THE MODELLING AND SIMULATION OF COMBINED DISCRETE/CONTINUOUS PROCESSES

by

Paul Inigo Barton, M.Eng, A.C.G.I.

May 1992

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

Imperial College of Science, Technology and Medicine
London SW7 2BY

To Amy

Abstract

Currently available dynamic simulation packages are mainly suitable for the continuous simulation of large industrial processing systems. In practice, however, few processes can be considered to operate in a completely continuous manner because discrete changes affect most operations to a greater or lesser extent. Even in a conventional 'continuous' process, start-up and shut-down operations, or the application of digital controllers will result in discrete changes superimposed on the predominantly continuous behaviour. Similarly, batch and semi-continuous processes always experience frequent discrete control actions in order to maintain operation in a dynamic, often cyclic, mode. Simulation of systems with these discrete components requires a more sophisticated tool - one that can perform simulations of a combined discrete/continuous nature.

This thesis considers the issues involved in the development of a general-purpose software package for the modelling and simulation of combined discrete/continuous processing systems of arbitrary complexity. The key requirements for such a package are analysed, and a new simulation language based on three distinct categories of entities, *models*, *tasks* and *processes*, is introduced. Model entities describe the continuous physico-chemical mechanisms governing the time dependent behaviour of unit operations, including any discrete changes resulting from these mechanisms, while task entities describe the external control actions or disturbances imposed on a system. A process entity represents a complete dynamic simulation experiment, and is formed by the application of tasks to instances of model entities.

This language is used as the basis for a new dynamic simulation package, gPROMS (**g**eneral **PRO**cess **M**odelling **S**ystem). The implementation of a prototype of this package is described, including details of the novel software architecture required for the simulation of these systems. The usefulness of this new approach, and the ability of gPROMS to address these issues, are then demonstrated through a set of detailed simulation examples generated by the prototype.

Acknowledgements

Preparing a Ph.D thesis takes up quite a chunk of one's life. During that period, there are many people who aid and abet the crime. I am probably most indebted to Amy Blake; without her love and support over the last three and half years, this thesis may never have been written.

I would like to thank my supervisor, Dr. Costas Pantelides, for his considerable intellectual input to this work, Professor Roger Sargent for inciting me to undertake a Ph.D in the first place, and Professor John Perkins for his contributions during the early stages of the project.

The work described in this thesis was partially supported by Air Products PLC and Air Products and Chemicals Inc. in the form of a bursary, which proved particularly useful during the first year. I would like to thank Alan Boyd and David Espie of Air Products for the enthusiasm and interest they have shown throughout the project.

Several people have helped considerably with the nuts and bolts of the project. I would like to thank Liam McLoughlin for the use of his Modula-2 version of Yacc, Richard Jarvis for co-operating on the interface to DASOLV and his patient advice concerning LaTeX, and Edward Smith and Rüdiger von Watzdorf for their work on some of the examples presented in chapter 6.

I must, of course, thank all my other friends and colleagues in the Centre for Process Systems Engineering as well. In particular, I would like to mention Felix Gross, Nilay Shah, Colin Crooks, Simeon Kassianides, Yishu Nanda, Shaun Dore, Larry Naraway, and Steve Walsh.

Finally, I would like to thank my family and friends for putting up with me through what has been a difficult period of my life.

Contents

List of Figures 7

List of Tables 9

1 Introduction 10

1.1 Continuous Process Simulation Packages 12

1.2 Combined Discrete/Continuous Process Simulation 15

1.3 General-purpose Combined Discrete/Continuous Simulation Languages . . . 17

1.4 A Mathematical Formulation of the Combined Process Simulation Problem . 21

1.4.1 The Initial Condition 24

1.4.2 Solution of the Initial Value Problems 25

1.4.3 Events 26

1.4.4 Discrete Changes to the Continuous Mathematical Model 30

1.4.5 Reinitialisation 30

1.5 Thesis Outline 31

2 Combined Discrete/Continuous Modelling - Model Entities 33

2.1 Continuous Modelling of Industrial Processes 34

2.2 Combined Discrete/Continuous Modelling 38

2.3 Primitive Model Entities 45

2.3.1 Parameter Attributes 47

2.3.2 Variable Attributes 49

2.3.3 Stream Attributes 50

2.3.4 Equation Attributes 51

2.3.5 Structured Equations 54

2.3.6 External Equation Declaration 64

2.4 Hierarchical Submodel Decomposition 67

2.4.1 Unit Attributes 69

2.4.2 The WITHIN Structured Equation 72

2.4.3 Parameter Value Propagation 73

2.4.4 Formal Definition of Model Entity 74

2.4.5 Continuous Connections 75

2.5 Hierarchical Model Development Through Inheritance 77

2.5.1 Single Inheritance 80

2.5.2 Polymorphism 80

2.5.3 Multiple Inheritance 83

2.5.4 Selective Inheritance 85

2.6 Summary 86

3 External Actions Imposed on a System - Task Entities **88**
3.1 Formalisms for the Interaction of External Actions with a Processing System 89
3.2 Elementary Tasks 94
3.2.1 The RESET task 96
3.2.2 The REPLACE task 99
3.2.3 The REINITIAL task 103
3.2.4 The MESSAGE task 105
3.3 Schedules of Tasks 106
3.3.1 The Basic Control Structures - Sequential and Concurrent Execution 109
3.3.2 Conditional Execution of Tasks 111
3.3.3 Iterated Execution of Tasks 112
3.3.4 Local Task Variables 113
3.3.5 The CONTINUE task 114
3.3.6 Task Entities 116
3.4 Reuse of Tasks 116
3.4.1 Task Parameters 118
3.4.2 Polymorphism 120
3.5 Hierarchical Subtask Decomposition 120
3.6 Explicit Concurrency Versus Process Interaction 123
3.7 Summary 124

4 Combined Discrete/Continuous Simulation – Process Entities **130**
4.1 The Distinction Between a Model and an Activity 130
4.2 Process Entities 132
4.2.1 The Combined Discrete/Continuous Model 132
4.2.2 Additional Equations 133
4.2.3 Initial Values for SELECTOR Variables 134
4.2.4 The Initial Condition 135
4.2.5 External Actions 137
4.2.6 Initial Guesses 137
4.2.7 Storage of Simulation Results 139
4.3 Reuse of Process Entities 140
4.4 Hierarchical Subprocess Decomposition 142
4.5 Summary 143

5 Implementation **144**
5.1 Implementation Philosophy and Assumptions 145
5.1.1 Intermediate Code Generation 145
5.1.2 A Software Architecture for Combined Simulation 146
5.1.3 Improving Translation Speeds 148
5.1.4 The Prototype Modelling Package 148
5.2 The Translator 149
5.2.1 The Scanner 150

5.2.2 The Parser . . . 151
5.2.3 The Semantic Routines . . . 152
5.2.4 Errors During Translation . . . 153
5.3 The Process Manager . . . 153
5.3.1 Instantiation of the Continuous Model . . . 156
5.3.2 The Simulation Algorithm . . . 158
5.3.3 A Multiple Process Manager . . . 164
5.4 The Environment . . . 164
5.5 Summary . . . 166

6 Combined Discrete/Continuous Simulation Examples 168
6.1 Physico-chemical Discontinuities . . . 169
6.1.1 Vessel Containing an Overflow Weir . . . 169
6.1.2 Vessel Fitted with a Bursting Disc . . . 171
6.1.3 Vessel Fitted with a Safety Relief Valve . . . 173
6.1.4 Equilibrium Flash Drum . . . 173
6.2 'Continuous' Processes . . . 181
6.2.1 Evaporator Pilot Plant . . . 182
6.2.2 pH Control of an Effluent Tank . . . 193
6.3 Periodic Processes . . . 197
6.3.1 Pressure Swing Adsorption . . . 198
6.4 Batch and Semi-Continuous Processes . . . 207
6.4.1 Batch Reactor . . . 209

7 Conclusions and Recommendations 215
7.1 Conclusions . . . 215
7.2 Recommendations for Future Research . . . 217
7.2.1 Modelling of Distributed Parameter Systems . . . 218
7.2.2 Dynamic Alteration to the Static Characteristics of Model Entities . 219
7.2.3 Multipurpose Equipment . . . 219
7.2.4 Broadening the Range of Activities . . . 221

A The DECLARE Block 223
1.1 The TYPE Section . . . 223
1.2 The STREAM Section . . . 224

B Yacc Input File for the Current Implementation of gPROMS 226

C gPROMS Input File for the Evaporator Pilot Plant 239

D Modelling Equations for the Equilibrium Flash Drum 265

References 269

List of Figures

2.1 Examples of Models Containing Physico-Chemical Discontinuities 41
2.2 Digraph Representation of a Combined Discrete/Continuous Model 44
2.3 Example PARAMETER section 49
2.4 Example VARIABLE section 50
2.5 Example STREAM section 50
2.6 Example EQUATION section 52
2.7 Model that Statically Mixes an Arbitrary Number of Streams 56
2.8 Model of a Tubular Reactor 57
2.9 Model of a Vessel Fitted with a Bursting Disc 60
2.10 Model of a Vessel Containing an Overflow Weir 62
2.11 Model of a Vessel Containing a Series of Overflow Pipes 63
2.12 Hierarchical Submodel Decomposition 68
2.13 Example UNIT section 70
2.14 Example WITHIN equation 72
2.15 Example of Parameter Propagation 73
2.16 Model of a Distillation Column 78
2.17 Model of a Tray Section 79
2.18 Inheritance Hierarchy 81
2.19 Tray Section Parameterised by the Trays Present 84

3.1 Two applications of the RESET task 98
3.2 Model of a Pump 99
3.3 Manipulation of selector variables by a RESET task 99
3.4 Applications of the REPLACE task 100
3.5 Automatic calculation of controller bias using a REPLACE task 102
3.6 Applications of the REINITIAL task 105
3.7 Simple Sequence of External Actions 110
3.8 Example IF Control Structure 112
3.9 Example WHILE Control Structure 113
3.10 Task to Model a Digital Proportional-Integral Control Law 117
3.11 Reusable Task to Model a Digital Proportional-Integral Control Law 126
3.12 Task to Open a Valve 127
3.13 Hierarchical Subtask Decomposition 128
3.14 Implied Concurrency 129
3.15 Example Employing the TERMINATE Task 129

4.1 Example UNIT and SET sections 133
4.2 Example EQUATION section 134

4.3 Example ASSIGN section 135
4.4 Example INITIAL section 136
4.5 A Complete Process Entity 138
4.6 Example PRESET section 139
4.7 A Reusable Process Entity 141

5.1 The Structure of the Translator 150

6.1 Simulation of a Vessel Containing an Overflow Weir 170
6.2 Simulation of a Vessel Fitted with a Bursting Disc 172
6.3 Extracts from the Model of a Vessel Fitted with a Safety Relief Valve 174
6.4 Simulation of a Vessel Fitted with a Safety Relief Valve 174
6.5 Flowsheet Involving Equilibrium Flash Drum 175
6.6 Finite Automaton for Flash Vessel 177
6.7 Simulation of Multicomponent Flash Drum 179
6.8 Simulation of Single Component Flash Drum 180
6.9 Evaporator Pilot Plant Schematic 183
6.10 Extract from the Model of the Calandria 185
6.11 Determination of the Intensive Properties of a Reversible Stream 186
6.12 Task Describing Start-Up Procedure for the Evaporator Pilot Plant 188
6.13 Task Describing Shut-Down Procedure for the Evaporator Pilot Plant 188
6.14 Process Entity Describing Simulation Involving the Evaporator Pilot Plant . 189
6.15 Liquid Levels in the Evaporator Pilot Plant 191
6.16 Bulk Solute Concentrations in the Evaporator Pilot Plant 191
6.17 Bulk Temperatures in the Evaporator Pilot Plant 192
6.18 Effluent Tank Schematic 193
6.19 Schedule Describing Simulation of Effluent Tank 195
6.20 pH and Volume Trajectories for the Effluent Tank 195
6.21 Stem Position Trajectory for the Alkali Valve 196
6.22 Skarstrom Two Column, Four Step Configuration 199
6.23 SO_2 Mole Fractions in the Gaseous Phase for Several Axial Positions 205
6.24 Moles of SO_2 Adsorbed per Gramme of Solid for Several Axial Positions . . . 206
6.25 Hypothetical Batch Plant 210
6.26 Yield, Conversion, and Selectivity for Controlled Cooling Load 211
6.27 Yield, Conversion, and Selectivity for Discrete Charges of Feed 213
6.28 Molar Holdups in the Batch Reactor 214

A.1 Simple DECLARE block. 224

D.1 Extract from the Model of the Flash Drum 268

List of Tables

2.1 Table of Built-in Vector Functions 54
2.2 Table of Built-in Scalar Functions 54

6.1 Table of Events During Simulation of Multicomponent Flash Drum 180
6.2 Table of Significant Events During Operation of Evaporator Pilot Plant . . . 190
6.3 Nomenclature for Model of Adsorption Column 201

D.1 Nomenclature for Model of Equilibrium Flash Drum 266

Chapter 1

Introduction

The ability to predict and analyse the time dependent behaviour of industrial processing plant is indispensable to the engineering of such systems. This thesis is concerned with one of the tools that may be employed for this purpose: *dynamic simulation.* The prediction is obtained from the numerical solution of a mathematical model of the system under a given set of experimental conditions. In particular, this thesis examines the dynamic simulation of processes that experience significant discrete changes superimposed on their predominantly continuous behaviour.

Dynamic simulation is crucial to the analysis of any process that is operated in a dynamic mode. A process operated in the periodic or batch modes is obviously a good example where dynamics dominate process behaviour. However, it is also important to recognise that even a process that is conventionally considered to be 'continuous' rarely operates at the notional steady-state. Perris (1990) lists commissioning and start-up/shutdown, process maintenance, feed stock and/or product campaign changes, and load/demand following as some of the many situations in which the dynamic performance of this latter category of process is of overriding concern.

Potential applications of dynamic simulation throughout the entire lifetime of a process have been advocated for many years. In addition to the traditional off-line application to the selection and design of regulatory control systems, Perkins (1985) points out that the identification of designs that may lead to dynamic instability, the prediction of potentially hazardous situations as a consequence of upsets in the operation of the process, and the selection of optimal operating procedures to move a process between operating points should all be considered equally worthwhile. Dynamic simulation can also aid in the synthesis and validation of safe and economical operating procedures in a broader sense. On-line applications include model predictive control, and process monitoring, estimation and prediction (Perkins and Barton, 1987).

The safe and environmentally friendly operation of all aspects of industrial processing plant is becoming increasingly important as the pressures of legislation and public opinion on industry continue to intensify. The contribution that dynamic models make in ensuring this is already significant, and can only increase in prominence as operating companies are forced to work within tighter constraints. A related issue is that of training. Ferney (1991) observes that there is little point in investing in a safety programme without also providing adequate training for the operators who will implement it. The necessity of accurate dynamic modelling of process behaviour as the basis for computer based operator training activities is well established (see, for example, Kassianides (1991)). Mani *et al.* (1990) describe a recent experience in which the general-purpose dynamic simulation package GEPURS^{TM} (Shinohara, 1987) was employed to build a training simulator for an entire fertiliser complex.

Despite the potential benefits, and sometimes necessity, of the application of dynamic simulation technology in the process industry, academic authors remain dissatisfied with the extent to which this technology is employed throughout the industry (Perkins, 1985; Marquardt, 1991). These authors cite two reasons for reluctance on the part of their industrial colleagues:

- Questions are still raised about the benefits (if any) accruing from the use of dynamic simulation technology.
- The inadequacies of currently available packages still makes dynamic simulation a very costly activity that can only be justified in special cases.

The former argument, however, is becoming outdated in light of the recent upsurge of industrial interest in dynamic simulation as a consequence of intensifying competition in worldwide markets and increasingly restrictive safety and environmental legislation in the developed world. While discussing the latter argument, Perkins and Barton (1987) point out that the large investment required to build a dynamic model can be justified if it becomes a knowledge base on which activities throughout the entire lifetime of a process, from preliminary design to decommissioning, can be based. The recent experience with the THORP nuclear fuel reprocessing plant (Evans and Wylie, 1990) is a modest example of the application of this philosophy: a dynamic model was originally built in order to facilitate the early testing of the control system software and thereby significantly reduce the time and cost of commissioning. However, the recognition of the value of employing the same model as a tool for operator training and dynamic experimentation extended its useful life way beyond the commissioning phase.

This thesis attempts to make a contribution towards addressing some of the inadequacies of currently available dynamic simulation packages. The reusability of process models will be a theme that is returned to repeatedly.

1.1 Continuous Process Simulation Packages

In order to address some of the inadequacies of existing dynamic simulation packages, it is first worthwhile to examine the range of techniques currently available. Marquardt (1991) has recently presented a thorough review of this field, so the reader is directed to this for an exhaustive list of the software packages available. Here, we will concentrate on a brief review of the features of the various classes of package.

Engineers have been building dynamic models of entire industrial processes ever since digital computers first became widely available. The earliest approach was based on the development of one-off programs coded in a procedural programming language such as FORTRAN. This medium offers the engineer immense flexibility, but is also extremely labour intensive, and requires a high level of expertise in numerical methods and programming in addition to the engineering knowledge required to pose models. Although the emergence of high quality codes implementing many standard numerical algorithms has alleviated these problems to a certain extent, this approach still remains a rather costly alternative. Nevertheless, it is important to recognise that the use of one-off programs is still probably the most common approach to the dynamic simulation of industrial processes. For example, although the developers of the dynamic model for THORP (Evans and Wylie, 1990) recognised the potential of more advanced techniques, they found that a specially developed program was the only alternative to offer sufficient flexibility.

In light of the expertise required by, and cost associated with, the above approach, the use of software packages specifically designed for the activity of dynamic simulation has had almost as long a history. Such packages are considered to improve productivity because, to a greater or lesser extent, they free the engineer to concentrate on the correct formulation of the process model, as opposed to the numerical algorithms and coding required to achieve a solution.

The general-purpose continuous simulation languages originating from the CSSL standard (Strauss, 1967), such as ACSL (Mitchell and Gauthier, 1976) and CSMP (Speckhart and Green, 1976), therefore seem to offer an attractive alternative. These packages have been available for many years and have reached a high level of sophistication, providing the engineer with a convenient and ready to use environment in which he or she can pose process models in either a block or equation-oriented manner. However, a major criticism of this appealing approach is that the modelling methodology and numerical techniques employed by these packages are most suitable for systems that can be modelled with relatively small numbers of explicit ordinary differential equations, and that they become impracticable when applied to the large numbers of mixed ordinary differential and algebraic equations typically required to model industrial processing systems. This argument has been used by Perkins (1986), amongst others, to justify the development of *continuous process simulation packages* specifically tailored to the demands of the process industry. Nevertheless, as Marquardt (1991) points out, any such continuous process simulation package must provide much more advanced numerical techniques and domain specific support than merely a library of precoded process models in order to be a viable alternative to CSSL-type languages.

Significant advances in numerical methods and computer hardware during the last ten years have brought about the development of several continuous process simulation packages. These packages are mainly suitable for the continuous simulation of industrial processes, although discrete changes in the process inputs and in the process model are tolerated to a limited extent. Leaving aside arguments concerning how the mathematical model is actually solved, for the purposes of this thesis it is worthwhile to examine how the engineer may employ these packages to pose a process model.

All continuous process simulation packages enable the process model to be posed in a similar manner to that of the majority of steady-state flowsheeting packages – through the connection of a series of library unit operation models in a process flowsheet. Interactive graphics and menus are sometimes also provided to support this activity. Some packages are based on extensions of the modular approach to steady-state process analysis, such as Dynamic FLOWPACK II (Aylott *et al.*, 1985) and GEPURS^{TM} (Shinohara, 1987). However, most employ the equation-oriented approach. Examples include Diva (Holl *et al.*, 1988), Dynamic QUASILIN (Smith and Morton, 1988), MASSBAL 3 (Shewchuk and Morton, 1990), and DYNSIM (Gani *et al.*, 1992).

In addition to providing the above facilities for unit based flowsheet modelling, some equation-oriented packages allow the engineer to pose additional unit operation models in terms of high-level equation-based symbolic languages. This category includes packages such as DPS (Wood *et al.*, 1984) and SpeedUp (Perkins and Sargent, 1982).

The latter approach provides the engineer with greater flexibility when constructing the dynamic model of an entire process. This degree of flexibility is desirable because the construction of a 'complete' library of standard dynamic unit operation models is extremely difficult; the level of detail often required of dynamic models has the consequence that each modelling exercise will usually demand its share of non-standard models. For example, vessels with different geometry or internal structure, employed for the same unit operation and described by identical steady-state models, will quite frequently have significantly dissimilar dynamic models.

The author therefore believes that the demands of potential industrial users of dynamic simulation can only be satisfactorily addressed by those packages that provide the engineer with the facility to add new unit operation models as required. In order to benefit from the improvements in productivity that dynamic simulation packages offer over procedural programming languages, the mechanisms that support the development of these new models should ideally require the engineer to provide only declarative information concerning the physical behaviour of the system under investigation.

Of course, this requirement places very stringent demands on the numerical methods employed to solve the simulation problem. Significant progress in the development of these methods has been achieved in recent years. They are therefore in an advanced and *relatively* satisfactory state. For example, current industrial practice with the SpeedUp package (Prosys, 1991) involves the regular solution of dynamic simulation problems involving tens of thousands of simultaneous equations. However, it is also important to recognise that this is still a very active field of research, and further developments are required to improve the reliability of solution methods (see, for example, recent work concerning the solution of problems where a process is far from its notional operating point (Jarvis and Pantelides, 1991)).

In spite of these developments, one of the remaining reasons why dynamic simulation is not receiving the broad application it deserves is that the engineer is still unable to pose an important class of problems in a satisfactory manner. This deficiency is considered in more detail in the next section.

1.2 Combined Discrete/Continuous Process Simulation

As already stated, the continuous process simulation packages were developed to address the solution of large dynamic process models of a continuous nature. However, few processes can be considered to operate in an entirely continuous manner. The majority of 'continuous' processes also experience significant discrete changes superimposed on their predominantly continuous behaviour. Such discrete changes typically arise from the application of digital regulatory control, plant equipment failure, or as a consequence of planned operational changes, such as start-up and shut-down, feed stock and/or product campaign changes, process maintenance etc. Moreover, the situations in which these discrete components affect the overall process behaviour usually correspond to those in which it is most worthwhile to perform dynamic simulation in the first place.[1]

The existing continuous process simulation packages provide very limited capabilities for the description of dynamic simulations in which discrete changes significantly affect the overall behaviour. This was, for example, one of the major reasons why the developers of the dynamic model for THORP (Evans and Wylie, 1990) did not find SpeedUp (Prosys, 1991) sufficiently flexible. Engineers therefore still have to resort to the use of procedural programming languages in order to be able to pose the very class of problems for which dynamic simulation can most easily be justified. In light of this, it is perhaps not surprising that dynamic simulation is widely held to be a costly and time consuming activity.

This thesis argues that, in many cases, an engineer involved in the analysis of the dynamic behaviour of a continuous process wishes to pose *combined discrete/continuous* simulation problems as opposed to purely continuous simulation problems. In order to meet this demand, and to improve productivity, it is necessary to consider the development of

[1]For example, from a safety or environmental point of view.

a general-purpose *combined discrete/continuous process simulation package* that will also encompass the capabilities of existing continuous process simulation packages.

Developments in the world economy have prompted renewed interest in the batch and semi-continuous modes of process operation amongst producers in the developed world (Parakrama, 1985). This trend has in part been caused by the increased competition from producers in the developing nations experienced in many bulk chemical markets. As a consequence, attention has shifted towards the production of relatively small quantities of high added value products in multipurpose/multiproduct batch plant. However, even in these markets it will become increasingly difficult to compete on the basis of patenting new products alone (Sawyer, 1992): simulation of batch processes to improve efficiency is becoming vital in the battle to remain competitive.

Batch and semi-continuous processes are always operated in an essentially dynamic manner. Dynamic simulation is therefore essential for the detailed prediction of their behaviour. Furthermore, this category of process always experiences frequent discrete control actions in order to maintain operation in this dynamic, often cyclic, mode. The combined discrete/continuous nature of these systems has been recognised for many years (Fruit *et al.*, 1974), and this has been reflected in the design of special purpose dynamic simulation packages for batch processes such as BATCHES (Joglekar and Reklaitis, 1984) and UNIBATCH (Czulek, 1988).

In his recent review, Marquardt (1991) argues that future dynamic simulation packages should support the numerical analysis of arbitrarily operated processes within a unified framework. Bearing in mind that the above discussion has emphasised the combined discrete/continuous nature of most complex industrial processing systems, regardless of their nominal mode of operation, it follows that this new generation of dynamic simulation packages should be combined discrete/continuous simulation packages. This thesis therefore considers the issues involved in the development of a general-purpose combined discrete/continuous process simulation package suitable for the analysis of the entire range of process operations, from purely continuous to batch. In doing so, two main objectives are perceived:

1. To develop a sound formal basis for the description of combined discrete/continuous process simulation problems, and to articulate this in the form of a simulation language.

2. To demonstrate that a practical implementation of such a modelling package is feasible.

1.3 General-purpose Combined Discrete/Continuous Simulation Languages

Before embarking on the task outlined at the end of the previous section, it is necessary to review the efforts of the system simulation community with regard to combined discrete/continuous systems. Their observations will at least provide useful insights for the development of the formal basis proposed above.

System simulation concerns itself with the prediction of the time dependent behaviour of real systems via the numerical solution of a mathematical model of these systems. Traditionally, systems have been classified as either *discrete event* or *continuous*:

- In order to obtain information concerning the detailed interactions of entities, the dynamic behaviour of a 'discrete event' system is abstracted to a series of *events* at specific points in time. The state of the system is only allowed to change discretely at these points in time – between events the system remains unchanged from the last event. Events can interact and trigger new events, so the dynamic behaviour is determined by the time order in which the events occur, and their interaction with each other.

- The dynamic behaviour of a 'continuous' system is abstracted to the point at which it can be represented by the smooth, continuous change of a series of state variables. This allows such a system to be represented mathematically as a set of differential equations, with time and possibly one or more spatial dimensions as independent variables. The simulation can therefore be posed as a initial value problem: the equations are integrated from an initial condition until the desired termination condition is satisfied. The set of equations remains unchanged throughout the simulation, and all variables and their time derivatives follow a continuous trajectory in time.

The simulation of systems that belong to either of these categories has a considerable history dating back to the 1950s, each 'school of thought' engendering several generations of general-purpose simulation languages. Kreutzer (1986) gives a detailed account of this history and a list of the languages available.

Obviously, neither classification is suitable for the detailed analysis of systems where both continuous and discrete changes take place and interact to a significant extent during part or all of the period under investigation. The assumptions made in either case, in order to simplify the solution of the simulation problem, preclude the analysis of such systems. Moreover, it appears that, for many years, the direct analysis of this class of system was not considered to be worthwhile by the simulation community – the behaviour of such systems was usually abstracted until it conformed to one of the categories above.

Fahrland (1970) was the first author to advocate the development of 'combined discrete event and continuous' simulation languages in order to handle those systems that exhibit both characteristics concurrently. This new class of simulation language was justified from a representational point of view: it would enable an engineer to model a physical system in its most natural form, whether continuous, discrete, or combined, and hence allow a more exact representation with fewer approximations. In addition to this argument, Cellier (1979a) was later to advocate combined simulation languages from the point of view of the efficient and accurate solution of the continuous model.

In his original paper, Fahrland (1970) identifies the fundamentals of combined discrete/continuous systems, and describes how this class of problem may be solved as a sequence of initial value problems involving a continuous model, interspersed by events at which a discrete model become instantaneously active. The foundations were also laid for the representational methodology employed by all subsequent combined simulation languages, to the author's knowledge.

This methodology requires a combined system to be decomposed into a continuous subsystem and a discrete subsystem. The two subsystems are then allowed to interact as equals during the course of a simulation experiment. The continuous subsystem can interact with the discrete subsystem in either of the following fashions:

- The discrete subsystem makes reference to the values of the variables describing the continuous subsystem at some discrete point in time.

- The reactivation of the discrete subsystem at some point in time (an event) is triggered by a condition becoming satisfied as a consequence of the continuous change of the relevant subsystem (subsequently termed *state events*).

Similarly, the discrete subsystem can interact with the continuous subsystem in the following fashions:

- The discrete subsystem can instantaneously change the values of one or more of the input variables of the continuous subsystem.
- The discrete subsystem can instantaneously change the values of one or more of the state (or differential) variables of the continuous subsystem.
- The discrete subsystem can cause an instantaneous and arbitrary structural modification of the continuous subsystem.

In fact, the first form of interaction of the discrete subsystem with the continuous subsystem can be considered to be a special case of the third form, if the relationships that determine the values of the input variables are included in the continuous subsystem as additional equations (see section 1.4). Although this latter form of interaction could encompass arbitrary changes to the number and functional form of the equations describing the behaviour of the continuous subsystem, Fahrland considered this capability too esoteric.[2] Instead, he concentrated on systems that exhibit a 'population-change' feature. Here, continuous behaviour of fixed dimensionality is nested within the description of a discrete entity, and is duplicated each time an instance of the discrete entity is created during a simulation experiment. A frequently quoted example of this category of system is a steel soaking pit in which individual steel ingots are created at random intervals, with randomly distributed initial temperatures. Following their creation, the ingots enter a furnace, within which they are heated continuously at different rates, according to the difference between their individual temperatures and the bulk temperature of the furnace.

In closing, Fahrland briefly observes that a more frequently occurring class of combined system is composed of a fixed dimension continuous subsystem which acts as the

[2]And difficult to implement!

facilities for some form of large scale processing sequence. In this case, the discrete subsystem would presumably model the control actions or disturbances imposed on this facility. However, this class of combined system does not seem to have been considered thoroughly in any of the subsequent literature, although it is of primary interest to process engineers.

The simulation language GSL (Golden and Schoeffler, 1973) was created on the basis of Fahrland's proposals. This language enables a system to be described in terms of a series of discrete and continuous *blocks*, and set the mould for most subsequent languages. During a simulation experiment, instances of either type of block are dynamically created, and then co-exist and interact with one another. By the time of Oren's (1977) review, at least eighteen software packages for combined discrete/continuous system simulation could be identified, although some confusion seemed to exist with simulation languages designed for hybrid digital/analogue computers.

In a later review, Cellier (1979b) observed that most of the more complex systems traditionally considered to be 'continuous' were in fact combined systems, as we have already observed for industrial processing systems in the previous section. He therefore concluded that combined simulation should have a much bigger impact on the continuous simulation community than on the discrete event simulation community. However, Cellier also observed that most combined simulation languages of the day had been developed as extensions of existing discrete event simulation languages, mainly because the complex language structures employed by this class of simulation language were more suitable for extension in order to encompass combined problems. As a consequence, he argued that the continuous simulation capabilities of these languages were not adequate for many applications. Languages based on Simula (Birtwistle *et al.*, 1979) such as CADSIM (Sim, 1975) and DISCO (Helsgaun, 1980), and the FORTRAN subroutine libraries descended from GASP II (Pritsker and Kiviat, 1969), GASP IV (Pritsker and Hurst, 1973), SLAM II (Pritsker, 1986), and SIMAN (Pegden, 1982), can all be considered to suffer from this deficiency.

In order to address these deficiencies, a second generation of combined discrete/continuous simulation languages that attempt to add continuous simulation capabilities comparable to those of the languages descended from the CSSL standard has emerged in more recent years. These might be considered to be the first *truly* combined simulation software packages (Cellier, 1986), and include COSY (Cellier and Bongulielmi, 1980), SYSMOD (Smart and Baker, 1984), and COSMOS (Kettenis, 1992).

It seems, however, that a simulation language suitable for the description of combined discrete/continuous process simulation problems does not exist at present. In summary, there are two major arguments to justify this statement:

- Very little attention has been paid to the class of combined systems of primary interest to the process engineer, namely those that are composed of a fixed dimension continuous subsystem which has actions imposed on it by the discrete subsystem.
- Even the most advanced combined simulation languages only offer continuous modelling and simulation capabilities comparable to those of the CSSL-type languages. They are therefore unsuitable for exactly the same reasons that make CSSL-type languages unsuitable for continuous process applications (Perkins, 1986).

The following section is concerned with a detailed analysis of the mathematical characteristics of the combined discrete/continuous process simulation problem. This discussion will further emphasise the unsuitability of CSSL-type languages for the description of this class of problem.

1.4 A Mathematical Formulation of the Combined Process Simulation Problem

It is important to recognise that many diverse modelling formalisms have been proposed for the representation of the physical behaviour of the real world, each of which has been developed to address the particular requirements of an application domain. The suitability of a modelling formalism is not, however, solely dictated by the characteristics of the system under investigation, but also by the questions that the modeller wishes to pose about the behaviour of this system. A non-exhaustive list of the modelling formalisms that have been proposed for the dynamic behaviour of physical systems includes:

- Differential-algebraic and partial differential-algebraic equations.
- Ordinary differential and partial differential equations.
- Difference equations.

- The various world views of discrete simulation (Kreutzer, 1986).
- Petri nets (Petri, 1962).

The utility of modelling environments in which a wide range of modelling formalisms can co-exist harmoniously has been advocated for several years (Oren and Ziegler, 1979).

The physico-chemical mechanisms that govern the time dependent behaviour of industrial processing systems are predominantly continuous. Therefore, the combined discrete/continuous simulation of such systems requires a modelling formalism that is compatible with the fundamental characteristics of this continuous behaviour. Modelling of these physico-chemical mechanisms from first principles typically yields large, sparse, and stiff nonlinear equation sets of mixed type. A process that can be entirely described in terms of lumped parameters will give rise to a model composed of a mixed set of ordinary differential and algebraic equations (see, for example, Pantelides *et al.* (1988)).

A process that also contains variables distributed in one or more spatial dimensions will give rise to a model composed of a mixed set of partial differential, ordinary differential, and algebraic equations (Heydweiller *et al.*, 1977). Particulate system modelling by means of population balances, or terms that must be integrated over one or more spatial dimensions, may also add integral terms to the above set of equations (Marquardt, 1991). However, methods for the direct solution of these equations when there are one or more spatial independent variables in addition to time are still in their infancy (see, for example, Pipilis (1990)). This latter class of problems will not therefore be considered further here. A more common practice at present involves the manual reduction of the set of partial differential-algebraic equations to a set of differential-algebraic equations through an appropriate discretisation of the spatial independent variables (the method of lines).

A natural modelling formalism for the continuous time dependent behaviour of industrial processing systems is therefore expressed mathematically as a set of nonlinear equations of the form:

$$\mathbf{F}(\mathbf{z}, \dot{\mathbf{z}}, \mathbf{u}, t) = 0 \tag{1.1}$$

$$\mathbf{u} = \mathbf{u}(t) \tag{1.2}$$

where $\mathbf{z}, \dot{\mathbf{z}} \in \Re^s$, $\mathbf{u} \in \Re^l$, $\mathbf{F} : \Re^s \times \Re^s \times \Re^l \times \Re \mapsto \Re^s$. $\mathbf{z}$ is the set of unknown system variables with time, t, as the independent variable, and $\dot{z} \equiv dz/dt$. $\mathbf{u}$ is the set of known system inputs.

Equations of the above form can be classified according to their *index*, which may be defined as the smallest non-negative integer I such that equation 1.1 and its first I derivatives with respect to time uniquely define $\dot{\mathbf{z}}$ as a function of $\mathbf{z}$, $\mathbf{u}$ (and its time derivatives), and t (Brenan *et al.*, 1989). For the purposes of this thesis, we will concentrate on a limited category of the above equations that occur frequently in the modelling of industrial processing systems, namely:

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}, \mathbf{u}, t) = 0 \tag{1.3}$$

$$\mathbf{u} = \mathbf{u}(t) \tag{1.4}$$

where $\mathbf{x}, \dot{\mathbf{x}} \in \Re^n$, $\mathbf{y} \in \Re^m$, $\mathbf{u} \in \Re^l$, and $\mathbf{f} : \Re^n \times \Re^n \times \Re^m \times \Re^l \times \Re \mapsto \Re^{n+m}$, such that:

$$\text{Rank} \begin{bmatrix} \mathbf{f}_{\dot{\mathbf{x}}} & \mathbf{f}_{\mathbf{y}} \end{bmatrix} = n + m \tag{1.5}$$

everywhere.

This, together with the assumption that a solution to the set of equations 1.3 does exist, is a sufficient, but not necessary, condition for these equations to have an index equal to or less than unity. $\mathbf{x}$ are usually referred to as the differential variables, whereas $\mathbf{y}$ are referred as the algebraic variables. It should be noted that well-posed purely differential ($m = 0$) and purely algebraic ($n = 0$) equation sets fall into this category.

Combined discrete/continuous simulation of industrial processing systems requires the solution of a sequence of initial value problems, described by equations of the above form, interspersed by instantaneous events that may cause some form of discrete change to the initial value problem currently being solved. The describing equations and initial condition of the first initial value problem are determined by an individual simulation description. The describing equations and initial condition of the succeeding initial value problems will be determined from a combination of the final state of the preceding initial value problem and the consequences of the corresponding event(s).

The following sections consider various aspects of this mathematical problem in more detail.

1.4.1 The Initial Condition

Before simulation can commence, consistent initial values for the variables $\mathbf{x}$, $\dot{\mathbf{x}}$, and $\mathbf{y}$ at the initial time t_0 are required. A necessary condition for a set of initial values $\{\mathbf{x}(t_0), \dot{\mathbf{x}}(t_0), \mathbf{y}(t_0)\}$ to be consistent is that they satisfy the equations 1.3 at t_0. In the most general case this is not a sufficient condition for consistency, because the initial values may also be constrained by additional equations which are derived from differentiation of the original set of equations with respect to time (Pantelides, 1988a).

However, for the limited category of equations that satisfy the constraint shown in equation 1.5, the above is also a sufficient condition. In this case, the equations 1.3 represent a set of $n + m$ equations in the set of $2n + m$ unknowns $\{\mathbf{x}(t_0), \dot{\mathbf{x}}(t_0), \mathbf{y}(t_0)\}$. In order to determine consistent initial values for these unknowns, an *initial condition* composed of n additional specifications is therefore required. For a set of explicit ordinary differential equations of the form:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t) \tag{1.6}$$

the term 'initial condition' normally refers to a set of values for the variables $\mathbf{x}$ at t_0. For a system of differential-algebraic equations of the form shown in equation 1.3 a more general approach is both possible and desirable. For example, it may be necessary to specify the initial state of the system in terms of values for the algebraic variables and/or the time derivatives of the differential variables.[3]

Conventional continuous process simulation packages such as SpeedUp (Pantelides, 1988b) provide a flexible facility for the specification of this initial condition: they allow for the specification of initial *values* for any subset of n variables in the set $\{\mathbf{x}, \dot{\mathbf{x}}, \mathbf{y}\}$, subject to the nonsingularity of the equations 1.3 in the remaining variables. The equations 1.3 can then, in principle, be solved for these remaining variables to yield consistent initial values for all the unknowns.

However, the initial condition can be expressed in even more general terms by nonlinear equations of the form:

$$\mathbf{r}(\mathbf{x}(t_0), \dot{\mathbf{x}}(t_0), \mathbf{y}(t_0), \mathbf{u}(t_0), t_0) = 0 \tag{1.7}$$

[3]After all, the specification of steady-state, $\dot{\mathbf{x}}(t_0) = 0$, is probably the most frequently encountered initial condition.

where $\mathbf{r} : \Re^n \times \Re^n \times \Re^m \times \Re^l \times \Re \mapsto \Re^n$. Providing the set of equations resulting from 1.7 and 1.3 represents a mathematically well-posed problem, consistent initial values for all the unknowns can be determined, in principle, from the simultaneous solution of these equations by a general-purpose nonlinear equation solver. From a practical point of view, we expect the following condition to hold for the set of values $\{\mathbf{x}(t_0), \dot{\mathbf{x}}(t_0), \mathbf{y}(t_0)\}$:

$$\text{Rank} \begin{bmatrix} \mathbf{f_x} & \mathbf{f_{\dot{x}}} & \mathbf{f_y} \\ \mathbf{r_x} & \mathbf{r_{\dot{x}}} & \mathbf{r_y} \end{bmatrix} = 2n + m \tag{1.8}$$

1.4.2 Solution of the Initial Value Problems

A combined discrete/continuous simulation is advanced by the solution of a sequence of initial value problems. Strategies for the numerical solution of these initial value problems fall broadly into the categories (Marquardt, 1991):

- Direct integration – the simultaneous solution of all the modelling equations with a single algorithm.
- Modular integration – the solution of a suitably partitioned system by means of different algorithms applied to each subsystem.

