION NPump-Control_IC1-1_Enable

  STATE StopDraining     SPEC
   VARIABLES Low , Active , Enabled , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88
    TRANSITION CPump_P1_Stop
    TRANSITION Pump-Control_IC1-1_Disable

   STATE ReStartDraining     SPEC
    VARIABLES High , Idle , Disabled , Open , Open , Open , 88 , 88 , 88 , 88 , 88 , Closed , Open , 88 , 88
     TRANSITION Pump-Control_IC1-1_Enable
     TRANSITION CPump_P1_Start

   STATE DontReStart     SPEC
    VARIABLES Low , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88 , 88
     TRANSITION NPump-Control_IC1-1_Enable
     TRANSITION NCPump_P1_Start

  END

END # CONTROLLER Det-Drain
```

Appendix B

Sample *gPROMS* File

```
****************************************
In the HODEL Complete_Batch_Plant,
Within The UNIT Section Place where it says "Place Procedure Declarations Here":
        ProcCIP-Fill     AS       Procedure

In the PROCESS Section of your gPROHS place these lines of code:

  Within the ASSIGN Section inside the WITHIN PLANT, where it says "Place Procedure variable Assignments Here"    :
        ProcCIP-Fill.command_Action  := 0;
        ProcCIP-Fill.external_command  := 0;
        ProcCIP-Fill.Indicator0  := 0;

  Within the SCHEDULE, where it says "Place code to call controllers here" :
        Procedure_OBSERVER(PLANT IS PLANT, Procedure IS PLANT.ProcCIP-Fill);
        CIP-Fill(PLANT IS PLANT, Procedure IS PLANT.ProcCIP-Fill);

 To run this procedural controller and any below it.
 Place this command in the SCHEDULE where it says "Place Code to initiate the main controller here" :
         SEQUENCE
           CONTINUE FOR 1
           RESET
             Plant.ProcCIP-Fill.command_Action := 1.0;
           END # RESET
           WHILE (PLANT.Endd <> 1) DO
             SEQUENCE
               IF ((Plant.ProcCIP-Fill.Indicator0 = 2) OR (TIHE > PLANT.Simulation_Time)) THEN
                 RESET PLANT.Endd := 1; END
               END # IF
               CONTINUE FOR Plant.Sample_Time
             END # SEQUENCE
           END # WHILE
         END # SEQUENCE
****************************************


Place the following tasks between the last HODEL and the PROCESS section where it says "PLACE CONTROLLER TASKS HERE"

TASK CIP-Fill

#
#
#----------- SYSTEH STATS ---------------
#   0 lines of inhibit code.
#   53 lines of precheck code.
#   137 lines of logic.
#   1 lines of comments.
#---------------------------------------
#
  PARAHETER
    PLANT        AS       HODEL Complete_Batch_Plant
    Procedure    AS       HODEL Procedure

  VARIABLE
    state        AS       INTEGER
    marked       AS       INTEGER
    endd         AS       INTEGER
```