Although in principle a modelling package for combined discrete/continuous simulation should be designed to be independent of such solution strategies, much of the work in this thesis is based on the contention that a simulation description should only contain declarative information concerning the physical behaviour of the system under investigation. If this is the only information that can be made available when a model is submitted to a numerical solver, it is unlikely that the problem specific improvements in efficiency advocated by the proponents of modular strategies can be realised given the current state of numerical techniques. Therefore, for the purposes of this thesis the benefits of a direct strategy, in terms of easily quantifiable accuracy and guaranteed numerical stability, far outweigh those of a modular strategy.

It is worthwhile to mention briefly the methods currently available for the direct integration of equation 1.3. Two relatively mature approaches to the solution of these equations have emerged. The first approach involves the use of a numerical differential-algebraic equation solver which accepts problems in the form shown in equation 1.3 directly. The only requirement is a routine to calculate the residuals of these equations given values for all the arguments. The majority of codes are based on Gear's (1971) multi-step Backward Differentiation Formulae (BDF) method, including, for example, DASSL (Petzold, 1982) and DASOLV (Jarvis and Pantelides, 1992).

Currently available numerical differential-algebraic equation solvers are suitable for the solution of equations that satisfy constraint 1.5. However, it is important to recognise that most encounter problems in controlling the error of integration if the index of the equations exceeds unity (see, for example, Gritsis *et al.* (1988)). A conceptual algorithm that can solve any problem of index exceeding unity directly has been published recently (Chung and Westerberg, 1990). Alternatively, the index of the equations can be reduced to unity by differentiation with respect to time (see, for example, Bachmann *et al.* (1990)) and then solved with one of the codes described above.

The second approach involves the solution of the differential-algebraic equation set 1.3 by a numerical *ordinary* differential equation solver. This can be done in the most general manner by transformation of equation 1.3 into the form:

$$\mathbf{f}(\mathbf{x}, \mathbf{v}, \mathbf{y}, \mathbf{u}, t) = 0 \tag{1.9}$$

$$\dot{\mathbf{x}} = \mathbf{v} \tag{1.10}$$

through the introduction of a set of velocity variables, $\mathbf{v} \in \Re^n$, to replace all occurrences of the time derivatives of the differential variables in the original set of equations (Westerberg and Benjamin, 1985). The numerical ordinary differential equation solver is applied to equation 1.10, and consequently requires a routine to solve the equations 1.9 for $\mathbf{v}$ and the algebraic variables $\mathbf{y}$, given values for the differential variables $\mathbf{x}$. This approach is therefore only suitable for the index one differential-algebraic equations that satisfy constraint 1.5.

1.4.3 Events

The termination of each initial value problem in the sequence described above is marked by the occurrence of an event. An event takes place instantaneously with respect to the independent variable time, and will have the consequence, for example, of some form of discrete change to the functional form of the describing equations 1.3, or the scheduling of new event(s) to occur at some future time.

Two types of event can occur during a combined discrete/continuous simulation, distinguished by the manner in which the time of occurrence is determined:

- *Time events* – the exact time of occurrence of these events is known in advance, so solution of the initial value problems can proceed to these events in time order. They may be either *exogenous*, if the time of occurrence is known a priori, or *endogenous*, if the time of occurrence is computed as a consequence of the occurrence of a previous event during the simulation.

- *State events* – the time of occurrence of these events is not known in advance because it is dependent on the system fulfilling certain conditions (called *state conditions*). An initial value problem must instead be advanced speculatively to the point in time at which the state condition becomes satisfied.

The nature of state events dictates that state conditions should ideally be monitored continuously throughout the solution of an initial value problem. However, the numerical integration algorithms employed for the solution of these initial value problems advance time in a step-wise fashion. As a consequence, the system status is only calculated at discrete values of the independent variable time. State events can therefore only practically be detected at the earliest discrete point in time that the system status indicates that a state condition has become satisfied.

Subsequent action can take several forms. The simplest approach is to assume that the state event merely occurs at the point in time that it was first detected. Although efficient, this approach will introduce a timing error that, depending on the size of the current integration step, may seriously affect the results of any subsequent initial value problems.

A second approach relies on the step size control mechanism of the integration algorithm to locate the exact time of occurrence of a state event. In this case, the consequences of a state event are implemented whenever the system status indicates that a state condition is satisfied, even if this occurs in the middle of an iterative calculation such as the 'corrector' iteration of an implicit integration algorithm. Sophisticated numerical integration algorithms, such as those based on BDF approximations, assume a certain degree of continuity in the solution trajectories, so if a state event causes a discrete change to the set of equations in mid-step, either the corrector iteration will fail, or an artificially large error estimate will ensue. In either case, this will cause the step size control mechanism to cut the step repeatedly until a much smaller step length enables the discontinuity to be negotiated. Use of the step size control mechanism to locate events in this manner often requires many failed steps as the step size is progressively reduced, and may even result in failure of the simulation at the discontinuity. In addition to these inefficiencies, this approach can, in certain situations, lead to incorrect solution trajectories (Cellier, 1979a).

An alternative approach adopted in most modern simulation packages (for example, BATCHES (Joglekar and Reklaitis, 1984) and SpeedUp (Pantelides, 1988b)) therefore requires that the describing equations of any initial value problem remain unchanged or 'locked' until the exact time of occurrence of a state event has been located. Numerical integration of this initial value problem is advanced in a stepwise fashion until, at the end of a *successful* time step, a state condition indicates that a state event has occurred at some point during the previous step. The exact time of occurrence is then located by one of the methods described below, the consequences of the state event are implemented, and the new initial value problem is started from that point.

The most frequently reported approach to the location of the exact time of occurrence of state events relies on the notion of a *discontinuity function* (Hay *et al.*, 1974; Cellier, 1979a; Preston and Berzins, 1991). This is a continuous scalar function that crosses zero at the state event. Integration of the current initial value problem proceeds until one or more discontinuity functions change sign, at which point it is known that one or more state events have occurred during the previous step. The actions then taken depend on whether the integration algorithm employed provides accurate interpolation of variable values within a step:

- If an accurate interpolation is not possible, the values of the discontinuity function at the beginning and the end of the step can be utilised to derive a new estimate for the state event time. An integration step is then taken from the *previous* time point to this estimated event time. With the normal step-size control mechanism of the integration algorithm disabled, the integration step length can thus be adjusted iteratively until the solution 'marches up' to the state event.

- If accurate interpolation of variables values is possible, the zero crossing of the discontinuity function, and thus the exact time of occurrence of the state event, can be located merely by interpolation of the discontinuity function over the previous time step. No further integration steps are required for this purpose.

It follows from the above discussion that the location of state events can be considerably more computationally expensive when an accurate interpolation is not available.

A discontinuity function may be employed to represent any state condition expressed in the form of a relational expression between the system variables (a relational expression can be rearranged to a zero crossing by simple subtraction of the real expressions on either side of the relational operators $=, <, \leq, >$, or $\geq$ (Cellier, 1979a)). A general-purpose combined discrete/continuous simulation package should, however, allow state conditions to be expressed in the most general terms possible. In particular, it will be desirable to express state conditions as general logical expressions involving multiple relational expressions linked by the AND/OR/NOT operators. In this case, a *boolean* discontinuity function must be utilised to locate the exact time of occurrence of state events in a slightly different manner:

- If an accurate interpolation is not available, the boolean discontinuity function restricts the iterative step adjustment algorithm to that of bisection.

- If an accurate interpolation is available, the exact time at which the boolean discontinuity function changes value can only be located through a bisection iteration on this function. Interpolation of the system variables over the previous step may be employed to determine the value of this function at any intermediate point in time. This iteration will therefore be inexpensive.

In both cases, therefore, the time of occurrence of the state event can only be located to a certain *state event tolerance.* The new initial value problem must always be started from the upper estimate of this event time in order to ensure that the system has entered the new domain. Moreover, particular care must be taken with the use of the relational operators $=$ and $\neq$ when a boolean discontinuity function is employed.

1.4.4 Discrete Changes to the Continuous Mathematical Model

One of the consequences of the events that delimit the sequence of initial value problems characterising a combined discrete/continuous simulation is a discrete change to the mathematical model of the continuous time dependent behaviour of the system. In the most general terms, this means that a completely new initial value problem is posed at the event time t. At time t^-, immediately before the event, the continuous behaviour of the system is described by the equations 1.3 and 1.4, whereas at time t^+, immediately after the event, the continuous behaviour of the system is described by the new equations:

$$\mathbf{f}'(\mathbf{x}', \dot{\mathbf{x}}', \mathbf{y}', \mathbf{u}', t) = 0 \tag{1.11}$$

$$\mathbf{u}' = \mathbf{u}'(t) \tag{1.12}$$

where $\mathbf{x}', \dot{\mathbf{x}}' \in \Re^{n'}$, $\mathbf{y}' \in \Re^{m'}$, $\mathbf{u}' \in \Re^{l'}$, and $\mathbf{f} : \Re^{n'} \times \Re^{n'} \times \Re^{m'} \times \Re^{l'} \times \Re \mapsto \Re^{n'+m'}$. It is therefore possible for both the number of variables in the various categories and the functional form of the describing equations to change in a completely general manner. From a practical point of view, however, this new set of equations must also satisfy constraint 1.5.

1.4.5 Reinitialisation

Another potential consequence of an event at time t is that consistent values for the variables $\mathbf{x}'$, $\dot{\mathbf{x}}'$, and $\mathbf{y}'$ at time t^+ must be determined before the new initial value problem can be solved. This will always be necessary after a discrete change to the continuous mathematical model at time t. Moreover, even if an event leaves the continuous mathematical model unchanged, it may still be necessary to recalculate consistent values for the variables. This is the case when impulses are employed to model phenomena that take place on a much smaller time scale than that of primary interest (see, for example, Mattsson (1989)).

The *reinitialisation* calculation required to determine a consistent set of values $\{\mathbf{x}'(t^+), \dot{\mathbf{x}}'(t^+), \mathbf{y}'(t^+)\}$ is similar to the *initialisation* calculation required at the beginning of a simulation. However, the values to be determined in the former case may also depend on the values of the variables immediately *before* the discontinuity. The initial condition can therefore be expressed as a set of nonlinear equations:

$$\mathbf{r}'(\mathbf{x}'(t^+), \dot{\mathbf{x}}'(t^+), \mathbf{y}'(t^+), \mathbf{u}'(t^+), \mathbf{x}(t^-), \dot{\mathbf{x}}(t^-), \mathbf{y}(t^-), \mathbf{u}(t^-), t^+) = 0 \tag{1.13}$$

where $\mathbf{r}' : \Re^{n'} \times \Re^{n'} \times \Re^{m'} \times \Re^{l'} \times \Re^{n} \times \Re^{n} \times \Re^{m} \times \Re^{l} \times \Re \mapsto \Re^{n'}$ and $\{\mathbf{x}(t^-), \dot{\mathbf{x}}(t^-), \mathbf{y}(t^-)\}$ are the (known) set of values of the describing variables at the end of the preceding initial value problem.

1.5 Thesis Outline

In the chapters that follow, we will consider the development of a general-purpose combined discrete/continuous process simulation package based on the mathematical formulation described above. The next three chapters are concerned with a formal basis for the description of this class of problems, and the articulation of this in the form of a simulation language.

Chapter 2 concentrates on the description of the underlying physical behaviour of processing systems. A review of recent work in this area is included, which emphasises the important issues of managing model complexity and promoting model reusability. Although this physical behaviour is predominantly continuous, discrete changes are also common. A general formalism for the description of this combined discrete/continuous behaviour is therefore considered.

In chapter 3, the modelling of the external actions imposed on a processing system by its environment is considered. An examination of the fundamental characteristics of most industrial processing systems is intended to yield a more convenient formalism for the interaction of these discrete actions with the predominantly continuous physical behaviour. The issues of complexity management and reusability are raised again. Chapter 4 then brings together these two disparate categories of information to form the description of individual dynamic simulation experiments.

By this point, the first main objective of the thesis will have been addressed. Chapter 5 therefore contains an overview of the current implementation of a new modelling package based on the simulation language introduced in the preceding chapters. The special demands of combined discrete/continuous process simulation, particularly in the form of arbitrary structural modification of the continuous model, and the need for interactive responses from the modelling package are examined in detail.

Chapter 6 employs this prototype modelling package to demonstrate the usefulness and necessity of combined discrete/continuous process simulation through a series of detailed examples covering the entire range of process operations.

The thesis concludes in chapter 7 with a discussion of the contribution made, and suggestions for future research.

Chapter 2

Combined Discrete/Continuous Modelling - Model Entities

Detailed dynamic simulation requires a mathematical model that describes the time dependent behaviour of the system under investigation. As already discussed in chapter 1, models of the physico-chemical mechanisms that characterise the continuous time dependent behaviour of most industrial processes can be expressed naturally in terms of differential-algebraic equations (DAEs) or partial differential-algebraic equations (PDAEs). The solution of these equations determines the time trajectories of the variables describing the system, from which the dynamic behaviour of such systems can predicted.

This chapter is concerned with the development of a high level declarative language for the description of systems modelled by DAEs. This language will enable the specification of the continuous aspects of a combined discrete/continuous simulation description. Ideally, the language should enable an engineer to declare the describing equations of a system in a manner that is completely decoupled from the details of the individual activities for which the model may subsequently be employed, and the procedural knowledge required to solve the equations. Within the scope of this thesis, the language is *not* intended to facilitate the definition of mathematical models other than those based on DAEs, or to include rigorous model documentation (Stephanopoulos *et al.*, 1990a), although scope for the addition of both these features exists.

The major problem encountered during the development of a continuous model for a processing system is the size and complexity of such systems. The proposed modelling language must provide structures for the management of this complexity that correspond to the engineer's perception of the structure of processing systems. These structures must also encourage the development of models that are both correct and reusable. Although correctness is the prime objective of any modelling exercise, reusability is also an issue of great importance. A model that is reusable can be repeatedly used for many different purposes, thus making the best use of the original effort required for model development.

The chapter begins with a review of recent work concerned with the development of the large continuous models required for industrial processing systems. This leads into a discussion of combined discrete/continuous models, and the conclusion that the existing view of continuous models must be extended to include certain discrete elements.

Drawing on these ideas, the elements of the proposed language that enable the description of primitive models are introduced. Finally, language structures for managing model complexity and model reuse are presented.

2.1 Continuous Modelling of Industrial Processes

It is practically impossible for an engineer to grasp simultaneously all the knowledge required to develop the model of a large system. To manage this complexity, the engineer must analyse the structure of the system in order to divide it into a set of connected components. The details of each component may then be considered as a problem in its own right, independently of the details of the complete problem. This methodology is reflected in the universally accepted representation of process plant as a flowsheet of interconnected unit operations.

Many of the components of complex systems appear repeatedly within the same structure. For example, pumps and valves may appear many times in the flowsheet of a complex process. In order to avoid repeated modelling of these identical components, all the languages reviewed here support some form of model *type* (or *class*) concept. A model type declares the behaviour of a set of components with similar characteristics. When a component is actually required to form part of a larger structure, an instance of the model type is created and inserted in the structure.

One of the earliest continuous simulation languages designed to support the structural decomposition methodology is DYMOLA (Elmquist, 1978). The behaviour of a DYMOLA model may be declared in terms of the continuous connection of a set of submodels. This is directly analogous to the decomposition of a complex system into interconnected components. The decomposition is taken to its logical conclusion by allowing any submodel also to be declared in terms of the continuous connection of submodels, enabling hierarchies of arbitrary depth to evolve. Elmquist terms this *hierarchical submodel decomposition.*

The descriptive power of DYMOLA is augmented further by the ability to declare relationships between submodels to represent the connection mechanisms, such as pipes or electrical wires, that occur in physical systems. The engineer building a complex model from existing components through the use of these connection mechanisms is thereby freed to concentrate on the structure of the system under consideration.

Hierarchical submodel decomposition also promotes reuse of models. Many of the components of complex systems, such as pumps, capacitors, or even complex structures such as distillation columns, are common to a wide range of systems. The models of these components are therefore suitable for storage in libraries of components for later reuse. When constructing a model of a new system, many submodels may be reused from these libraries.

DYMOLA was designed to be a completely declarative language. Unlike languages descended from the CSSL standard (Strauss, 1967), the DAEs describing the time dependent behaviour of models do not have to be declared in assignment form. In order to solve equations declared in this manner, Elmquist proposed analytical rearrangement of the equations to assignment form during compilation of the simulation description. However, the DAEs that are required to model the continuous time dependent behaviour of most process industry applications cannot always be rearranged into this form. This has led to interest in simultaneous methods for the solution of DAEs amongst the process engineering community.

An early example of a declarative language specifically designed for the continuous modelling of chemical processes is the SpeedUp input language (Perkins and Sargent, 1982; Prosys, 1991).

The representation of process plant as flowsheets is a natural application for Elmquist's hierarchical submodel decomposition, which is supported to a limited extent by SpeedUp. Using the input language, primitive MODELs may be connected together to form MACROs that represent complex augmented unit operations such as distillation columns. Both MODELs and MACROs may then be connected together to form the top level of the model hierarchy, the FLOWSHEET. This hierarchy is, however, limited to three levels because it is impossible to define MACROs in terms of the interconnection of other MACROs. Moreover, the FLOWSHEET is unique to a particular simulation, preventing reuse of this top level of the hierarchy.

SpeedUp provides two connection mechanisms. STREAMs represent the flow of material and energy in the pipes between unit operations, while CONNECTIONs represent the electrical or pneumatic information signals associated with the control system of any process.

In recent years, the increasing awareness of ideas popularised by the object-oriented programming paradigm has prompted the development of a new generation of continuous modelling languages. Those languages designed for process engineering applications will be reviewed here.

The common feature of all these new languages is the enhancement of the model type (or class) concept through model *inheritance*. Inheritance enables the declaration of a new model type in terms of the extension or restriction of a previously declared type. Thus, inheritance is another mechanism by which model reusability and consistency can be encouraged. Reusability is promoted because existing models may now be easily extended for new applications, and consistency is guaranteed because common information need only be declared and validated once.

OMOLA (Andersson, 1989; Andersson, 1990; Mattsson and Andersson, 1990) builds on the hierarchical submodel decomposition of DYMOLA with the introduction of an object-oriented modelling framework. An important feature is the ability to use inheritance in the declaration of both model types and complex connection mechanisms.

Although model reuse through model parameterisation is a feature of all the languages reviewed here, the design of OMOLA pays particular attention to this issue. The use of parameters extends the model type concept by enabling a model type to describe the behaviour of a wide range of similar, albeit not identical, components. The values assigned to the parameters of an individual model instance then customise it to its application.

Finally, OMOLA introduces the representation of model behaviour as a number of different mathematical realisations (such as DAEs, transfer functions, and state space descriptions), rather than a single realisation.

The application of the concepts embodied in OMOLA has been demonstrated through the development of the continuous model of a complete chemical process (Nilsson, 1989a; Nilsson, 1989b). This application has also highlighted some shortcomings of OMOLA, particularly in the representation of the regular structures, such as distillation tray sections, common to many unit operations.

ASCEND (Piela, 1989; Piela *et al.*, 1991) is a language for the declaration of continuous mathematical models, particularly models of chemical processes. A complex model type is constructed from primitive types using a range of language operators. The model eventually developed should represent a well-posed mathematical problem that can then be submitted for solution to a suitable numerical method. Both hierarchical submodel decomposition and model inheritance are supported by the language operators. However, the authors note that it is not strictly necessary, or even suitable, for a modelling language to adopt the entire object-oriented paradigm.

The most interesting features of the ASCEND language are three of the operators introduced for complex model type development:

- The IS_REFINED_TO operator is employed to refine an attribute to a new type that is developed by inheritance from its original type. For example, it is possible to refine the generic tray attributes of a distillation column model to sieve plate or bubble cap tray models for the purposes of a particular simulation.

- Attributes that are associated using the ARE_ALIKE operator adopt the most refined of their types. Used in conjunction with the IS_REFINED_TO operator, it can propagate type changes through entire structures. In the distillation column example above, it is only necessary to apply the IS_REFINED_TO operator to a single tray in order to refine all the trays in the column, provided these are all associated by the ARE_ALIKE operator.

- Attributes associated using the ARE_THE_SAME operator are merged into a single attribute of the most refined type. This attribute can be referenced by any of the identifiers of the original attributes that were merged. The operator is used, for example, to form the continuous connections between submodels, which has the effect of reducing the total number of variables and equations present in a system model by eliminating the equality constraints sometimes used for this purpose.

MODEL.LA (Stephanopoulos *et al.*, 1990a; Stephanopoulos *et al.*, 1990b) is presented as a language suitable for the description of models to be used for the entire range of process engineering activities. The language is fully object-oriented and hierarchical submodel decomposition is represented in five levels of abstraction; plant, plant-section, augmented unit, unit and sub-unit. The language has been integrated with the DESIGN-KIT package (Stephanopoulos *et al.*, 1987), an object-oriented environment for computer-aided process engineering.

An important feature of MODEL.LA that distinguishes it from the other languages reviewed in this section is the manner in which models of individual unit operations are declared. All the other languages require a basic unit operation model to be declared in terms of mathematical relationships between system variables. In contrast, MODEL.LA only requires a declaration of the relationships required (such as mass or energy balances) and a set of assumptions concerning physical and chemical phenomena. The model executive can then automatically generate the correct mathematical relationships from this information. This approach has many advantages, including rigorous model documentation and consistency checking, and greater support for the inexperienced modeller, but may ultimately be restricted by the scope of the knowledge base from which equations are automatically generated. Currently, researchers are attempting to combined the advantages of high level model description with the flexibility of access to individual equations (Hutton *et al.*, 1991; Vázquez-Román, 1992).

In addition, MODEL.LA introduces a framework for multifaceted modelling. This recognises the need to consider a process model at several different levels of abstraction during the evolution of a design. A multifaceted model consists of an arbitrary number of facets that exchange and share information concerning the physical object under consideration. The facet used for a particular activity is determined by the level of abstraction required.

The remainder of this chapter is concerned with the development of a high level modelling language for the declaration of the continuous aspects of a combined discrete/continuous simulation. The design of this language incorporates many of the ideas introduced above. However, the language is deliberately focussed on the description of dynamic simulations and does not address many other important modelling issues raised by these workers.

2.2 Combined Discrete/Continuous Modelling

Before the design of the modelling language can be detailed, the special modelling requirements of combined discrete/continuous systems must be identified and addressed. A consideration of these issues leads to a natural extension of the traditional concept of a continuous model to that of a combined discrete/continuous model: a continuous model that contains certain discrete components.

A combined simulation progresses through periods of continuous simulation, characterised by the solution of the describing equations, interspersed by instantaneous events, which *may* result in some form of discrete change. A detailed characterisation of these discrete changes was given in section 1.4. It is proposed that the discrete changes a processing system may experience can be split naturally into two distinct categories:

- Those that are a result of the physico-chemical mechanisms that characterise the continuous time dependent behaviour of a system and thus occur independently of any direct external intervention or interference with the system. Examples include the transitions from laminar to turbulent to choked flow in a pipe, phase changes, and flow (or not) over a weir. These changes are termed *physico-chemical discontinuities.*

- Those that result directly from the interaction of a system with its environment, such as external disturbances and control actions. Examples include the opening or closing of manual valves, switching a control loop from manual to automatic control, or the action of a discrete controller at the end of each sampling interval. These changes will be dealt with in detail in the next chapter.

Discrete changes that result from physico-chemical mechanisms are most naturally declared, together with the describing equations, within the system model. In this manner, all the knowledge concerning the physical behaviour of a system can be encapsulated in a single entity. In order to include these physico-chemical discontinuities, the concept of a combined discrete/continuous model is introduced. A consideration of the nature of these discontinuities and the mechanisms by which they occur will lead to a formalism for these combined models.

An examination of the nature of physico-chemical discontinuities leads to the conclusion that they can be modelled by dynamic changes to the describing equations of a system (see section 1.4). For example, the transition from laminar to turbulent flow in a pipe involves a discrete change in the relationship between the Fanning friction factor and the Reynolds number. Similarly, when the liquid level in a vessel rises above a weir, the flow over the weir is related to the liquid level above the weir, whereas before it was zero. At a phase change, not only does the functional form of the modelling equations change, but it is also possible that the number of variables, and therefore equations, required to describe the system changes.

A continuous model is composed of a set of variables that describe the dynamic behaviour of a system, and a set of equations that relate these variables. At any given point in time these two sets characterise the current *state*[1] of the model. Many models have a unique state; the composition of the sets of variables and equations associated with them remains unchanged (although, of course, the *values* of the variables will change with time). On the other hand, models that include physico-chemical discontinuities must be declared in terms of several states, each characterised by a different set, and possibly number, of describing equations. For example, the model of a flash drum will be declared in terms of at least three states corresponding to the presence of both vapour and liquid phases, liquid phase only, and vapour phase only. During a simulation the active state of a model determines the equations that describe the system at that point in time. Events during simulation may result in changes in the active state of a model, and the attendant changes to the describing equations.

In the process engineering community, the occurrence and explicit declaration of physico-chemical discontinuities in modelling equations was first discussed in the context of steady-state simulation (Westerberg and Benjamin, 1985). In this case, the steady-state solution procedure must search for the correct model state as the calculation proceeds. This has been represented formally by the CASE structured statement of ASCEND (Piela, 1989), where a series of globally applied logical conditions are combined in a truth table to determine the active state. The occurrence of physical discontinuities during a dynamic simulation is also reflected in the design of several existing continuous process simulation packages. For example, the IF equation of SpeedUp (Pantelides, 1988b) defines two system states linked by a logical condition. While the condition is true, the first state will be active, but if the condition becomes false, the second state will become active. Multiple states may be accommodated by nesting these IF equations.

It should be recognised, however, that neither formalism provides a sufficiently

[1]This should not be confused with the dynamic state of the system, i.e. the values of the differential variables $\mathbf{x}(t)$ in equation 1.3.

general representation of the mechanisms that result in dynamic transitions between model states. This will be illustrated by three simple examples. In figure 2.1 models containing physico-chemical discontinuities are represented as digraphs, with nodes denoting model states and arcs signifying instantaneous transitions between these states.

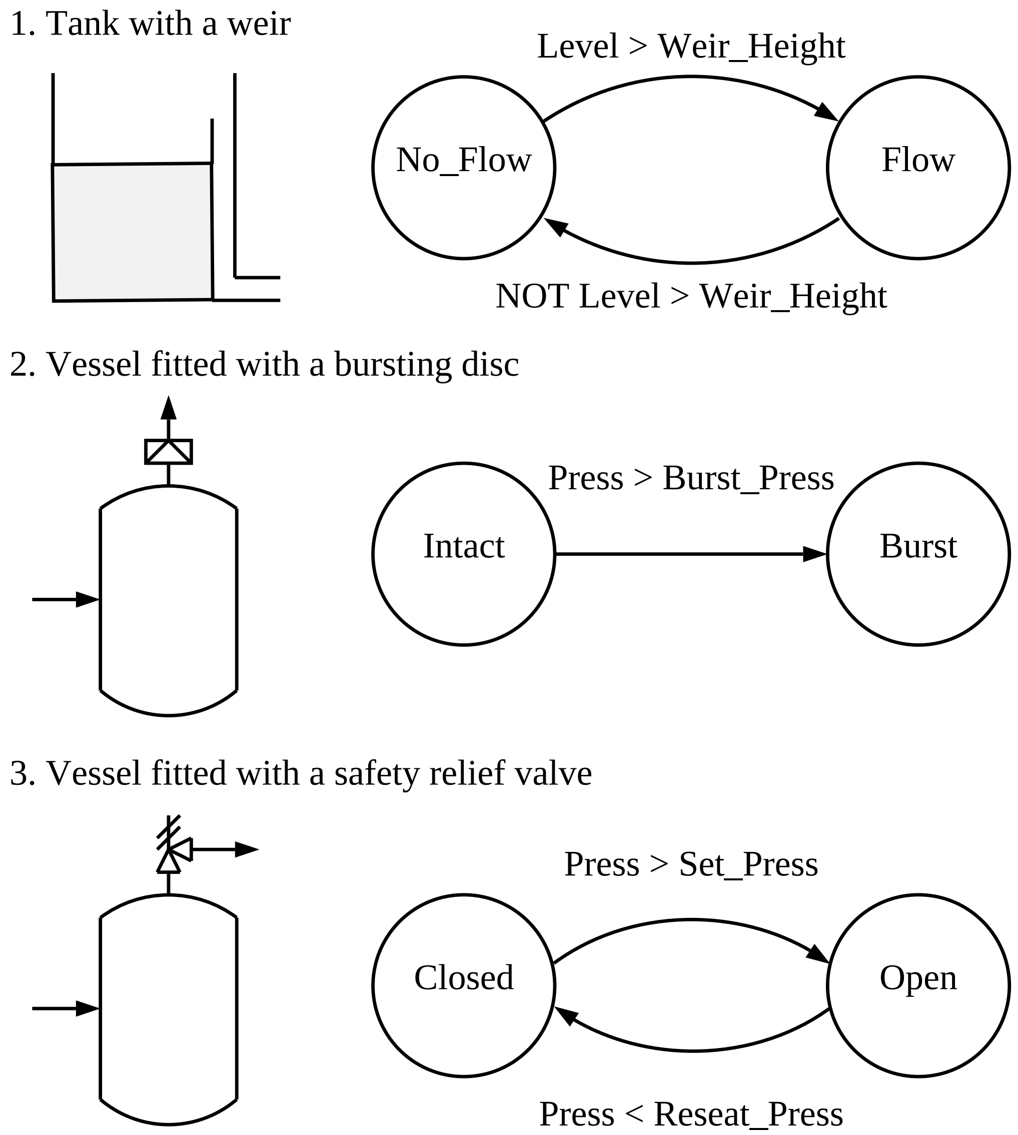

Figure 2.1: Examples of Models Containing Physico-Chemical Discontinuities

The first example is that of a vessel containing an overflow weir. The model can be declared in terms of two states, corresponding to whether or not fluid flows over the weir. A transition from the No_Flow to the Flow state occurs when the liquid level rises above the weir, and a transition from the Flow to No_Flow state occurs when the liquid level drops below the weir. This is termed a *reversible discontinuity* because the condition for one state transition is the negation of the condition for the other. Consequently, the two state transitions can be characterised by a single logical condition, a fact that is reflected by the IF equation of SpeedUp and its counterparts. This is also the limitation of these representations: they are only suitable for declaration of reversible discontinuities.

The second example is that of a vessel fitted with a bursting disc. The bursting disc can either be intact, with no gas flow from the vessel, or burst, with gas venting from the disc to a flare stack. The model of the vessel is thus declared in terms of these two states with only one possible transition between them occurring when the pressure in the vessel rises above the set pressure and the disc shatters. The disc can never return to the intact state once it has shattered: the plant must be shut down and the disc replaced. This is termed an *irreversible discontinuity* and clearly demonstrates the limitations of the IF equation representation, which would incorrectly return the disc to the intact state as soon as the pressure dropped below the set pressure.

The final example is that of a vessel fitted with a safety relief valve. This valve can be either open or closed, which again corresponds to two model states. A transition from the closed to the open state occurs when the pressure in the vessel rises above the set pressure, and a transition from the open to the closed state occurs when the pressure falls below a (lower) reseat pressure. This is termed an *asymmetric discontinuity* because, although there are possible transitions in both directions, the transition conditions that must be satisfied are not directly related. Other examples of systems that contain asymmetric discontinuities include a thermostat, and the mechanism employed to periodically flush public urinals. Again, the IF equation representation cannot be employed to declare asymmetric discontinuities in a natural manner.

A formalism that does give a sufficiently generalised representation of these discontinuities was first proposed by the system simulation community (Pearce, 1978), and has been implemented in the general-purpose combined simulation languages COSY (Cellier and Bongulielmi, 1980), SYSMOD (Smart and Baker, 1984), and COSMOS (Kettenis, 1992). As stated earlier, the model of a system is declared in terms of a finite number of distinct states, one of which will be active at any point in time. In general, each state S is characterised by:

- a set of equations, $\mathbf{f}$.
- a set of variables, $\mathbf{x}$, $\dot{\mathbf{x}}$, $\mathbf{y}$ and $\mathbf{u}$.
- a (possibly empty) set of *transitions* to other states.

and a transition is characterised by:

- an initial state, S^I.
- a terminal state, S^T.
- a logical condition, $l(\mathbf{x}^I, \dot{\mathbf{x}}^I, \mathbf{y}^I, \mathbf{u}^I, t)$, expressed in terms of the variables in the initial state S^I.
- a set of relationships allowing the determination of consistent initial values for the variables in S^T from the final values of the variables in S^I.

If we assume that the set of variables is the same for all states, and that the differential variables $\mathbf{x}$ are continuous across all transitions, consistent initial values for state S^T can be determined automatically from the mapping:

$$\mathbf{x}^T(t^*) = \mathbf{x}^I(t^*) \tag{2.1}$$

where t^* is the earliest time at which $l(\mathbf{x}^I, \dot{\mathbf{x}}^I, \mathbf{y}^I, \mathbf{u}^I, t^*)$ becomes true.

Each model that uses this formalism is equivalent to a deterministic finite automaton (Hopcroft and Ullman, 1979), albeit one whose states can be immensely complex, according to the nature of the describing equations. A system containing an irreversible discontinuity includes at least one state with an empty transition set.

From a practical point of view, many of the describing equations of a system will remain unchanged regardless of the state the system is in. In order to avoid duplication of these equations in the declaration of each state, a model should be able to contain an *invariant* set of equations. Also, the *variant* equations of a model can often be divided into a series of smaller groups each characterised by an independent finite automaton. For example, the model of a vessel fitted with both a safety relief valve and a bursting disc would be declared in terms of the invariant balance equations and two finite automata, determining the flow from the relief valve and the bursting disc respectively.

The formalism for a combined discrete/continuous model is thus complete. Such a model will consist of *variant* and *invariant* sets of equations, either of which may be empty. The variant set of equations are described in terms of one or more *finite automata*. Each finite automaton is characterised by one or more *states*. At any point in time, the describing equations of a model are the union of the equations of the active states of the finite automata and the invariant equations. This definition is also recursive; each state of a finite automaton can be declared with variant and invariant parts. Figure 2.2 demonstrates how a combined model can be represented diagrammatically as a digraph.

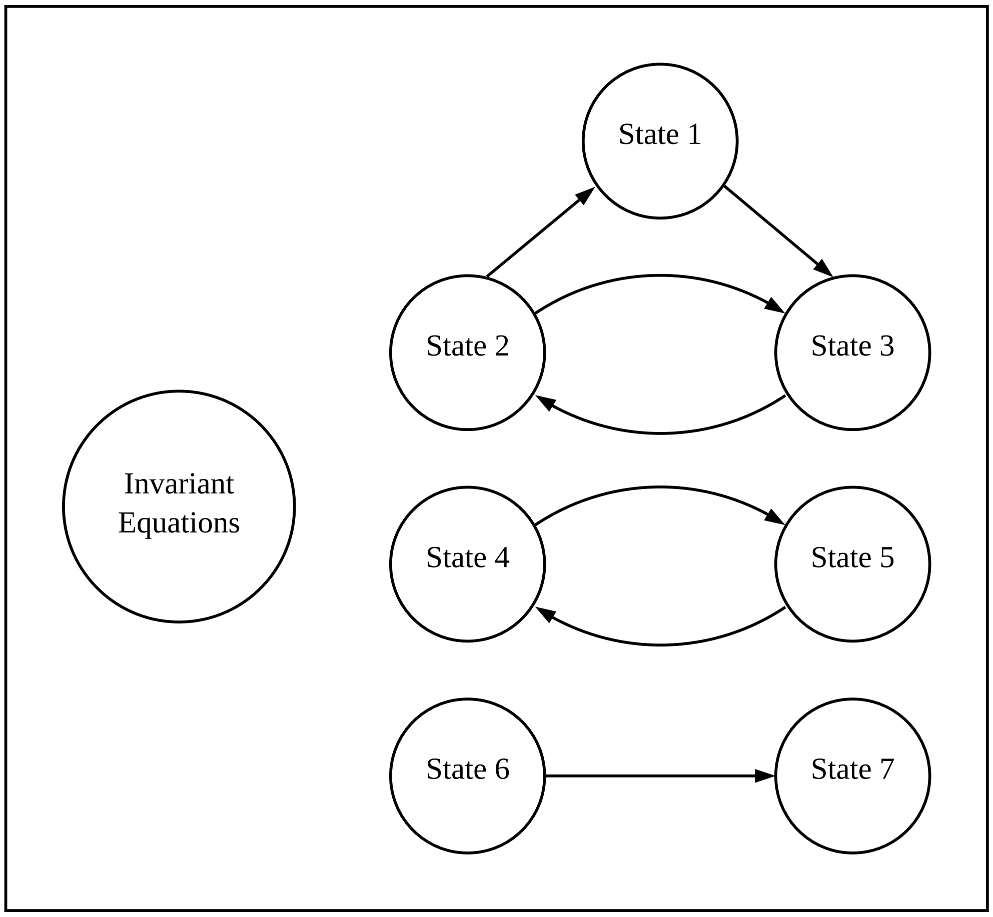

Figure 2.2: Digraph Representation of a Combined Discrete/Continuous Model

This formalism will be used in the next section as a basis for the development of syntactic structures for the declaration of combined discrete/continuous models.

2.3 Primitive Model Entities

This section is concerned with the development of language structures for the declaration of *primitive model entities*. As will be seen later in this chapter, these can form the basis for the construction of higher level model entities.

A model entity captures all the knowledge regarding the physical behaviour of a system, including the equations that determine the continuous time dependent behaviour and any physico-chemical discontinuities that may change the functional form of these equations. Once a model entity has been declared, the information it contains may be used by instantiating it in the description of an individual activity.

The review at the beginning of this chapter has already pointed out that high level declarative languages for the description of continuous models, such as that employed by SpeedUp (Perkins and Sargent, 1982), have now been established for several years, and that recent developments, such as the ASCEND system (Piela *et al.*, 1991), have introduced new concepts first popularised by the object-oriented programming paradigm. The language structures described here are based on these existing packages, although a number of important features, related to combined discrete/continuous simulation in particular, have been introduced, and others have been significantly enhanced.

A model entity is a complex data type that encapsulates a declaration of the following information regarding the structure and physical behaviour of a system:

- A set of variables that *describe* the time dependent behaviour of the system.
- A set of relationships between these variables, in the form of DAEs, that *determine* the time dependent behaviour of the system, including any physico-chemical discontinuities that may cause discrete changes to their functional form.
- A set of time invariant parameters that characterise the system and promote reuse of the model entity.
- A set of complex terminals (streams) that represent the model's interface with its environment. These terminals may subsequently be used in the construction of more complex structures involving the model entity as a component.

Each item of information, such as a variable, is termed an *attribute* of the model entity. An attribute must have a unique identifier associated with it, by which the attribute may be referenced, for example, in expressions. An exception to this rule is made in the case of equation attributes, for which the identifier is only optional. The set of attributes encompass the information declared within a model entity.

Regular structures that provide an aggregated description of a number of related items of information are a feature of most programming and modelling languages, and can be used to model conveniently many physical quantities or phenomena that occur frequently in processing systems. Examples include:

- A variable attribute that represents the component flowrates in a multi-component process stream.
- A parameter attribute that represents the stoichiometric coefficients of a chemical reaction.
- A terminal attribute that represents the inlet streams of a unit operation that mixes an arbitrary number of process streams.

In reflection of this, all the attributes of a model entity may be declared as a regular structure, or *array*, of a base type. Attribute arrays may have an arbitrary number of *dimensions*,[2] although a particular implementation may impose a practical upper limit on the number of dimensions. The total number of scalar quantities, or *elements*, represented by an attribute array is determined from the product of the number of elements in each dimension of that array. The number of elements in each dimension is declared in terms of a scalar integer expression involving integer constants and/or any previously declared integer parameters of the model entity in question (e.g. `Flow_In AS ARRAY(3,NoStream+1) OF REAL`).[3]

References to array attributes may be made in several different fashions. For example, a reference to an entire array is made through use of the attribute identifier alone, and a reference to an individual element of an array is made by an explicit index to the element

[2]It is important here to distinguish between the dimensions of an array or regular structure, and the fundamental physical dimensions of a quantity, such as mass or length.