```
SCHEDULE
  SEQUENCE

    state := 0;
    marked := 0;
    endd := 0;
    WHILE (PLANT.Endd <> 1) AND (TIHE < PLANT.Simulation_Time) AND (marked = 0) AND (endd <> 1) DO
      SEQUENCE
       IF (Procedure.Indicator0 = 1) THEN
        PARALLEL
# Prechecks as for whether Controller can start
IF (state = 0) THEN
  SEQUENCE
    IF (Plant.CIP_Unit.T7.LS1.Indicator0 = 1) AND (Plant.CIP_Unit.T7.LS2.Indicator0 = 1) AND
       (Plant.CIP_Unit.P6.Indicator0 = 0) AND (Plant.CIP_Unit.HE3.IC1_8.Indicator0 = 0) AND
       (Plant.CIP_Unit.AV1_10.Indicator0 = 0) AND (Plant.CIP_Unit.AV1_14.Indicator0 = 0) AND
       (Plant.CIP_Unit.AV1_15.Indicator0 = 0) AND (Plant.CIP_Unit.AV1_16.Indicator0 = 1) AND
       (Plant.CIP_Unit.AV1_22.Indicator0 = 1) AND (Plant.CIP_Unit.AV1_25.Indicator0 = 0) AND
       (Plant.Unit_P5_7.AV1_24.Indicator0 = 0) AND (Plant.CIP_Unit.P10.Indicator0 = 0)  THEN
    state := 1;
    END # If
    IF (Plant.CIP_Unit.T7.LS1.Indicator0 = 0) AND (Plant.CIP_Unit.T7.LS2.Indicator0 = 0) AND
       (Plant.CIP_Unit.P6.Indicator0 = 0) AND (Plant.CIP_Unit.HE3.IC1_8.Indicator0 = 0) AND
       (Plant.CIP_Unit.AV1_10.Indicator0 = 0) AND (Plant.CIP_Unit.AV1_14.Indicator0 = 0) AND
       (Plant.CIP_Unit.AV1_15.Indicator0 = 0) AND (Plant.CIP_Unit.AV1_16.Indicator0 = 1) AND
       (Plant.CIP_Unit.AV1_22.Indicator0 = 1) AND (Plant.CIP_Unit.AV1_25.Indicator0 = 0) AND
       (Plant.Unit_P5_7.AV1_24.Indicator0 = 0) AND (Plant.CIP_Unit.P10.Indicator0 = 0)  THEN
    state := 11;
    END # If
    IF (Plant.CIP_Unit.T7.LS1.Indicator0 = 0) AND (Plant.CIP_Unit.T7.LS2.Indicator0 = 1) AND
       (Plant.CIP_Unit.P6.Indicator0 = 0) AND (Plant.CIP_Unit.HE3.IC1_8.Indicator0 = 0) AND
       (Plant.CIP_Unit.AV1_10.Indicator0 = 0) AND (Plant.CIP_Unit.AV1_14.Indicator0 = 0) AND
       (Plant.CIP_Unit.AV1_15.Indicator0 = 0) AND (Plant.CIP_Unit.AV1_16.Indicator0 = 1) AND
       (Plant.CIP_Unit.AV1_22.Indicator0 = 1) AND (Plant.CIP_Unit.AV1_25.Indicator0 = 0) AND
       (Plant.Unit_P5_7.AV1_24.Indicator0 = 0) AND (Plant.CIP_Unit.P10.Indicator0 = 0)  THEN
    state := 2;
    END # If
    IF (state = 0) THEN
      SEQUENCE
        MESSAGE "Controller CIP-Fillis NOT in AN ALLOWABLE INITIAL STATE!!"
        RESET
          Procedure.external_command := 3;
        END #RESET
      END # SEQUENCE
    END # IF
  END # SEQUENCE
END #IF

IF (state = 10) THEN
  SEQUENCE
    Digital_Valve_Close(Digital_Valve IS Plant.CIP_Unit.AV1_25);
    state := 1;
    MESSAGE "TRANSITION Close OF Valve_AV1-25 EXECUTED"
  END # Sequence
END # IF

IF (state = 12) THEN
  SEQUENCE
    Digital_Valve_Open(Digital_Valve IS Plant.CIP_Unit.AV1_25);
    state := 13;
    MESSAGE "TRANSITION Open OF Valve_AV1-25 EXECUTED"
  END # Sequence
END # IF

IF (state = 13) THEN
  SEQUENCE
    Digital_Valve_Open(Digital_Valve IS Plant.CIP_Unit.AV1_10);
    state := 14;
    MESSAGE "TRANSITION Open OF Valve_AV1-10 EXECUTED"
  END # Sequence
END # IF

IF (state = 14) THEN
  SEQUENCE
    C_SISO_Pump_Start(C_SISO_Pump IS Plant.CIP_Unit.P6);
    state := 15;
    MESSAGE "TRANSITION Start OF SISO-Pump_P6 EXECUTED"
```

```
  END # Sequence
END # IF

IF (state = 15) THEN
  SEQUENCE
    IF (Plant.CIP_Unit.T7.LS2.Indicator0 = 1) THEN
      # InvLevel-Switch_IS1-2 IS Covered
      SEQUENCE
        state := 6;
        MESSAGE "TRANSITION Cover OF InvLevel-Switch_IS1-2 OCCURRED"
      END # SEQUENCE
    END # IF
  END # Sequence
END # IF

IF (state = 3) THEN
  SEQUENCE
    Digital_Valve_Open(Digital_Valve IS Plant.CIP_Unit.AV1_25);
    state := 4;
    MESSAGE "TRANSITION Open OF Valve_AV1-25 EXECUTED"
  END # Sequence
END # IF

IF (state = 4) THEN
  SEQUENCE
    Digital_Valve_Open(Digital_Valve IS Plant.CIP_Unit.AV1_10);
    state := 5;
    MESSAGE "TRANSITION Open OF Valve_AV1-10 EXECUTED"
  END # Sequence
END # IF

IF (state = 5) THEN
  SEQUENCE
    C_SISO_Pump_Start(C_SISO_Pump IS Plant.CIP_Unit.P6);
    state := 6;
    MESSAGE "TRANSITION Start OF SISO-Pump_P6 EXECUTED"
  END # Sequence
END # IF

IF (state = 6) THEN
  SEQUENCE
    IF (Plant.CIP_Unit.T7.LS1.Indicator0 = 1) THEN
      # Level-Switch_IS1-1 IS Covered
      SEQUENCE
        state := 7;
        MESSAGE "TRANSITION Cover OF Level-Switch_IS1-1 OCCURRED"
      END # SEQUENCE
    END # IF
  END # Sequence
END # IF

IF (state = 7) THEN
  SEQUENCE
    C_SISO_Pump_Stop(C_SISO_Pump IS Plant.CIP_Unit.P6);
    state := 8;
    MESSAGE "TRANSITION Stop OF SISO-Pump_P6 EXECUTED"
  END # Sequence
END # IF

IF (state = 8) THEN
  SEQUENCE
    Digital_Valve_Close(Digital_Valve IS Plant.CIP_Unit.AV1_10);
    state := 9;
    MESSAGE "TRANSITION Close OF Valve_AV1-10 EXECUTED"
  END # Sequence
END # IF

IF (state = 9) THEN
  SEQUENCE
    Digital_Valve_Open(Digital_Valve IS Plant.CIP_Unit.AV1_22);
    state := 10;
    MESSAGE "TRANSITION Open OF InvValve_AV1-22 EXECUTED"
  END # Sequence
END # IF

IF (state = 1) THEN
  SEQUENCE
  END # Sequence
```

```
END # IF

IF (state = 11) THEN
  SEQUENCE
    Digital_Valve_Close(Digital_Valve IS Plant.CIP_Unit.AV1_22);
    state := 12;
    HESSAGE "TRANSITION Close OF InvValve_AV1-22 EXECUTED"
  END # Sequence
END # IF

IF (state = 2) THEN
                SEQUENCE
                  Digital_Valve_Close(Digital_Valve IS Plant.CIP_Unit.AV1_22);
                  state := 3;
                  HESSAGE "TRANSITION Close OF InvValve_AV1-22 EXECUTED"
                END # Sequence
              END # IF

               END # PARALLEL
              ELSE
                state := 0;
              END # IF
              CONTINUE FOR PLANT.SAHPLE_TIHE
            END # SEQUENCE
          END # WHILE
        END # SEQUENCE
      END # TASK CIP-Fill
```

Appendix C

Sample PARACODE Source File

```
.THIS_SEQ_CONT 2
/
.SEQ 2.536
/-------------------------------------------------------
/
/
/ Imperial College of Science, Technology and Hedicine
/ Centre for Process Systems Engineering
/ LONDON SW7 2BY, UK.
/
/ Procedural Controller : CIP-Fill
/ Date Generated : Wed Sep 15 16:29:37 GHT+01:00 1999
/
/ This controller executes the filling of the CIP tank from the
/ feed water supply. The basic execution is to Close AV1-22 open
/ Av1-25 Open AV1-10 and then start P6. It then waits for both
/ level sensors to read covered and then stops the system in the
/ reverse oreder to start up.
/
/
/---------- SYSTEH STATS ---------------
/   0 lines of inhibit code.
/   85 lines of precheck code.
/   74 lines of logic.
/   45 lines of comments.
/---------------------------------------
/
/ PRECHECKS
/
S536A1  HOVN    1, R2.537       / SET SEQUENCE FLAG REGISTER TO 1 FOR RUNNING
/
        HOVN    -1, R2.536
        HOVN    1, R2.536
        IFN     R2.363, EQ,1, S536A2    / CHECK Level-Switch_IS1-1 IS Covered
        HOVN    2, R2.536
        IFN     R2.364, EQ,1, S536A2    / CHECK InvLevel-Switch_IS1-2 IS Covered
        HOVN    3, R2.536
        IFN     R2.350, EQ,0, S536A2    / CHECK SISO-Pump_P6 IS Idle
        HOVN    4, R2.536
        IFN     R2.374, EQ,0, S536A2    / CHECK Temp-Control_IC1-8 IS Disabled
        HOVN    5, R2.536
        IFN     R2.300, EQ,0, S536A2    / CHECK Valve_AV1-10 IS Closed
        HOVN    6, R2.536
        IFN     R2.302, EQ,0, S536A2    / CHECK Valve_AV1-14 IS Closed
        HOVN    7, R2.536
        IFN     R2.304, EQ,0, S536A2    / CHECK Valve_AV1-15 IS Closed
        HOVN    8, R2.536
        IFN     R2.328, EQ,1, S536A2    / CHECK InvValve_AV1-16 IS Open
        HOVN    9, R2.536
        IFN     R2.330, EQ,1, S536A2    / CHECK InvValve_AV1-22 IS Open
        HOVN    10, R2.536
        IFN     R2.306, EQ,0, S536A2    / CHECK Valve_AV1-25 IS Closed
        HOVN    11, R2.536
        IFN     R2.308, EQ,0, S536A2    / CHECK Valve_AV1-24 IS Closed
        HOVN    12, R2.536
        IFN     R2.354, EQ,0, S536A2    / CHECK Acid-Base-Pump_P10 IS Idle
```