[3]In all language examples given in this thesis, upper case is employed to denote standard keywords, whereas user-selected identifiers are shown in mixed case.

in question. This index is determined from a list of scalar integer expressions enclosed by brackets following the attribute identifier (e.g. `Flow_In(2,NoStream-1)`). Each expression in this list represents an index into one dimension of the array. Individual elements of a dimension are indexed from one to the number of elements in that dimension.

A reference to a subset of the elements in one or more dimensions of an array is termed a reference to a *slice* of that array. The elements that are included within a slice is again determined by a list of references into each dimension of the array in question enclosed by brackets. A subset of the elements in a particular dimension is denoted by two scalar integer expressions separated by a colon, representing the lower and upper bounds of the reference into that dimension respectively (e.g. `Flow_In(2:3,1:NoStream)`). The value of the upper bound must be greater than or equal to that of the lower bound, and both values must lie with the lower and upper indices of the dimension itself. A reference to an entire dimension is made by leaving a blank, so a list of blanks enclosed in brackets and separated by commas is identical to the use of an attribute identifier alone. A reference to an individual element is again made by a single scalar integer expression (e.g. `Flow_In(2:3,1)`).

The declaration of a model entity begins with the keyword MODEL followed by a unique identifier by which it may be referred to globally. The remainder of the declaration is split into a series of optional *sections* in order to gather all the attributes belonging to a particular category in one place and thus aid model documentation. The following text details how each category of attribute is declared.

2.3.1 Parameter Attributes

The merits of *parameter attributes* that extend the notion of a model type to describe a broad range of similar, although not identical, components have already been discussed. Parameter attributes are distinguished from variable attributes by a characterisation as time invariant quantities that are not, under any circumstances, determined from the results of a calculation. Quantities such as the number of elements in an array dimension, Arrhenius coefficients, and stoichiometric coefficients are therefore suitable parameter attributes. Otherwise, the degree of flexibility, particularly from the point of view of regular structures, allowed in the declaration of parameter attributes is identical to that of variable attributes.

The categorisation of certain real quantities as parameter attributes as opposed to variable attributes is obviously rather tenuous. Designating a quantity as a parameter attribute has the advantage of reducing the total number of variables in the system, but there is the disadvantage that this quantity may never be treated as an unknown to be calculated in any future use of the model. Consider, for example, the time invariant quantities that characterise the size and geometry of a vessel. From the point of view of dynamic simulation, these quantities can always be designated as parameter attributes, although from the point of view of steady-state design calculations performed with the same model, these quantities may be considered unknowns under certain circumstances.

The PARAMETER section is employed for the declaration of the parameter attributes of a model entity. All parameter attributes are declared as instances of a parameter type. Parameter declarations may optionally include the assignment of default values, which may be specified in terms of expressions involving previously declared parameters. The current language definition offers the following parameter types:

- Real, integer or logical *values*.
- Real, integer of logical *expressions* that are passed in symbolic form.
- Model type parameters, which are explained in section 2.5.2.

The difference between value and expression parameters is rather subtle. In fact, whenever a model entity is instantiated, an expression of the appropriate type may be assigned to *both* categories of parameters. The actual difference lies in the fact that value parameters are replaced by the *value* of the expression at the time of instantiation of the model entity and remain constant thereafter. On the other hand, an expression parameter is replaced symbolically by the expression itself, in a manner somewhat similar to parameter transmission by name in Simula procedures (Birtwistle *et al.*, 1979). Figure 2.3 demonstrates how this range of parameter types may facilitate the parameterisation of a reactor model according to the reaction(s) taking place.

Clearly, before an instance of a model entity can actually be used in a simulation, all of its parameters must be assigned appropriate values.[4] As we shall see later, this can

[4]Unless they have been given default values

```
MODEL Reactor

PARAMETER
  NoComp                 AS INTEGER
  NoReactions            AS INTEGER DEFAULT 1
  Stoich                 AS ARRAY(NoReactions,NoComp) OF INTEGER
  K0, Activation_Energy  AS ARRAY(NoReactions)        OF REAL
  Reaction_Rate          AS ARRAY(NoReactions)        OF REAL_EXPRESSION
  G                      AS REAL
```

Figure 2.3: Example PARAMETER section

be done in higher-level models (see section 2.4.3) or in process entities describing entire simulations (see section 4.2). However, a model is also capable of fixing the values of some of its own parameters.

The SET section enables any of the parameter attributes declared in the PARAMETER section to be assigned fixed values that may not be redefined on instantiation of a model entity. These values may be determined from expressions of the appropriate type involving other parameters, although circular value assignments will be automatically detected and rejected. For example, the declaration:

```
SET
  G := 9.81 ;
```

effectively renders the parameter attribute G a constant with a fixed value in all instances of the above model.

2.3.2 Variable Attributes

Variable attributes represent the quantities describing the time dependent behaviour of a system, such as the temperature or the material holdup of a vessel. It is advantageous to group these quantities according to *variable types* that define such properties as the range of physically meaningful values, the fundamental physical dimensionality, and the units. The declaration of variables types is discussed in appendix A.

The VARIABLE section contains a declaration of *all* the variables that describe the time dependent behaviour of the system represented by the model entity. Variable attributes must be declared as instances of already declared variable types. An example VARIABLE section is shown in figure 2.4.

```
VARIABLE
  Flow_In, Flow_Out                     AS ARRAY(NoComp) OF Flowrate
  HoldUp                                AS ARRAY(NoComp) OF Moles
  Vessel_Temperature                    AS Temperature
```

Figure 2.4: Example VARIABLE section

2.3.3 Stream Attributes

Streams attributes are subsets, not necessarily disjoint, of the variables describing the time dependent behaviour of a system. They represent a system's interface with its environment, and are useful in specifying the complex connection mechanisms that exist between different components of a physical system (see section 2.4), or for displaying and manipulating simulation results.

The STREAM section is used to declare stream attributes, which must be declared as instances of already declared *stream types* (see appendix A). This declaration also includes a specification of the subset of variable attributes that is to be included in the stream. The number and types of the variable attributes in a stream must normally match directly those in the stream type declaration. This type conformance is relaxed for variable attributes that have been declared as AnyType in the stream type declaration (see appendix A). An example of a STREAM section is shown in figure 2.5.

```
STREAM
  Inlet  : Flow_In,  Temp_In,  Press_In,  Enth_In    AS MainStream
  Outlet : Flow_Out, Temp_Out, Press_Out, Enth_Out   AS MainStream
```

Figure 2.5: Example STREAM section

It should be noted that no assumptions concerning the *dimensionality* of the variable attributes included in a stream are made in a stream type declaration (see appendix A). Therefore, a slice or an entire array of variable attributes may appear in any field of a stream attribute, provided the base type of the array matches the variable type of the corresponding field in the stream type. For instance, the following is a valid stream declaration:

```
STREAM
  Inlet : Flow_In(1:NoComp-1),Temp_In,Press_In,Enth_In  AS MainStream
```

Stream attributes may themselves be declared as arrays of the basic stream types. For instance, a mixer involving several inlet streams could have a corresponding stream declaration of the form:

```
STREAM
  Inlet : Flow_In, Press_In    AS ARRAY (NoStream) OF MainStream
```

Each variable attribute in a k-dimensional stream must have at least k dimensions, and each of its *first* k dimensions must have exactly the same number of elements as the corresponding dimension of the stream. For instance, a possible declaration of the variables in the above example would be:

```
VARIABLE
  Flow_In                      AS ARRAY (NoStream,NoComp) OF Flowrate
  Press_In                     AS ARRAY (NoStream)        OF Pressure
```

This rule allows a natural identification of the variable attributes to be associated with each element of the stream array.

2.3.4 Equation Attributes

The EQUATION section contains the declaration of the *equation attributes* of a model entity. These form part of the set of DAEs that determine the time trajectories of the variables already declared in the VARIABLE section. In general, this set of equations will be under-determined with respect to these variables, but will include the declaration of any physico-chemical discontinuities that may result in dynamic changes to their functional form.

```
EQUATION
  MassBalance   AS $Holdup = Flow_In - Flow_Out + Total_Reaction_Rate ;
  EnergyBalance AS $U_Holdup = Enth_In - Enth_Out + Total_Reaction_Heat ;
```

Figure 2.6: Example EQUATION section

A simple equation attribute is declared using a high level symbolic language in terms of an equality constraint between two real expressions. These real expressions may be written in terms of the following primitive operands:

- Real or integer constants.
- References to real or integer value or expression parameter attributes.
- References to variable attributes.[5]
- References to the built-in system variables representing time.
- Built-in *functions*.

These operands may be related by the real operators $+, -, *, /$ and ˆ (raising to a power) common to most programming and modelling languages. The raising to a power operator has the highest precedence, followed by the division and multiplication operators, and the addition and subtraction operators have the lowest precedence. Brackets may be employed in order to alter these precedence rules for certain operations. All equation attributes can optionally be associated with a unique identifier. These identifiers are required for some of the more sophisticated manipulations of the continuous model allowed during dynamic simulation (see chapter 3). In addition, they are often useful for diagnostic purposes. Figure 2.6 demonstrates how mass and energy balance equations might be declared in a particular model.

Arrays of equation attributes are not declared explicitly, but are implied by their declaration in terms of expressions involving references to arrays or slices of variable and/or

[5]The symbol $ preceding a variable attribute identifier indicates the derivative with respect to time of the latter.

parameter attributes. The dimensionality[6] of a unary expression is the same as that of its operand. For binary expressions, three cases are distinguished:

- If both operands are scalar, then the expression is scalar.
- If only one operand is scalar, then the expression adopts the dimensionality of the other operand.[7]
- If neither operand is scalar, then both operands must be of the same dimensionality, which is also adopted by the expression itself.[8]

A simple equation is declared as two real expressions separated by an equality operator (denoted by '=' or 'IS'). The dimensionality of the equation itself is obtained by applying the rules for binary expressions to the equality operator.

Expressions may include built-in functions as operands. A function performs a mathematical operation on its arguments that would be difficult or even impossible to declare using the language operators. At present, there are two categories of built-in function:

- *Vector functions* take a single argument and return a set of values with dimensionality equal to that of the argument.
- *Scalar functions* take an arbitrary number of arguments of arbitrary dimensionality and return a scalar value.

All function arguments may themselves be expressions of the appropriate type. Table 2.1 contains a summary of the vector functions currently included in the language definition. Similarly, table 2.2 contains a summary of scalar functions. The implementation enables this set of built-in functions to be extended easily as needs arise.

If any of the arguments of a scalar function are references to an array or a slice, the operation is applied to the entire array or slice. For example, if an array is passed as an

[6]The dimensionality of any entity is defined as the number of dimensions and the number of elements in each dimension.

[7]Each element of this expression is obtained by the binary operation between the scalar operand and the corresponding element of the other operand.

[8]Each element of the resulting expression is obtained by the binary operation between the corresponding elements of the two operands.

Identifier	Function
ABS	The absolute value of the argument
SGN	The sign of the argument
SQRT	The square root of the argument
ERROR	The error function of the argument
SIN	The sine of an argument in radians
COS	The cosine of an argument in radians
TAN	The tangent of an argument in radians
ASIN	The arcsine in radians of the argument
ACOS	The arccosine in radians of the argument
ATAN	The arctangent in radians of the argument
SINH	The hyperbolic sine of the argument
COSH	The hyperbolic cosines of the argument
TANH	The hyperbolic tangent of the argument
EXP	The exponential of the argument
LOG	The natural logarithm of the argument
LOG10	The logarithm to base 10 of the argument
INT	Truncate real argument towards negative infinity

Table 2.1: Table of Built-in Vector Functions

Identifier	Function
SIGMA	The sum of the arguments
PRODUCT	The product of the arguments
MIN	The smallest argument
MAX	The largest argument

Table 2.2: Table of Built-in Scalar Functions

argument to the function SIGMA, a scalar value equal to the sum of all the elements of that array will be returned (e.g. `Total_Flow_Out = SIGMA(Flow_Out);`). All function identifiers may be used in the declaration of model attributes, thereby locally overriding the built-in function definitions.

2.3.5 Structured Equations

Some or all of the equation attributes of a model entity may also be *structured equations* which define complex operations on the equality constraints declared in simple equations. These operations include the dynamic changes to the describing equations that occur as a result of physico-chemical discontinuities. Below, we introduce the structured equations currently included in the language definition.

2.3.5.1 The FOR Equation

As described in section 2.3.4, the language already allows the implied declaration of arrays of equation attributes. Situations do exist, however, in which these rules are not sufficient, e.g. in defining matrix-vector multiplications.

One solution to this problem is to introduce a special language operator for each non-scalar operation a user might conceivably want to perform. However, this approach cannot guarantee completeness and has the disadvantage of significantly increasing the size of the basic language definition. Alternatively, a syntactic structure for the expression of complex array notation in an explicit manner can be introduced. It is this second approach that is adopted by the introduction of the FOR structured equation. This equation defines the repeated expansion of a list of equations for a sequence of values assigned to a special integer *counter* variable. Two examples of how complex array manipulations may be expressed through this structured equation are illustrated in figures 2.7 and 2.8. Note that text following a # symbol is always treated as commentary.

Figure 2.7 demonstrates how a FOR equation may be used in a unit operation that is parameterised by its number of inlet streams. In particular, the FOR equation is necessary for summing the input streams on a component-by-component basis. Note that the desired effect could not be obtained simply through `SIGMA(Flow_In)` because this would sum the flowrates of *all* components of all streams together, returning a scalar!

Figure 2.8 illustrates one possible model for an isothermal liquid phase tubular reactor. The partial differential equation that normally describes the mass balance of such a system has been reduced to a set of ordinary differential equations with respect to time by a backward finite difference approximation based on a fixed spatial discretisation. The FOR equation is particularly necessary in this example in order to express the reaction term of the mass balance. Note that explicitly and implicitly declared array equations can be mixed freely. For instance, the last equation in figure 2.8 could also be written as:

```
FOR J := 2 TO NoSlice DO
```

```
MODEL Multi_Mix

PARAMETER
  NoComp                        AS INTEGER
  NoStream                      AS INTEGER

VARIABLE
  Flow_In                       AS ARRAY(NoStream,NoComp) OF Molar_Flowrate
  Flow_Out                      AS ARRAY(NoComp)          OF Molar_Flowrate
  Press_In                      AS ARRAY(NoStream)        OF Pressure
  Press_Out                     AS Pressure

STREAM
  Inlets : Flow_In, Press_In    AS ARRAY(NoStream) OF Mixer_Stream
  Outlet : Flow_Out, Press_Out  AS Mixer_Stream

EQUATION

  # Static mass balance
  FOR I := 1 TO NoComp DO
    Flow_Out(I) = SIGMA(Flow_In(1:NoStream,I)) ;
  END # for

  # Outlet pressure drops to minimum of the inlet pressures
  Press_Out = MIN(Press_In) ;

END # Multi_Mix
```

Figure 2.7: Model that Statically Mixes an Arbitrary Number of Streams

```
    $Conc(,J) = - Velocity*(Conc(,J) - Conc(,J-1))/Dl
                 + Stoich*Rate_Const*Conc(1,J)*Conc(2,J) ;
  END # for
```

Here, the expansion over the various components is implied in the declaration of the simple equation.

In general, the declaration of a FOR equation begins with:

FOR Identifier := <exp> TO <exp> STEP <exp> DO

The increment is defaulted to unity if not specified.

```
MODEL Tubular_Reactor

PARAMETER
  NoComp, NoSlice                     AS INTEGER
  Length, Area, Rate_Const, Dl        AS REAL
  Stoich                              AS ARRAY(NoComp) OF INTEGER

VARIABLE
  Flow_In, Flow_Out                   AS Vol_Flowrate
  Velocity                            AS Velocity
  Conc_In                             AS ARRAY(NoComp) OF Molar_Conc
  Conc                                AS ARRAY(NoComp,NoSlice) OF Molar_Conc

STREAM
  Inlet  : Flow_In, Conc_In           AS Reactor_Stream
  Outlet : Flow_Out, Conc(,NoSlice)   AS Reactor_Stream

SET
  Dl := Length/NoSlice ;

EQUATION

  # Reaction in dilute solution
  Flow_In = Flow_Out ;

  # Fluid velocity
  Velocity*Area = Flow_In ;

  # Inlet node
  $Conc(,1) = - Velocity*(Conc(,1) - Conc_In)/Dl          # Convection term
              + Stoich*Rate_Const*Conc(1,1)*Conc(2,1) ;   # Reaction term

  # Interior nodes
  FOR I := 1 TO NoComp DO
    FOR J := 2 TO NoSlice DO

      $Conc(I,J) = - Velocity*(Conc(I,J) - Conc(I,J-1))/Dl
                   + Stoich(I)*Rate_Const*Conc(1,J)*Conc(2,J) ;

    END # for
  END # for

END # Tubular_Reactor
```

Figure 2.8: Model of a Tubular Reactor

It must be possible to distinguish the identifier of the special counter variable from any attribute identifiers of the model entity in question, and this identifier may only be referenced by equations enclosed by the FOR structure. The list of equations may involve any combination of simple and structured equations, enabling nesting to arbitrary depth, although the identifiers of any counter variables introduced must be distinguishable from those of any enclosing FOR equations. The number of times a FOR equation is expanded is determined from the starting value, upper limit and the optional positive increment for the counter variable. These values are declared immediately after the counter variable and may be expressed as integer expressions not involving the counter variable itself. The expansion of a FOR structure proceeds according to algorithm 2.1.

Algorithm 2.1 *Expand* FOR *Equation*

```
1. Assign the starting value to the counter variable.
2. While the counter variable does not exceed the upper limit do
    (a) Expand the enclosed equation list for the current value of the counter variable.
    (b) Increment the counter value by the increment specified (default 1).
   end
end
```

2.3.5.2 The CASE Equation

As already discussed in section 2.2, a combined discrete/continuous model consists of a set of invariant and variant equations, the variant equations being described in terms of one or more finite automata. The CASE structured equation is one of two syntactic structures that enable the declaration of physico-chemical discontinuities in terms of a finite automaton.

The states of the finite automaton represented by a CASE equation are enumerated by a special model attribute known as a *selector* variable. All selector attributes must be declared in the SELECTOR section of a model entity in terms of a list of symbolic identifiers that enumerate these states. A default initial value for the selector attribute may also be included in this declaration and several selector attributes may share the same enumeration.

A CASE equation is declared in terms of a set of two or more *clauses*, each corresponding to one state of the finite automaton. The selector variable associated with each CASE equation provides the enumeration of these states, and stores the active state of that particular automaton at any point in time. Each selector variable may therefore only appear in one CASE equation.

Each of these clauses is declared in terms of a list of equations and an optional list of *switch* statements defining transitions from this state to other states of the CASE equation. Again, in order to allow the recursive declaration of each state of a finite automaton, this list of equations may include any combination of simple and structured equations.

The switch statements associated with a clause correspond to the set of possible transitions from that state. Each switch statement is declared in terms of the logical expression that must be satisfied in order for a transition to occur, and the new state which becomes active as a result of the transition. Of course, this new state must belong to the set of states of the automaton concerned. Switch statements are evaluated in the order in which they are declared, so if two or more transition conditions are satisfied simultaneously, the new active state will be determined from the first switch statement to be evaluated. If a clause contains no switch statements, it is impossible to leave that particular clause once it is entered.

Figure 2.9 demonstrates how a CASE equation can be used in the declaration of the model for a vessel fitted with a bursting disc. This example also demonstrates that the specification of the initially active state of a system is necessary; the user must specify whether or not a disc has been installed in the vessel. In this case the initial state of the finite automaton can be inferred from a default value, although this default could be overridden by an individual simulation description.

2.3.5.3 The IF Equation

Reversible discontinuities are by far the most commonly encountered physico-chemical discontinuity in industrial processing systems. As has already been noted, the state tran-

```
MODEL Vessel_With_Bursting_Disc

PARAMETER
  Vessel_Volume, R                        AS REAL
  Burst_Pressure, Disc_Const              AS REAL

VARIABLE
  Flow_In, Flow_Out, Relief_Flow          AS Molar_Flowrate
  Holdup                                  AS Moles
  Temp                                    AS Temperature
  Press                                   AS Pressure

STREAM
  Inlet  : Flow_In, Press                 AS MainStream
  Outlet : Flow_Out, Press                AS MainStream
  Relief : Relief_Flow, Press             AS MainStream

SELECTOR
  Disc_Flag                               AS (Intact,Burst) DEFAULT Intact

SET
  R := 8.314 ;

EQUATION

  # Mass balance
  $Holdup = Flow_In - Flow_Out - Relief_Flow ;

  # Equation of state - perfect gas
  Press*Vessel_Volume = Holdup*R*Temp ;

  # Relief flow from bursting disc - assume choked flow
  CASE Disc_Flag OF
    WHEN Burst  : Relief_Flow = Disc_Const*Press/SQRT(Temp) ;
    WHEN Intact : Relief_Flow = 0 ;
                  SWITCH TO Burst IF Press >= Burst_Pressure ;
  END # case

END # Vessel_With_Bursting_Disc
```

Figure 2.9: Model of a Vessel Fitted with a Bursting Disc

sitions associated with this type of discontinuity can be characterised by a single logical condition. Although reversible discontinuities can be declared using a CASE equation, the IF structured equation is specifically introduced as a convenient shorthand, involving a single logical condition, for the declaration of this common form of discontinuity. CASE and IF structured equations are known collectively as *conditional* equations.

IF structured equations are almost identical to the similar structures supported by many other modelling languages. The declaration consists of a logical expression and two clauses, each characterised by a list of structured and/or simple equations. The active clause may change dynamically during a simulation and is determined from the value of this logical expression at any point in time. If the expression is true, the equations declared in the first clause are included in the system model, otherwise the equations of the second clause are included. The use of an IF structured equation in the declaration of a model for a vessel containing an overflow weir is demonstrated in figure 2.10.

The logical expression associated with an IF equation may contain references to non-scalar quantities. Obviously, this results in an array of scalar logical expressions representing the expansion of the expression for each element of these non-scalar quantities. An array of logical expressions may only be used in situations where the number of equality constraints in each clause of the IF equation exactly matches the number of scalar logical expressions in the array. Each equality constraint can then be assigned its own scalar logical expression that determines any dynamic changes independently of any change to the other equality constraints declared in the same IF equation. Figure 2.11 demonstrates the application of this rule to the model of a vessel containing overflow pipes at several different levels.

A similar rule is applied to CASE equations declared in terms of an array of selector variables. Here, each element of the selector array determines the active state of one equality constraint and an array of switch statements determines the independent state transitions.

It is worth noting that, unlike CASE equations, the clause of an IF equation that is initially active cannot be specified explicitly. Instead, it is determined automatically by the initialisation calculation, which ensures that the consistent initial values obtained (see section 1.4.1) satisfy both the logical condition *and* the equations in this clause. However, the solution of nonlinear systems involving such conditional equations is far from trivial (see, for example, Zaher and Westerberg (1991)). Furthermore, in some cases there may exist valid

```
MODEL Vessel_With_Weir

PARAMETER
  Area, Weir_Length, Weir_Height    AS REAL

VARIABLE
  Flow_In, Flow_Out                 AS Molar_Flowrate
  Total_Holdup                      AS Moles
  Liquid_Level                      AS Length
  Bulk_Density                      AS Liquid_Density

STREAM
  Inlet  : Flow_In                  AS MainStream
  Outlet : Flow_Out                 AS MainStream

EQUATION

  # Mass balance
  $Total_Holdup = Flow_In - Flow_Out ;

  Liquid_Level*Area*Bulk_Density = Total_Holdup ;

  # Francis formula for flow over a weir
  IF Liquid_Level > Weir_Height THEN
    Flow_Out = Bulk_Density*1.84*Weir_Length*
                                ABS(Liquid_Level - Weir_Height)^1.5 ;
  ELSE
    Flow_Out = 0 ;
  END # if

END # Vessel_With_Weir
```

Figure 2.10: Model of a Vessel Containing an Overflow Weir

solutions in either clause of an IF equation; the solution found will depend on the initial guesses and the numerical method employed. In view of these factors, it may sometimes be preferable to use a CASE equation, provided the initial state of the system is known a priori.

2.3.5.4 The UNDEFINED Construct and Conditional Equations

In certain circumstances it is possible for a system to enter a domain in which a subset of the describing variables have no physical meaning and therefore become undefined. For example, if a flash drum moves dynamically into a domain in which only a liquid phase

```
MODEL Overflow_Tank

PARAMETER
  NoPipes                             AS INTEGER
  Area                                AS REAL
  Pipe_Height, Valve_Const            AS ARRAY(NoPipes) OF REAL

VARIABLE
  Liquid_Volume                       AS Volume
  Liquid_Level                        AS Length
  Flow_In                             AS Vol_Flowrate
  Flow_Out                            AS ARRAY(NoPipes) OF Vol_Flowrate

EQUATION

  $Liquid_Volume = Flow_In - SIGMA(Flow_Out) ;

  Liquid_Level*Area = Liquid_Volume ;

  IF Liquid_Level >= Pipe_Height THEN
    Flow_Out = Valve_Const*ABS(Liquid_Level - Pipe_Height)^0.5 ;
  ELSE
    Flow_Out = 0 ;
  END # if

END # Overflow_Tank
```

Figure 2.11: Model of a Vessel Containing a Series of Overflow Pipes

is present, then the variables relating to the component mole fractions of the vapour phase are no longer necessary for the description of the system. These undefined variables can be dropped from the continuous model because they are irrelevant to the rest of the system while it remains in this domain. As a consequence of this, a number of describing equations equal to the number of undefined variables can also be dropped from the continuous model.

Despite the fact that the set of variables required to describe a given system can vary dynamically, from a syntactic point of view it is more convenient to define a continuous model in terms of the *maximal* set of such variables. It is then necessary to introduce a mechanism for dropping one or more variables from this maximal set. This is provided by the UNDEFINED construct. For example:

```
UNDEFINED(y) ;
```

declares all vapour mole fractions y to be undefined.

The UNDEFINED construct has the status of an equation, and can therefore appear anywhere in the EQUATION section, in a fashion identical to that of a simple equation. If the boundaries of a domain in which a subset of variables becomes undefined can be expressed in terms of logical conditions, the UNDEFINED construct may be used in conjunction with a conditional equation to declare explicitly which variables and equations become irrelevant. For example:

```
IF Total_Holdup > 0 THEN
  x*Total_Holdup = Component_Holdup ;
ELSE
  UNDEFINED(x) ;
END # if
```

defines the mole fractions x in a vessel in terms of the component and total holdup. Clearly x is undefined if the vessel is empty. Any graphical or textual results produced from a simulation should indicate clearly the periods during which a system variable is undefined.

It should be noted that the introduction of the UNDEFINED construct allows us to demand that the number of equations be the same in all clauses of a conditional equation. This facilitates semantic checks against incorrect model definitions without restricting the generality of the language.

Of course, beyond purely syntactic and notational considerations, the use of UNDEFINED constructs raises fundamental questions regarding the definition of what constitutes a solution of a DAE set. Furthermore, if a variable declared as 'undefined' occurs in other equations, other system variables may also become undefined implicitly. Some mathematical and numerical issues relating to such partially determinable DAE sets have recently been examined by Jarvis and Pantelides (1991).

2.3.6 External Equation Declaration

The equation definition language described above is quite general. In spite of this, situations may arise in which it is desirable to allow alternative mechanisms for declaring relationships between the variable attributes of a model entity. This is particularly true in process engineering applications where it is often necessary to exploit the large investments already made in existing code.

SpeedUp (Prosys, 1991) provides one such mechanism through *procedures*. These introduce equations of the form:

$$\mathbf{z} - \mathbf{g}(\mathbf{w}) = 0 \tag{2.2}$$

where $\mathbf{w}$ and $\mathbf{z}$ are mutually exclusive subsets of the set of model variables and are known as the procedure input and output variables respectively. The values returned by the function $\mathbf{g}(\mathbf{w})$ are calculated by user supplied code. The number of equality constraints represented by these equations will be equal to the number of members of $\mathbf{z}$. The Speedup implementation places certain restrictions on the use of procedures, but in general a reference to a procedure should be able to appear anywhere that a simple equation may appear. In particular, it is important that procedure references may appear in the clauses of a conditional equation.

Procedures are frequently employed to improve the solution robustness of problems involving subsets of equations that may cause numerical difficulties, in particular those equations arising from the prediction of the physical properties of multi-component systems. Procedures offer the advantage of solving these equations with specially tailored methods that exploit the physical or mathematical properties of the system under consideration. In addition, a significant reduction in the overall dimensionality of the problem to be solved can be achieved in circumstances in which many intermediate variables can remain internal to the procedure. For example, the equation that determines the bubble point of a multi-component mixture at a given pressure can be solved efficiently with a Newton-Raphson iteration scheme. If this equation is encapsulated in a procedure, the efficiency of this iteration scheme can be exploited, and the variables that represent the equilibrium vapour-liquid distribution coefficients of each component at the bubble point can remain internal to the procedure.

A further application of procedures is for the insertion of a common set of equations in many different model entities. In this case, the equations encapsulated by a procedure represent abstract relationships between a set of input variables and a set of output variables, which are related to actual variable attributes when the procedure is inserted in a particular model entity. If, for example, plant measurements have yielded an empirical correlation between the enthalpy, temperature, and composition of the material present, this relationship may be encapsulated by a procedure, and then inserted in every model of the unit operations that make up the process.

Traditionally, procedures have also been applied to the description of equations that could not be expressed in the syntax of the modelling language in question. In many cases, however, the use of procedures in this role can impair the solution efficiency of a simulation and can even lead to incorrect results. In SpeedUp, for example, procedures may be applied to the description of physico-chemical discontinuities, particularly those that cannot be described in terms of a reversible mechanism. As a consequence, the events that lead to physico-chemical discontinuities become hidden from the simulation executive, and the step size control mechanism of the integration algorithm must be relied on to locate the exact time of occurrence of the event (see section 1.4.3). It is hoped that the use of procedures for this purpose will disappear as the descriptive power of modelling languages increases with, for example, the introduction of CASE equations.

For identical reasons, hidden events should not form part of procedures that are used in a more legitimate role. For example, the equation determining the enthalpy of a fluid stream is discontinuous at both the bubble and dew temperatures of the mixture, a fact that is best declared explicitly in the modelling language itself.

In summary, there appear to be at least three legitimate roles for procedures or *externally declared equations*:

- Exploitation of tailored solution algorithms and existing investments in model code.
- Decomposition of the problem through removal of internal variables from the global iteration.
- Insertion of abstract relationships in multiple model entities.

It may be desirable to separate these roles through the introduction of three categories of externally defined equations (EDEs). The first category would be employed to declare a set of equations, and any internal variables, in terms of the high level symbolic language outlined in the preceding sections. These equations and variables would then be inserted symbolically in the overall system model wherever a reference to the EDE appears, thereby enabling the insertion of abstract relationships in multiple model entities.

The second category of EDE would be declared in an identical fashion, except that the equations and variables would not be inserted in the overall system model. Instead, the equations would be solved for their output variables each time a value for the function $\mathbf{g}(\mathbf{w})$ is required by the global iteration. This category of EDE would therefore facilitate the removal of internal variables from the global iteration, and the insertion of abstract relationships in multiple model entities.

The final category of EDE would be identical to the procedures of SpeedUp, representing an interface to an external subroutine written in a general purpose programming language such as FORTRAN, thereby fulfilling any or all of the roles outlined above. Further consideration of these issues is, however, beyond the scope of this thesis.

2.4 Hierarchical Submodel Decomposition

The advantages of hierarchical submodel decomposition during the development of the large continuous models required for industrial processes have already been discussed. An example of the application of this methodology can be drawn from the development of a model for the hypothetical process shown in figure 2.12. At the top level of the hierarchy, the flowsheet can be decomposed into three interconnected component models representing the pretreatment, reaction and separation sections of the process respectively. After this initial decomposition, the development of models for each of these sections can be considered in isolation. The separation section may, for example, consist of a train of three distillation columns, which would be modelled by the connection of three column models in the desired configuration. At the next level in the hierarchy, the model of one of the distillation columns also consists of the connection of several different components in a flowsheet, but here some of the component models, such as the reboiler or condenser, are primitive models and some, such as the tray sections, can be further decomposed into a series of primitive tray models connected in a regular structure. The net result is a model for a complex process that has been completely constructed, through the use of a number of intermediate hierarchical levels, from the interconnection of many primitive models.

The use of several intermediate hierarchical levels has at least two additional benefits: as each level of the hierarchy is developed, the component models required at that stage

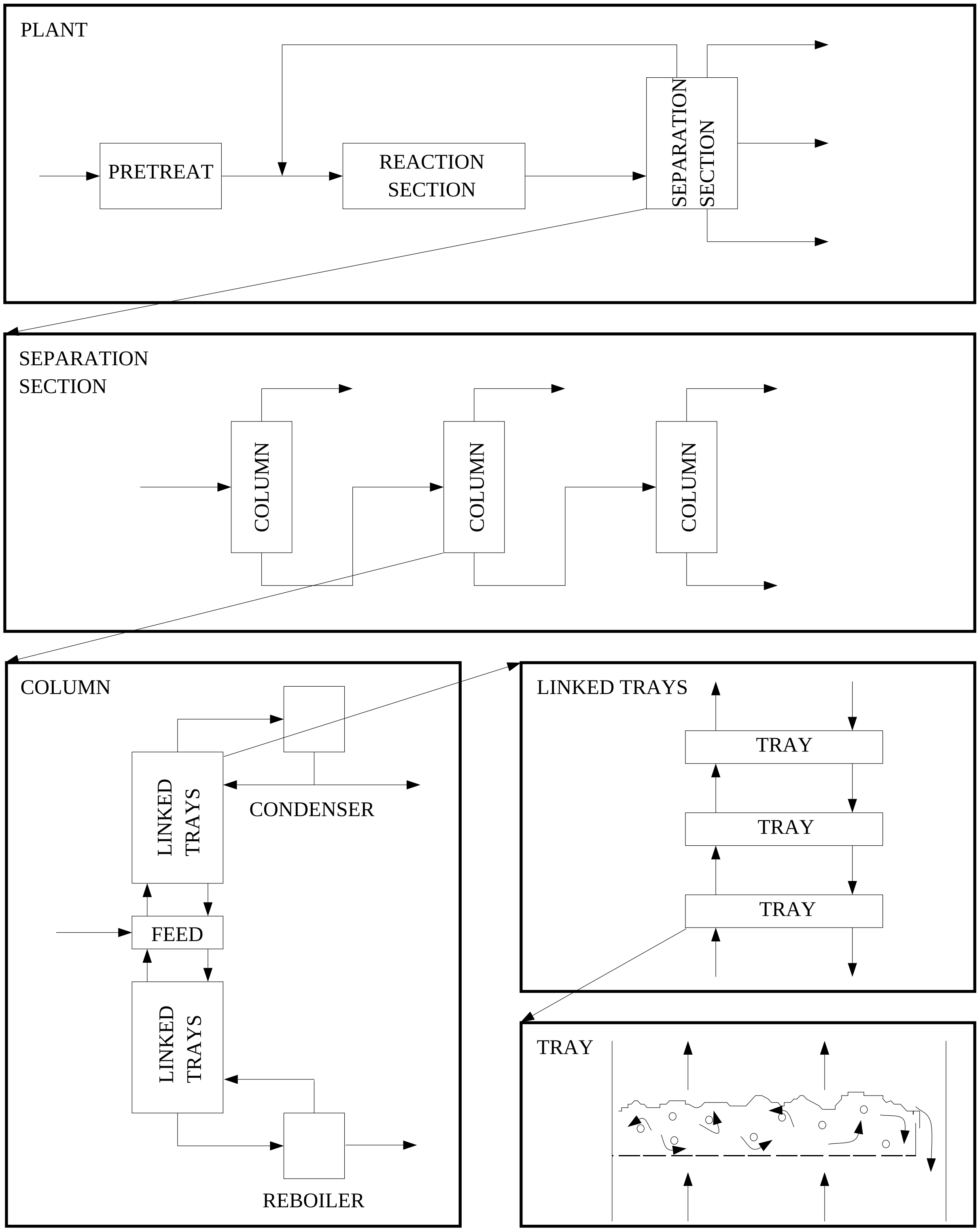

Figure 2.12: Hierarchical Submodel Decomposition

can be validated individually before insertion in the overall structure, and when the model is complete, a suitable hierarchical decomposition will enable the user to perform studies that focus on a subsection of the overall process merely by selecting a model at the appropriate point in this hierarchy.

This section is concerned with the language structures that support the hierarchical decomposition of models.

2.4.1 Unit Attributes

The hierarchical decomposition of a system requires that the behaviour of any model at a hierarchical level above that of primitive models must be declared, at least partially, in terms of a set of component submodels. Language structures that support this decomposition can therefore distinguish between the complex models whose behaviour is described in terms of a set of component submodels on one hand, and the primitive models whose behaviour is entirely described by a set of equations on the other. This approach has been adopted by the developers of OMOLA (Andersson, 1990) and in the MACROs of SpeedUp (Prosys, 1991). Alternatively, all models can be considered to be equal, in that their behaviour can be described by equations, submodels or any mixture of the two. This approach has most notably been adopted by the developers of ASCEND (Piela, 1989) and has at least two advantages. Firstly, because a model can be declared in terms of any combination of submodels and equations, it is possible to introduce new equations at any level in the hierarchy. The benefits of this can be seen in several examples throughout the text. Secondly, the language itself becomes considerably less cluttered by avoiding the need for separate language structures to distinguish between primitive and complex models. This second approach is therefore adopted by the addition of the optional UNIT section to a model entity declaration.

The UNIT section contains a declaration of the set of submodels that partially describe the behaviour of the model entity in question. All these *unit attributes* must be instances of a previously declared model entity. The fact that the behaviour of *any* model entity may be declared in terms of a set of submodels enables a hierarchical decomposition to extend to an arbitrary number of intermediate levels and, because no distinction between complex or primitive models is made, each of these intermediate levels may include the

```
UNIT
  Valve                          AS Control_Valve
  Flow_Controller                AS PI_Controller
  Flow_Sensor                    AS Orifice_Plate
  Pump                           AS Centrifugal_Pump
```

Figure 2.13: Example UNIT section

declaration of other model attributes (such as equations and variables). Figure 2.13 shows the UNIT section from a model entity representing an analogue flow control loop.

Systems containing regular structures of components occur frequently in processing systems, particularly in those unit operations comprised of a series of identical stages. The example used at the beginning of this section demonstrates that the tray sections of a distillation column are in fact regular structures of tray components, and similar structures appear in many other unit operations, such as countercurrent liquid/liquid extractors, multiple effect evaporators, or plate and fin heat exchangers. These regular structures are modelled by the declaration of arrays of unit attributes in exactly the same manner as any other attribute array would be declared.

Regular structures of component models have also been proposed as the means by which *model approximation decomposition* can be implemented (Nilsson, 1989a). This methodology approximates a distributed parameter system, that is usually modelled by a set of partial differential equations, as a series of well mixed *slices* in series. Alternative methods for the modelling of such systems include reduction of the partial differential equations to a set of ordinary differential equations through use of a suitable finite difference approximation based on a fixed spatial discretisation, as demonstrated in figure 2.8, or direct solution of the partial differential equations with a suitable general-purpose PDAE solver (Pipilis, 1990). As the performance of general-purpose solvers improves, this latter approach will probably prove to be the most accurate and efficient.

The remaining information concerning the composite system represented by a model entity containing a UNIT section can be declared in the usual manner. For example:

- Integer parameter attributes may be used in the definition of the dimensionality of unit attribute arrays.

- Variable attributes may define new quantities that only exist at the hierarchical level represented by the composite model.
- Stream attributes may define the interface of this composite system to its environment.
- Equation attributes may relate new variable attributes to the variable attributes of the submodels, or define the connection mechanisms between submodels.

A *pathname* mechanism provides the means by which the attributes of submodels can be referenced by higher hierarchical levels. The pathname to a particular attribute consists of a list of unit attribute references separated by full stops terminated by a reference to the attribute in question. The members of this list define the path through the hierarchy from the level at which the reference is made down to the level of the attribute itself. This mechanism therefore gives a model global access to all the attributes of the hierarchical levels extending beneath it.

In contrast, some modelling languages enforce strict modularisation on component models. For example, an OMOLA model is declared in terms of a set of clearly defined interfaces or *terminals*, and an internal description. A higher hierarchical level may only access the information contained within these terminals, and is denied access to the remainder of the information declared within submodels. Strict modularisation is a principle that has been adopted from modern programming languages, such as Modula–2 or Ada, where it has been demonstrated as a valuable aid to software development. The benefits it brings to modelling languages, however, are rather less tangible. From the specific point of view of a combined discrete/continuous modelling language, strict modularisation would deny the control actions imposed on a continuous model access to much of the system model, and thereby significantly limit the scope of these control actions.[9] Furthermore, requiring a comprehensive set of interfaces to anticipate all future uses for a model entity at the time of its declaration is, in our opinion, a severe restriction on the model developer.