```
        GOTO    S536S1
S536A2  HOVN    -1, R2.536
        HOVN    1, R2.536
        IFN     R2.363, EQ,0, S536A3    / CHECK Level-Switch_IS1-1 IS Uncovered
        HOVN    2, R2.536
        IFN     R2.364, EQ,1, S536A3    / CHECK InvLevel-Switch_IS1-2 IS Covered
        HOVN    3, R2.536
        IFN     R2.350, EQ,0, S536A3    / CHECK SISO-Pump_P6 IS Idle
        HOVN    4, R2.536
        IFN     R2.374, EQ,0, S536A3    / CHECK Temp-Control_IC1-8 IS Disabled
        HOVN    5, R2.536
        IFN     R2.300, EQ,0, S536A3    / CHECK Valve_AV1-10 IS Closed
        HOVN    6, R2.536
        IFN     R2.302, EQ,0, S536A3    / CHECK Valve_AV1-14 IS Closed
        HOVN    7, R2.536
        IFN     R2.304, EQ,0, S536A3    / CHECK Valve_AV1-15 IS Closed
        HOVN    8, R2.536
        IFN     R2.328, EQ,1, S536A3    / CHECK InvValve_AV1-16 IS Open
        HOVN    9, R2.536
        IFN     R2.330, EQ,1, S536A3    / CHECK InvValve_AV1-22 IS Open
        HOVN    10, R2.536
        IFN     R2.306, EQ,0, S536A3    / CHECK Valve_AV1-25 IS Closed
        HOVN    11, R2.536
        IFN     R2.308, EQ,0, S536A3    / CHECK Valve_AV1-24 IS Closed
        HOVN    12, R2.536
        IFN     R2.354, EQ,0, S536A3    / CHECK Acid-Base-Pump_P10 IS Idle
        GOTO    S536S2
S536A3  HOVN    -1, R2.536
        HOVN    1, R2.536
        IFN     R2.363, EQ,0, S536A4    / CHECK Level-Switch_IS1-1 IS Uncovered
        HOVN    2, R2.536
        IFN     R2.364, EQ,0, S536A4    / CHECK InvLevel-Switch_IS1-2 IS Uncovered
        HOVN    3, R2.536
        IFN     R2.350, EQ,0, S536A4    / CHECK SISO-Pump_P6 IS Idle
        HOVN    4, R2.536
        IFN     R2.374, EQ,0, S536A4    / CHECK Temp-Control_IC1-8 IS Disabled
        HOVN    5, R2.536
        IFN     R2.300, EQ,0, S536A4    / CHECK Valve_AV1-10 IS Closed
        HOVN    6, R2.536
        IFN     R2.302, EQ,0, S536A4    / CHECK Valve_AV1-14 IS Closed
        HOVN    7, R2.536
        IFN     R2.304, EQ,0, S536A4    / CHECK Valve_AV1-15 IS Closed
        HOVN    8, R2.536
        IFN     R2.328, EQ,1, S536A4    / CHECK InvValve_AV1-16 IS Open
        HOVN    9, R2.536
        IFN     R2.330, EQ,1, S536A4    / CHECK InvValve_AV1-22 IS Open
        HOVN    10, R2.536
        IFN     R2.306, EQ,0, S536A4    / CHECK Valve_AV1-25 IS Closed
        HOVN    11, R2.536
        IFN     R2.308, EQ,0, S536A4    / CHECK Valve_AV1-24 IS Closed
        HOVN    12, R2.536
        IFN     R2.354, EQ,0, S536A4    / CHECK Acid-Base-Pump_P10 IS Idle
        GOTO    S536S11
/
S536A4  HESS    1.200, 1.1
        VARN    2
        VARN    536
        VARR    R2.536
        VARR    R2.536
        ALSQ    2.536
        GOTO    S536A1
/
/
/ BEGIN OPERATION
/
S536S1  HOVN    1, R2.536               / SET STATE VARIABLE
        GOTO    S536S                   / STEP TO STATE
/
S536S2  HOVN    2, R2.536               / SET STATE VARIABLE
        ENGE    AV1-22                  / Close InvValve_AV1-22
S536SB2 WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S3                  / STEP TO STATE 3
/
S536S3  HOVN    3, R2.536               / SET STATE VARIABLE
        ENGE    AV1-25                  / Open Valve_AV1-25
S536SB3 WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
```