[9]The advantages of giving control actions global access to the underlying continuous model are discussed at length in the next chapter.

```
WITHIN Separation_Section.Column_C.Top_Section DO
  FOR I := 1 TO NoTrays DO
    WITHIN Stage(I) DO
      Q = UA*(Ambient_Temp - Temp) ;
    END # within
  END # for
END # within
```

Figure 2.14: Example WITHIN equation

2.4.2 The WITHIN Structured Equation

As the number of intermediate hierarchical levels increases, so does the length of the pathnames required to reference attributes at the bottom of the hierarchy. Writing equations in terms of these attributes becomes increasingly tedious, especially if a large part of the pathname is common to many of the attributes referenced by an equation. The WITHIN structured equation helps to relieve some of this burden.

A WITHIN equation defines a *scope* for the list of equations it encloses. This scope is a reference to one of the submodels in the hierarchy that extends beneath the model in which the equation appears. A pathname can be used to refer to a submodel several hierarchical levels down. All attribute references that appear in the enclosed equations will first be interpreted as if they were attributes of this scope, relieving the burden of prefixing all the references with a pathname to this scope. If no interpretation for a reference exists within this scope, it will then be interpreted in the usual manner. An example of a WITHIN equation declared at the top of level of the hierarchy illustrated in figure 2.12 is shown in figure 2.14, where `Ambient_Temp` is a variable attribute declared at this top level of the hierarchy.

As the list of equations may be comprised of any combination of simple and structured equations, it is possible to nest WITHIN equations. In this case, the search for an interpretation of an attribute reference will start at the innermost scope and progress outwards through all the other enclosing scopes until an interpretation is found. An error occurs if this search fails, and no interpretation exists in the model in which the reference is actually made.

```
PARAMETER
  Top_NoTrays, Bottom_NoTrays              AS INTEGER

UNIT
  Top_Section, Bottom_Section              AS Linked_Trays

SET
  Top_Section.NoTrays    := Top_NoTrays    ;
  Bottom_Section.NoTrays := Bottom_NoTrays ;
```

Figure 2.15: Example of Parameter Propagation

2.4.3 Parameter Value Propagation

As the hierarchical decomposition of a system evolves, it becomes increasingly important to be able to propagate values to parameter attributes of submodels from higher hierarchical levels. For example, in the distillation column model outlined at the beginning of this section, it may be necessary to propagate values to the integer parameters of the linked tray submodels that determine the number of trays in the top and bottom sections respectively. These values may be known a priori, or may be parameter attributes of the column model itself, thus parameterising the column model by the number of trays in each section. Obviously, values may only be propagated to parameter attributes that have not already been assigned fixed values.

Parameter values can be propagated through a hierarchy of models by means of two mechanisms. The first mechanism is merely the explicit assignment of values to the parameters of submodels, in the SET section of a model entity at a higher hierarchical level. In order to make this feasible, the language definition requires the UNIT section to be declared before the SET section. Through the pathname mechanism described above, the parameters assigned values may be any number of hierarchical levels beneath that of the SET section. However, if the parameters concerned have already been assigned fixed values at an intermediate hierarchical level, including that of the model entity in which they were originally declared, the assignment will be rejected. The excerpts from the model of a distillation column shown in figure 2.15 demonstrate how the assignment of values to the parameters that determine the number of trays in each section would be accomplished.

If, on instantiation of the model of a complete system, it is found that a parameter of a submodel has not been explicitly assigned a fixed value from any source, the hierarchical levels extending above this submodel will be searched for a compatible parameter attribute. A compatible parameter is defined as having the same identifier, same type, and identical dimensionality to that of the original parameter. If such a compatible parameter is found, the original submodel parameter will be assigned the same value as that assigned to the compatible parameter. It is important to recognise that a submodel parameter adopts the value of the *first* compatible parameter that is found as the search progresses upwards through the model hierarchy. This is the preferred mechanism by which the number of components that exist in a multicomponent system can be propagated through an entire hierarchy (merely by the assignment of a value at the top level of this hierarchy).

If a value for a parameter cannot be found through either mechanism, and no default value is given, an error has occurred.

2.4.4 Formal Definition of Model Entity

It is now possible to define formally, in a recursive manner, the information declared within a model entity:

- The set of unit attributes associated with a model entity are those unit attributes declared in the UNIT section.
- The set of parameter attributes of a model entity is the union of the sets of parameter attributes associated with the unit attributes and the set of parameter attributes declared in its own PARAMETER section.
- The set of variable attributes of a model entity is the union of the sets of variable attributes associated with the unit attributes and the set of variable attributes declared in its own VARIABLE section.
- The set of stream attributes of a model entity is the union of the sets of stream attributes associated with the unit attributes and the set of stream attributes declared in its own STREAM section.

- The set of selector attributes of a model entity is the union of the sets of selector attributes associated with the unit attributes and the set of selector attributes declared in its own SELECTOR section.
- The set of equation attributes of a model entity is the union of the sets of equation attributes associated with the unit attributes and the set of equation attributes declared in its own EQUATION section.

2.4.5 Continuous Connections

Stream attributes, and their equivalents in other modelling languages, model the complex continuous connection mechanisms that exist between the components of a physical system. A continuous connection between two submodels is set up by declaring a relationship between stream attributes of the two submodels. The simulation executive can then automatically generate the same relationship between all the fields of the two stream attributes and thereby free the engineer to concentrate on the structure of the system under consideration.

The most frequently encountered relationship between two stream attributes is that of an equality constraint. As a result, a series of equality constraints between each field of the two streams is automatically generated. It is also possible for more complex relationships to exist between stream attributes. For example, Kirchhoff's law requires the sum of all currents at any node in an electrical circuit to be zero, which would be most naturally modelled by summing the fields representing current of two or more streams to zero (Elmquist, 1978).

The developers of ASCEND have attempted to exploit the frequent occurrence of equality constraints between stream attributes by the application of the ARE_THE_SAME operator. Stream attributes that are related by this operator are actually merged into a single attribute that can be referenced by either identifier, automatically eliminating the equality constraints that would otherwise be required and in the process significantly reducing the size of the overall system model.

An important feature of combined simulation is the ability to alter dynamically the topology of a system, either by replacing one connection with another or by replacing a continuous connection with an equivalent number of equations. This, however, requires the flexibility afforded by the implementation of connections as equality constraints that can simply be added to or dropped from the overall system model as the need arises. Overall, we believe that this requirement negates any potential benefits from the use of the ARE_THE_SAME operator.

The advantages of both approaches can be realised if variable merging is performed at a lower level than that proposed by the developers of ASCEND. In order to allow the topology of a system to be altered dynamically, all continuous connections are represented as equality constraints in the overall system model. However, when this system model is submitted to the solution routines (e.g. during a period of continuous simulation between events), the solution routines themselves can a priori eliminate *all* equality constraints from the model actually solved, thereby ensuring that the smallest possible model is always solved.

For the reasons outlined above, it is most natural to consider relationships between stream attributes as members of the set of describing equations of a system. These relationships are therefore declared in the EQUATION section of a model entity, where they may also appear within structured equations. Conditional equations may then be used to define dynamic changes to the topology of a system. The only relationship between stream attributes allowed by the current language implementation is that of an equality constraint, but an extension to allow generalised equations involving stream attributes would in principle be straightforward.

An equality constraint between two stream attributes requires the stream attributes to be compatible. Stream attributes are compatible with each other if they are of the same stream type and, for each field, the dimensionality of the variables matched is identical.

In the case of complex models involving several submodels, some or all of the model's interface with its environment will correspond exactly to some of the interfaces of its components. For example, in a distillation column model, the feed to the column corresponds to feed of the feed tray submodel, and the top and bottom product streams correspond to the product streams of the condenser and reboiler submodels respectively. This is one situation in which the two stream attributes should be merged into a single instance that can be referenced by two different pathnames. The stream attributes of a model may therefore be merged with the stream attributes of its submodels within the STREAM section.

Figure 2.16 demonstrates how the model of a distillation column might be declared.

In particular, it demonstrates the advantages of allowing other model attributes to be declared in the same model as unit attributes and the use of stream merging to define the interfaces to composite systems. The model of a tray section is itself declared in terms of the connection of a series of submodels, as is shown in figure 2.17. Here, the model is parameterised by the number of trays in the section, and uses an array of unit attributes to represent the regular tray structure.

2.5 Hierarchical Model Development Through Inheritance

Inheritance, a concept first popularised by the object-oriented programming languages, is an aid to model development that particularly enhances a model's reusability. Through inheritance, a new type may be declared as an extension or restriction of one or more previously declared types. An inheritance hierarchy evolves as new types are declared that inherit characteristics from already defined types.

Inheritance is introduced briefly here because in later chapters it will be demonstrated that language structures employed to model the control actions applied to a process can take considerable advantage of model entities developed through use of an inheritance hierarchy. Before this discussion can begin, it is necessary to introduce some terms relating to inheritance:

- A type that inherits directly or indirectly (via intermediate hierarchical levels), from type X is said to be a descendant of type X. A type is always considered to be a descendant of itself.

- If type Y is a descendant of type X, type X is said to be an ancestor of type Y. If type Y inherits directly from type X, then type X is a parent of type Y.

An inheritance hierarchy can be represented diagrammatically by a tree, where the nodes signify types and the arcs signify the relationship *inherits from*. An arc from one node to another indicates that the latter node is a parent of the former.

In the context of model development, only the facilities to extend or amend (*selective* inheritance) a previously defined type are required. A model that is directly descended from another model contains all the information associated with the parent, plus any new

```
MODEL Distillation_Column

PARAMETER
  No_Trays, Feed_Position            AS INTEGER

UNIT
  Condenser                          AS Total_Condenser
  Top_Section, Bottom_Section        AS Linked_Trays
  Feed                               AS Feed_Tray
  Reboiler                           AS Partial_Reboiler

VARIABLE
  Net_Energy_Requirement             AS Energy_Flow

STREAM
  Feed_Stream                        IS Feed.Feed_Stream
  Top_Product                        IS Condenser.Liquid_Product
  Bottom_Product                     IS Reboiler.Liquid_Product

SET
  Top_Section.NoTrays    := NoTrays - Feed_Position ;
  Bottom_Section.NoTrays := Feed_Position - 1       ;

EQUATION

  # Define the net energy requirement for the column
  Net_Energy_Requirement = Reboiler.Heat_Load + Condenser.Heat_Load ;

  # Continuous connections
  Condenser.Reflux           IS Top_Section.Liquid_In     ;
  Condenser.Vapour_Feed      IS Top_Section.Vapour_Out    ;
  Top_Section.Vapour_In      IS Feed.Vapour_Out           ;
  Top_Section.Liquid_Out     IS Feed.Liquid_In            ;
  Feed.Vapour_In             IS Bottom_Section.Vapour_Out ;
  Feed.Liquid_Out            IS Bottom_section.Liquid_In  ;
  Bottom_Section.Vapour_In   IS Reboiler.Vapour_Out       ;
  Bottom_Section.Liquid_Out  IS Reboiler.Liquid_Feed      ;

END # Distillation_Column
```

Figure 2.16: Model of a Distillation Column

```
MODEL Linked_Trays

PARAMETER
  NoTrays                          AS INTEGER

UNIT
  Stage                            AS ARRAY(NoTrays) OF Tray

STREAM
  Liquid_In                        IS Stage(NoTrays).Liquid_In
  Vapour_Out                       IS Stage(NoTrays).Vapour_Out
  Liquid_Out                       IS Stage(1).Liquid_Out
  Vapour_In                        IS Stage(1).Vapour_In

EQUATION

  # Continuous connections
  FOR I := 1 TO NoTrays - 1 DO
    Stage(I).Vapour_Out IS Stage(I+1).Vapour_In  ;
    Stage(I).Liquid_In  IS Stage(I+1).Liquid_Out ;
  END # for

END # Linked_Trays
```

Figure 2.17: Model of a Tray Section

information declared within the model itself. Models may therefore be developed in a hierarchical manner through a series of intermediate stages of increasing complexity.

Inheritance is also a powerful tool for avoiding the repetition of common information during model development. Careful development of an inheritance hierarchy will ensure that information common to several models need only be specified once. In addition, if this common information is declared correctly in the first place, the possibility of errors occurring during the repeated specification of the same information is eliminated.

The present language definition only caters for the development of model types through inheritance, although, as several workers have recently demonstrated (Piela, 1989; Andersson, 1990), it is equally applicable to the development of both variable and stream types (see appendix A).

2.5.1 Single Inheritance

The basic form of inheritance, *single* inheritance, requires that any type has at most one parent, although a type may have any number of direct descendants. Even with single inheritance, it is possible to declare complex hierarchies of model types. Single inheritance is implemented by the keyword INHERITS, and the identifier of the parent model entity, immediately following the declaration of the unique identifier of the new model entity. For instance, `MODEL Mixer INHERITS Tank`.

The root model of an inheritance hierarchy will often only be used to declare a few attributes that are common to all models in the particular hierarchy. In fact, such a root model may not contain sufficient information for it to be used in any useful calculations and will therefore only ever be used as a template for building other models. Any number of models can then be declared that inherit the information contained within this root and augment it with additional information. This additional information will distinguish models at the same hierarchical level from each other. Further levels of the hierarchy will evolve as these new models and their descendants are used as parents themselves. Note that a model type contains no information concerning its descendants, so it remains unaltered by changes to the inheritance hierarchy that extends beneath it.

An example is shown in figure 2.18, where a series of models for operations performed in a well mixed vessel are developed through an inheritance hierarchy. The root model is used to declare information common to all mixed tanks, such as geometric parameters and many of the describing variables. Its direct descendants then add more information to this, in particular the balance equations that characterise the operation in question, creating models that could actually be used for simulation. Finally, a third level of the hierarchy is shown where the model of a continuous stirred tank reactor is specialised by the kinetic mechanisms of the reactions. When a simulation is actually performed, the user can select an appropriate model of the operation under consideration from this hierarchy.

2.5.2 Polymorphism

Polymorphism permits the full exploitation of the advantages of model development in inheritance hierarchies. In chapter 3 we will demonstrate that polymorphism, in

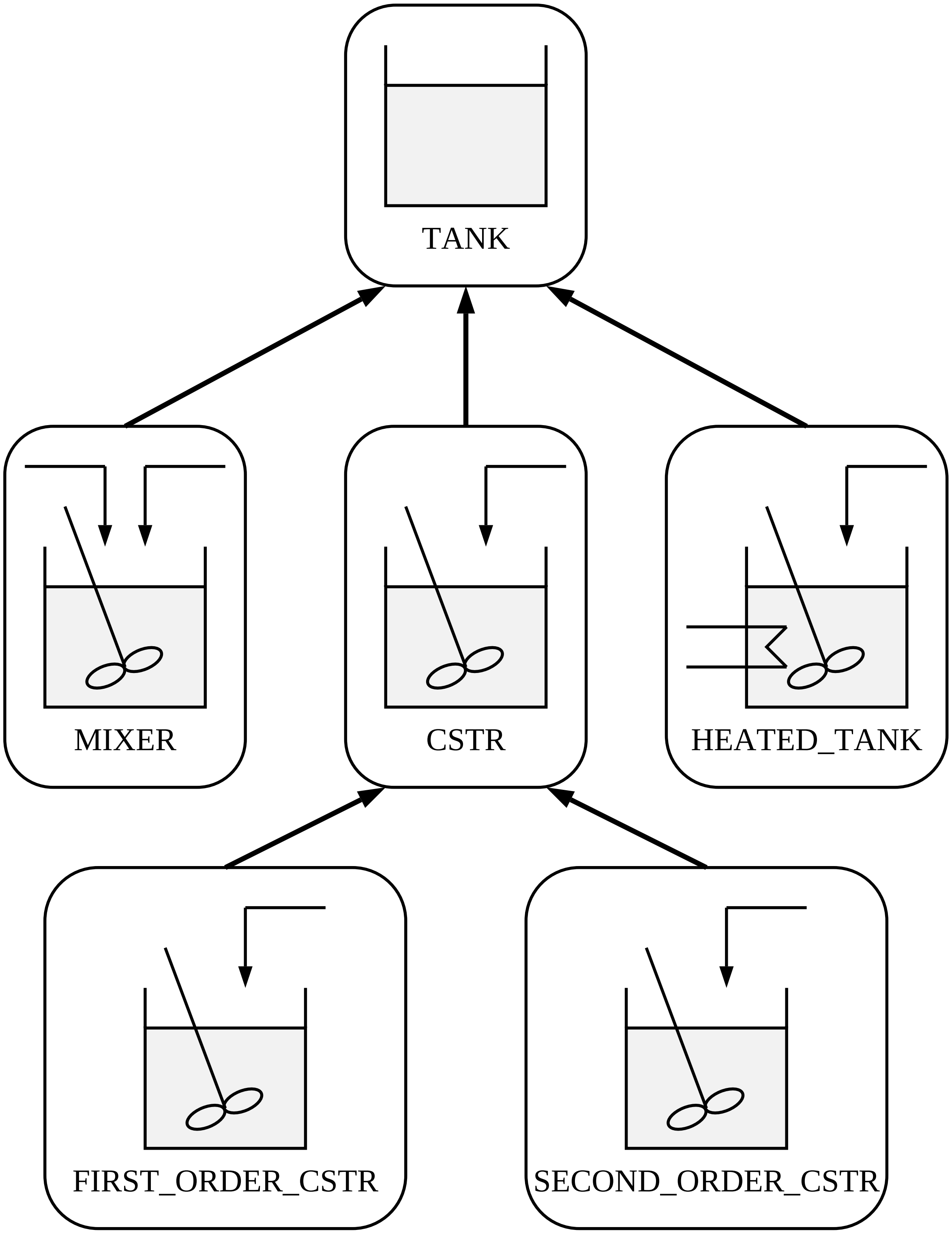

Figure 2.18: Inheritance Hierarchy

conjunction with inheritance hierarchies, can greatly enhance the reusability of the language structures employed to model the control actions applied to a process. Here, the basic concept is introduced and its application to model parameterisation is illustrated.

Returning again to the model of a distillation column, the need to parameterise tray sections by the submodel that describes the behaviour of individual trays has recently been stressed (Nilsson, 1989a). The effect of changing the type of tray in a column (e.g. to use bubble caps as opposed to a sieve plate) can then be studied easily. Furthermore, the development of special models for columns containing different tray types in each section (a situation that occurs relatively frequently) becomes unnecessary because the basic column model, with appropriate parameter assignments, may be used instead.

This degree of flexibility can be achieved by declaring a parameter which determines the tray submodel used by a particular instance of the distillation column. Such a parameter must be set to an entire model entity rather than merely a numeric value or an algebraic expression (cf. section 2.3.1). There must, however, be some restrictions on the model entities that can be assigned to this parameter: the model of a heat exchanger would clearly not be suitable in the example above, whereas the model of a sieve plate would. An inheritance hierarchy of tray models can be employed to achieve this restriction. The *polymorphic* parameter will be associated with a base type that is also the root model of the tray model hierarchy, and type checking will ensure that only members of this hierarchy are assigned to the parameter.

A polymorphic entity is one that has the ability to take several forms. In the object-oriented programming languages, this term has been applied to variables that may refer to instances of several different types during the execution of a program. Obviously, there must be some restrictions on the range of types that are compatible with a polymorphic entity, otherwise the advantages of a strongly typed environment are lost. Inheritance is used to constrain this range of types: a polymorphic entity is only compatible with instances of descendants of its base type. A polymorphic entity is always compatible with instances of its base type because a type is always considered to be a descendant of itself. As a consequence, an entity is no longer tied to values of a single type, but may be assigned values of a range of similar types. The development of these similar types in inheritance hierarchies ensures that they are suitable for the application of identical operations.

As has already been mentioned, model entities may be declared in terms of model type parameter attributes. These special parameter attributes are polymorphic entities, so their declaration includes the specification of a base type. They may be used to specify the type of any unit attributes of the model entity in question. This effectively defers a decision on the nature of the submodel actually included until a value is assigned to the parameter upon instantiation of the model entity, although the base type defines a broad class of models that are suitable. Note that this value is in fact a model entity type itself, and any attempt to assign a value that is not a descendant of the base type of the parameter would be rejected immediately.

Figure 2.19 demonstrates how a model type parameter attribute may be used to create a model for a tray section that is parameterised by the type of tray present in the section. The base type of this parameter is Generic_Tray, the root model of an inheritance hierarchy for tray models.

The only attributes of submodels, whose type is determined by a parameter, that may be referenced by higher hierarchical levels are those declared in the base type of the parameter. In this example, the liquid and vapour streams entering and leaving a tray must have been declared in the model entity `Generic_Tray` in order for references to them to be made in the STREAM and EQUATION sections.

The IS_REFINED_TO operator of ASCEND can be used to perform a similar role to that of model type parameters because it enables attributes declared in terms of generic models to be refined for a specific application. Both mechanisms are similar in that they require the model developer to anticipate future refinement, but parameterisation removes the need for the declaration of a new model each time a different refinement is required and helps to prevent inappropriate use of a model by demanding assignment of a parameter value upon each instantiation. Obviously, the IS_REFINED_TO operator still has a very powerful role to perform in selective inheritance, as will be seen later.

2.5.3 Multiple Inheritance

Multiple inheritance allows a type to be the direct descendant of two or more other types. A type with more than one parent inherits the union of the information declared in its set of parents. An inheritance hierarchy that tolerates multiple inheritance becomes a

```
MODEL Linked_Trays

PARAMETER
  NoTrays                        AS INTEGER
  Tray_Type                      AS MODEL Generic_Tray DEFAULT Sieve_Tray

UNIT
  Stage                          AS ARRAY(NoTrays) OF Tray_Type

STREAM
  Liquid_In                      IS Stage(NoTrays).Liquid_In
  Vapour_Out                     IS Stage(NoTrays).Vapour_Out
  Liquid_Out                     IS Stage(1).Liquid_Out
  Vapour_In                      IS Stage(1).Vapour_In

EQUATION

  # Continuous connections
  FOR I := 1 TO NoTrays - 1 DO
    Stage(I).Vapour_Out IS Stage(I+1).Vapour_In  ;
    Stage(I).Liquid_In  IS Stage(I+1).Liquid_Out ;
  END # for

END # Linked_Trays
```

Figure 2.19: Tray Section Parameterised by the Trays Present

general acyclic digraph as opposed to a tree.

An example frequently employed to illustrate the need for multiple inheritance in a process engineering context is that of a model for a reactor vessel, where one parent defines the detailed information concerning the geometry and behaviour of the vessel while the other parent defines the details of the reaction kinetics. Alternatively, the model could be described in terms of a machine (geometry and behaviour) and media (reaction kinetics and other physical properties) decomposition (Nilsson, 1989a). An obvious inconvenience associated with this second approach is the need to prefix all references to the attributes of the machine and media submodels with the appropriate pathname. However, due to the lack of other convincing examples to demonstrate the need for multiple inheritance, Nilsson states that it is unnecessary to include multiple inheritance when the machine/media model is sufficient.

There is even greater scope for the exploitation of a multiple inheritance hierarchy by polymorphism than there is in a single inheritance hierarchy, because it becomes possible to apply operations to a type that are compatible with any of its parents, or even any of the ancestors of these parents. In the reactor example above, it would be possible to apply operations compatible with the model that describes the geometry of the vessel and operations compatible with the model that describes the reaction kinetics.

If a type is allowed to have more than one parent, problems with identifier clashes may occur. For example, if a model receives an attribute with the same identifier from two of its ancestors, how can the two attributes be distinguished from each other for future reference? The problem of identifier clashes has been considered in detail by researchers developing object-oriented programming languages (see, for example, (Meyer, 1988)), including the situations in which repeated inheritance occurs (where an inheritance hierarchy results in a type inheriting more than once from the same type). In the context of a modelling language, it is proposed that all identifier clashes result in the model declaration being rejected, unless the attributes whose identifiers clash are identical in every respect, in which case the two attributes are merged. This is compatible with polymorphism because a model will always contain a superset of the information declared in its ancestors.

Although it was originally intended to explore the application of multiple inheritance in detail, time constraints have prevented this. Multiple inheritance will not therefore be discussed any further in this thesis, and has not been included in the current implementation of the modelling language.

2.5.4 Selective Inheritance

In many situations it may be necessary for a type to inherit most of the attributes of its parent or parents, but to also exclude certain attributes. An attribute may be excluded in order for it to be redefined by the descendant, or if it is not required by the descendant type. This is termed *selective* inheritance because it enables a type to inherit only the information it requires from its parent type(s). With selective inheritance, truly evolutionary development of models becomes possible. For example, at the beginning of a simulation exercise, only a small amount of information concerning the system being modelled will be known, so only simple models may be declared. As knowledge of the system increases, new models may developed that inherit the information in the early models and add to it, but also reject some of the simplifications made in those models.

If the balance equations of a system are included in the early stages of the development of an inheritance hierarchy, it becomes extremely important for descendant models to be able to add or drop terms from them. Although perhaps a rather unsatisfactory solution, selective inheritance, combined with the association of an identifier with the equations, enables them to be rejected by descendants and replaced by equations containing the correct terms.

In practice, selective inheritance in its purest form poses severe problems, especially for the safe exploitation of polymorphism, because a model is no longer guaranteed to contain a superset of the information contained in its ancestors. Both OMOLA and ASCEND (through the IS_REFINED_TO operator) tolerate a limited form of selective inheritance by allowing an attribute to have its type redefined to that of a descendant of the original type. This approach guarantees that a model will always contain a superset of the information contained in its ancestors, and is thus compatible with polymorphism.

Again, time constraints have prevented a thorough consideration of the issues associated with selective inheritance.

2.6 Summary

Increasing awareness of the object-oriented programming paradigm has led to recent experiments with object-oriented model definition languages, some of which were reviewed at the beginning of this chapter. Object-oriented principles state that data (or declarative knowledge) should be declared with the routines (or procedural knowledge) that operate on that data in a single entity known as a *class*. However, we argue that model entities should only contain declarative knowledge concerning the physical behaviour of a system. Procedural knowledge is only required when an instance of a model entity is used for a particular activity, such as dynamic simulation, and is specific to that activity. The declarative knowledge encapsulated by a model entity should therefore only be matched with procedural knowledge in the description of an individual activity. This enables the information declared within a model entity to be reused for a wide range of activities, including both steady-state and dynamic simulation, optimisation, and parameter estimation (see chapter 4). As a consequence, model entities are considered to be complex *type* declarations, not *class* declarations.

Drawing on the contributions of several existing modelling languages, a powerful language for the declaration of primitive model entities was introduced. In particular, this new language offers a more general representation of the physico-chemical discontinuities that occur frequently in physical systems, and extensive support for the complex regular structures that are also a common feature of such systems.

The development of model entities for large, complex systems via the hierarchical combination of a series of submodels was then considered. The key feature of the language structures thus developed is the uniform treatment of model entities regardless of their position in this hierarchy. This will have important consequences for the uniformity of the language structures employed to model the control actions applied to a processing system (see chapter 3). Moreover, the advantages, from the point of view of combined discrete/continuous simulation, of the uniform treatment of equations and the continuous connections (streams) between submodels were stressed.

Finally, hierarchical model development through inheritance was briefly discussed, with particular reference to the notion of polymorphism, which will be exploited extensively in the chapters that follow.

Chapter 3

External Actions Imposed on a System - Task Entities

In the preceding chapter, it was argued that the discrete changes that a processing system may experience fall naturally into one of two categories. From this categorisation the two fundamental structures of the simulation language have evolved. The first of these, the *model entities* described at length in the previous chapter, are used to describe the physico-chemical mechanisms governing the continuous time dependent behaviour of unit operations, including any discontinuities arising from these mechanisms. This chapter introduces the second of these fundamental structures, the *task entities*, that are used to describe the control actions and disturbances imposed on a processing system by its environment.

It may be worth noting at this point that the distinction between categories of discrete change is primarily intended as an aid to the model builder. In some practical situations, it may not be entirely clear (or important) whether a discrete change occurs within the physical system being modelled, or as the result of an external action. In such situations, the model builder is free to choose the description which is more appropriate to his or her needs. Consider for instance the case of a pressure-relief valve that the control system opens whenever the pressure in a vessel exceeds a certain critical value. If one is particularly interested in the precise behaviour of the control system (e.g. the effects of the nonzero sampling time on the safety of the system), then one could consider the opening of the valve as one of the control actions imposed on the vessel model. On the other hand, if these details are not important, the valve behaviour could be described by an appropriate physico-chemical discontinuity within the continuous model.

It is also important to recognise that the control actions or disturbances a processing system may experience are again combined discrete/continuous in nature. Some, such as the action of a discrete controller or the opening and closing of manual valves, are purely discrete, whereas others, such as a disturbance that is ramped between two steady values, have both discrete (initiation and termination) and continuous (ramping) characteristics. In the case of an analogue controller, the continuous aspects of the control action are normally considered to form part of the continuous model of a system, but the discrete actions that place the control loop under manual or automatic control are considered to be imposed on the system by its environment.

A simulation language suitable for the description of processing systems must provide a formalism for the interaction of control actions or disturbances (represented by task entities) with the processing equipment (represented by the combined model) that corresponds closely to the engineer's perception of the system behaviour. In addition, suitable mechanisms for the detailed description of the complex operations taking place must be provided. The following section includes a critical review of the formalisms provided for this interaction by existing simulation languages, and proposes a novel approach that forms the basis of the design of this aspect of the simulation language.

3.1 Formalisms for the Interaction of External Actions with a Processing System

The search for a suitable formalism for this interaction begins with an examination of the facilities offered by continuous process simulation packages, taking SpeedUp (Prosys, 1991) as an example.

Although originally developed as a continuous simulation package, Speedup does allow discrete changes to be imposed on the continuous model during a simulation in order to model perturbations from its previous state. Discrete changes and the events that trigger them are declared in the OPERATION section of the SpeedUp language with an IF/THEN/ENDIF structure that only allows the definition of exogenous time events (see section 1.4.3) relating to changes in the system inputs. The description of simple sequences of external actions is possible with this methodology, but its limitations soon become apparent if the description of a complex sequence of operations such as a start-up procedure or a batch recipe is attempted.

The External Data Interface (EDI) of SpeedUp (Prosys, 1991) enables external programs, such as a supervisory control system (Cott, 1989) or a computer based operator training system (Kassianides, 1991), also to impose instantaneous changes to the system's inputs. Through the EDI, an external program may be used to drive a SpeedUp simulation through a fairly complex sequence of operations.

Overall, current continuous process simulation packages provide a very poor representation for the external actions applied to a processing system, the key disadvantages being:

- Only discrete changes to the inputs are tolerated.
- The built-in formalism for the interaction of a system with its environment is not suitable for the description of complex sequences of operations.
- Therefore, in practice, these operations can only be coded in an external program which must then be interfaced to the simulation package.

On the other hand, workers interested in developing simulation packages for batch processing systems have recognised the need for combined discrete/continuous simulation for many years (Fruit *et al.*, 1974), so a study of one of these systems, BATCHES (Joglekar and Reklaitis, 1984; Clark and Kuriyan, 1989), may yield useful insights.

The key representational issue in the development of models for batch production facilities is the flexible, multiproduct/multipurpose nature of such systems. In a multiproduct plant, different products requiring the same sequence of operations share the same equipment at different times, whereas in a multipurpose plant several products with different production routes are produced simultaneously. Representations for batch systems are therefore usually based on batches of the various products moving through the plant; a material-oriented approach as opposed to the equipment-oriented (or unit operation based) approach taken by continuous process simulation packages.

A BATCHES simulation description is based on the set of products produced by a facility. The manufacture of a product requires a series of operations, each performed in a different item of process equipment. Each product therefore has a network of *tasks*[1] associated with it that describes this series of operations. A task describes an operation carried out in its entirety in one item of equipment, so the simulation of continuous processes, or periodic processes such as pressure swing adsorption, could be considered to be simulations that only require a single task in order to describe them fully.

[1]These should not be confused with the task entities (introduced later in this section) that are the main subject of this chapter.

An operation performed on a product, carried out in its entirety in a single item of process equipment, normally requires a sequence of elementary processing steps. For example, a batch reaction operation may consist of the following elementary steps:

1. Charge the reactor with the various reactants.
2. Preheat the reactor to the operating temperature.
3. Add catalyst.
4. Wait until the desired conversion has been achieved.
5. Allow the reactor to cool to a temperature suitable for discharge.
6. Discharge the reaction mixture into a downstream vessel.
7. Clean the vessel in anticipation of future operations.

Elementary processing steps are represented by the fundamental building blocks of a BATCHES simulation, the *subtasks*. Each subtask is characterised by the set of DAEs that determine the continuous time dependent behaviour of the operation in question for the duration of the elementary step. A library of subtasks representing elementary processing steps occurring frequently in batch systems is made available to the user.

The model of a complete task is built by specifying the order of execution of a series of library subtasks. A task therefore defines a sequence of initial value problems: each problem is characterised by a different set of equations (provided by the subtask in question), and the initial condition for one problem is determined automatically from the final condition of the preceding problem. A Product Processing Sequence then co-ordinates the entire simulation by initiating production of batches of the various products, terminating when the required number of batches or amount of material has been processed.

The decomposition of a simulation description into products, and then a network of tasks provides a powerful representation for many batch processing systems. Here, however, it is argued that the description of tasks in terms of the execution of a series of subtasks, each characterised by a unique set of describing equations, also has considerable drawbacks, especially for a simulation package designed for the entire range of process operations, from purely continuous to batch.

The use of a different continuous model for each subtask has one obvious advantage: the number of equations required to describe the entire process at any point in time is kept to the absolute minimum. For example, during the preheating and cooling phases of the reaction operation described above, it may be possible to drop the component balance equations and only solve the energy balance equation. In the simulation of small batch plant (Joglekar and Reklaitis, 1984), the use of the subtask representation resulted in the dimensionality of the simulation problem fluctuating dynamically between 2 and 27 equations.

The disadvantage of the subtask approach lies in the fact that a continuous model must be posed separately for each elementary step involved in an operation. This can be tedious even for small models in which the bulk of the describing equations are common to all steps, and becomes impracticable as the number of equations required increases, both from the point of view of the sheer programming effort required, and in terms of ensuring correctness of the models. For example, it would be impracticable to re-pose the model of a continuous process each time an external action is performed during a complex start-up schedule, especially as each external action would usually affect only a small part of the overall system.

A subtask describes a period of continuous simulation terminated by some form of discrete change to the continuous model. Usually these discrete changes will only affect a subset of the describing equations, and in many cases all that will distinguish one elementary processing step from another is the forcing functions. It would therefore seem more natural to pose one continuous model for an entire operation, and then provide a means for the discrete manipulation of this continuous model or its forcing functions at the end of each elementary processing step.

This conclusion is fundamental to the design of the simulation language and provides the justification for the development of a novel formalism for the interaction of external actions with a processing system. According to this new approach, an operation performed in its entirety in a single item of process equipment[2] is considered to be described by a single combined discrete/continuous model entity over the entire time horizon of the operation. The simulation description is then completed by a *schedule* of tasks representing the external actions applied to the system during the particular operation; execution of this schedule will

[2]An item of process equipment here could be a single batch unit or an entire continuous plant

drive the continuous model through the desired elementary steps.

The new approach is also introduced in an attempt to emulate the manner in which processes are operated in reality. A schedule of tasks mirrors the action of an operator or control system on a process. To accommodate the diverse needs of many different applications, it has global access to the underlying continuous model and modifies this model according to the desired objectives. During the simulation of a start-up procedure, for example, the schedule of tasks will manipulate the continuous model, just as operators will manipulate the plant itself by opening valves or placing control loops under automatic control.

The above representation can be contrasted to that employed by conventional combined discrete/continuous simulation languages (for example, SYSMOD (Smart and Baker, 1984)), where the simulation of an operation performed in its entirety in a single item of process equipment would be described in terms of continuous and discrete *blocks*. The continuous block contains a declaration of the continuous describing equations that characterise the operation, analogous to a model entity, and the discrete block contains a declaration of all possible time or state events that may occur during the simulation. Associated with each event is a sequence of operations that are executed instantaneously upon occurrence of the event. These operations may include discrete manipulations of the describing equations declared in the continuous block, or the scheduling of a time event at some future time. Control of the simulation is passed back and forth between the continuous and discrete blocks as a result of the occurrence of events and the termination of the actions associated with events.

However, the conditions that result in state events must be included in the declaration of the continuous block. Consequently, the declaration of the continuous model of a system becomes intimately entwined with the control actions imposed during a particular simulation. This has severe and adverse consequences regarding model reusability, as it is usually impossible to predict all the future external actions that may be applied to a system at the time its model is declared. This thesis argues that the set of control actions applied to a system is procedural knowledge specific to one or more dynamic simulations, which should remain decoupled from the declarative information encapsulated by the continuous model.

On the other hand, a formalism involving a schedule of tasks that drives a model entity through the desired simulation greatly enhances the possibilities for reuse of the model entity, because it is not tied to the external actions imposed during an individual simulation. A combined discrete/continuous model entity is a declaration of truth concerning the physical behaviour of a system which is decoupled from any external actions applied to that system during a particular operation. The set of schedules that may then be applied to an individual model entity is potentially infinite.

The remaining sections of this chapter deal with the development of language structures for the description of the external actions imposed on a system during a particular operation. This discussion begins with a description of the *elementary tasks*. These represent the fundamental discrete manipulations that a schedule of tasks may impose on the underlying continuous model, thereby providing the interface between the processing system and the external actions it experiences. Next, the ordering and execution of these elementary tasks in the time domain using a *schedule* is detailed. Finally, the issues involved in managing the complexity of external actions, and creating tasks that may be reused for many different simulations are considered.

3.2 Elementary Tasks

Elementary tasks define the fundamental discrete manipulations that a schedule may impose on the underlying continuous model. They are a formalism of the mathematical requirements outlined in section 1.4, providing language primitives that enable the consequences of simple external actions to be modelled.

At the most fundamental level, the external actions imposed on a processing system during a particular operation can be modelled in terms of discrete changes to some aspect of the continuous model of that system. Mathematically, there are, in fact, only two such aspects of a continuous model: the set of *variables* that describe the time dependent behaviour of the system, and the set of *equations* that determine this behaviour.

A discrete change to the set of equations takes the form of the instantaneous disposal of a subset of these equations and replacement with new equations, similar to the effect of a physico-chemical discontinuity. In order for an equation to be replaced, it must be possible to identify it uniquely. For convenience, each equation is considered to belong to one of three categories:

- The model or *general equations*, expressing general relationships between the system variables. These will include all the equations declared explicitly within the EQUATION section of model entities, and must be associated with an identifier on declaration in order to be uniquely identified during a simulation.

- The *connectivity equations*, established by stream connections between submodels. At present, these always take the form of equality constraints between variables on each side of the connection and again must be associated with an identifier on declaration in order to be uniquely identified during a simulation.

- The *input equations* or forcing functions, defined as the assignment of a function dependent on time alone to a single system variable, thus rendering the latter an *input variable* for the period of simulation in which the forcing function is included in the continuous model. Input equations are uniquely identified by the system variable to which the forcing function is assigned.

For the purposes of this thesis, it is assumed that the set of variables remains unchanged for the duration of an operation performed in its entirety in a single item of process equipment. The total number of equations must therefore remain constant, although the number of equations in each of the above categories may vary. In spite of this restriction, the UNDEFINED construct of section 2.3.5.4 does enable the specification of periods in which certain members of the set of variables become undefined, thereby reducing the number of equations that actually have to be solved during these periods.

Discrete changes to the set of equations, arising from either external actions or physico-chemical discontinuities, will usually lead to discontinuities in the values of a subset of the system variables or their time derivatives. In order to determine consistent values for these variables immediately following such discrete changes, it is normally assumed that the values of the variables whose time derivatives appear explicitly in the new set of equations (the *differential* variables) are continuous across the discontinuity, but that their time derivatives and the values of the remaining *algebraic* variables may be discontinuous.

Nevertheless, there are also certain external actions that can be modelled as Dirac delta functions which may cause a discontinuous change in the values of one or more differential variables. We therefore need mechanisms for defining such discrete changes (see section 3.2.3).