```
        GOTO    S536S4                  / STEP TO STATE 4
/
S536S4  HOVN    4, R2.536               / SET STATE VARIABLE
        ENGE    AV1-10                  / Open Valve_AV1-10
S536SB4 WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S5                  / STEP TO STATE 5
/
S536S5  HOVN    5, R2.536               / SET STATE VARIABLE
        ENGE    P6                      / Start SISO-Pump_P6
S536SB5 WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S6                  / STEP TO STATE 6
/
S536S6  HOVN    6, R2.536               / SET STATE VARIABLE
S536T6  IF      R2.537, EQ, 0, S536A1   / SET STATE VARIABLE
        IF      R2.363, EQ, 1, S536S7   / Level-Switch_IS1-1 Cover
        GOTO    S536T6                  / LOOP IN STATE 6
/
S536S7  HOVN    7, R2.536               / SET STATE VARIABLE
        DENG    P6                      / Stop SISO-Pump_P6
S536SB7 WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S8                  / STEP TO STATE 8
/
S536S8  HOVN    8, R2.536               / SET STATE VARIABLE
        DENG    AV1-10                  / Close Valve_AV1-10
S536SB8 WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S9                  / STEP TO STATE 9
/
S536S9  HOVN    9, R2.536               / SET STATE VARIABLE
        DENG    AV1-22                  / Open InvValve_AV1-22
S536SB9 WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S10                 / STEP TO STATE 10
/
S536S10 HOVN    10, R2.536              / SET STATE VARIABLE
        DENG    AV1-25                  / Close Valve_AV1-25
S536SB10        WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S1                  / STEP TO STATE 1
/
S536S11 HOVN    11, R2.536              / SET STATE VARIABLE
        ENGE    AV1-22                  / Close InvValve_AV1-22
S536SB11        WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S12                 / STEP TO STATE 12
/
S536S12 HOVN    12, R2.536              / SET STATE VARIABLE
        ENGE    AV1-25                  / Open Valve_AV1-25
S536SB12        WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S13                 / STEP TO STATE 13
/
S536S13 HOVN    13, R2.536              / SET STATE VARIABLE
        ENGE    AV1-10                  / Open Valve_AV1-10
S536SB13        WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S14                 / STEP TO STATE 14
/
S536S14 HOVN    14, R2.536              / SET STATE VARIABLE
        ENGE    P6                      / Start SISO-Pump_P6
S536SB14        WAIT    5                       / WAIT FOR GRACE TIHE
        IF      R2.537, EQ, 0, S536A1   / IF SEQUENCE RESTARTED GO BACK TO BEGINNING
        GOTO    S536S15                 / STEP TO STATE 15
/
S536S15 HOVN    15, R2.536              / SET STATE VARIABLE
S536T15 IF      R2.537, EQ, 0, S536A1   / SET STATE VARIABLE
        IF      R2.364, EQ, 1, S536S6   / InvLevel-Switch_IS1-2 Cover
        GOTO    S536T15                 / LOOP IN STATE 15
/
/
S536Z   ABEY                    / END OF SEQUENCE 2.536
        STOP
```

References

Alsop, N., L. Sulfaroli Camilloci, A. Sanchez and S. Macchietto (1996). Synthesis of Procedural Controllers - application to a Batch Pilot. *Computers in Chemical Engineering* **20**, S1481–S1486.

Alsop, Nicholas (1996). Formal Techniques for the Procedural Control of Industrial Processes. PhD thesis. University of London.

APV (1994). *ACCOS 30 Training Manual.* Crawley, U.K. Issue 1.

Årzén, Karl-Erik (1994). Grafcet for Intelligent Supervisory Control Applications. *Automatica* **30**(10), 1513–1525.

Baird, Ross (1997). Progress Report: An Investigation of Decomposition Methods in the Development of a Procedural Controller for an Automatic Drinks Service. *Centre for Process Systems Engineering, Imperial College.*

Bates, I.D., E.G. Chester and D.J. Kinniment (1998). Automatic Generation of IEC 1131 PLC Code from Statemate. In: *International Workshop on discrete Event Systems.* pp. 167–172.

Brandin, B.A. (1998). The modelling and Supervisory Control of Timed DES. In: *International Workshop on discrete Event Systems.* pp. 8–14.

Brandin, B.A. and W.M. Wonham (1994). Supervisory Control of Timed Discrete-Event Systems. *IEEE Transactions on Automatic Control* **39**(2), 329–341.

Brave, Y. and M. Heymann (1993). Control of Discrete Event Systems Modelled as Hierarchical State Machines. *IEEE Transactions on Automatic Control* **38**(12), 1803–1819.

Bridges, W. G. and T. R. Williams (1997). Create Effective Safety Procedures and Operating Manuals. *Chemical Engineering Progress* **93**(12), 23–37.

Burch, J.R., E.M. Clarke, K.L. McMillan, D.L. Dill and J. Hwang (1992). Symbolic Model Checking: 10^{20} and beyond. *Information and Computation* **98**(2), 142–170.

Camillocci, Luca Solfaroli (1995). Synthesis of Procedural Controllers for Chemical Processes. Master's thesis. Centre for Process Systems Engineering. Imperial College. London SW7 2BY.

Chung, P.W.H., E. Bloomfield and S.H. Yang (1998). Safety related questions for computer-controlled plant: derivation, organisation and application. *Journal of Loss Prevention in the Process Industries* **11**, 397–406.

Clements-Jewery, K. and W. Jeffcoat (1996). *The PLC Workbook*. Prentice-Hall, London.

Dimitriadis, V.D., J. Hackenberg, N. Shah and C.C. Pantelides (1996). A Case Study in Hydred Safety Verification. *Computers and Chemical Engineering* **20**(SA), S 503–S 508.

Duran, O. and A. Batocchio (1999). Automatic PLC software generator with natural interface. *International Journal of Production Research* **37**(4), 805–819.

Edwards, V. H. and D. G. Clark (1999). Protect Highly Hazardous Processes with Critical Incident Recorders. *Chemical Engineering Progress* **97**(7), 75–78.

Elsag-Bailey (1992). *BATCH 90 User Manual*. Canada.

Feldmann, K., C. Schnur and W. Colombo (1996). Modularised, Distributed Real-Time Control of Flexible Production Cells Using Petri Nets. *Control Engineering Practice* **4**(8), 1067–1078.

Gabrielian, A. and R. Iyer (1991). Verifying Properties of HMS Machine Specifications of Real-Time Systems. In: *Computer Aided Verification*. Lecture Notes in Computer Science. Vol. 575. pp. 421–431.

Gabrielian, Armen (1994). Formal specification and Verification of Real-Time Sequential Control Systems. In: *Rutgers' Conference on Computer Intergrated Manufacturing in the Process Industries*. pp. 384–395.

Gabrielian, Armen and Matthew K. Franklin (1991). Multilevel Specification of Real-Time Systems. *Communications of the ACM*.

Gallo, Erica (1996). Synthesis of Procedural Controllers for Industrial Processes : A Case Study. Master's thesis. Centre for Process Systems Engineering. Imperial College, London SW7 2BY.

Gibbs, W. Wayt (1994). Software's Chronic Crisis. *Scientific American* pp. 72–81.

Harel, David (1987). Statecharts: A Visual Formalism for Complex Systems.. *Science of Computer Programming* **8**, 231–274.

Harel, David and Amnon Naamad (1996). The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*.

Hubbard, P. and P.E. Caines (1998). A state aggregation approach to hierarchical supervisory control with applications to a tranfer line example. In: *International Workshop on discrete Event Systems*. pp. 2–7.

Huizing, C. and W.P. de Roever (1991). Introduction to Design Choices in the Semantics of Statecharts. *Information Processing Letters* **37**, 205–213.

Huizing, C., R. Gerth and W.P. de Roever (1988). Modelling Statecharts Behaviour in a Fully Abstract Way. *Proc. Colloq. on Trees in Algebra and Programming* Lecture Notes in Computer Science **299**, 271–294.

ISA (1995). *ISA-S88.01-1995 BatchControl, Part 1: Models and Terminology.*

Klein, Eberhard and Herbert Wehlan (1996). Systematic Design of Protective Controllers in Process Industries by Means of the boolean Differential Calculus. WODES96, Edinburgh, UK.

Kowalewski, S., Y. Lakhnech, B. Lukoschus and L. Urbina (1998). On the Composition of Condition/Event Systems. In: *4th International Workshop on Discrete Event Systems, Cagliari, Italy.*