Changes to the set of input equations could equally be regarded as discrete changes to the values of input variables, but their effect is more similar to changes to the set of equations. Moreover, considering the input equations as interchangeable with other equations enables the number of input variables to vary during the course of a simulation.

3.2.1 The RESET task

The RESET task is the first of two elementary tasks that causes an instantaneous change to the describing equations of a system: it defines the replacement of one or more input equations with an equal number of new input equations involving the same set of variables. Execution of a RESET task at some point during a simulation thus results in an instantaneous change to the forcing functions assigned to one or more of the current input variables. This is the simplest form of discrete manipulation a continuous model may experience during a simulation, and is, in fact, the only manipulation allowed by continuous process simulation packages such as SpeedUp.

The introduction of a special language primitive for the declaration of this limited form of discrete manipulation is justified by the fact that it is by far the most commonly encountered manipulation experienced by processing systems. Furthermore, the language structures required to express this special case can be considerably simplified. Step changes to an input, and the initiation or termination of the ramping of an input are both examples of external actions that are modelled by this form of manipulation.

A special syntactical form is used for the declaration of input equations. This consists of a reference to the input variable on the left hand side separated, by the assignment operator, from a real expression on the right hand side that must not involve references to any of the system variables. A distinction is made between the assignment operator (:=) and the equality operators (= or IS) used in general and connectivity equations, in order to emphasise the role of input equations in assigning a function of time alone to an input variable. In fact, because input equations define a unique value for the input variables concerned at any point in the time domain, it is not necessary to solve them simultaneously with the remaining system equations. At any point in time, the input equations can be used to determine the values of the input variables, which can then be considered to be fixed during the iterative solution of the remaining describing equations for the values of the remaining variables.

The declaration of a RESET task merely consists of a list of the new input equations to be inserted in the continuous model. The set of input equations to be dropped from the continuous model need not be specified explicitly since it can be determined automatically from the input variables that appear on the left hand side of the new set of equations. Obviously, this list of input equations may only involve those variables that are already considered to be input variables at the time of execution, and, in order to avoid an ambiguity concerning which equation is actually inserted in the continuous model, each input variable may only appear once. FOR and WITHIN structures, similar to those that exist for general equations, are provided to aid in the declaration of these new equations.

Figure 3.1 demonstrates two applications in which a RESET task is employed to manipulate dynamically the continuous model of a system. In the first example, a RESET task is used to model the opening of a manual valve by a process operator. The continuous model of the valve contains a variable representing the position of the valve stem and an equation that, according to the position of the stem and the inherent characteristic of the valve, relates the flowrate through the valve to the pressure drop across it. The stem position variable is considered to be an input variable, so its value is determined by an input equation. The RESET task shown reaches into the continuous model and directly manipulates the equation that determines this value, just as an operator would walk into a plant and manipulate the valve. The action is considered to occur in such a small time interval relative to the length of the overall simulation, that it can modelled as an instantaneous change. If the action in fact took an appreciable length of time to perform, it could be modelled by two RESET tasks separated by a time interval.

The second example demonstrates how the action of a digital controller at the end of its sampling interval might be modelled. Here, the expression on the right hand side is evaluated at the time of execution of the RESET task. The value obtained is used to update the position of the control valve stem discretely according to a proportional-integral control law. The movement of the valve stem is again considered to be both instantaneous and correct. If this were not the case, the control signal to the valve could be considered to be an input variable instead, and the stem position determined from a differential equation involving this signal included in the control valve model.

```
# Open a manual valve instantaneously
RESET
  Valve.Position := 1.0 ;
END

# Action of a digital controller at the end of its sampling interval
RESET
  Valve.Position := Bias + Gain*(Error + Integral_Error/Reset_Time) ;
END
```

Figure 3.1: Two applications of the RESET task

RESET tasks may also be used to alter the value of selector variables (see section 2.3.5.2). Manipulation of a selector variable by a RESET task forces the continuous model to change state as a result of an external external action as opposed to a physico-chemical mechanism. Applications include the modelling of the replacement of a shattered bursting disc by an operator, a transition that would not normally be described as a physico-chemical mechanism, and the switching on or off of a pump described by the model entity shown in figure 3.2.

Here, the model entity has two states designated by the selector variable `Status` and corresponding to whether or not the pump is switched on. When the pump is switched on, the pump characteristic relates the pressure rise across the pump to the flowrate of material through the pump, and when it is switched off, the pressure rise is set to zero (or even a negative value, e.g. a pressure drop, related again to the flowrate). No transitions link these states under normal operating conditions.[3] Whether the pump is initially switched on or off forms part of the initial condition of any simulation, and external actions during the simulation, modelled by RESET tasks (see figure 3.3), can cause dynamic changes to this status.

Of course, the use of external actions to alter a model's state must have physically meaningful consequences. An attempt, for example, to change the state of a flash vessel from supercooled liquid to two phase while the contents of the vessel remain below the bubble point would most probably lead to failure of the simulation.

[3]Of course, a series of trip conditions that result in a transition from the `On` state to the `Off` state could easily be included in the model, if necessary.

```
MODEL Pump

VARIABLE
  Flow_In, Flow_Out      AS Flowrate
  Press_In, Press_Out    AS Pressure
  Press_Rise             AS Positive

SELECTOR
  Status                 AS (On,Off)

EQUATION

  Flow_In = Flow_Out ;

  Press_Out = Press_In + Press_Rise ;

  CASE Status OF
    WHEN On  : Flow_Out = f(Press_Rise) ;
    WHEN Off : Press_Rise = 0 ;
  END # case

END # Pump
```

Figure 3.2: Model of a Pump

```
# Turn the pump on.
RESET
  Pump.Status := Pump.On ;
END
```

Figure 3.3: Manipulation of selector variables by a RESET task

3.2.2 The REPLACE task

The REPLACE task takes one or more describing equations of a system and replaces them with an equal number of new equations. The type of the equations discarded and the type of the new equations inserted in the system model are irrelevant, all equations being considered equally interchangeable for the purposes of this operation. It is therefore possible to replace an input equation involving one variable with a new input equation involving another variable, or to replace a general equation with an input equation and vice versa. A

```
# Close the control loop
REPLACE
  Control_Valve.Position
WITH
  Automatic AS Control_Valve.Position = Control_Valve.Signal ;
END

# Open the control loop
REPLACE
  Automatic
WITH
  Control_Valve.Position := 1.0 ;
END
```

Figure 3.4: Applications of the REPLACE task

REPLACE task also provides the means to change the topology of a plant dynamically, by allowing a connectivity equation to be replaced by another connectivity equation, or even an equation belonging to either of the other categories. A RESET task is therefore just a special form of the REPLACE task that can only replace an input equation with a new input equation involving the same variable.

It must, of course, be possible to identify an equation uniquely in order to replace it. As already stated, input equations can be identified by the associated input variable, whereas connectivity equations and general equations must be associated with an identifier when declared in a model entity (see section 2.3.4) in order to be replaced. If any of the new equations inserted in a continuous model by a REPLACE task are to be manipulated in this manner at some future point in time, their declaration must also be accompanied by an identifier.

An example of the application of a REPLACE is shown in figure 3.4, where a control valve is switched from manual to automatic analogue control. This is achieved by replacing an input equation involving the stem position with a general equation relating the control signal to the valve to the stem position. The new equation is associated with an identifier, so that the second REPLACE task shown in figure 3.4 may later be employed to switch the control valve back to manual control.

At any point in time during a simulation, the set of variables describing a system will be split into three categories (see equation 1.3):

- The *input* variables, already described earlier, the values of which are determined uniquely from the input equations.
- The *differential* variables, the time derivatives of which appear in the describing equations.
- The *algebraic* variables, being the remaining variables whose time derivatives do not appear explicitly in the describing equations.

The presence of REPLACE tasks in a schedule may result in the status of variables changing dynamically during a simulation. For example, execution of a REPLACE task that replaces an input equation with a general equation will result in the status of the variable concerned changing dynamically from input to algebraic or differential, depending on the way it appears in the new equation. It is therefore possible for the number of differential variables describing a system to vary during a simulation. For example, if all the equations involving the time derivative of a particular variable are dropped from the continuous model, the number of differential variables will be reduced by one. Similarly, it is possible for a physico-chemical discontinuity to vary the number of differential variables describing a system.

Changing the number of differential variables in this manner only poses problems when the number of differential variables increases. Each time a discrete change to the describing equations of a system occurs during a simulation, a new initial value problem is posed. This new problem starts from the point in time denoted by the event that triggered the change, and requires an initial condition in order to determine consistent initial values for all the system variables. The final values of the differential variables in the preceding initial value problem normally provide this initial condition. If, however, new differential variables are introduced as a consequence of the discrete change, initial values must also be found for these variables. At present, it is assumed that the values of these variables can also be carried over from the preceding problem.[4]

The new initial value problem resulting from a dynamic change to the describing equations must represent a well-posed dynamic simulation problem in its own right. As

[4]If this is not true, a REINITIAL task (see section 3.2.3) can be used to specify the correct value for the new differential variable.

already discussed in section 1.4, additional limitations may also be imposed by the numerical algorithms employed for the solution of the describing equations, in particular those imposed by the problems associated with the solution of DAEs of index exceeding unity (Pantelides *et al.*, 1988). It is, after all, very easy to increase dynamically the index of the describing equations with a REPLACE task, the addition of an input equation constraining the value of a differential variable being the simplest example.

A further interesting application of the REPLACE task is to the automatic calculation of the steady-state bias of an analogue controller. In order to determine this bias, a steady-state calculation is performed in which the controller error is constrained to zero by an input equation, and the bias is considered to be a calculated (algebraic) variable. The value for the controller bias determined by this calculation corresponds to the correct steady-state bias for the controller. Before dynamic simulation begins, execution of the REPLACE task in figure 3.5 can automatically replace the input equation involving the controller error with an input equation constraining the controller bias to its current value. The controller error, now an algebraic variable, is then free to fluctuate as disturbances are introduced and the controller attempts corrective action.

```
# Set the controller bias
REPLACE
  Controller.Error
WITH
  Controller.Bias := OLD(Controller.Bias) ;
END
```

Figure 3.5: Automatic calculation of controller bias using a REPLACE task

The above example also illustrates the use of the special built-in function OLD. This is used when it is desirable to be able to express the new equations inserted in the continuous model in terms of the values of the variables immediately *before* the discontinuity. When an expression involving this function is inserted in the continuous model, the function is symbolically replaced by a real constant corresponding to the value of its argument evaluated at that point in time.

OLD is a vector function. It may also appear in RESET and REINITIAL tasks (section 3.2.3), and can be used to include the current time as a constant in any expression. It has no meaning within model entities, where no well-defined values for the variables exist before simulation commences. As a consequence, for example, replacing two input equations simultaneously in the same RESET task becomes subtly different from replacing each equation individually using two RESET tasks executed in a sequence. This is because the reinitialisation calculation following the execution of the first RESET task in the sequence may alter the values of variables that appear in OLD functions on the right hand side of the second RESET task.

It is interesting to note that, although the model described in the example of figure 3.5 results in a perfectly legitimate steady-state calculation, any attempt at dynamic simulation without releasing the constraint on the controller error could encounter problems. This is because the input equation concerned effectively constrains the controlled variable to its set point, forcing the simulation to determine the trajectory of the manipulated variable that would achieve this 'perfect' control, a typical example of a problem of index exceeding unity. Only by releasing this constraint with a REPLACE task, thereby reducing the index to unity, is it possible to perform a simulation with existing numerical codes. This suggests that any analysis of the describing equations of a system to determine their index should be postponed until an attempt at integration is made, and that this analysis should be repeated before each attempt at integration following a manipulation of the describing equations.

3.2.3 The REINITIAL task

Whenever a discrete change occurs to the continuous model of a system, one of the key concerns is the precise definition of the condition of the system immediately following the discontinuity – this, together with the modelling equations, normally determines a unique trajectory for the system variables until the next discrete change. In continuous simulation packages such as SpeedUp, the usual assumption is that the values of the differential variables remain unaltered across any discontinuity, which, for most cases, is sufficient to define uniquely the system condition thereafter.

However, such assumptions preclude situations in which it may be desirable to define instantaneous changes to the values of differential variables. For example, when an analogue control loop involving integral action is switched to automatic control, the integral of the controller error is normally initialised to zero in order to eliminate any reset windup that may have occurred while the loop was under manual control: a situation that is most easily modelled by instantaneously changing the value of the variable representing this integral (a differential variable) to zero. Moreover, the characterisation 'instantaneous' may simply denote changes that occur on a much smaller time-scale than the rest of the simulation, the details of which are not considered to be important. Examples could include the 'instantaneous' dumping of solid reactant or catalyst into a batch reactor by an operator, or the very rapid heating of the reactor contents.

The REINITIAL task is introduced in order to accommodate this type of discrete change. Its declaration consists of a list of the differential variables that are to be considered discontinuous, and the relationships that will replace the corresponding continuity assumptions in determining the consistent initial values for system variables immediately following the discontinuity. The specified relationships are nonlinear equations that may involve any system variable, including the values of system variables immediately before the discontinuity. The OLD function introduced in the previous section may be used for the purpose of specifying the latter. Thus, overall, the REINITIAL task represents a complete implementation of the general types of discrete change envisaged in section 1.4.5.

Two examples of the application of the REINITIAL task are shown in figure 3.6. The first example demonstrates how such a task can be used to model the initialisation of an analogue controller. Execution of the task will result in a reinitialisation calculation in which the usual assumption concerning the continuity of the variable representing the integral of the controller error is replaced by an equation that constrains its new initial value to zero. The second example demonstrates how the equation that determines the new initial value of a differential variable does not necessarily have to involve the differential variable itself. Here, a step change in the temperature of the vessel (an algebraic variable) implies a step change in the internal energy holdup.

The number of equations specified in a REINITIAL task must match the number of discontinuous differential variables. These equations are declared in the same manner as equations declared in model entities, including all structured equations except the CASE equation. Use of the latter would be meaningless because a CASE equation must remain locked in a single clause during a reinitialisation calculation.

```
# Initialise the integral error of an analogue controller
REINITIAL
  PI_Controller.Integral_Error
WITH
  PI_Controller.Integral_Error = 0 ;
END

# Flash heating of the contents of a vessel
REINITIAL
  Batch_Reactor.U_Holdup
WITH
  Batch_Reactor.Temp - OLD(Batch_Reactor.Temp) = 20 ;
END
```

Figure 3.6: Applications of the REINITIAL task

The equations declared in a REINITIAL task only serve to define consistent initial values for the system variables immediately following the discontinuity, and are not included in the continuous model thereafter. Execution of a REINITIAL task is immediately followed by a reinitialisation calculation. This calculation determines consistent initial values for all system variables from the simultaneous solution of the current set of describing equations, the set of continuity assumptions for all differential variables *not* listed in the REINITIAL task, and the set of equations declared in the REINITIAL task. The latter two sets of equations are then dropped from the continuous model and numerical integration of the current set of describing equations proceeds as before.

3.2.4 The MESSAGE task

It is desirable that a simulation executive based on an implementation of the proposed simulation language be able to provide a report on the progress of a simulation. Examples of the information that could be derived automatically from a simulation description for inclusion in this report include:

- The time of occurrence of physico-chemical discontinuities, including the logical condition that led to the discontinuity, new and old values for selector variables, and even details of the new equations inserted in the continuous model.

- The time of occurrence of time and state events, including the logical condition triggering them.
- Details of the execution of elementary tasks, including the model manipulations performed by these tasks.

This information is often as important as that contained within the time trajectories of the describing variables, providing a means by which the discrete component of a combined discrete/continuous simulation may be recorded and analysed. In many situations, however, the user may wish to include more detailed information concerning the progress of the operation under investigation in this report. For example, it may be desirable to send messages concerning the completion of certain elementary processing steps, the outcome of decisions programmed into the schedule, or even warnings and alarms.

The MESSAGE task is introduced in order to include customised messages on the status of a system in the simulation report. Execution of such a task occurs instantaneously, and will result in the message declared within the task being inserted in the simulation report. At present, only messages composed of a predefined string of characters enclosed within quotes may be issued, e.g. `MESSAGE "Tank Full"`.

3.3 Schedules of Tasks

While elementary tasks are used to model the individual discrete actions applied to a processing system during a particular operation, a schedule provides a means by which the order and time of execution of these individual actions may be declared. The execution of this schedule drives the underlying model entity through a simulation of the operation under consideration.

In many situations several different agents may act on a processing system simultaneously. For example, if the start-up of a plant is to be conducted by a team of operators, each operator will move through the plant relatively independently in order to perform the allocated control actions, although the team will probably periodically communicate in order to synchronise and co-ordinate their actions. Furthermore, in certain situations, it is most natural to model digital controllers as agents that act on a system concurrently with each other and with any other external actions experienced by the system, whereas in others the complete computer control system can be modelled as a single agent that acts on a system concurrently with any disturbances. The schedule must therefore reflect the concurrent nature of the real world through appropriate language structures.

If a schedule is a means by which the order of execution of the external actions applied to a processing system can be declared, it may be argued that the language structures required to support this will probably be very similar to the control structures used by general-purpose programming languages to specify the order of execution of a series of statements by a computer. This conclusion is reflected in the design of many simulation languages, in which the consequences of an event are essentially declared using a programming language, and in special-purpose languages for the computer control of sequential process operations (Rosenof and Ghosh, 1987).

Unfortunately, most conventional programming languages require all statements to be executed in a sequence, although with more modern programming languages, such as Modula–2 (Wirth, 1988), it is possible to emulate concurrency through the use of coroutines. Attempts by developers of simulation languages to overcome the problems of representing essentially concurrent systems in a sequential environment have resulted in the various world views of discrete event simulation (Kreutzer, 1986). Probably the best solution to this problem is provided by the *process interaction* world view, typified by the SIMULATION class of Simula (Birtwistle *et al.*, 1979), a general-purpose programming language designed with discrete event simulation specifically in mind.

Extensions of the SIMULATION class of Simula have formed the basis of several combined discrete/continuous simulation languages, including DISCO (Helsgaun, 1980) and CADSIM (Sim, 1975). Although both these languages are extensions of a discrete event simulation language to include limited continuous simulation capabilities, the process interaction world view inherited from Simula provides a powerful formalism for the inherent concurrency of the real world. According to this world view, a system is modelled by a web of concurrent entities that periodically interact with each other. The life cycle of each entity is declared using a Simula class containing details of the sequence of actions undertaken by the entity during its life cycle, which may include creation or termination of other entities. The simulation is initiated and co-ordinated by a main program, or master procedure, that creates and terminates entities as required. On a sequential computer, the execution of all entities currently active in the simulation takes places in a quasi-parallel fashion; only one entity is actually executed at any point in time, but control is passed from entity to entity in order emulate concurrency. The simulation terminates when execution of the master procedure is complete, and all active entities have completed their life cycles.

The advent of parallel processing, in particular the MIMD (Multiple Instruction Multiple Data) computers, and the subsequent development of the so-called parallel programming languages, such as OCCAM (INMOS, 1984), has provided an alternative methodology for the description of concurrent systems. The ability of parallel machines to execute commands and manipulate data concurrently has prompted the developers of these parallel programming languages to provide structures that enable a programmer to express explicitly which tasks are to be executed concurrently.

The above discussion has highlighted the fact that there are at least two means by which the inherent concurrency of a system may be expressed:

- The concurrency can be declared explicitly through language structures similar to those of the parallel programming languages.
- The concurrency can be implied by adoption of the process interaction world view of discrete event simulation.

Experience suggests that there is a role for both methodologies in the description of the external actions experienced by a processing system. The nature of a system, or the features of the particular study being conducted, will usually dictate which approach is adopted, although in certain situations a combination of both approaches could be the most appropriate. Consider, for example, a process in which several digital controllers are employed in order to maintain the plant at a nominal steady-state. One simulation study may attempt to assess the performance of these digital controllers in response to a series of disturbances. In this case, because the study focuses on the performance of the controllers, the schedule would be comprised of an explicit declaration of the execution of these controllers in parallel with any disturbances. However, in another situation it may be necessary to use simulation in order to validate the feasibility of a proposed start-up procedure. The start-up procedure will consist of a sequence of actions to be undertaken by the operator in order to achieve the steady-state operating point. At several points in this sequence, the operator will be required to place digital control loops under automatic control. The process interaction world view is much more convenient in this situation – the act of closing the control loop would result in the activation of a separate digital controller entity. The operation of the latter in parallel with the remaining sequence of actions would be implied, without the need for an explicit declaration. In effect, once the controller is spawned by the main schedule, it forms an independent entity and the main schedule has no further responsibilities for, or direct knowledge of, its actions.

Language structures for the explicit declaration of concurrency in a schedule will be introduced in this section. We will return to the process interaction world view at the end of this chapter and demonstrate that the two approaches are in fact equivalent, although it must be recognised that there are circumstances in which one approach is more convenient notationally than the other.

3.3.1 The Basic Control Structures - Sequential and Concurrent Execution

A schedule may be used to define explicitly the execution of a series of tasks in either a sequential or concurrent fashion, or any combination of the two. The definitions of sequential and concurrent execution are strict:

- **Sequential execution** – Execution begins with the first task and will only proceed to the next task when execution of the preceding task has terminated. Execution of the sequential structure is complete when execution of the last task has terminated.

- **Concurrent execution** – Execution of all tasks begins simultaneously and then proceeds concurrently. Execution of the concurrent structure is only complete when *all* tasks have terminated.

A schedule is written in a similar manner to a programming language. The order in which the tasks declared therein are executed is determined by control structures. The sequential control structure consists of a list of tasks between the keywords SEQUENCE and END, whereas the concurrent control structure consists of a list of tasks between the keywords PARALLEL and END. An entire control structure is itself a task, so nesting of control structures can be used to describe complex schedules in a simple and elegant manner. In fact, an

```
SCHEDULE
  SEQUENCE
    CONTINUE FOR 10.0
    RESET Manual_Valve.Position := 1.0 ; END
    CONTINUE FOR 53.5
    RESET Manual_Valve.Position := 0.0 ; END
    CONTINUE FOR 5.0
  END # sequence
```

Figure 3.7: Simple Sequence of External Actions

entire schedule is defined by the execution of a single task, although this task will usually be a control structure that defines the order of execution of a set of other simpler tasks.

Figure 3.7 demonstrates how the sequential control structure may be utilised to model a simple sequence of external actions applied to a processing system. The CONTINUE task, which specifies the duration of periods of continuous change between discrete actions, will be discussed in section 3.3.5.

The provision of suitable mechanisms to reflect the concurrent nature of the real world will potentially result in the concurrent execution of two or more tasks that are able to manipulate the same underlying continuous model. If two or more elementary tasks are executed simultaneously with respect to the simulation clock, it is possible that concurrent manipulation of the same aspect of this model will be attempted. A physical analogy to this is an attempt by two operators to open the same manual valve simultaneously. As the consequences of such actions are ambiguous, the simulation language excludes them semantically. For example, in order for the simultaneous execution of two RESET tasks to be accepted, the sets of input equations manipulated by these tasks must be mutually exclusive. Similarly, the sets of differential variables manipulated by REINITIAL tasks executed simultaneously must be mutually exclusive. This discussion also highlights the difference between concurrent and sequential execution of the same list of elementary tasks, even though the simulation clock is not advanced in either case. The execution of the tasks in a sequence will result in a reinitialisation calculation immediately following the execution of each task, each task being able to manipulate the same aspects of the continuous model if it so wishes. On the other hand, the concurrent execution of the tasks will result in a single reinitialisation calculation following implementation of all the manipulations defined by the tasks, and will reject any attempts to manipulate the same aspect of the model more than once.

The two control structures described above provide an excellent basis for the explicit description of the order in which tasks are to be executed during a simulation. However, as with programming languages, other control structures must also be introduced to describe more complex situations. The following sections introduce some of these features.

3.3.2 Conditional Execution of Tasks

In many circumstances the correct external actions to apply to a system cannot be fully determined a priori, and must therefore be established from decisions that can only be made during the progress of the operation in question. Consider, for example, a batch operation involving a series of elementary processing steps applied to a batch of material, after which a decision is made as to whether the batch is acceptable, should receive further processing, or should be discarded. This decision depends on the quality of the batch, so the result can only be established after the preceding processing steps have been completed.

Conditional control structures are introduced to enable selection between alternative external actions based on decisions involving the current status of the system under investigation. As with all other control structures, conditional control structures are considered to be tasks, and may be nested in an arbitrary manner.

The conditional control structure proposed is identical to that offered by most modern programming languages. The basic form of the IF control structure permits a choice between the execution of two alternative tasks to be based on the value of a logical expression. This logical expression may include relationships involving describing variables, and will only be evaluated at the point in simulation time that the control structure is encountered. The ELSE clause defining the alternative task may be omitted. In this situation, the control structure will define a single task that is only to be executed if a certain condition is satisfied.

An example of the application of the IF control structure is shown in figure 3.8, where it used to clip a digital control signal sent to a control valve.

There is a subtle difference between the interpretation of the conditional control structures used to co-ordinate the execution of tasks on one hand, and the conditional equations that are used to declare physico-chemical discontinuities on the other (see sections

```
SCHEDULE
  IF Control_Signal > 1.0 THEN
    RESET Control_Valve.Position := 1.0 ; END
  ELSE
    IF Control_Signal < 0.0 THEN
      RESET Control_Valve.Position := 0.0 ; END
    ELSE
      RESET Control_Valve.Position := OLD(Control_Signal) ; END
    END # if
  END # if
```

Figure 3.8: Example IF Control Structure

2.3.5.2 and 2.3.5.3). Conditional control structures are *procedural* in an identical sense to their counterparts in general-purpose programming languages: the control structure is encountered during execution of the schedule and results in a branch of control, after which it has no further effect on the simulation unless it is encountered again as a result of normal execution. In contrast, conditional equations represent *declarative* knowledge concerning the physical nature of the system under investigation that holds throughout the period in which the equations concerned form part of the continuous model of the system.

3.3.3 Iterated Execution of Tasks

Many processing systems are characterised by the repetitive nature of the external actions required to achieve the desired mode of operation. For example, periodic processes, such as pressure swing adsorption, are usually brought to and maintained at a 'cyclic steady-state' by a sequence of external actions that is applied repeatedly. Moreover, the action of a digital control system on a process can be considered to consist of a regular cycle of continuous operation followed by sampling and discrete actions. The iterative control structures are therefore introduced in order to define the repeated execution of a task. In some cases it may be possible to determine the number of times an operation is to be repeated either a priori or, at least, just before the first execution of the operation, whereas in other cases this is not feasible and the operation is repeated while a logical condition holds true.

The WHILE control structure defines the repeated execution of the task that it

```
SCHEDULE
  WHILE Time < 1000 DO
    SEQUENCE
      CONTINUE FOR 5
      RESET
        Control_Valve.Position := OLD(Control_Signal) ;
      END # reset
    END # sequence
  END # while
```

Figure 3.9: Example WHILE Control Structure

encloses as long as an associated logical expression remains true. When the control structure is encountered, this logical expression is evaluated. If it is found to be true, the enclosed task is executed. The expression is then evaluated again, and if the expression is still true, the enclosed task is executed once more. This process continues until the expression is no longer true, at which point execution of the WHILE structure is considered to be complete. It is important to recognise that the logical expression is only evaluated before the first and each subsequent attempt to execute the enclosed task. If, therefore, the logical expression is initially false, execution of the control structure will terminate instantaneously. Otherwise, the control structure will only terminate if the logical expression evaluates as false at the beginning of an iteration.

The WHILE control structure shown in figure 3.9 demonstrates the regular cycle of continuous operation followed by discrete action exhibited by a digital controller.

3.3.4 Local Task Variables

The preceding sections have shown how the execution of tasks within a schedule may be co-ordinated by control structures. As with any programming language, there is often a need for auxiliary variables that facilitate this co-ordination. These special variables are local to the task: they only exist for the duration of the task's life cycle and are only accessible from within it.

Local task variables may be of an integer, real, or logical type. Integer variables are obviously necessary for the implementation of the FOR control structure. Real variables can be used to accumulate quantities, such as the total material produced by a series of batches, or as an intermediate variable in a calculation, such as the controller error in figure 3.1. Logical variables may be used to store information concerning the status of the system for subsequent use in logical expressions. An example of the declaration of such variables is:

```
VARIABLE
  Error, Integral_Error, Control_Signal              AS REAL
```

Assignment statements are the only means by which the values of these local variables can be altered during a simulation. These statements may appear anywhere in a schedule, are executed instantaneously with respect to the simulation clock, and result in the assignment of a new value to the variable which is derived from the evaluation of an expression of a type compatible with that of the variable. This expression may contain references to the *continuous* variables of the underlying model, which are evaluated at the time of execution, just as elementary tasks may include references to local task variables that are also evaluated at the time of execution. For example:

```
Error := 150 - Temp_Sensor_5.Measurement ;
```

Execution of an assignment statement obviously has no effect on the underlying continuous model, and is not therefore immediately followed by a reinitialisation calculation. Simultaneous attempts to alter the value of the same local variable are rejected.

Extensions to the range of types available, such as arrays of the basic types or LIFO and FIFO queues, may also prove useful but have not been considered in this thesis.

3.3.5 The CONTINUE task

The execution of all the elementary tasks described so far takes place instantaneously with respect to the simulation clock. The final vital component is therefore a mechanism by which the duration of periods of continuous change between discrete actions can be specified. The CONTINUE task is introduced in order to achieve this objective.

Execution of a CONTINUE task suspends execution of the control structure within which it appears, and in doing so schedules the event that will result in resumption of this execution. In order for execution to resume, the simulation clock must be advanced to this event by integration of the underlying continuous model. The CONTINUE task merely has the effect of scheduling an external action at some future point in time, and does not in itself manipulate the continuous model.

The CONTINUE task comes in two forms, distinguished by the nature of the event scheduled. The first form schedules a time event and has already appeared in several examples:

CONTINUE FOR <real expression>

Execution of the control structure concerned will resume after a specified period of time has elapsed. The time period is established from the evaluation of the real expression on execution of the task; if the result turns out to be non-positive, execution of the control structure will resume immediately. The real expression may involve any real quantities that the schedule has access to.

The second form schedules a state event:

CONTINUE UNTIL <logical expression>

Execution of the control structure concerned will only resume when the logical expression becomes true. This logical expression may again involve any quantities that the schedule has access to.

The two forms described above may also be combined in a single CONTINUE task through the use of the OR and AND operators. The OR operator will result in suspension of execution until either the state or time event, whichever occurs first, whereas the AND operator will suspend execution at least until the time event, after which the state event will determine when the schedule is resumed. The OR operator is particularly useful for debugging a simulation description, by providing a upper time limit on a state event that might otherwise never be satisfied. For instance, the task:

```
CONTINUE FOR 100 OR UNTIL Reactor.Conversion > 0.9
```

ensures that simulation is advanced for at most 100 time units even if the reactor conversion never reaches the required value.

3.3.6 Task Entities

We are now in a position to formalise the notion of a *task entity*, which describes a complex set of external actions applied over a finite period of time. All task entities are user-defined, encapsulating a complete schedule and a set of local variables. The life cycle of an instance of a task entity is determined through execution of this schedule.

Task entities are declared in a similar manner to the model entities that were the subject of chapter 2: the keyword TASK followed by a unique identifier by which it may be referred to globally. The remainder of the declaration is split into a series of sections in order to gather the information belonging to a particular category in one place. The VARIABLE section is employed to declare local variables, and the SCHEDULE section defines the schedule which determines the life cycle of the task entity.

Figure 3.10 demonstrates a task entity that models the action of a simple digital controller. Three local control variables are employed to calculate the controller error, a discrete approximation to the integral of this error, and the actual control signal respectively. The life cycle of the task entity involves the execution of two other tasks in sequence: an assignment statement that initialises the integral of the controller error, and another sequence representing the action of the controller, which is executed repeatedly until the termination condition is satisfied (in this case, when one thousand seconds have passed on the simulation clock). Execution of the iterated sequence is suspended by a CONTINUE task for the duration of the sampling interval (5 seconds), after which the controller error is calculated. This value is then used as an intermediate variable in the calculation of the integral error (an accumulated quantity) and the control action. The latter is then implemented by a RESET task, although IF control structures are employed to clip the signal if necessary.

3.4 Reuse of Tasks

The task entity shown in figure 3.10 is extremely specific, being suitable only for the description of one particular digital controller in the context of an individual simulation. It is tied to a unique sensor and valve pair (`Temp_Sensor_5` and `Valve_101`), the sampling interval and tuning parameters are expressed as constant values, and the task must always terminate after one thousand seconds have elapsed. If it were necessary to alter any of this

```
TASK Digital_PI_Control

VARIABLE
  Error, Integral_Error, Control_Signal            AS REAL

SCHEDULE
  SEQUENCE
    Integral_Error := 0 ;
    WHILE Time < 1000 DO
      SEQUENCE
        CONTINUE FOR 5.0
        Error          := 150 - Temp_Sensor_5.Measurement       ;
        Integral_Error := Integral_Error + Error*5.0            ;
        Control_Signal := 0.5 + 1.2*(Error + Integral_Error/20) ;
        IF Control_Signal > 1.0 THEN
          RESET Valve_101.Position := 1.0 ; END
        ELSE
          IF Control_Signal < 0.0 THEN
            RESET Valve_101.Position := 0.0 ; END
          ELSE
            RESET Valve_101.Position := OLD(Control_Signal) ; END
          END # if
        END # if
      END # sequence
    END # while
  END # sequence

END # Digital_PI_Control
```

Figure 3.10: Task to Model a Digital Proportional-Integral Control Law

information, a new task entity would have to be developed. This is clearly unsatisfactory.

The example illustrates the importance of developing task entities in a manner that will maximise their potential for reuse in future simulations. Tasks developed in this manner become candidates for storage in a library of task entities, similar to the libraries of continuous models described in the chapter 2. Reuse of tasks from such a library can then considerably reduce the time required for the development of new simulation descriptions, as part, or all, of the description will already exist as reusable components. This, of course, requires that task entities can be declared independently of the details of an individual simulation.

Parameterisation, in conjunction with inheritance, is already used to enhance the reusability of continuous models. Similarly, the key to the development of reusable tasks is to allow a task entity declaration to be expressed in terms of a series of parameters, through which instances of the task can be tailored to individual applications. The role of inheritance in enhancing task reusability is less clear, because this will involve augmentation of the task's life cycle. The notion of inheritance was probably first introduced by the class prefixes of Simula (Birtwistle *et al.*, 1979), where the execution of a class begins with the life cycle (if any) of its parent class, and then proceeds with its own life cycle. However, process engineering applications for this facility are not immediately obvious. Nevertheless, it will be seen in section 3.4.2 that polymorphic task parameters permit exploitation of the development of continuous models in inheritance hierarchies.

3.4.1 Task Parameters

Task entities may be declared in terms a wide range of parameters in order to enhance their potential for reuse. Useful applications of task parameters include:

- Numerical or logical constants are vital in most applications, such as the specification of controller constants, or the size and duration of a batch.
- Numerical or logical expressions for symbolic insertion in the task can also prove invaluable, in particular for adapting decisions made during execution (e.g. termination conditions).
- Equations can be specified for symbolic insertion in the continuous model by a REPLACE task.
- The continuous model(s) on which the task operates. This is probably the most important application of all, through which a task can be designed to operate on any instance of a particular model entity independently of the details of an individual simulation.

The parameters of a task are distinguished by type, the latter determining the purposes to which the parameter may be applied within the body of the task. The range of parameter types that a task entity may require is thus far wider than those allowed by subroutines of general-purpose programming languages:

- Parameters of the REAL, INTEGER, and LOGICAL types assume a constant or expression of the appropriate type. References to these parameters are interpreted as constant values.

- Parameters of the REAL_EXPRESSION, INTEGER_EXPRESSION, and LOGICAL_EXPRESSION types assume an expression of the appropriate type. References are interpreted as though they have been symbolically replaced by this expression.

- Parameters of the EQUATION type assume an equation. References are interpreted as though they have been symbolically replaced by this equation.

- Parameters of the MODEL type assume an instance of the appropriate model entity. References are interpreted as references to this instance.

Note again the subtle difference between value parameters on the one hand, and expression parameters on the other (cf. section 2.3.1). Although both may be assigned an expression of the appropriate type, value parameters are interpreted as a constant value obtained by evaluation of this expression at the time of creation of the task instance, whereas expression parameters are interpreted as expressions, the value of which can change during the life cycle of the task.

The declaration of each parameter includes an identifier by which it may be referred to within the schedule of the task. An instance of a task entity is only fully defined if all its parameters are assigned values of the correct type.

The task shown in figure 3.11 demonstrates how careful parameterisation can facilitate the development of a reusable task implementation of the digital controller. The parameters of the task include real constants that determine the various tuning parameters and the sampling interval, a logical expression that determines the termination of control, and model type parameters that determine the sensor and the manipulated control valve. Note that the MODEL type parameters are used to access variable attributes of the underlying continuous model, thereby establishing the task entity's independence of a particular sensor and valve combination.

3.4.2 Polymorphism

This section demonstrates how the notion of polymorphism introduced in section 2.5.2 can be used to enhance significantly the potential for reuse of a task entity. In particular, all model type parameters of task entities are polymorphic. In this case, polymorphic parameters may be assigned an instance of any descendant of its base type.

The use of polymorphism enables the declaration of task entities that may operate on a broad range of similar continuous models provided they have some basic features in common (as opposed to being restricted to instances of a single type), whilst still ensuring the task is applied to a suitable model. Careful development of continuous models in inheritance hierarchies will ensure that many operations will only require a single task, where otherwise a task would have to be developed for the same operation applied to each member of the hierarchy. This will, of course, only apply to operations that can be described in terms of the attributes of the base type.

The task entity shown in figure 3.12 is designed to model the opening of a valve under manual control. The only parameter of this task is the continuous model, of type `Generic_Valve`, to which the operation is applied. The model entity `Generic_Valve` is the root model of an inheritance hierarchy of valve models that encapsulates a declaration of all the universal features of valves, including a variable that represents the position of the stem. Operations that open and close a valve can be described completely in terms of this variable, so an `Open_Valve` task that operates on this continuous model can be developed. The inheritance hierarchy that evolves from this root can then be used to declare the details of the entire range of valves encountered in processing systems. However, it is not necessary to develop a task to open each new member of this hierarchy – the task `Open_Valve` will suffice.

3.5 Hierarchical Subtask Decomposition

The language structures that have been introduced up until this point are, in principle, adequate for the description of the external actions applied to a processing system, although in practice the task entities required would become more difficult to develop and validate as the complexity of the external actions increases.

Chapter 2 discussed at length the need for mechanisms for managing the potential complexity of the continuous model of a processing system. Combined discrete/continuous simulation faces the additional problem of dealing with the potential complexity of the external actions imposed on these models. For all but the most trivial simulations, the complexity of the external actions applied to a system may increase just as rapidly as that of the underlying continuous model.

The management of complexity in the continuous model of a system through decomposition into a series of connected components provides a useful insight into the management of task complexity. A complex operation on an item of process equipment can usually be decomposed in terms of lower level operations involving the structural components of this equipment. Here, however, the decomposition is procedural as opposed to structural: the role of the more complex operation is to define the ordering in the time domain of the lower level operations, rather than the structural connections of components. Each of the lower level operations may in turn be decomposed in terms of other operations, the decomposition continuing until all operations can be described in terms of the manipulations of the underlying continuous model made possible by elementary tasks. Decomposition in this manner defeats complexity through restriction of the scope of the problem considered at any point to a manageable level.

So, hierarchies are again the most natural manner in which to manage the declaration of complex external actions. This also offers the advantage that, with the continuous model already decomposed in terms of a hierarchy, the external action hierarchy can be designed to correspond closely to the model hierarchy, and in so doing, exploit the structure of the system under consideration. Consider, for example, a complex operation such as the start-up procedure applied to a distillation column (in itself a system of significant complexity). This operation can be decomposed into a series of simpler operations ordered in the time domain, some of which involve manipulation of the column's reboiler or condenser submodels. These operations may then in turn be decomposed in terms of a series of operations that involve manipulation of the components of these submodels, such as valves and pumps. Moreover, tasks that operate on the submodels of a system can often be developed and tested on the submodel concerned in isolation, before insertion in a larger structure.

The construction of task hierarchies is facilitated through the introduction of the notion of a parameterised task entity (see section 3.4.1). In order that a hierarchical decomposition can extend to an arbitrary number of intermediate levels, any schedule may be defined in terms of the execution of instances of other task entities. An operation performed in its entirety in a single item of process equipment can, as a result, be declared in terms of a single hierarchy, and recursion is even possible, although there do not seem to be any immediately obvious applications of this to processing systems.

The task shown in figure 3.13 demonstrates a task whose life cycle is hierarchically decomposed in terms of the execution of two instances of the digital controller task shown in figure 3.11. The example also demonstrates how the parameters of a task are assigned values.

The introduction of concurrency implies that several instances of the same task entity may operate simultaneously on different submodels of the overall system. It also follows that the periods in which two tasks operate on the same submodel could overlap. At present, it is considered too restrictive to disallow this situation, and that the rules concerning the simultaneous manipulation of aspects of the continuous model (see section 3.3.1) are sufficient to avoid ambiguities.

Task hierarchies enable a user to develop a simulation description in either a top-down or bottom-up fashion. Employing the top-down approach, the user constructs a simulation problem by abstracting the operations taking place, and then considering each individual operation in more and more detail as the hierarchy evolves. The bottom-up approach takes the opposite view: a simulation description is constructed by the repeated combination of component tasks to form more complex tasks, until a task representing the desired simulation has been constructed. The advantage of the latter approach is that in most situations many of the component tasks required will already exist, so the modelling activity primarily involves the combination of existing tasks. In practice, no engineer is likely to work entirely with one approach; a top-down decomposition is usually the best way of initially analysing a system, just as useful component tasks should not be ignored if they are readily available.

3.6 Explicit Concurrency Versus Process Interaction

In section 3.3 it was observed that the inherent concurrency of the external actions imposed on a processing system may be expressed in either an explicit or implicit fashion. It follows that, in principle, the appearance of an instance of a task entity in one of the control structures described above can be interpreted in two different manners:

- The appearance of the task entity results in the execution of its complete life cycle, and control is only returned to the enclosing control structure on termination of this life cycle.
- The appearance of the task entity merely results in activation of the task entity, whose life cycle is then executed in a quasi-parallel fashion with all other active tasks, and control returns to the enclosing control structure immediately.

It is therefore necessary for the user to specify the manner in which the appearance of a task entity in a control structure is to be interpreted. The most common interpretation is execution in the explicit manner, so this is implied by employing the identifier of the task alone, as demonstrated in figure 3.13. Execution in the implicit manner, thereby merely activating an instance of the task entity, can be specified by the keyword ACTIVATE preceding the identifier of the task. If the execution of a task occurs instantaneously with respect to the simulation clock, then either interpretation will have exactly the same consequences. Therefore, only the user-defined task entities described above, that encapsulate potentially complex life cycles, may be executed in the implicit manner.

As already stated, the two approaches to the expression of concurrency are in fact equivalent. This can be demonstrated by the task shown in figure 3.14 which employs implicit concurrency to achieve exactly the same simulation description as that given by the task shown in figure 3.13.

The normal conditions under which execution of a task will terminate are determined by the various rules outlined above. Under certain circumstances, however, it may be necessary for events elsewhere in the simulation to trigger the premature termination of a task. Consider again the example of a process in which several digital controllers are applied in order to maintain the plant at a nominal steady-state. For the purposes of this discussion, it is considered necessary to use simulation in order to validate the feasibility of a proposed shut-down procedure. This procedure will consist of a sequence of actions to be undertaken by the operator with the intention of taking the process safely from steady-state to a cold and empty state, and at several points in this sequence, digital control loops will be opened to manual control. If these control loops are modelled as self-contained tasks executed in a quasi-parallel fashion with the rest of the simulation, actions within the shut-down sequence must be able to terminate their life cycles prematurely.

The TERMINATE task is introduced in order to allow one task to cause the termination of another task operating in parallel with the former. Execution of a TERMINATE task results in the immediate termination of one or more of the tasks being executed concurrently with it, regardless of the normal life cycle of these tasks. In order to apply this operation, a task instance must be associated with an optional identifier by which it may be distinguished from other active instances of the same task.

The application of the TERMINATE task to the simulation of the shut-down procedure described above is shown in figure 3.15.

3.7 Summary

The second category of discrete changes that a processing system may experience, i.e. external actions imposed on the processing system, were considered in this chapter. A critical review of the facilities offered by existing simulation packages led to the proposal of a novel formalism based on a schedule of tasks that drives the underlying continuous model of a system through the desired simulation. This formalism is based on the conjecture that any external action can be modelled by some form of incremental manipulation of the underlying continuous model. The fundamental mathematical manipulations were identified, and language structures to implement them were introduced.

The problem of ordering the execution of these fundamental manipulations in the time domain using a schedule was then considered, with particular reference to the inherent concurrency of the real world. A series of control structures, similar to those of general-purpose programming languages, were proposed to express this ordering and concurrency. This ultimately led to the introduction of the task entity, which encapsulates a complex set of external actions applied over a finite period of time.

The importance of developing generic task entities that may be employed to describe a broad range of similar, albeit not identical, operations was then highlighted. This was facilitated by the introduction of task parameterisation, in particular polymorphic model type parameters which enable the same task to be applied to a broad range of similar continuous models. Finally, the management of the potential complexity in the external actions applied to complex processing system was addressed by hierarchical subtask decomposition.

```
TASK Digital_PI_Control

PARAMETER
  Set_Point, Reset_Time, Gain, Bias           AS REAL
  Sampling_Interval                           AS REAL
  Termination_Condition                       AS LOGICAL_EXPRESSION
  Sensor                                      AS MODEL Generic_Sensor
  Valve                                       AS MODEL Generic_Valve

VARIABLE
  Error, Integral_Error, Control_Signal       AS REAL

SCHEDULE
  SEQUENCE
    Integral_Error := 0 ;
    WHILE NOT Termination_Condition DO
      SEQUENCE
        CONTINUE FOR Sampling_Interval
        Error          := Set_Point - Sensor.Measurement                 ;
        Integral_Error := Integral_Error + Error*Sampling_Interval       ;
        Control_Signal := Bias + Gain*(Error + Integral_Error/Reset_Time) ;
        IF Control_Signal > 1.0 THEN
          RESET Valve.Position := 1.0 ; END
        ELSE
          IF Control_Signal < 0 THEN
            RESET Valve.Position := 0.0 ; END
          ELSE
            RESET Valve.Position := OLD(Control_Signal) ; END
          END # if
        END # if
      END # sequence
    END # while
  END # sequence

END # Digital_PI_Control
```

Figure 3.11: Reusable Task to Model a Digital Proportional-Integral Control Law

```
TASK Open_Valve

PARAMETER
  Valve                             AS MODEL Generic_Valve

SCHEDULE
  RESET
    Valve.Position := 1.0 ;
  END

END # Open_Valve
```

Figure 3.12: Task to Open a Valve

```
TASK Effluent_Control_System

PARAMETER
  Plant                           AS MODEL Effluent_Plant
  Termination_Condition           AS LOGICAL_EXPRESSION

SCHEDULE
  WITHIN Plant DO
    PARALLEL
      Digital_PI_Control(Sensor                IS pH_Meter,
                         Valve                 IS Alkali_Valve,
                         Termination_Condition IS Termination_Condition,
                         Set_Point             IS 4.0,
                         Gain                  IS -0.02,
                         Bias                  IS 0.7,
                         Reset_Time            IS 210,
                         Sampling_Interval     IS 8.0) ;
      Digital_PI_Control(Sensor                IS Level_Sensor,
                         Valve                 IS Exit_Valve,
                         Termination_Condition IS Termination_Condition,
                         Set_Point             IS 5.0,
                         Gain                  IS 1.0,
                         Bias                  IS 0.5,
                         Reset_Time            IS 100,
                         Sampling_Interval     IS 20.0) ;
    END # parallel
  END # within

END # Effluent_Control_System
```

Figure 3.13: Hierarchical Subtask Decomposition

```
TASK Effluent_Control_System

PARAMETER
  ...

SCHEDULE
  WITHIN Plant DO
    SEQUENCE
      ACTIVATE Digital_PI_Control(...) ;
      ACTIVATE Digital_PI_Control(...) ;
    END # sequence
  END # within

END # Effluent_Control_System
```

Figure 3.14: Implied Concurrency

```
TASK Shut_Down_Procedure

SCHEDULE
  SEQUENCE
    # activate the digital control system at steady-state
    ACTIVATE Controller1 : Digital_PI_Control(...) ;
    ACTIVATE Controller2 : Digital_PI_Control(...) ;
    # shut-down
    ... sequence of actions
    TERMINATE Controller1
    ... more actions
    TERMINATE Controller2
    ... final actions
  END # sequence
END # within

END # Shut_Down_Procedure
```

Figure 3.15: Example Employing the TERMINATE Task

Chapter 4

Combined Discrete/Continuous Simulation – Process Entities

This chapter is concerned with the description of one of the activities for which a combined discrete/continuous model may be employed – that of dynamic simulation. The previous two chapters have described how both the continuous time dependent behaviour of a system, and the aspects of a dynamic simulation description relating to the external actions imposed on this system, may be encapsulated within model and task entities respectively. It is now necessary to detail how the information declared in these disparate entities may be brought together to form the description of an individual dynamic simulation experiment.

The chapter begins with a discussion concerning the distinction between the continuous model of a system and the activities in which this model may subsequently be employed. The discussion lays the foundation of a formalisation for the activity of dynamic simulation in the form of the *process entity*.

The remainder of this chapter deals briefly with issues associated with the reuse of simulation experiments and introduces a multiple simulation experiment management facility.

4.1 The Distinction Between a Model and an Activity

One of the major concerns of the previous two chapters has been the provision of language structures that facilitate the declaration of the continuous model of a system in a manner that is completely decoupled from the details of the individual activities for which the model may subsequently be employed. As already discussed, this promotes reuse of the exclusively declarative knowledge concerning the physical behaviour of a system in a wide range of activities. This disengagement also enables the design of language structures for describing an individual activity in a fashion that provides a clear and unambiguous specification of its functional elements.

It is now possible to consider the activities for which these models may be employed.

A model developed according to a particular formalism may in fact be used for several different activities, and several different modelling formalisms may be used for the same activity. For example, models developed according to any one of the formalisms listed at the beginning of section 1.4 are routinely utilised for dynamic simulation, although each formalism will yield different information concerning the dynamic behaviour of a system. One of the advantages of a modelling formalism based on DAEs or PDAEs is the wide range of activities in which the resulting models may be usefully employed. A non-exhaustive list of such activities relating to process engineering includes:

- Simulation, both steady-state and dynamic.
- Optimisation, both steady-state and dynamic.
- Parameter estimation, both steady-state and dynamic.

However, the subject of this thesis is dynamic simulation, so other activities will not be considered further.

By analogy to experimentation in the real world, the notion of an *experiment* can be introduced to denote an individual application of an activity to a model (Oren and Ziegler, 1979). Each experiment will consist of three elements: the object investigated by the experiment (in the form of a model developed according to a specified formalism), the *experimental frame*, and the data generated by execution of the experiment. The experimental frame denotes the circumstances under which the system is to be investigated, and therefore encapsulates the procedural information required to apply an individual activity to a model. In order to clarify the declaration of this information, an experimental frame can be broken down into a series of *functional elements*.

The functional elements required by the activity of dynamic simulation will be discussed in the next section. However, many of these are not unique to the activity of dynamic simulation, so the language structures developed for them may also be employed in the declaration of experimental frames for other activities, thereby promoting the overall consistency of the modelling environment.

Each experimental frame is a description of a unique set of experimental conditions. A numerous, if not infinite, number of experimental frames may be developed for the same model. However, it is important to recognise that the reverse is also true: it is conceivable that the same experimental frame could be applied to several different models. There will obviously have to be tight constraints on this latter mapping, with provisions for the automatic determination of the applicability of a model to a particular experimental frame.

4.2 Process Entities

A process entity encapsulates the dynamic simulation of a period of operation of one or more items of process equipment described by an appropriate continuous model and driven by a particular set of external actions, and is therefore a formalisation of the requirement for a dynamic simulation experimental frame outlined above. Application of a process entity to a continuous model will result in a complete dynamic simulation experiment.

The declaration of a process entity begins with the keyword PROCESS followed by a unique identifier by which it may be referred to globally. The remainder of the declaration is partitioned into sections, each containing information pertaining to one of the functional elements required by the activity of dynamic simulation. These are described below.

4.2.1 The Combined Discrete/Continuous Model

The first item of information required is a declaration of the process equipment under investigation and the combined discrete/continuous model entity that will determine its continuous time dependent behaviour during the simulation experiment in question. A UNIT section is employed to declare this information in the form of one of more instances of previously declared model entities. The set of variable attributes associated with these model instances will completely describe the time dependent behaviour of the system, and the set of equation attributes will partially determine it. Execution of the enclosing process entity will result in instantiation of the continuous model thus declared, followed by a dynamic simulation experiment involving this copy. Note that alterations to this copy of the continuous model as a consequence of events during simulation has no effect whatsoever on the *static* model and process entities.

The declaration of a process entity also includes a SET section, within which parameter attributes of these model entities may be assigned values explicitly, provided they have not already been assigned a value from some other source. For a simulation description to be valid, all parameter attributes must have received a value through one of the mechanisms outlined in section 2.4.3 by this point.

An excerpt from a process entity demonstrating the declaration of these two sections is shown in figure 4.1. The example involves the simulation of a well mixed vessel with two inlet streams and one outlet stream, and will be used throughout the discussion of process entities. The UNIT section is employed to create an instance of the continuous model of the vessel, and the SET section is employed to assign values to the parameter attributes that determine the number of components present and the cross-sectional area of the vessel.

```
PROCESS Simulate_Tank

UNIT
  Tank101               AS Tank

SET
  WITHIN Tank101 DO
    NoComp := 2   ;
    Area   := 1.3 ;
  END # within

...
```

Figure 4.1: Example UNIT and SET sections

The appearance of a UNIT section in the declaration of a process entity may at first appear to be a contradiction of the arguments above advocating the disengagement of an experimental frame from the model it is applied to. In fact, we will demonstrate later how the UNIT section can be employed to declare a broad range of models to which the experimental frame is applicable.

4.2.2 Additional Equations

The set of describing equations derived from the model instances declared in the UNIT section is typically underdetermined with respect to the set of variables derived in the same manner. For the experiment to be fully defined, additional relations between the variables must be introduced to make the set square.

The EQUATION section permits the specification of additional *general* equations to be combined with those derived from the UNIT section to form a complete description of the continuous time dependent behaviour of the system under investigation. Any of the structured equations introduced in chapter 2 may be employed in the declaration of these relationships, and connectivity equations may define stream connections between components of the overall system model. Figure 4.2 demonstrates how this section is employed to declare the equation that will determine the total flowrate from the vessel described above. This flowrate would normally be determined by downstream units such as pumps and valves, but must be declared here when the vessel is considered in isolation.

```
EQUATION
  WITHIN Tank101 DO
    Total_Flow_Out = 0.5*SGN(Press_Out - Press_Atm)*
                                        ABS(Press_Out - Press_Atm)^0.5 ;
  END # within
```

Figure 4.2: Example EQUATION section

However, by far the most common type of equation introduced in a process entity is that of *input* equations (cf. equation 1.4). The ASSIGN section is used to designate a subset of the variables as ‘inputs’ and to set them to constant values or functions of time. In order to emphasise the special form of these specifications, input equations are declared with the assignment operator described in section 3.2.1. FOR and WITHIN structures are also provided to aid declaration when necessary. The ASSIGN section shown in figure 4.3 illustrates the declaration of the inlet flowrates and the atmospheric pressure as the initial set of input variables for the simulation of the well-mixed vessel. Note the use of the `[ ... ]` notation as a shorthand for assigning values to vectors.

4.2.3 Initial Values for SELECTOR Variables

During a simulation, the active clause of a CASE equation (see section 2.3.5.2) will determine which set of equations is inserted in the overall continuous model at any particular

```
ASSIGN
  WITHIN Tank101 DO
    Flow_In_1 := [ 50,0 ] ;
    Flow_In_2 := [ 0,25 ] ;
    Press_Atm := 1.013    ;
  END # within
```

Figure 4.3: Example ASSIGN section

point in time. A specification of which clause is active at the *beginning* of a simulation forms part of the initial condition of that simulation. This information is deduced from the initial value of the selector variable concerned which, unless a default value is specified, must be supplied explicitly.

All selector variables not already assigned a default value *must* therefore be supplied with an initial value in the SELECTOR section of a process entity. For instance:

```
SELECTOR
  Plant.Tank101.Disc_Status := Intact ;
```

With the input equations and assignments of initial values to selector variables, the initial continuous model is now complete. At this point the set of describing equations must be fully determined with respect to the system variables, and in addition must represent a well-posed dynamic simulation problem. Of course, both the set of input variables and their specifications may be changed *during* the simulation as a consequence of the execution of RESET and/or REPLACE tasks (see sections 3.2.1 and 3.2.2). Similarly, the values of selector variables may change as a result of physico-chemical discontinuities or the execution of RESET tasks.

4.2.4 The Initial Condition

The description of any dynamic simulation experiment must include a specification of the initial condition of the system under investigation. As already discussed in section 1.4, the initial condition of a continuous simulation based on a system of DAEs is determined in the most general terms possible by the simultaneous solution of the describing equations and a number of additional equality constraints.

The INITIAL section is employed to declare this functional element of a dynamic simulation experimental frame, which is expressed in terms of the requisite number of additional nonlinear equations. This can be contrasted to the facilities offered by existing continuous process simulation packages, where the initial condition can only be expressed in terms of the assignment of initial values to a subset of the system variables or their time derivatives (cf. section 1.4.1).

The excerpt from the description of the simulation of the well-mixed vessel shown in figure 4.4 demonstrates the declaration of an initial condition. The first equation illustrates the conventional assignment of a value to a system variable, whereas the second equation demonstrates the use of a more general equation.

```
INITIAL
  WITHIN Tank101 DO
    X(2) = 0 ;
    SIGMA(Flow_In_1 + Flow_In_2) = Total_Flow_Out ;
  END # within
```

Figure 4.4: Example INITIAL section

The initial condition that is most frequently employed for the simulation of processing systems is the assumption of steady-state, which can be expressed by the requisite number of equations constraining the time derivatives of differential variables to zero. Alternatively, the keyword STEADY_STATE may be utilised as a convenient shorthand. In this case, no further specifications are required:

```
INITIAL
  STEADY_STATE
```

is sufficient.

4.2.5 External Actions

The task entities introduced in chapter 3 provide the means by which the external actions imposed on processing system can be modelled without impairing the reusability of the underlying continuous model. They are therefore only employed by those activities, such as dynamic simulation, that may involve external actions. The language structures described in chapter 3 recognise the potential complexity of the information declared in this functional element, and the fact that it may be desirable to reuse much of it in other experiments.

The SCHEDULE section of a process entity facilitates the declaration of the schedule of task entities in terms of a time ordered set of discrete manipulations to the underlying continuous model. A continuous model and the external actions applied to it are therefore only brought together in the description of an experimental frame. Execution of this schedule will drive the process entity, and hence the underlying continuous model, through its life cycle. Execution of the process entity will terminate when execution of this schedule is complete.

It is now possible to show in figure 4.5 the declaration of the entire process entity that describes the simulation of the well-mixed vessel. The SCHEDULE section in this example only involves one simple action, but the schedule of tasks that appears in such a section may be composed of any combination of elementary tasks and reusable task entities (see section 3.4.1). In particular, all previously defined task entities are made available globally and can be used in any process entity.

All the functional elements of a dynamic simulation experimental frame outlined above are obviously necessary to form a complete experimental description. The remaining text relates to useful optional functional elements that are not absolutely vital to the description of an experiment.

4.2.6 Initial Guesses

The initialisation calculation performed at the beginning of any continuous simulation, in order to determine consistent initial values for all the system variables, is achieved through the numerical solution of a system of nonlinear equations (see section 1.4.1). This calculation requires an initial guess for the values of all the variables concerned. The PRE-

```
PROCESS Simulate_Tank

UNIT
  Tank101               AS Tank

SET
  WITHIN Tank101 DO
    NoComp := 2   ;
    Area   := 1.3 ;
  END # within

EQUATION
  WITHIN Tank101 DO
    Total_Flow_Out = 0.51*SGN(Press_Out - Press_Atm)*
                                    ABS(Press_Out - Press_Atm)^0.5 ;
  END # within

INPUT
  WITHIN Tank101 DO
    Flow_In_1 := [ 50,0 ] ;
    Flow_In_2 := [ 0,25 ] ;
    Press_Atm := 1.013    ;
  END # within

INITIAL
  WITHIN Tank101 DO
    X(2) = 0 ;
    SIGMA(Flow_In_1 + Flow_In_2) = Total_Flow_Out ;
  END # within

SCHEDULE
  SEQUENCE
    CONTINUE FOR 10.0
    RESET
      Flow_In_2(2) := 50 ;
    END # reset
    CONTINUE UNTIL ABS($HoldUp(2)) < 1E-3
  END # sequence

END # Simulate_Tank
```

Figure 4.5: A Complete Process Entity

SET section enables the default initial guesses associated with variable type declarations to be overridden by values more appropriate to the calculation in question.

The assignment operator, complemented by FOR and WITHIN structures, is also employed in this section. The upper and lower bounds on the value of this variable may also be altered at the same time. Any initial guess must, of course, lie within these bounds. Initial guesses and bounds on the time derivatives of differential variables may also be specified in this manner.[1] Figure 4.6 demonstrates a PRESET section.

```
PRESET
  WITHIN Tank101 DO
    X(1)      := 1.0                 ;
    X(2)      := 0.0                 ;
    Press_Out := 1.013 : 1.013 : 2.0 ;
  END # within
```

Figure 4.6: Example PRESET section

4.2.7 Storage of Simulation Results

Modern modelling environments regard interpretation of the data gathered during an experiment to be the role of the environment in which activities are performed (see, for example, the ASCEND environment (Piela, 1989)), as opposed to the language structures employed to describe individual activities. After all, in many situations the interpretations desired will in part be determined by the results of the experiment itself. The environment must, therefore, provide a variety of tools that enable the data collected from an experiment to be manipulated and displayed in an interactive manner. Moreover, provisions must be made for the storage of selected data from an experiment for later reference. This data must be accompanied by a record of the model and experimental frame pair from which it was generated. The interactive tools described above may then retrieve this data for manipulation and display at a later date. This approach, however, assumes that it is more

[1]By default, the initial guess assumed for a time derivative is zero, and upper and lower bounds are set to the machine limits.

efficient to collect the maximum amount of data during the course of an experiment,[2] rather than repeat the experiment in order to collect data that were not previously observed.

The data observed and recorded during a simulation experiment will always include the time trajectories of all or some of the system variables. Other data that should, by default, be recorded as a simulation progresses will include the state transitions of discrete variables, such as selector and local task variables, and any dynamic changes to the describing equations.

However, the resources available for the storage of data will always to a certain extent be limited. Although this problem can be partially alleviated by careful exploitation of modern techniques for data compression, situations will always arise in which facilities for explicitly restricting the data observed and recorded during a particular experiment will become invaluable. This observation is particularly relevant for circumstances in which only a few key unit operations, in an overall system model involving thousands of quantities, are actually of interest from the point of view of the post-simulation analysis. An optional REPORT section could therefore be introduced that enables a specification of the data to be observed and recorded during a simulation. If no REPORT section is included, *all* relevant information will be recorded for future reference.

4.3 Reuse of Process Entities

The process entity, as presented so far, is a limited implementation of the concept of an *experimental frame*, as introduced at the beginning of this chapter, because there is no potential for reuse of the frame in several experiments of a similar nature. In particular, all aspects of the simulation, such as the form of the input functions, the initial conditions etc., are entirely fixed.

Parameterisation of process entities, just as with the task entities of chapter 3, is the key to the development of reusable experimental frames. A process entity declared in terms of a suitable set of parameters no longer represents a single dynamic simulation experiment, but a framework for a wide range of potential experiments. All the parameter types introduced in section 3.4.1 can be useful in the declaration of reusable process entities.

[2]Which may result in the collection of large amounts of uninteresting data

Figure 4.7 demonstrates how the process entity that encapsulates the description of the simulation experiment involving the well mixed vessel may be altered to describe a broad range of experiments with varying values for the cross-sectional area of the vessel and the constant that determines the total flowrate from the vessel. Execution of this parameterised process entity must obviously be accompanied by the assignment of suitable values to all its parameters.

```
PROCESS Simulate_Tank

PARAMETER
  Cross_Sectional_Area, Flow_Constant   AS REAL

UNIT
  Tank101                               AS Tank

SET
  WITHIN Tank101 DO
    NoComp := 2                   ;
    Area   := Cross_Sectional_Area ;
  END # within

EQUATION
  WITHIN Tank101 DO
    Total_Flow_Out = Flow_Constant*SGN(Press_Out - Press_Atm)*
                                    ABS(Press_Out - Press_Atm)^0.5 ;
  END # within

...

END # Simulate_Tank
```

Figure 4.7: A Reusable Process Entity

The discussion at the beginning of this chapter also suggested that there may be some potential for limited disengagement of an experimental frame from the model to which it is actually applied for an individual experiment, provided tight constraints on the possible mappings were maintained. In fact, the notion of polymorphic MODEL type parameters may again be exploited for this purpose. The MODEL type parameters of a process may be employed to specify the type of any unit attributes of that process, effectively deferring a decision concerning the model to which the process is actually applied until execution of an individual experiment. The polymorphic nature of these parameters will ensure the suitability of the models eventually used.

4.4 Hierarchical Subprocess Decomposition

It has already been observed that a process entity may encapsulate any dynamic simulation experiment that might be performed by a conventional continuous simulation package such as SpeedUp (Prosys, 1991). In many situations, however, it is also desirable to be able to initiate and co-ordinate automatically multiple simulation experiments. This *multiple case management* facility would enable routine experiments to be set up and executed to completion without the need for intervention from the user. One particularly good application is to Monte-Carlo type experiments, where the same simulation is performed many times, with randomly distributed initial conditions and/or parameters, in order to gather statistics regarding the system performance and reliability. Moreover, this facility need not just be available for dynamic simulation experiments. If process entities are introduced to encapsulate, for example, optimisation or parameter estimation activities, different activities may be combined to form a larger experiment in which the results from one activity are used as the basis for another.

The need for a multiple case management facility highlights the fact that, in the most general terms, the progress of an experiment may in fact be dictated by the execution of multiple primitive experimental frames. In direct analogy with task entities, this additional complexity can be accommodated by establishing a procedural decomposition based on the control structures introduced in chapter 3. The process entities that appear in a hierarchy thus created will fall into two categories:

- The *primitive* process entities that encapsulate the description of single activity such as dynamic simulation or optimisation.
- The *composite* process entities, the life cycles of which are determined by the execution of a schedule of other process entities.

A process entity therefore becomes a description of an experimental frame that may involve a single activity, or a complex hierarchy of interacting activities. Moreover, a suitable hierarchical decomposition will again enable components to be developed and tested in isolation before insertion in a larger structure. However, such considerations are beyond the scope of this thesis.

4.5 Summary

The importance of maintaining a distinction between the continuous model of a system and any of the activities in which it may subsequently be employed was explored with particular reference to a model formalism based on DAEs or PDAEs. This argument was used to justify the introduction of the concept of an experiment, denoting the individual application of an activity to a model. An experiment is composed of a model, an experimental frame, and the data generated by the experiment. A consideration of the functional elements of a dynamic simulation experimental frame then led to a formalisation in the form of the process entity.

It was then observed that certain experiments may actually require the execution of several interacting activities, which identified the need for a multiple case management facility based on the procedural decomposition and parameterisation of process entities. A primitive process entity encapsulates the description of an activity such as dynamic simulation or optimisation, whereas a composite process entity is decomposed in terms of the execution of other process entities.

Chapter 5

Implementation

The simulation language introduced in the previous three chapters provides a general framework for the description of combined discrete/continuous simulation experiments involving industrial processing systems of arbitrary complexity. This chapter is concerned with the implementation of a prototype process modelling package based on this language, known as gPROMS (**g**eneral **PRO**cess **M**odelling **S**ystem).

Implementation of the prototype serves two purposes. In the first instance, it provides a means by which the ideas embodied in the simulation language can be tested and refined. Many of the features introduced in the preceding chapters have been reconsidered or revised as a direct consequence of the evolving implementation. Secondly, a working prototype can be used to demonstrate the usefulness of general-purpose tools for combined discrete/continuous systems through a series of detailed simulation examples that would otherwise be difficult or even impossible to construct (see chapter 6).

The chapter provides an overview of the current implementation detailing the issues that had to be addressed and the decisions made. It is not intended to act as a detailed documentation of the code, which would probably occupy a volume as large as the entire thesis. A discussion concerning the philosophy and assumptions on which the implementation is based begins the chapter. This leads to the presentation of a software architecture based on three major components: the *translator*, which checks and converts the information declared in an input file into an internal representation, the *process manager*, which employs this representation to form, modify, and solve individual experiments, and the *environment*, from which the user controls and co-ordinates the various actions performed during a session. The implementation of each of these components is then considered in more detail in the sections that follow. The chapter concludes with a summary of the versions of the software available.

5.1 Implementation Philosophy and Assumptions

First, it is enlightening to consider briefly the motivations that have led to the software architecture traditionally adopted by continuous modelling packages that employ a simulation language.

5.1.1 Intermediate Code Generation

Packages such as ACSL (Mitchell and Gauthier, 1976) or SpeedUp (Prosys, 1991) can be classified as *intermediate code generators.* They typically utilise a language processor to translate input files written in the simulation language into an internal representation, from which is generated some form of intermediate code, often in the form of a series of subroutines written in the programming language FORTRAN. This code must then be compiled and linked with a library of simulation subroutines to form an executable module which, when executed, will perform the desired simulation experiment. Alternatively, some language processors compile the input file directly into machine code that is again linked with utility object code and then executed.

This architecture evolved at a time when large mainframe computers were the only machines that offered the computational and memory resources required to perform simulation experiments of any size. Dynamic simulation was therefore not considered to be an activity that could be conducted interactively, and most packages were designed to make the best use of limited, shared resources accessed in a batch mode. The advent of engineering workstations and personal computers has, however, changed this situation dramatically. These new machines make computational power and memory resources previously only associated with mainframe computers exclusively available to individual users. As a consequence, dynamic simulation involving several hundred or even thousand simultaneous equations is now considered to be an activity that can be conducted interactively. Moreover, as the computational power and memory offered by this class of machines continues to grow each year, the size of problems that can be dealt with in an interactive manner can only increase.

Simulation packages have attempted to keep pace with this trend, but those based on the architecture described above have been at a significant disadvantage. SpeedUp, for example, is now an interactive package available on a wide range of engineering workstations, but the cycle of translation, code generation, compilation, linking of object code, and then execution proves to be a very time consuming barrier to a truly interactive environment. This is particularly frustrating in the case of problems involving a relatively small number of equations, where it often takes considerably longer to generate an executable module from an input file than it does to execute the entire simulation experiment.

As a consequence, the developers of more modern continuous modelling packages such as OMOLA (Andersson, 1992) and ASCEND (Piela, 1989) have attempted to reduce the time taken between the successful translation of a problem description and execution of the actual simulation experiment. In OMOLA this has been achieved by a new architecture based on translation of the problem description followed by instantiation of the continuous model in memory. This copy is then employed to generate intermediate simulation code that is *interpreted* directly by a simulation algorithm. The advantage of this approach lies in the complete elimination of the compilation and linking steps from the cycle, but has the disadvantage of being more memory intensive and slightly less computationally efficient than object code.

5.1.2 A Software Architecture for Combined Simulation

Combined discrete/continuous simulation places even greater demands on software architecture that cannot in general be addressed by either of the approaches outlined above. The simulation language described in the previous chapters provides facilities for the discrete manipulation of the underlying continuous model of a system as a result of either physico-chemical discontinuities or externally imposed control actions. As a consequence, the set of equations that determines the continuous time dependent behaviour of the system, and the status of the variables that describe this behaviour, is likely to change frequently.

Although it is possible, in principle, to analyse a priori all the physico-chemical discontinuities declared in a continuous model and thereby generate code that will manipulate the set of active equations accordingly as a simulation progresses, it is clearly impossible in general to analyse the manipulations implemented by a schedule of tasks in a similar manner. This conclusion, allied with the special requirements of combined simulation outlined above, is used to justify a software architecture for the prototype modelling package somewhat similar to that proposed for different reasons by the developers of the ASCEND system

(Piela, 1989).

As in the case of OMOLA, this architecture is based on a translation phase followed by instantiation of the continuous model in memory. However, no intermediate simulation code is generated. The simulation algorithm instead employs the instance of the continuous model created in memory to determine the dynamic behaviour of the system under investigation. As a consequence, all the symbolic information relating to the continuous time dependent behaviour of the system is readily accessible for the entire duration of the experiment. The significant improvements in the reporting and diagnosis of problems or errors during simulation facilitated by this approach was alone sufficient to convince the developers of the ASCEND system to adopt it.

From the point of view of combined discrete/continuous simulation, however, this approach appears to be absolutely essential. It enables the schedule of tasks, execution of which is also co-ordinated by the simulation algorithm, to manipulate the copy of the continuous model directly as the simulation progresses, and therefore satisfies all the special requirements of combined discrete/continuous simulation outlined above. Moreover, the time taken between a successful problem translation and execution of the actual simulation experiment is further reduced by the complete elimination of the code generation phase. This is particularly important in the case of a simulation experiment involving multiple process entities (see section 4.4), where the simulation executive would otherwise have to halt all simulation calculations and enter a code generation phase each time a new process entity was activated, considerably slowing the execution of the overall experiment as a consequence.

Holding a copy of the continuous model in memory for the entire period in which it is employed by a simulation experiment is obviously more memory intensive than either of the two other approaches described above. In light of the dramatic and continuing increase in the availability and decrease in the cost of computer memory in recent years, this is not, however, considered to be a significant obstacle. For example, experiments with the prototype modelling package have demonstrated that a problem involving ten thousand equations can comfortably fit into sixteen megabytes of computer memory (even without any particular efforts to promote efficient use of memory). Further advantages of holding a copy of the continuous model in memory during a simulation experiment are discussed in section 5.3.

5.1.3 Improving Translation Speeds

Another disadvantage of SpeedUp in particular is the relatively long time taken to translate an input file into the internal representation employed by the simulation package. This is in part attributable to the fact that this representation is stored on file in a data base, considerably increasing the time required to access each item of information. The implementation of the prototype modelling package has demonstrated that dramatic increases in translation speeds can be achieved through storage of this representation in the form of symbol tables held entirely in memory.[1] Careful data structure design ensures that this representation occupies a relatively small quantity of memory in comparison to that occupied by the copies of continuous models actually employed by simulation experiments. However, if an input file contains references to entities that have already been translated and archived in libraries, these must also be imported into the symbol tables before or during application of the translator.

5.1.4 The Prototype Modelling Package

It is now possible to describe the functions of the three major components of the prototype modelling package: the *environment*, the *translator*, and the *process manager*. Simulation experiments are instigated and interpreted by the user from the environment. However, before a simulation experiment can be performed, it is necessary to build the symbol tables that hold the internal representation of the simulation description. This can be achieved by creating an input file in the simulation language described in the preceding chapters, and then submitting the file to the translator, which checks the file for syntactic and semantic correctness. Successful translation will result in the creation of symbol table entries for all the entities declared in the input file. More information may be imported from libraries of archived entities, or via translation of additional input files.

The final component, the process manager, is employed to form, modify, and solve the mathematical description of individual dynamic simulation experiments. A process entity is submitted to the process manager by issuing the command EXECUTE to the environment (see section 5.4). Execution of an individual process entity by the process manager will

[1]This is also the approach adopted by many modern programming language compilers.

result in the creation of a copy of the continuous model it encapsulates in memory, followed by application of the accompanying schedule of tasks by the simulation algorithm. On termination of the process entity, the copy of the continuous model will be destroyed and the memory recovered. Simulation experiments involving multiple process entities will obviously require the dynamic creation and destruction of several continuous models.

The implementation of the prototype modelling package is obviously a task of considerable magnitude, and certain features of the simulation language have had to be omitted due to time constraints. There are, however, modern software development tools that, if used correctly, can significantly reduce the effort required. Two tools have been found to be particularly useful. The first is an automated compiler construction tool which greatly reduced the time required for the development and modification of the translator, and promoted the strict modularity of its various structural components. Secondly, there is the contribution of the programming language Modula–2 (Wirth, 1988), employed to code the modelling package. Modula–2 is a modern programming language specifically designed to support large software projects involving teams of programmers through the decomposition of programs into a series of modules that can only communicate through strict interfaces. Strict modularisation in this manner has proved to be an invaluable discipline, and will greatly improve the future maintainability of the code. It has also been possible to design the code with as few prespecified limits as possible: through the exploitation of dynamic data structures, the only limit on the size of problem that can be solved by the prototype is effectively the amount of computer memory available to a particular user.

5.2 The Translator

The translator is intended to take an input file containing part or all of a problem description coded in the simulation language, check that it is both syntactically and semantically correct, and then store the information thus declared in an internal representation for later use by the process manager (see section 5.3).

The structure of the translator is similar to that of a single pass language translator, or *compiler* (Fischer and LeBlanc, 1988), employed to convert input files written in a high level general-purpose programming language into machine code that can be interpreted by a

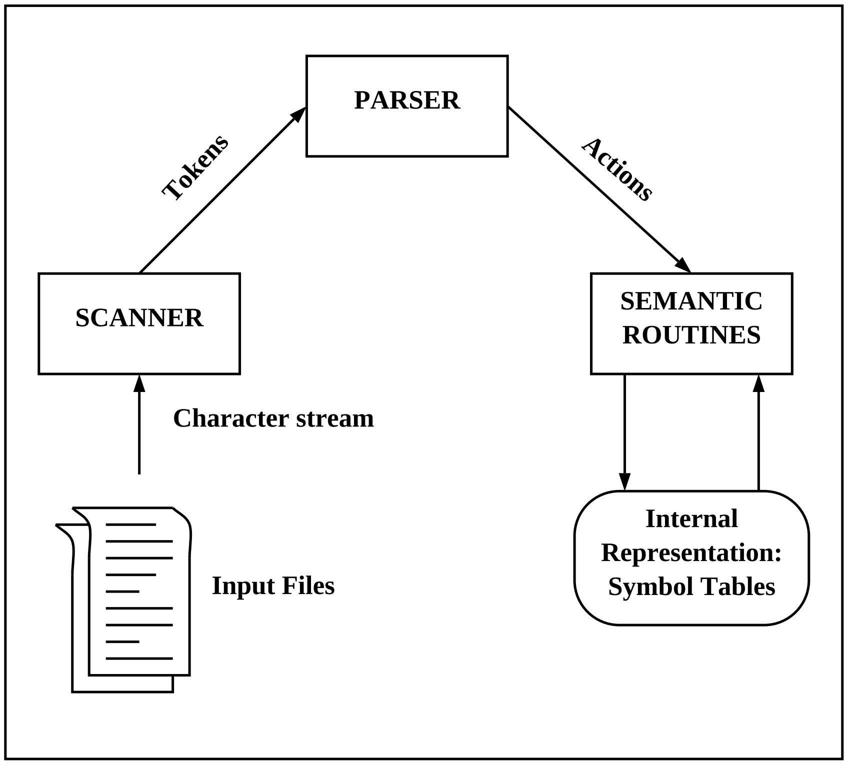

Figure 5.1: The Structure of the Translator

computer. The three structural components are: the *scanner*, the *parser*, and the *semantic routines*, each of which is covered in more detail in the sections that follow. This structure is illustrated in figure 5.1.

As with most modern compilers, the translator is *syntax-directed*. That is, the translation is driven by the syntactic structure of the source code, as recognised by the parser. However, the translator differs from a compiler in the sense that the purpose of the entire activity is to build the data structures that hold the internal representation of a problem description, rather than to generate machine code.

5.2.1 The Scanner

The scanner is the simplest component of the translator. It reads the input file character by character, and aggregates these characters into *tokens* (e.g. identifiers, literals, keywords, delimiters) which are then passed to the parser on request. The tokens that currently make up the language definition are summarised at the beginning of appendix B.

The implementation employs a standard algorithm based on deterministic finite automata (Fischer and LeBlanc, 1988) and is designed to make the addition of new tokens straightforward. The processing of keywords is managed by a separate module, which is again designed to expedite the addition of new keywords.

5.2.2 The Parser

The entire translator is driven by the parser, which checks the structure of the input file against the formal definition of the simulation language grammar. The parser calls the scanner whenever a new token is required, and the appropriate semantic routine when an action is required by the language definition.