Kristoffersen, K., K Larsen, P. Pettersson and C. Weise (1999). Experimental Batch Plant: VHS Case Study 1 using UPPAAL. In: *VERIMAG, VHS Study First Year Report.* University of Dortmund.

Lano, K., P. Kan and A. Sanchez (1998). Linking hazard analysis to formal specification and design in B. *Lecture Notes in Computer Science* **1516**, 60–74.

Leduc, Ryan (1996). PLC Implementation of a DES Supervisor for a Manufacturing Testbed: An Implementation Perspective. Master's thesis. Department of Electrical and Computer Engineering, University of Toronto.

Lee, Sang-Heon and Kai C. Wong (1998). Decentralised Control of Concurrent Discrete-Event Systems with Non-Prefix Closed Local Specifications.

Lemetais, Charlotte (1999). More Recipes for Success. *The Chemical Engineer* (678), 22.

Leveson, Nancy G., Mats Per Erik Heimdahl, Holly Hildreth and Jon Damon Reese (1994). Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering* **20**(9), 684–707.

Liu, Zhenhai (1995). An Advanced Process Manufacturing System – Design and Application to a Food Processing Pilot Plant. PhD thesis. University of London.

Luder, A., H-M. Hanisch and M. Rausch (1996). Synthesis of Forcing/Locking Controllers Based on Net Condition/Event Systems. WODES96, Edinburgh, UK.

Maler, O. (1999). On the Programming of Industrial Computers. In: *VERIMAG, VHS Study First Year Report.* University of Dortmund.

Marikar, M. T., G. E. Rotstein, A. Sanchez and S. Macchietto (1998). Computer Aided Analysis and Synthesis of Procedural Controllers. *4th International Workshop on Discrete Event Systems, Cagliari, Italy* pp. 420–425.

Moon, I. and S. Macchietto (1993). Overview of the Model Checking Method. Centre for Process Systems Engineering, Imperial College of Science, Technology and Medicine, London.

Moon, Il (1994). Modeling Programmable Logic Controllers for Logic Verification. *IEEE Control Systems Magazine* **14**(2), 53–59.

Moon, Il, Gary J. Powers, Jerry R. Burch and Edmund M. Clarke (1992). Automatic Verification of Sequential Control Systems using Temporal Logic. *AIChE* **38**(1), 67–75.

Mouille, Philippe (1997). *Synthesis of Stable and Optimal Procedural Controller for Chemical Processes.* Centre for Process Systems Engineering, Imperial College of Science, Technology and Medicine.

Nimmo, Ian (1993). Start Up Plant Safely. *Chemical Engineering Progress* **89**(12), 66–69.

Nimmo, Ian (1994). Extend HAZOP to Computer Control Systems. *Chemical Engineering Progress* **90**(10), 32–44.

Nimmo, Ian (1995). Adequately Address Abnormal Operations. *Chemical Engineering Progress* **91**(9), 36–45.

Ostroff, Jonathan S. (1997). A Visual Toolset for the Design of Real-Time Discrete Event Systems. *IEEE Transactions on Control Systems Technology.*

Palmer, Stephen (1995). *MALPAS Executive Overview.* TA Consultancy Services Limited.

Park, Taeshin and Paul I. Barton (1996). Binary Logic Diagrams Imply Nonunique Functionality. *ISA Transactions* **35**, 337–343.

Park, Taeshin and Paul I. Barton (1997). Implicit Model Checking of Logic-based Systems. *AIChE Journal* **43**(9), 2246–2260.

Parra-Mora, L. F. (1998). *Control System Design for a Milk Pasteurisation Plant II: Unit Operation Raw Milk Intake with Failures.* Centre for Process Systems Engineering, Imperial College of Science, Technology and Medicine.

Pinzon, L. and H-M. Hanisch (1998). Sequential Specification Modelling with temporal Logic and Net Condition/Event Systems. In: *4th International Workshop on Discrete Event Systems, Cagliari, Italy.*

Pinzon, L., H-M. Hanisch, M. Jafari and T. Boucher (1997). A Comparative Study of Synthesis Methods for Discrete Event Controllers. Technical report. Centre for Operations Research, Rutgers University.

Pnueli, A. and M. Shalev (1991). What is in a Step: On the Semantics of Statecharts. In: *Proceedings of the Symposium on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science. Vol. 526. pp. 244–264.

Presto Project P4 (1996). Final Report – Integrated Design of Control and Automation Systems for Flexible Batch Plants. Technical report. Centre for Process Systems Engineering. (confidential).