In order to facilitate the rapid prototyping of the language translator, a modification of the parser generator Yacc (Johnson, 1975) was chosen to generate the parsing routines automatically. This decision was justified by the fact that Yacc is a well understood and widely available tool, particularly on systems running the UNIX operating system and its variants. In addition, the version actually employed (McLoughlin, 1981) has the option to produce parser routines coded in the programming languages Modula–2 or Pascal (instead of the more usual C).

The input to Yacc takes the form of a file containing a formal declaration of the context-free grammar that the required parser routines will accept. This declaration, which is composed of a list of tokens and the production rules that define the grammar, including calls to semantic routines as required, is given in a special Yacc input language. The Yacc input file for the current implementation of the simulation language is given in appendix B. The input file is translated and checked by Yacc, which will then attempt to generate a LALR(1) parser (Fischer and LeBlanc, 1988) for the grammar. Output consists of a set of parser routines coded in the target programming language and a file containing the parse tables. The parser routines implement a shift-reduce parsing algorithm, manage the semantic stack, and call semantic routines as required. The parse tables are employed by the shift-reduce algorithm to determine the syntactic correctness of an input file written in the simulation language.

Yacc is therefore a very powerful tool for rapidly prototyping a language translator. Changes to the syntax of the simulation language merely involve modification of the Yacc input file, followed by automatic generation of new parser routines and tables. It is important to recognise that the semantic routines are completely decoupled from this process, and can therefore be developed independently of changes to the language syntax.

5.2.3 The Semantic Routines

The purpose of the semantic routines is to check the semantic correctness (or meaning) of the information declared in an input file, and then to generate the data structures that will store this information in a form that facilitates the subsequent construction of the mathematical description of an individual simulation experiment. Adoption of Yacc has the consequence that coding of the semantic routines represents most of the effort required in the implementation of the translator. The semantic rules of the simulation language have been covered in detail as the various features were introduced, so this section will only be concerned with the intermediate data structures employed to store the internal representation of a problem description, in the form of a series of *symbol tables*.

5.2.3.1 The Symbol Tables

Each category of entity included in the language definition (model entities, task entities, process entities, variable types, and stream types) is stored in its own individual symbol table. This has the consequence that the same identifier may be used by entities belonging to different categories. AVL balanced binary trees (Wirth, 1986) ordered by the identifier of the entity in question have been chosen to store these symbol tables, because they combine rapid search speed with low memory overhead. A generic module (King, 1988) is employed to implement the AVL balanced binary trees.

Those entities (e.g. models) that possess attributes (e.g. parameters, variables etc.) also have individual symbol tables in which these attributes are ordered by their identifier. A hash table with collision resolution by chaining (Fischer and LeBlanc, 1988) has been chosen for the implementation of these symbol tables because very rapid access is required during translation and instantiation of a continuous model. The number of attributes declared in such entities will typically be less than fifty,[2] so the memory overhead associated with these

[2]Otherwise it is likely that the facilities that gPROMS provides for hierarchical decomposition are not being exploited properly!

hash tables can be kept small.

5.2.3.2 The String Space Array

The *string space* (Fischer and LeBlanc, 1988) is a character array into which all identifiers that appear in a problem description are packed. This is the most efficient way to store a large number of identifiers of varying length. An upper limit of eighty is set on the number of characters that can make up an identifier.

In order to ensure that the memory occupied by the string space is minimised, the string space is allocated dynamically as required in segments of equal length. A reference to an identifier stored in the string space requires three quantities: a pointer to the segment in which the identifier is stored, an index to the beginning of the identifier in this segment, and the length of the identifier.

5.2.4 Errors During Translation

An input file is echoed to the terminal as it is translated. If lexical or semantic errors are encountered, an error message is constructed by the routine that detected the error. This message is then issued at the end of the line of input in which the error occurred, but is also stored for inclusion in a error summary issued at the end of translation. At present facilities for syntax error recovery are primitive: translation terminates immediately on detection of a syntax error. Yacc does provide more sophisticated syntax error recovery features, but it was felt that it was not entirely justified to exploit them while the language syntax was still evolving rapidly.

5.3 The Process Manager

The process manager, as its name suggests, manages the execution of process entities. The current version of the implementation only supports experiments involving the execution of a single process entity, so most of this section is devoted to a discussion of this function. The issues relating to the execution of multiple process entities during an experiment are considered briefly in section 5.3.3.

The execution of a process entity involves the following four phases:

1. Instantiation in memory of both the continuous model and the schedule encapsulated by the process.
2. Initialisation of the continuous model.
3. Application of the schedule to the continuous model by the simulation algorithm in order to advance the simulation clock from the initial condition to the termination condition.
4. Recovery of the memory occupied by the continuous model and the schedule.

The novel implementation that has been dictated by the special requirements of combined discrete/continuous simulation also leads to a trade-off between the time taken to prepare an experiment for execution and the time it takes to execute the experiment. This is best illustrated by a comparison with the SpeedUp (Prosys, 1991) continuous process simulation package.

As already discussed, preparation for a simulation experiment employing SpeedUp effectively involves the automatic generation of several FORTRAN subroutines, which must then be compiled and linked before the experiment can be executed. The two most important subroutines will determine the residuals of the describing equations and their partial derivatives with respect to the system variables. This approach leads to a time consuming code preparation phase, but does normally ensure the most efficient execution of the experiment, because advanced FORTRAN compilers can be relied upon to generate very efficient machine code for residual and derivative evaluations.

In contrast, the prototype modelling package creates an image of the continuous model in memory which is then employed by the simulation algorithm to determine the residuals and partial derivatives of the describing equations. The expressions that determine these values are stored in a binary tree data structure. Whenever the value of an expression is requested, a subroutine traverses the corresponding binary tree recursively in order to calculate the value. This has the consequence that each residual and Jacobian evaluation will always take slightly longer than those performed by the equivalent subroutines generated by SpeedUp, although the additional integer operations required to traverse the binary trees will be relatively cheap in comparison to the floating point operations required to evaluate the expressions in either format. Hence the trade-off between preparation and execution of an experiment.

On the other hand, there are significant additional benefits to be gained from having a copy of the continuous model held in memory for the duration of an experiment. The first of these is the fact that more accurate structural information concerning the current set of describing equations can be passed to the solution routines. For example, whenever dynamic changes to the set of describing equations occur, the pattern of non-zero elements in the Jacobian is also likely to change. In SpeedUp this problem is evaded by identifying a priori the superset of the variables appearing in all the clauses of an IF equation, and then creating non-zero entries for all these variables in the Jacobian. In contrast, if the continuous model is available for analysis for the entire duration of an experiment, the exact pattern of non-zero elements in the Jacobian can be updated each time a change to the set of describing equations occurs.

Secondly, *exceptions*, such as division-by-zero errors, can be trapped by the subroutine that evaluates an expression, rather than relying on any exception handling facilities provided by the operating system. This has two advantages. In the first instance, the response of the expression evaluation subroutine can be standardised across all machine implementations of the software, whereas the availability, form, and responses of standard exception handlers are all highly machine and operating system dependent. Secondly, transient exceptions that may occur during an iterative calculation that actually has a valid solution can be trapped and recovered from, avoiding unnecessary failure of the simulation.

Finally, it is envisaged that the availability of detailed symbolic information relating to the continuous time dependent behaviour of a system throughout a simulation may also be exploited to improve the efficiency and robustness of the solution of the describing equations, e.g. by implementing automatic scaling of equations etc.

Despite the advantages outlined above, situations can arise in which it is desirable to generate intermediate code with which an experiment may be executed (Holl *et al.*, 1988), rather than relying on the modelling package. This requirement has been addressed in the current implementation by a special environment command that results in instantiation of the continuous model encapsulated by a process entity, followed by the generation of FORTRAN code that may then be used to perform the desired simulation. However, only a limited subset of the features of the simulation language may be utilised by experiments coded in this manner, corresponding closely to those offered by a conventional continuous process simulation package such as SpeedUp.

5.3.1 Instantiation of the Continuous Model

Instantiation of the continuous model effectively involves generation of the data structures which the simulation algorithm employs in order to determine the continuous time dependent behaviour of the system under investigation. In addition, the introduction of extensive model parameterisation dictates that a small number of semantic checks on a problem description must be postponed until this point. For example, because expressions involving integer parameters may be utilised to specify the number of elements in attribute arrays, certain checks to determine the dimensional correctness of expressions can only be done during model instantiation. This, however, has not proved to cause problems in practice because model instantiation takes place much more rapidly than the traditional cycle of code generation and compilation.

The data structures, known collectively as the *active arrays*, all take the form of packed arrays in order to facilitate rapid access during residual and Jacobian evaluations. They are allocated dynamically as required in segments of equal length, thereby ensuring that the amount of memory available is the only limit on problem size. The following active arrays are required:

- The Active Parameter Array (APA) stores all the parameter attributes included in the problem description.

- The Active Variable Array (AVA) stores all the variable attributes included in the problem description. Each entry stores information such as the current value of the variable, the current status of the variable, and the individual bounds on the value.

- The Active Equation Array (AEA) stores the set of equations that currently determines the continuous time dependent behaviour of the system, in the form of the expressions that determine their residuals. The entry for a conditional equation includes a structure that stores all the equations that could potentially be inserted in the system model, and an indication of which equation is currently active.

- The Active Selector Array (ASA) stores the set of selector attributes included in the problem description and the set of transitions associated with each clause of any corresponding CASE equation. The values of these entries are employed to determine which clause of a CASE equation is inserted in the system model at a given point in time. This information is stored separately from the equations in which the selectors are used, because the same scalar selector attribute can determine the status of an array of CASE equations.

- The Active Logical Condition Array (ALCA) stores the set of logical conditions associated with the IF equations currently included in the system model. Again, these expressions are stored separately from the equations in which they are used, because the same scalar logical condition can determine the active clause of an array of IF equations.

- The Active Scope Array (AScA) holds an entry for each submodel included in the overall system model. Associated with each entry is an AVL balanced binary tree holding all the attributes that make up the submodel. An attribute entry includes a specification of its dimensionality and an indication of where it may be found. If, for example, an attribute is a variable, this indication will take the form of a reference into the AVA, whereas in the case of a unit attribute, it will take the form of a reference to another entry in the AScA. The AScA is primarily employed to locate attributes rapidly by their pathname.

Whenever an equation is inserted in the AEA, it is analysed in order to determine the set of variables that occur in it. This occurrence (or incidence) information is then stored in yet another active array. Each expression stored in the AEA, including those corresponding to the various clauses of a conditional equation, is therefore accompanied by a reference to its occurrence information. Whenever the set of equations currently included in the continuous model changes, the information concerning the pattern of non-zero elements in the Jacobian passed to the solution algorithms can be rapidly updated from this structure.

After generation of its incidence information, each equation is differentiated analytically in order to generate expressions for its partial derivatives with respect to the variables occurring in it. Analytical differentiation is performed according to algorithms discussed by Pantelides (1988c). These exploit the frequent occurrence of subexpressions that are common to both the original equation and its partial derivatives, in order to save both memory and computations. These partial derivative expressions are stored with the occurrence information, hence the data structure is known as the Active Jacobian Array (AJA).

5.3.2 The Simulation Algorithm

The current implementation of the simulation algorithm applies the schedule of tasks encapsulated by a single process entity to the copy of the continuous model created by the instantiation algorithm. As a consequence, the simulation clock is advanced from the initial condition to the termination condition of the process entity.

The procedural nature of this schedule dictates that it must be interpreted directly by the simulation algorithm as the experiment proceeds. It should be noted that it is impossible in general to determine in advance if any particular discrete manipulation will be valid at its time of execution. For example, whether the set of variables designated as being discontinuous by a REINITIAL task (see section 3.2.3) are currently differential variables or whether the resulting initial condition is both consistent and sufficient can often only be determined on execution of the task. Similarly, whether an equation to be discarded by a REPLACE or RESET task currently forms part of the continuous model can only be determined on execution, and even if this operation is successful, the corresponding occurrence information and analytical Jacobian elements must then be generated, followed by reanalysis of the dynamic simulation problem in order to determine if it is still well-posed. Moreover, as in the case of model instantiation, a small number of semantic checks including dimensional correctness of expressions must be postponed until this point.

Algorithm 5.1 co-ordinates the application of a schedule to a continuous model. Following the system initialisation, execution of the schedule (step 2) is instigated, and will continue until execution is complete or all active control structures have been suspended by the execution, for example, of CONTINUE tasks (see section 3.3.5). Note that the simulation clock is not advanced by the execution of the schedule, but entries in the agendas of pending events will be created.

An agenda of pending events must be maintained throughout a combined discrete/continuous simulation in order to determine when suspended portions of the schedule

Algorithm 5.1 *Simulation*

```
1. Initialise the continuous model (see section 1.4.1).
2. Execute the schedule
3. Process any events currently active
4. While events are pending and no error has occurred do
     (a) Advance the simulation clock to the next event
     (b) Process any events currently active
   end
end
```

are to be resumed or discrete changes occur to the underlying continuous model. Time and state events are located in different manners, so they are stored on separate agendas. Furthermore, the state conditions that lead to physico-chemical discontinuities are stored in the active arrays (see section 5.3.1) instead of the state event agenda. They are therefore distinguished as *equation discontinuities* from those state events scheduled by CONTINUE UNTIL tasks.

If the initial execution of the schedule (step 2) has created active entries in these agendas, the associated events must now be processed. At any point in time, active events are processed (steps 3 and 4b) according to algorithm 5.2. The processing of any event causes the immediate resumption of execution of the control structure that was originally suspended by scheduling of the event. The execution of control structures proceeds according to the rules outlined in chapter 3. If a CONTINUE task is executed as a consequence of the resumption of a control structure, this control structure is immediately suspended, and a new entry is added to the appropriate pending event agenda. In particular, CONTINUE FOR tasks create entries on the time event agenda, whereas CONTINUE UNTIL tasks create entries on the state event agenda.

On the other hand, if one of the elementary tasks of section 3.2 (e.g. RESET or

Algorithm 5.2 *Process Current Events*

```
1. Process any active events on the state event agenda
2. Process any active events on the time event agenda
3. While reinitialisation calculation requested or dummy events pending do
    (a) If reinitialisation calculation has been requested then
          i. Reinitialise the continuous model
        end
    (b) If dummy events are pending then
          i. Process the dummy event agenda
        end
   end
end
```

REPLACE) is executed, the specified discrete manipulations of the continuous model must be implemented according to the conditions prevailing at time t^-, corresponding to the point in time immediately preceding the event.

As noted in section 3.3.1, a complication arises from the possibility of several discrete manipulations of the underlying continuous model occurring simultaneously at time t. In this case, the reinitialisation calculation to determine the conditions prevailing at time t^+ immediately following the event must be postponed until *all* active events at time t have been processed. Moreover, the continued execution of the enclosing control structure must be suspended until this reinitialisation calculation has been completed. Execution of an elementary task therefore also involves the issuing of a request for a reinitialisation calculation, and the scheduling of resumption of execution of the enclosing control structure in the form of a *dummy* event. This approach has the computational additional benefit that at most one reinitialisation calculation is performed as a consequence of one or more events occurring at time t.

When all currently active state and time events have been processed, algorithm 5.2 enters a loop (step 3) that will be executed repeatedly until no reinitialisation calculation is currently requested and no dummy events are pending. Processing of the agenda of dummy events may lead to requests for another reinitialisation calculation and/or the creation of a new agenda of dummy events, so this loop may pass through several cycles of reinitialisation followed by processing of dummy events before control can return to the simulation algorithm. Consider, for example, a sequential control structure involving a series of elementary tasks. Execution of the first task will issue a request for a reinitialisation calculation and schedule resumption of the sequence in the form of a dummy event. The algorithm will then enter the above loop, reinitialise the continuous model and then resume the control structure, which will lead to the execution of the next elementary task. This cycle will continue, without the simulation clock being advanced, until execution of the control structure is complete, or a time or state event is scheduled by the sequence as a consequence of the execution of a CONTINUE task.

Once all initially active events have been processed, algorithm 5.1 enters its main loop (step 4), which iterates until all the agendas of pending events become empty. The main loop merely entails repeatedly advancing the simulation clock to the time of the next event(s) followed by processing of these event(s). Advancing the simulation clock between events corresponds to the solution of an initial value problem involving the current set of describing equations from an initial point, dictated by the conditions prevailing immediately following the previous event, until the time of occurrence of the next event. This is accomplished by algorithm 5.3.

The nature of state events (see section 1.4) dictates that their time of occurrence cannot be determined a priori: the solution must be advanced until the occurrence of one or more state events during the previous time step is detected. The algorithm must then locate the exact time of occurrence of the event within this step (see section 1.4.3). Algorithm 5.3 ensures that the current set of the describing equations is altered only after successful location of an event.

The discussion of algorithms for the location of state events in section 1.4.3 highlighted the fact that the time of occurrence of a state event can never be located exactly.

Algorithm 5.3 *Advance to Next Event*

```
1. Gather occurrence information for the current set of describing equations
2. Initialise the DAE solver
3. While no event now do
   (a) Take a time step to desired accuracy (an upper bound on the length of this step is
       also provided by the next time event if one exists)
   (b) If state event or equation discontinuity has occurred during the last step then
         i. Locate the earliest event to within the state event tolerance through bisection on
            the interpolant of the solution over the previous step
        ii. Advance the simulation clock to the upper estimate of the time of occurrence of
            the state event
       iii. Mark any active state events
       else
         i. Advance the simulation clock to the end of the time step
       end
   (c) If equation discontinuity now then
         i. Report the equation discontinuity
        ii. Update the values of any selector variables that experience a transition as a result
            of the equation discontinuity
       iii. Issue request for a reinitialisation calculation
       end
   end
end
```

Instead, it can be determined only within a short period of time known as the state event tolerance. It is therefore possible for several state events to occur within this short period of time and to be treated as effectively simultaneous by the simulation algorithm. Moreover, it is conceivable that more than one state event may occur at *exactly* the same time – if, for example, two separate state events share the same logical condition. In any case, once the earliest event time has been located, *all* active state events must immediately be marked for future processing. This is a necessary safeguard against the possibility of the discrete manipulations caused by one event affecting the logical condition of another event occurring at the same time.

The implementation of algorithm 5.3 requires three routines from a general-purpose DAE solver:

- A routine that will initialise the DAE solution routines at the beginning of each initial value problem.
- A routine that will take a time step to the desired accuracy, or to the next time event, whichever occurs earlier.
- A routine that will interpolate the polynomial approximation to the solution over the previous step.

The current implementation employs the corresponding routines from the general-purpose DAE solver DASOLV (Jarvis and Pantelides, 1992), although in principle any DAE solver that provides these three basic routines could be used. DASOLV pays particular attention to problems that may become *partially determined* during solution and the integration of systems with very fast transients (Jarvis and Pantelides, 1991), but is currently restricted to equations of index not exceeding unity.

The occurrence of an equation discontinuity leads to a request for a reinitialisation calculation and the return of control to the main simulation algorithm.[3] At present, initialisation and reinitialisation calculations suitable for the restricted class of index one DAEs described in section 1.4 are performed using the nonlinear algebraic equation solver SPARSE

[3]This calculation, however, will not be performed until the consequences of any other simultaneous events have been implemented.

(Pantelides, 1988c). All logical conditions relating to IF equations (see section 2.3.5.3) are checked at the beginning of each Newton step, and the current set of describing equations and the corresponding occurrence information are updated if necessary. On the other hand, the equations inserted in the system model by CASE equations are determined by the value of the corresponding selector variable, and can therefore only change immediately before the initiation of a reinitialisation calculation as a consequence of an equation discontinuity or execution of a RESET task.

5.3.3 A Multiple Process Manager

A discussion in section 4.4 described a multiple case management facility, in which the progress of an experiment could be dictated by the execution of multiple process entities. The implementation of this feature will require the simulation algorithm described above to be applied simultaneously to all the currently active process entities. The data structures employed by this algorithm to determine the continuous time dependent behaviour of a system have been designed with this eventual requirement in mind: in this situation, each active process entity will have its own unique set of active arrays (see section 5.3.1).

5.4 The Environment

The environment is intended to provide facilities for the user to build and edit problem descriptions, submit experiments for execution, and interpret the results of experiments. It is now possible to describe how a typical session in this environment might progress.

The session will begin with a specification of the model entities to be used during the session. Some of these may be imported from libraries of previously declared and validated model entities; the definition of others may have to be created in the simulation language and translated. The user will then proceed to import or declare a set of task entities that operate on these model entities. As a consequence, the need for more model entities may be identified and dealt with.

At this point it becomes possible to declare or import one or more process entities that describe simulation experimental frames involving some or all of the model and task entities introduced so far. Once one of these process entities has been translated, and found to be to valid, an entire simulation experiment may be performed simply by issuing the command EXECUTE to the environment, accompanied by a specification of the process entity which describes the desired simulation. Control will return to the environment as soon as this experiment is complete, at which point the results can be analysed, or another cycle of declaration and experiment may commence.

In many situations, correct exploitation of the hierarchical decomposition mechanisms provided will dictate that simulation experiments performed near the beginning of a session or at an early stage in a project will merely involve the validation of components of the overall system under consideration. As the session progresses, and as more validated components become available, the scope of the problem considered by each experiment can broaden, until the intended experiment, involving the complete system, can be performed with a high probability of success. Moreover, by the end of the session a series of experimental frames involving components of this system will also be available for future use. The termination of a session should always be preceded by the archiving of all useful entities developed therein.

Although the design of such an environment is a subject of considerable importance (Stephanopoulos *et al.*, 1987; Bar and Zeitz, 1990; Westerberg *et al.*, 1991), this thesis has concentrated on the fundamental characteristics of combined discrete/continuous simulation as opposed to the user interface, so the current implementation of the environment only supports rudimentary features. In particular, the entire problem description (including model, task, and process entities) must be prepared a priori in the form of an input file. An example of such as input file is shown in appendix C.

A session is commenced by typing the command ‘gPROMS’. This enters the user into a simple command line interpreter, from which commands to direct the progress of the session may be issued. The following set of commands is currently available:

- **SELECT** – submits a specified input file to the translator.
- **DIRECTORY** – provides a list of the input files currently available to the user.
- **EDIT** – enables a specified input file to edited or created. On exit, the input file is automatically submitted to the translator.
- **LIST** – provides a list of all the process entities currently available for execution.

- **EXECUTE** – submits a specified process entity to the process manager for execution. Control will return to the environment on termination of the experiment.
- **CODE** – submits a specified process entity for code generation. This leads to the generation of a set of FORTRAN subroutines for residual and derivative evaluations. These may be exported for use by other software packages but are not otherwise used by gPROMS.
- **PLOT** – enters a graph plotting environment which may be employed to display time trajectories of the describing variables generated by an experiment.
- **QUIT** – exits the environment.

5.5 Summary

The implementation of a prototype modelling package, gPROMS, based on the simulation language introduced in earlier chapters has been discussed. The special requirements of combined discrete/continuous simulation dictate a software architecture involving the creation of a copy of the continuous model in memory, which is then employed by the simulation algorithm to determine the dynamic behaviour of the system under investigation. The main advantage of this approach lies in the ability to manipulate this copy of the continuous model directly for the entire duration of a simulation, but a significant reduction in the time required to prepare a simulation experiment for execution can also be achieved. The implementation of the two major components of the modelling package, the translator and the process manager, are considered in detail.

The current version of the prototype has been implemented in SUN Modula–2 Release 2.1, on engineering workstations running the variant of the UNIX operating system SunOS Version 4.1.1. Earlier versions of the prototype have also been successfully ported to personal computers running the DOS and OS/2 operating systems, although the DOS version is severely limited by the memory restrictions imposed by the operating system. The development of standard utility library routines for the modelling package that are then interfaced to the library routines provided with an individual Modula–2 compiler have ensured that the code can potentially be ported to a wide range of platforms.

In section 5.1 it was claimed that significant improvements in the speed of problem translation and subsequent problem preparation could be achieved with the software architecture adopted by gPROMS. Typical translation speeds are illustrated by the evaporator pilot plant example described in section 6.2.1. The corresponding input file (see appendix C) is composed of 1232 lines of input, contains the declaration of twelve model entities, eight task entities and a single process entity. Its translation on a SUN IPX workstation with 16Mb of memory takes approximately 8.8 seconds. Moreover, this time is more than halved if the input file is not echoed to the terminal during translation!

Model instantiation speeds are illustrated by the tubular reactor model shown in figure 2.8. Although translation time for this example is trivial, increasing the number of spatial mesh points enables the creation of a model containing an arbitrary number of equations. For example, a model composed of 4950 equations takes approximately 42.4 seconds to instantiate.[4]

The above figures are quite typical of gPROMS' performance. It can be seen that today's engineering workstations enable complex dynamic simulation problems involving several thousand equations to be posed interactively.

[4]This includes the generation of analytical partial derivatives for all the equations.

Chapter 6

Combined Discrete/Continuous Simulation Examples

The introduction to this thesis argued that combined discrete/continuous simulation is a potentially powerful tool for the analysis of process dynamics, particularly for those systems that experience significant discrete changes superimposed on their predominantly continuous behaviour. It also identified the lack of suitable representational methodologies as a major outstanding obstacle to the development of a general-purpose simulation package for this category of systems. In chapters 2 to 4, these shortcomings where addressed through the introduction of a new simulation language to encompass the description of this class of problems, and chapter 5 detailed the implementation of a prototype modelling package based on this language.

This chapter presents a collection of detailed simulation examples designed to illustrate potential applications of combined discrete/continuous simulation. These examples also demonstrate the descriptive capabilities of the simulation language described in earlier chapters when applied to this class of problems. All the simulation results presented here were generated using the current implementation of the prototype modelling package.

The chapter begins with a series of brief examples involving physico-chemical discontinuities. These illustrate how simulations with significant discrete characteristics can arise merely as a consequence of the physical mechanisms that determine the underlying continuous behaviour of any processing system. Each of the remaining examples focuses on a complete processing system, and is categorised according to the mode in which the process is operated. This demonstrates the application of combined discrete/continuous simulation to the entire range of process operations, from processes that are normally considered to be 'continuous', through processes operated in a periodic or semi-continuous manner, to batch processes. In all these examples, discrete changes arising from physico-chemical mechanisms occur *in addition* to the discrete changes caused by the control actions applied to maintain the desired mode of operation.

6.1 Physico-chemical Discontinuities

The facility to include discrete changes to the describing equations that occur as a result of physico-chemical mechanisms is vital to all but the simplest of modelling exercises involving processing systems. The first three examples presented here, concerning a weir, a bursting disc, and a safety relief valve respectively, all concentrate on how an individual feature of a larger system might be modelled as a physico-chemical discontinuity. The arguments put forward in chapter 2 to establish the necessity of a general formalism based on finite automata are further reinforced by the results of these simulations.

In the final example, the more complex model of a flash drum is considered. This example is primarily intended to demonstrate the application of a finite automaton to a system which can move dynamically through more than two states.

6.1.1 Vessel Containing an Overflow Weir

The first example, regarding a vessel containing an overflow weir that regulates the flow of liquid from it, has already been introduced in chapter 2, and one possible model is shown in figure 2.10. During normal operation, this device will maintain a relatively constant holdup of material in the vessel, but if, for any reason, the level of liquid in the vessel drops to the height of the weir, or below it, flow from the vessel will cease until this level rises above that of the weir again. A reversible discontinuity, implemented by an IF equation, is the most convenient way to model this phenomenon.

The results of a simulation employing this model are shown in figure 6.1. The vessel is considered to be initially empty, with no material flowing into it. After a short period material begins to flow into the vessel, and, as a consequence, the liquid level eventually rises to that of the weir (one metre), at which point material begins to flow over the weir. The liquid level continues to rise until the flow over the weir exactly balances that of the flow into the vessel. The flow of material to the vessel is then stopped, and the liquid level drops to that of the weir.

As already stated, this example is merely an illustration of how an individual feature of a larger system might be modelled. This representation of a weir could just as easily be applied within the model of a distillation column tray. If instances of this tray model are

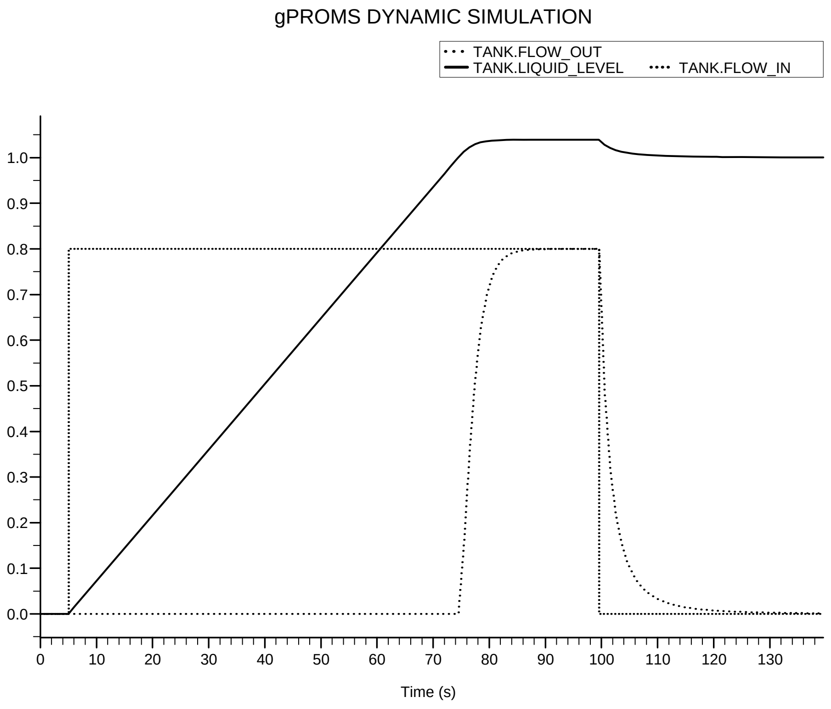

Figure 6.1: Simulation of a Vessel Containing an Overflow Weir

then inserted as components of a distillation column model that is employed for start-up or shut-down studies, the model of the overall system would be composed of many finite automata, corresponding to the behaviour of the weir on each tray of the column.

6.1.2 Vessel Fitted with a Bursting Disc

In certain safety critical situations, pressure vessels are fitted with a protective device known as a bursting disc. This is a disc of a material such as graphite, sealing a vent outlet of the vessel, which provides a form of 'one-off' protection for the vessel: if the pressure in the vessel rises to a critical value, the disc shatters and the contents of the vessel are exhausted through the vent. The disc must then be replaced before the vessel can be pressurised again. The physical mechanism that results in the disc shattering is irreversible, so a model for this phenomenon must be implemented through use of a CASE equation. One possible model for this device, shown in figure 2.9, has already been discussed in chapter 2.

The model shown in figure 2.9 assumes that the inlet and normal outlet flowrates of gas are determined by equations declared in upstream and downstream units respectively. For the purposes of the simulation presented in figure 6.2, the outlet valve has been closed and the flowrate into the vessel is considered to be an input variable. After an initial period of five seconds in which the vessel is held at atmospheric pressure, gas is introduced at a constant flowrate, and the pressure in the vessel rises to the set pressure of the disc. At this point, the disc shatters and the pressure in the vessel drops until the relief flowrate balances the inlet flowrate. Note that, unlike an IF equation, the CASE equation employed here does not return the vessel to the intact state as soon as the condition that triggered the original transition is negated.

The model employed in this example makes the assumption that the relief flow is always choked. For more accuracy, the equation that determines the choked flowrate could be replaced by an IF equation that determines the correct flow/pressure relationship by comparing the downstream pressure with the critical pressure, thereby exploiting the recursive definition for a finite automaton given in chapter 2.

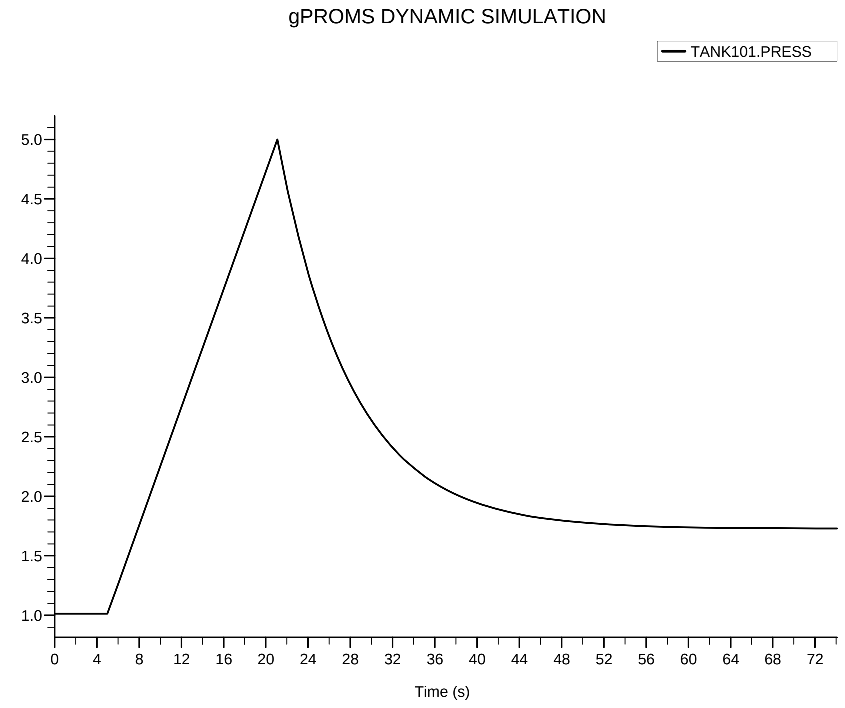

Figure 6.2: Simulation of a Vessel Fitted with a Bursting Disc

6.1.3 Vessel Fitted with a Safety Relief Valve

One of the devices most frequently used to protect pressure vessels against deformation as a result of excessive pressure is the safety relief valve. As already discussed in chapter 2, the mechanism that causes one of these valves to open and close is usually asymmetric, and must therefore be modelled by a CASE equation. The model employed in this example is almost identical to that of the vessel fitted with a bursting disc, so figure 6.3 only demonstrates the CASE equation that determines the relief flowrate.

In figure 6.4, the vessel again experiences a initial period in which the pressure is maintained at atmospheric pressure, after which gas is introduced at a constant flowrate and the pressure rises to the set pressure of the relief valve. At this point, the relief valve opens and the pressure in the vessel begins to drop. In this example, however, when the pressure drops to the reseat pressure the valve closes and the pressure in the vessel begins to rise again. The system will continue to repeat this cycle until some form of external intervention occurs.

It is sometimes considered necessary to install both a safety relief valve and a bursting disc in order to protect a vessel. The bursting disc provides protection in the event of the safety relief valve being unable to supply the relief flow required. A model of this system would be composed of a set of invariant equations that remain valid regardless of the state the system is in (see section 2.2) and two independent finite automata that determine the relief flows through the safety relief valve and bursting disc respectively.

6.1.4 Equilibrium Flash Drum

The final example that concentrates on physico-chemical discontinuities in isolation involves the equilibrium flash drum shown in figure 6.5. A liquid stream at high pressure passes through a valve at the inlet to the flash drum, which results in the sudden irreversible and isenthalpic expansion of the stream. If the drum pressure is lower than the bubble point pressure of the feed, some of the liquid will vapourise. Vapour escapes through a valve at the top of the drum, and liquid is drawn off through a valve at the bottom of the drum. For the purposes of this simulation, it is assumed that the vessel is fitted with devices similar to a stream trap that will only allow liquid to flow through the lower valve, and likewise will

```
SELECTOR
  Valve_State                     AS (Closed,Open) DEFAULT Closed

EQUATION

  # Relief flow from safety valve - assume choked flow
  CASE Valve_State OF
    WHEN Closed : Relief_Flow = 0 ;
                  SWITCH TO Open IF Press > Set_Pressure ;
    WHEN Open   : Relief_Flow = Valve_Const*Press/SQRT(Temp) ;
                  SWITCH TO Closed IF Press < Reseat_Pressure ;
  END # case
```

Figure 6.3: Extracts from the Model of a Vessel Fitted with a Safety Relief Valve

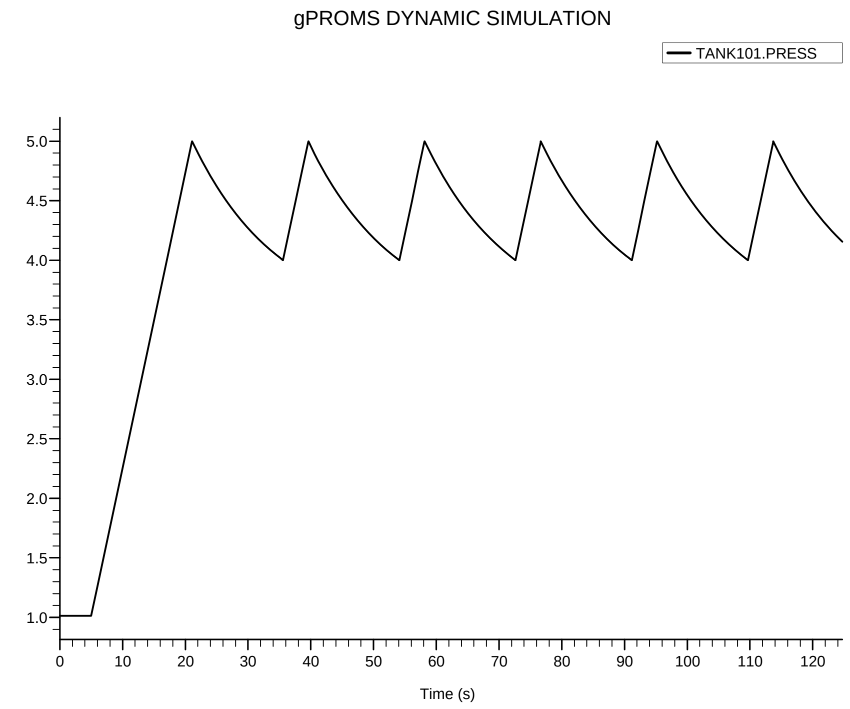

Figure 6.4: Simulation of a Vessel Fitted with a Safety Relief Valve

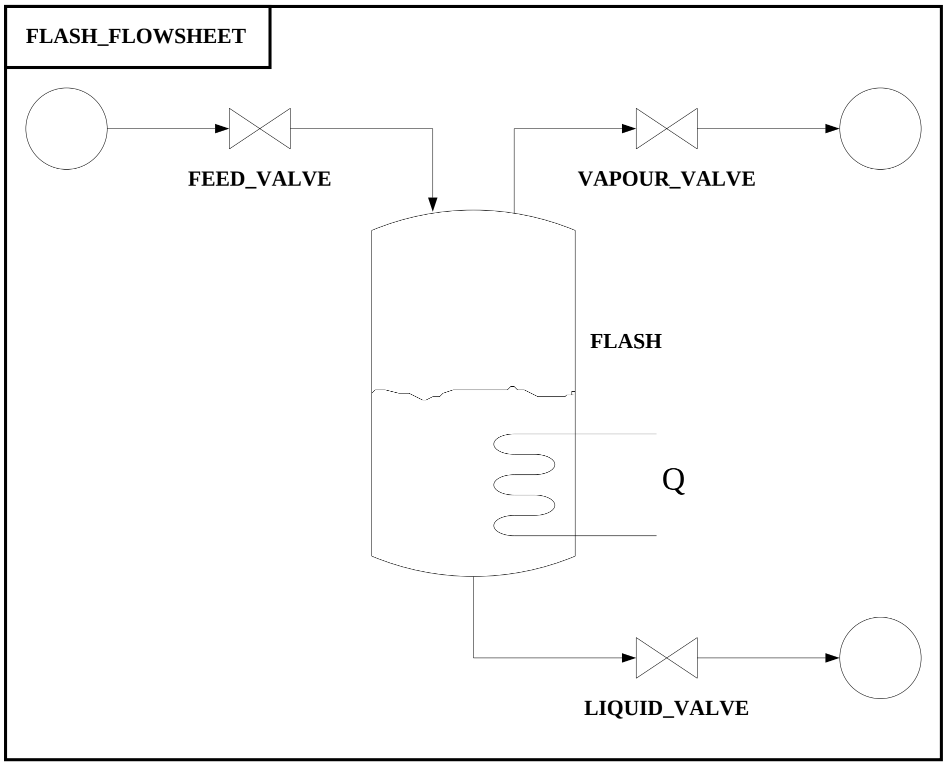

Figure 6.5: Flowsheet Involving Equilibrium Flash Drum

only allow vapour to escape from the upper valve. Heat may be supplied to or removed from the vessel via a coil.