Probst, Scott T. and Gary J. Powers (1994). Automatic Verification of Control Logic in the Presence of Process Faults. In: *AIChE Annual Meeting on Process Operations and Fault Tolerant Control.*

Probst, S.T., G.J. Powers, D.E. Long and I.Moon (1997). Verification of a Logically Controlled Solids Transport Using Symbolic Model Checking. *Computers and Chemical Engineering* **21**(4), 417.

Process Systems Enterprise Ltd (1997). gPROMS *Advanced User Guide*. London, United Kingdom.

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

Rausch, M., A. Luder and H-M. Hanisch (1996). Combined synthesis of Locking and Sequential Controllers. WODES96, Edinburgh, UK.

Rausch, M. and B.H. Krogh (1998). Symbolic Verification of Stateflow Logic. In: *4th International Workshop on Discrete Event Systems, Cagliari, Italy*. pp. 489–494.

Rotstein, G. E. and S. Macchietto (1997). Synthesis of Stable and Optimal Procedural Controllers. Workshop on Analysis and Design of Event-Driven Operations, London.

Rotstein, G.E., A. Sanchez, N. Alsop, L. Sulfaroli Camilloci and S. Macchietto (1995). Synthesis of Procedural Controllers and the Automatic Generation of Sequential Control Code. Workshop on Analysis and Design of Event-Driven Operations, London.

Sakao, T., Y. Umeda, Y. Tomiyama and Y. Shimomura (1997). Model-based automatic generation of sequence-control programs from design information. *IEEE Expert – Intelligent Systems & their Application* **12**(3), 54–61.

Sanchez, A., C. Gollain and S. Macchietto (1996). Rapid Development of Structured Simulation Models for the Verification of Discrete-Event Systems. In: *IFAC DYCOPS-5. Corfu, Greece, June (8-10).*

Sanchez, A., G. Rotstein, N. Alsop and S. Macchietto (1999). Synthesis and Implementation of Procedural Controllers for Event-Driven Operations. *AIChE* **45**(8), 1753-1775.

Sanchez, Arturo (1994). Synthesis of Sequential Logic Controllers for Process Systems. PhD thesis. University of London.

Sanchez, Arturo (1996). *Formal Specification and Synthesis of Procedural Controllers for Process Systems.* Lecture Notes in Control and Information Sciences 212, Springer.

Schubach, S. (1997). A modified computer hazard and operability study procedure. *Journal of Loss Prevention in the Process Industries* **10**(5–6), 303–307.

Sreenivas, R. and B. Krogh (1991). On Condition/Event Systems with Discrete State Realizations. *Discrete Event Dynamic Systems: Theory and Applications* **2**(1), 209–236.

Tinham, B. (1996). OPEN CONTROL will IEC1131 provide an answer?. *Control and Instrumention* **28**(10), 31–32.

University of Bremen (1998). daVinci. Bremen Institute for Safe and Secure Systems, University of Bremen, Germany.

von der Beeck, Michael (1994). A Comparison of Statecharts Variants. In: *Formal Techniques in Real-Time and Fault-Tolerance Systems,* Lecture Notes in Computer Science. Vol. 863. pp. 128–148.

Wong, K. C. and W. M. Wonham (1996*a*). Hierarchical control of Discrete-Event Systems. *Discrete Event dynamic Systems* **6**, 241–273.

Wong, K. C. and W. M. Wonham (1996*b*). Hierarchical Control of Timed Discrete-Event Systems. *Discrete Event dynamic Systems* **6**, 275–306.

Wonham, W.M. (1996). *Notes on Control of Discrete–Event Systems.* Systems Control Group, Dept. of Electrical & Computer Engineering, University of Toronto. ECE 1636F/1637S.

Wren, A. (1998). Heuristics Ancient and Modern: Transport Scheduling Through the Ages. *Journal of Heuristics* **4**(1), 87–100.

Yamalidou, E.C., E.P. Patsidou and J.C. Kantor (1990). Modelling Discrete-Event Dynamical Systems for Chemical Process Control - A Survey of Several New Techniques. *Computers in Chemical Engineering* **14**(3), 281–299.

Yang, S. and P.W.H. Chung (1998). Hazard analysis and support tool for computer controlled processes. *Journal of Loss Prevention in the Process Industries* **11**, 333–345.

Yao, W.L. and X.D He (1997). Mapping Petri nets to concurrent programs in C++. *Information and Software Technology* **39**(7), 485–495.