This example is of interest because the model of the vessel is again composed of an invariant set of equations, and a finite automaton that determines the equations specific to each phase regime. The automaton involves at least three states, corresponding to whether the vessel contains both liquid and vapour phases, subcooled liquid, or superheated vapour. Only the simple case of a single liquid phase is considered here, but if two liquid phases were to form in addition to a vapour phase, the number of states would have to be increased accordingly.

In order to develop a relatively simple model of the continuous time dependent behaviour of the flash drum, the following assumptions concerning the conditions prevailing inside the vessel are made:

- Apart from heat supplied or removed via the coil, the vessel is operated under adiabatic conditions.

- The contents of each phase are perfectly mixed at all times.
- The rate at which both thermal and phase equilibrium are reached is much faster than the rate at which changes in the bulk properties of the contents occur. The vessel can therefore be considered to be at thermal and phase equilibrium at all times.

As a consequence of these assumptions, the control volume employed to derive the balance equations can encompass the entire vessel, as opposed to each individual phase. The minimal set of properties required to determine the state of the system, combined with a specification of the volume of the vessel (a time invariant parameter), are the total molar holdups for each component and the total internal energy holdup.

When both a liquid and a vapour phase are present in the vessel at equilibrium, a set of differential equations defining the component and energy balances, and a set of auxiliary algebraic equations including constitutive relationships (Ponton and Gawthrop, 1991), phase equilibrium relationships, and the volume constraint, will form an adequate dynamic model. If, however, only a single phase is present in the vessel, a number of auxiliary algebraic quantities, including the component mole fractions of the phase not present, are no longer necessary for the description of the system, and the number of auxiliary algebraic equations required drops significantly. For the purposes of this simulation, the model is declared in terms of the maximal set of describing variables, as required by the two phase regime, and while the system is in a single phase regime, the UNDEFINED construct (see section 2.3.5.4) is employed to eliminate the unnecessary variables. The modelling equations derived in this manner are included in appendix D.

In addition, the volume constraint creates problems for the numerical algorithms employed for the solution of the describing equations. If the vessel is full of subcooled (effectively incompressible) liquid, this constraint defines a relationship between the component holdups, as opposed to determining the pressure of the contents. A change in the mathematical properties of the describing equations, the index of which increases from one to two, therefore occurs as the system moves dynamically from the two phase regime into the liquid regime. This will precipitate the immediate failure of the simulation as a consequence of the limitations imposed by the solution routines employed by the current implementation of the modelling package. One way of posing a model that can move dynamically through all

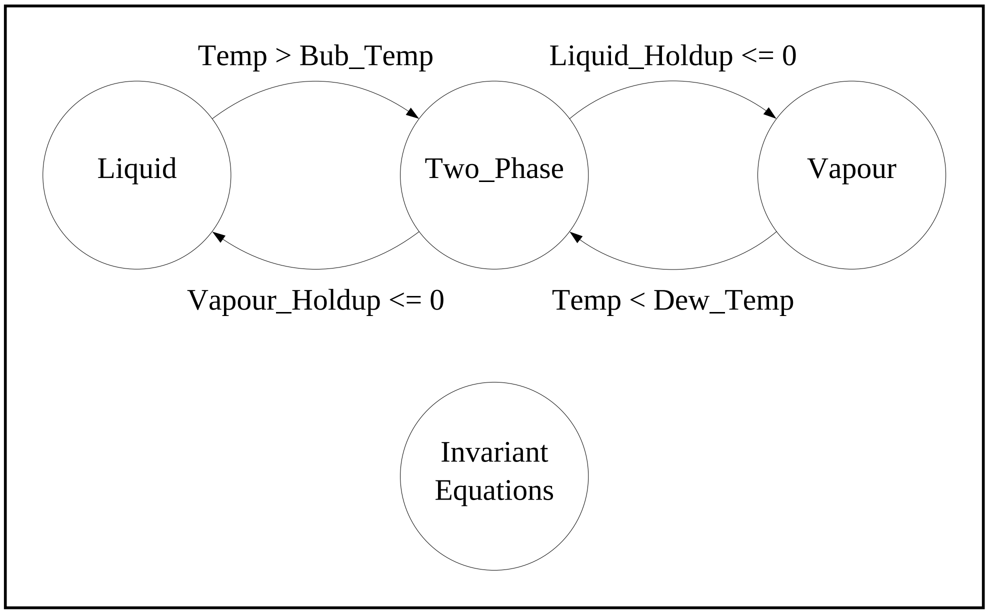

Figure 6.6: Finite Automaton for Flash Vessel

three phase regimes is to reduce the index of the describing equations in the liquid regime by differentiation of this volume constraint with respect to time. If the liquid phase is considered to be an ideal mixture and the temperature dependence of its density is neglected, this differentiation, accompanied by algebraic manipulation, yields an algebraic relationship defining equality between the volumetric flow into the vessel and the volumetric flow from the vessel. For the model employed in this example, the volume constraint is added to the set of variant equations and is replaced by its differentiated form whenever the system enters the liquid regime.

At any point in time, the variant set of equations inserted in the system model is determined by the active state of the finite automaton illustrated in figure 6.6. It now only remains necessary to define the possible transitions between these states, and the logical conditions that must be satisfied in order to trigger them. As can be seen from figure 6.6, four transitions are possible, but at no time is it possible to move from the vapour regime to the liquid regime without first passing through the two phase regime. The logical conditions that trigger a transition from a single phase regime to the two phase regime are intuitively obvious: when the temperature of a subcooled liquid rises to its bubble temperature, two phases will appear, and when the temperature of a superheated vapour falls to its dew temperature, two phases will also appear. The logical conditions that trigger a transition from the two phase regime to a single phase regime are not, however, as obvious, because there are, in both cases, two equivalent conditions:

- To move from the two phase regime to the liquid regime, the vapour holdup in the vessel must drop to zero, *or* equivalently the temperature must drop to the bubble temperature of the contents.
- To move from the two phase regime to the vapour regime, the liquid holdup in the vessel must drop to zero, *or* equivalently the temperature must rise to the dew temperature of the contents.

If the dew and bubble temperature conditions are employed, the finite automaton can be implemented by two nested IF equations, because of the symmetry of the transition conditions. On the other hand, if the holdup conditions are employed, a CASE equation is required to implement the resultant asymmetric discontinuities. There are, however, strong arguments advocating the adoption of the holdup conditions. In the first instance, if the model is used for a single component fluid,[1] the dew and bubble temperatures of the fluid become indistinguishable, and therefore the holdup conditions *must* be used to determine transitions from the two phase regime. Secondly, experimentation with the model has demonstrated that, although the two forms of a condition are physically equivalent for multicomponent mixtures, the numerical nature of the equation solution will not always guarantee that they will be satisfied simultaneously. In certain circumstances, a period of fluttering between phase regimes will occur as a consequence of slight numerical inaccuracies in the bubble or dew temperatures that lead to the triggering of a temperature condition before the corresponding holdup reaches zero. This fluttering will continue until integration has advanced sufficiently for the holdup to approach zero, but is eliminated when the holdup conditions are employed.

Two dynamic simulation experiments involving the model described above are presented here, employing simple assumptions in order to determine the physical properties of the fluids flowing through the vessel. The disturbances introduced in both cases are rather artificial, but they are sufficient to demonstrate the model moving dynamically through all

[1]This can be implemented merely by setting the integer parameter that determines the number of components in the model to one.

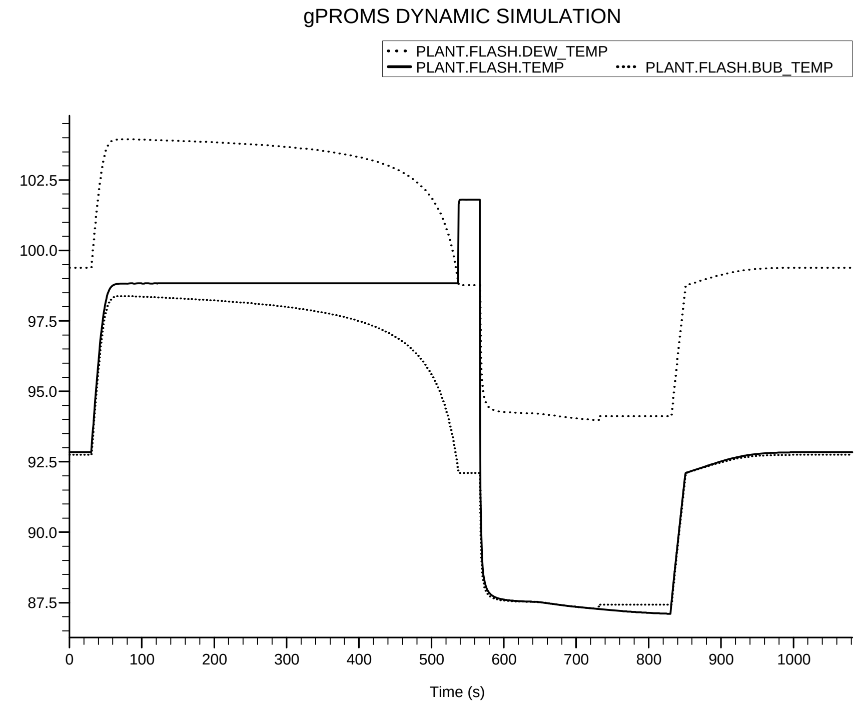

Figure 6.7: Simulation of Multicomponent Flash Drum

three states. The time trajectories from a simulation involving a feed stream containing an equimolar mixture of benzene and toluene are shown in figure 6.7. Table 6.1 contains a summary of the events occurring during the simulation.

From steady-state, a step increase in the heat input to the vessel is introduced that eventually drives the model into the vapour phase regime. The heat input is then decreased sufficiently to drive the model from the vapour regime, through the two phase regime, and into the liquid regime. Finally, the heat input is returned to its original value, and the vessel approaches a steady-state in the two phase regime. This simulation therefore demonstrates all four of the possible transitions described in the previous section. The time trajectories from a similar simulation involving a feed stream containing pure benzene is shown in figure 6.8.

Time (s)	Event
0	Initial condition in the two-phase regime.
30	Step increase in the heat input introduced.
537	Vessel enters the vapour regime.
567	Step decrease in the heat input introduced.
567	Vessel re-enters the two phase regime.
730	Vessel enters the liquid regime.
830	Heat input returned to original value.
831	Vessel re-enters the two phase regime.

Table 6.1: Table of Events During Simulation of Multicomponent Flash Drum

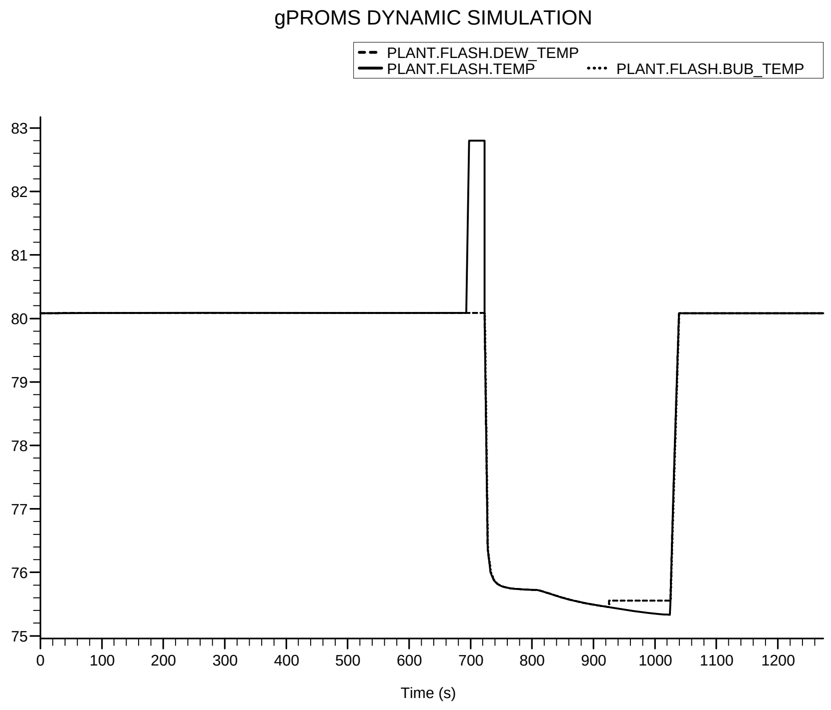

Figure 6.8: Simulation of Single Component Flash Drum

6.2 'Continuous' Processes

It is particularly important to emphasise the role of combined discrete/continuous simulation in the analysis of systems or processes that are normally considered to be 'continuous', because therein lies one of the biggest weaknesses of the existing continuous process simulation packages. In any application of dynamic simulation where external actions or environmental influences can take a continuous process far from its nominal steady-state operating point (such as safety and environmental studies, operating procedure validation, or computer based operator training) it is important to be able to describe the control actions and disturbances that drive the process into the abnormal state, or return it to the nominal operating point. Moreover, the complexity of the continuous models required for the accurate simulation of these large deviations is often much higher than that of many continuous models in use today, which, by and large, have only been designed for the simulation of small perturbations around a nominal steady-state.

On the other hand, the simulation language described in earlier chapters has been specifically designed with these requirements in mind. Hopefully, it will ease considerably the development of many simulations of continuous processes that are at present considered problematic. Also, it should render feasible simulations that were hitherto considered infeasible or impracticable within the context of a general-purpose package.

Two applications of combined discrete/continuous simulation to 'continuous' processes are presented here. The first example, involving a model of the evaporator pilot plant at Imperial College, demonstrates the description of the complex sequential operations that are often applied to continuous processes in order to switch between operating points, or to accomplish start-up and shut-down. Furthermore, the obstacles encountered during the development of a continuous model to encompass the large deviations experienced by the process are a graphic illustration of the increased model complexity required by this class of problems.

The second example concentrates on an effluent tank in which a digital control strategy is employed to regulate the pH of the discharge stream, and is primarily intended to demonstrate how the primitive elements of the simulation language may be enlisted for the simple and elegant expression of digital control laws. The conventional approach to the description of digital control laws in conjunction with continuous process simulation packages has either involved a language extension specific to digital control, or external code inserted in the continuous model (effectively introducing the hidden events of section 2.3.6).

6.2.1 Evaporator Pilot Plant

The continuous process considered in this example is the evaporator section of the evaporator/crystalliser pilot plant installed in the Chemical Engineering Department at Imperial College. The pilot plant is employed extensively for both undergraduate teaching and research, so a dynamic model has many potential applications.

The pilot plant, a schematic of which is shown in figure 6.9, is a single effect forced circulation evaporator employed to concentrate an aqueous solution of potassium nitrate. A weak solution (5% wt.) is fed to the evaporator loop, which operates at atmospheric pressure and between 100°C and 104°C. The product (approximately 10% wt.) is collected in a receiver tank, whilst the vapour is condensed and subcooled. During normal operation, solution enters the shell of the forced circulation heat exchanger (the calandria) and is heated by a tubeside steam stream as it rises vertically. The solution subsequently rises up through a vertical tube and overflows into the splitter, where the vapour is separated from the concentrated solution. Some of this solution is directed into the product tank, whereas the remainder leaves from the bottom of the splitter, mixes with the dilute feed, and is then pumped back into the calandria.

Earlier attempts at modelling this process with the SpeedUp dynamic simulation package (Bernal, 1986; Rozo, 1987) encountered considerable difficulties as a direct result of the limitations imposed by the aforementioned package. The inability to introduce new equations at intermediate hierarchical levels, and to express asymmetric physico-chemical discontinuities, meant that the relationships that determined the direction and magnitude of material flows in the process effectively destroyed the modular nature of the model. Moreover, the simulation of sequential operations applied to the process could only be accomplished by an external software package that manipulated the continuous model via SpeedUp's External Data Interface (Prosys, 1991). Nevertheless, a simplified version of this original model has been used to build a computer based operator training system for the process (Kassianides, 1991).

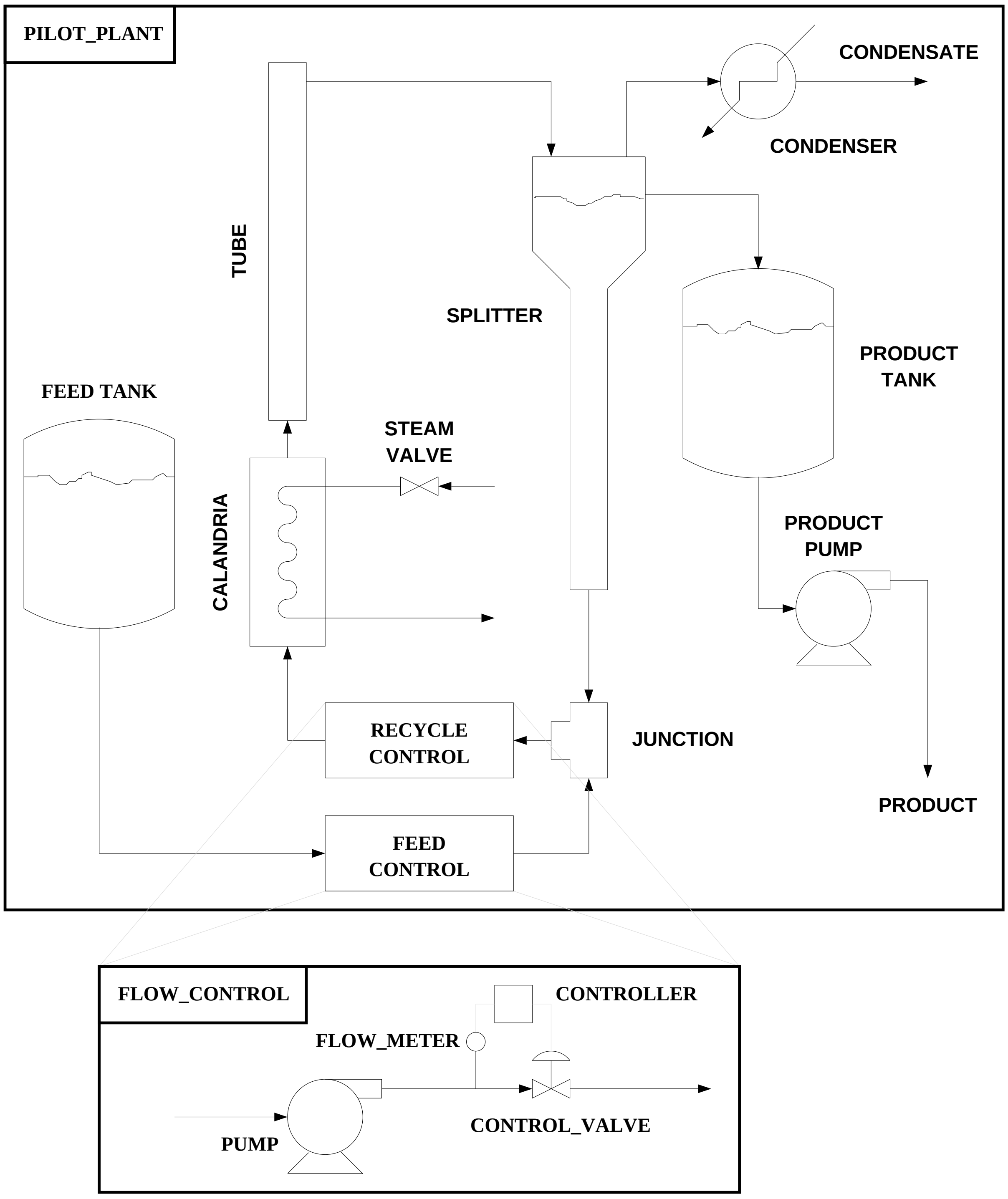

Figure 6.9: Evaporator Pilot Plant Schematic

One object of this exercise is to demonstrate how the simulation language described in earlier chapters may be employed to express the direction and magnitude of material flows in the process in terms of relatively realistic flow/pressure relationships, thereby enabling development of the continuous model of each unit operation in a manner that preserves its modularity. As a consequence, separate task entities may be developed to describe operations involving each of these individual units, such as a task that switches on an instance of the pump model entity. Furthermore, instances of the individual unit operation models may be combined in a flowsheet to form a continuous model of the complete process, and the task entities developed to act on the components of this overall model may be combined to describe any sequential operation applied to the process. More details of the continuous models are given by Smith (1991).

As an illustration of the increased complexity required in continuous models employed for the simulation of large deviations from steady-state, even in such a relatively simple system as an evaporator, the model of the calandria will be described in detail. Although at steady-state the calandria will always be full of liquid, during certain operations such as start-up or shut-down, the liquid level will rise and fall through the vessel. This phenomenon may be modelled by a physico-chemical discontinuity with two states corresponding to whether or not the calandria is completely full of liquid. While the liquid level remains below the vessel height, it may rise and fall freely, and no liquid flows upwards into the tube. However, when the level reaches the vessel height, it becomes constrained to that value, and any excess liquid flows upwards into the tube.

This discontinuity is, in fact, asymmetric. As already stated, a transition to the 'full' state occurs when the liquid level reaches the vessel height. The return transition, however, may not be determined from the negation of the condition based on the liquid level, because the equations that characterise the 'full' state require this level to remain equal to the vessel height at all times. This transition will actually occur only when the level in the tube above drops to zero, which is equivalent to the pressure at the top of the calandria dropping to atmospheric pressure. Hence the asymmetry.

Moreover, as in the flash drum example, the liquid height constraint causes the index of the describing equations to increase dynamically from one to two as the system enters the 'full' state. The describing equations in the 'full' state must therefore be reformulated to

```
CASE Calandria_State OF
 WHEN Not_Full : OverFlow_Flow = 0 ;
                 SWITCH TO Full IF Liquid_Height >= Height_Liquid_Max ;
 WHEN Full     : OverFlow_Flow = Gain*(Liquid_Height - Height_Liquid_Max) ;
                 SWITCH TO Not_Full IF Press_Top <= Press_Atm ;
END # case
```

Figure 6.10: Extract from the Model of the Calandria

reduce the index to one. This can be achieved rigorously by differentiating the height constraint with respect to time, thereby obtaining an explicit expression for the outlet flowrate. Alternatively, an approximate solution may be obtained by relaxing the height constraint to one that tightly controls the liquid level through suitable adjustment of the flowrate upwards into the tube (e.g. by using a proportional controller with high gain).

The latter solution was adopted in this example (see figure 6.10). Note that adoption of this controller equation has no consequences whatsoever regarding the asymmetric nature of the discontinuity. In the 'full' state, the controller equation will allow the liquid level in the calandria to rise slightly if the level in the tube above is rising, and to drop slightly if the level in this tube is dropping. However, a transition to the 'not full' state can only occur if the level in the tube above drops to zero, thus reducing the pressure at the top of the calandria to atmospheric.

Further physico-chemical discontinuities similar to the above are required to model other large deviations from steady-state within the calandria (e.g. the contents of this vessel may, independently, be boiling or not), and in most of the other major unit operations of the process.

A further complication arises from the need to determine the intensive properties (in this case temperature and solute concentration) of process streams in which the flow direction is dependent on the conditions prevailing in the plant. The intensive properties of such a stream are determined from the bulk properties of the unit that is currently upstream with respect to the direction of flow. If this direction reverses, so does the unit that determines these properties. This problem is resolved by eliminating the equations that determine the intensive properties of reversible streams from the models of individual unit operations, and using conditional equations at the hierarchical level at which the stream connection is declared to determine these intensive properties from the current direction of flow in the stream in question. A simple model demonstrating this approach is shown in figure 6.11.

```
MODEL Reversible_Flow

UNIT
  First_Tank, Second_Tank            AS Tank

EQUATION

  # Stream connection
  First_Tank.Outlet IS Second_Tank.Inlet ;

  # Conditional equation that determines the intensive properties of the
  # stream.
  IF First_Tank.Total_Outlet_Flow >= 0 THEN
    WITHIN First_Tank DO
      Outlet_Conc = Bulk_Conc ;
      Outlet_Temp = Bulk_Temp ;
    END # within
  ELSE
    WITHIN Second_Tank DO
      Inlet_Conc = Bulk_Conc ;
      Inlet_Temp = Bulk_Temp ;
    END # within
  END # if

END # Reversible_Flow
```

Figure 6.11: Determination of the Intensive Properties of a Reversible Stream

Having described the continuous models of the unit operations in a modular manner, it is possible to develop a series of task entities that describe typical operations involving these units. This study, for example, requires task entities to open and close a valve, to start and stop a pump, and to switch an analogue controller from manual to automatic control and vice versa. The task entity that starts a pump, for instance, may then be applied to any of the three instances of the pump model entity that appear in the evaporator process shown in figure 6.9.

It is now possible to construct a simulation of an operation involving the entire process. The operation considered is typical of an experiment performed by undergraduate students in the course of an afternoon, in which the plant is started from a cold and empty state, run for a short period, and then shut-down. As part of this exercise, the students are required to propose safe start-up and shut-down procedures. A situation can therefore be envisaged in which the students use a *simulation* of their proposed procedures, constructed from the general-purpose tasks described above, for the purposes of validating them before application to the process itself (Macchietto *et al.*, 1987). One possible start-up procedure for the process involves the following sequence of elementary steps:

1. Start the feed pump and close the feed flow control loop.
2. Wait until all units in the recycle loop are full of liquid and solution begins to overflow into the product tank.
3. Start the product pump.
4. Start the recycle pump and close the recycle flow control loop.
5. Start the flow of steam to the calandria.
6. Wait until the temperature in the splitter reaches 99°C.
7. Start the flow of cooling water to the condenser.

A task entity that drives the continuous model of the pilot plant through this procedure is shown in figure 6.12. Note that a model of the condenser was not included in this example, so no operations on it appear in the schedule. Similarly, a possible shut-down procedure involves:

1. Stop the flow of steam to the calandria.
2. Stop the product pump.
3. Continue to feed cold, dilute solution to the plant until the calandria has cooled to 60°C.
4. Shut off the feed and recycle pumps, and open the flow control loops.
5. Stop the flow of cooling water to the condenser.

A task entity that drives the continuous model of the pilot plant through this procedure is shown in figure 6.13. All that remains is to declare a process entity (excerpts from

```
TASK Start_Up_Pilot_Plant

PARAMETER
  Plant                         AS MODEL Complete_Plant_Flowsheet

SCHEDULE
  SEQUENCE
    PARALLEL
      Start_Pump(Pump IS Plant.Feed_Control.Pump) ;
      Close_Loop(Controller IS Plant.Feed_Control.Controller) ;
    END
    CONTINUE UNTIL Plant.Splitter.Product_Flow > 1E-4
    Start_Pump(Pump IS Plant.Product_Pump) ;
    CONTINUE FOR 20
    PARALLEL
      Start_Pump(Pump IS Plant.Recycle_Control.Pump) ;
      Close_Loop(Controller IS Plant.Recycle_Control.Controller) ;
    END
    CONTINUE FOR 20
    Open_Valve(Valve IS Plant.Steam_Valve) ;
    CONTINUE UNTIL Plant.Splitter.Temp_Bulk > 99.0
  END
END # Start_Up_Pilot_Plant
```

Figure 6.12: Task Describing Start-Up Procedure for the Evaporator Pilot Plant

```
TASK Shut_Down_Pilot_Plant

PARAMETER
  Plant                         AS MODEL Complete_Plant_Flowsheet

SCHEDULE
  SEQUENCE
    Close_Valve(Valve IS Plant.Steam_Valve) ;
    CONTINUE FOR 20
    Stop_Pump(Pump IS Plant.Product_Pump) ;
    CONTINUE UNTIL Plant.Calandria.Temp_Bulk < 60.0
    PARALLEL
      Stop_Pump(Pump IS Plant.Feed_Control.Pump) ;
      Stop_Pump(Pump IS Plant.Recycle_Control.Pump) ;
      Open_Loop(Controller IS Plant.Recycle_Control.Controller) ;
      Open_Loop(Controller IS Plant.Feed_Control.Controller) ;
    END
  END
END # Shut_Down_Pilot_Plant
```

Figure 6.13: Task Describing Shut-Down Procedure for the Evaporator Pilot Plant

```
PROCESS A_Pilot_Plant_Simulation

...

SCHEDULE
  SEQUENCE
    CONTINUE FOR 100
    # Apply the start-up procedure
    Start_Up_Pilot_Plant(Plant IS Pilot_Plant) ;
    # Run the plant for a while
    CONTINUE FOR 5000
    # Apply the shut-down procedure
    Shut_Down_Pilot_Plant(Plant IS Pilot_Plant) ;
    # Let the plant drain
    CONTINUE UNTIL Time > 21000
  END
END
```

Figure 6.14: Process Entity Describing Simulation Involving the Evaporator Pilot Plant

which are shown in figure 6.14) which combines these two tasks to drive the continuous model through a simulation of the entire afternoon's activities. The description of a complex operation involving the entire pilot plant has therefore been constructed from the hierarchical combination of a set of simple operations involving individual items of process equipment. The entire gPROMS input file for this problem is included in appendix C.

During the afternoon, the proposed start-up procedure is applied, the process is then run for a while, and finally the shut-down procedure is applied. Table 6.2 contains a summary of the significant events during this period, and trajectories of some of the key process variables are shown in figures 6.15 to 6.17. It is interesting to note that steady-state is not reached within the 5000 seconds allowed between the end of the start-up procedure and the beginning of the shut-down procedure. In fact, because it takes approximately nine hours to achieve, it is not possible to reach steady-state within a single afternoon! The liquid level and concentration trajectories correspond closely to those of the real process (Bernal, 1986), but improvement of the heat transfer relationships employed in the unit operation models would lead to more accurate temperature transients (Smith, 1991).

Time (s)	Event
100	Feed pump is started and feed control loop closed
461	Calandria becomes full and the tube starts to fill
1458	Splitter widens from a pipe to a vessel
4728	Splitter starts to overflow into product tank
	Product pump starts
4748	Recycle pump starts and recycle control loop closed
4768	Steam is introduced into the calandria
6945	Splitter temperature reaches 99^{o}C
	Start-up procedure completed
6946	Boiling occurs in the calandria
11945	Steam ceases to flow to the calandria
	Boiling ceases in the calandria
11965	Product pump switched off
14447	Calandria temperature drops to 60^{o}C
	Feed and recycle pumps switched off
	Feed and recycle flow control loops opened
	Shut-down procedure completed
	Solution begins to drain back into the feed tank
19013	Tube becomes empty and the calandria starts to drain
21000	Liquid levels in splitter, feed tank and calandria equalise

Table 6.2: Table of Significant Events During Operation of Evaporator Pilot Plant

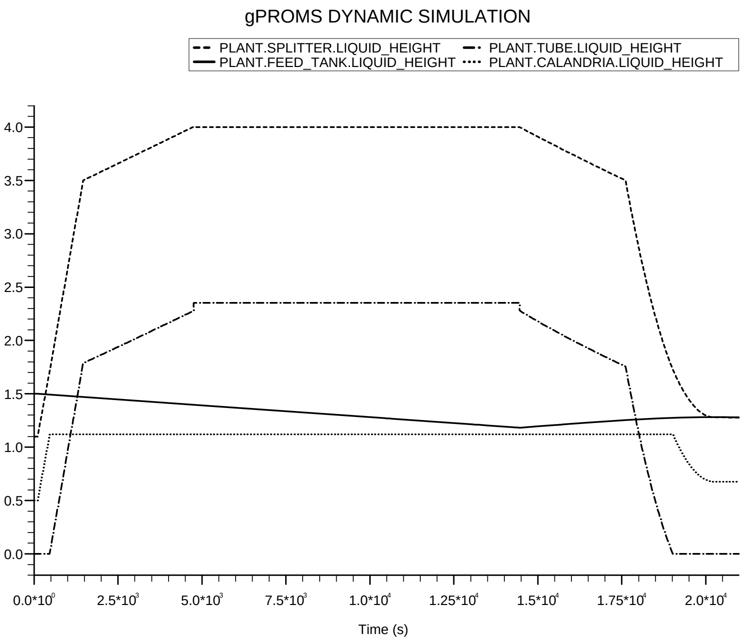

Figure 6.15: Liquid Levels in the Evaporator Pilot Plant

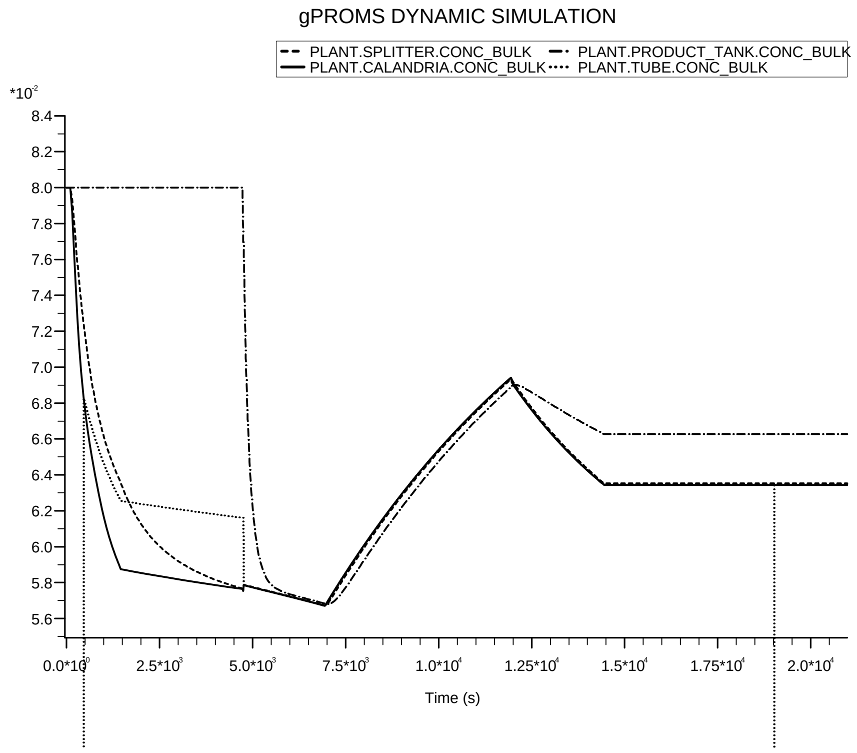

Figure 6.16: Bulk Solute Concentrations in the Evaporator Pilot Plant

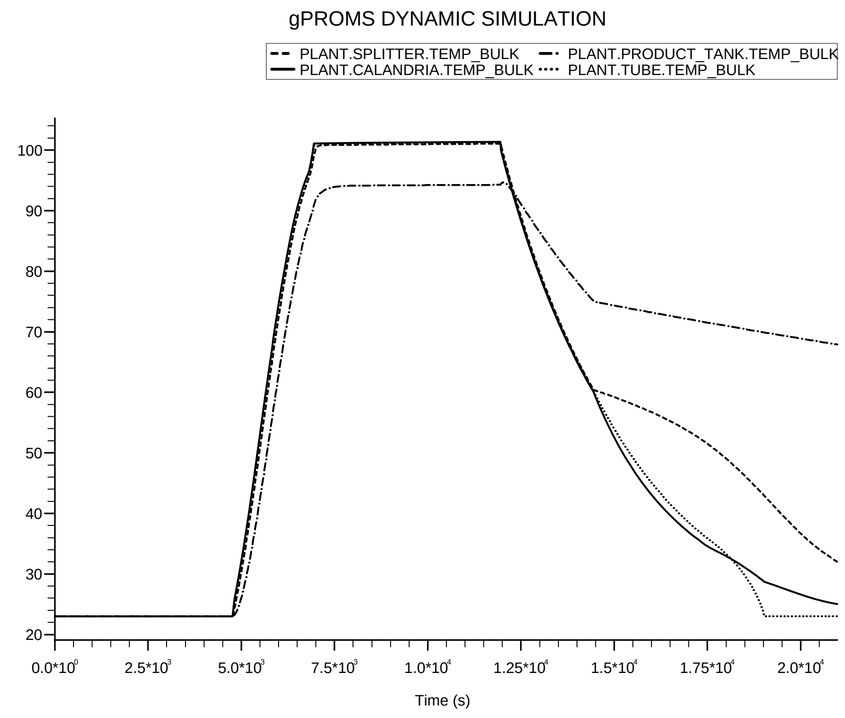

Figure 6.17: Bulk Temperatures in the Evaporator Pilot Plant

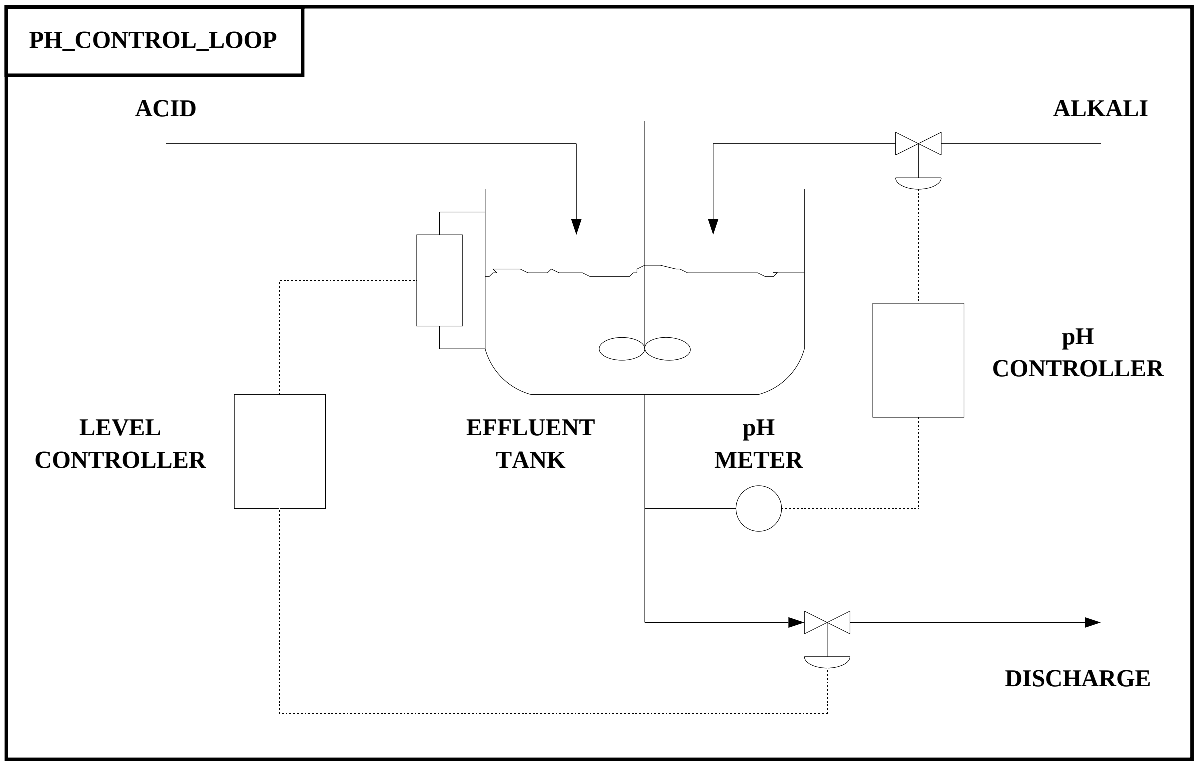

Figure 6.18: Effluent Tank Schematic

6.2.2 pH Control of an Effluent Tank

The first stage of a typical effluent treatment process is considered, in which the pH of a stream containing a solution of strong acid must be increased by the addition of a strong alkali before it is sent for further processing. The flowsheet shown in figure 6.18 is employed to achieve this objective. The pH of the discharge stream is regulated by adjusting the flow of alkali to the effluent tank, and the liquid level in the vessel is maintained by adjusting the discharge flowrate. A dynamic model of this process is potentially useful, for example, when determining if environmental standards are violated as a result of a range of process disturbances. However, this example is mainly concerned with a demonstration of the ability to express digital control laws using the primitive elements of the simulation language.

Because only disturbances around a steady-state are being considered, less detail in the continuous model of this process is required than in the previous example. Imperfect mixing in the effluent tank is modelled by a region of perfect mixing followed by a pure time delay, and the pH meter model takes into account the time delays and lags that characterise pH measurement. Full details are given by Smith (1991).

Again, having declared a continuous model for each unit operation, it is possible to develop task entities that describe typical operations involving these units. In this case, the task entity already shown in figure 3.11 is employed to model the action of a digital proportional integral control law. Note that this task is designed to take a measurement from an instance of a sensor model, and then update the stem position of an instance of a control valve model according to the control law. Through the wide range of parameter types introduced in chapter 3, it is possible to encapsulate a fairly general digital control law in a single task entity and then reuse this task in many different applications.

The schedule shown in figure 6.19 describes the disturbances and control actions that the effluent tank experiences during the simulation presented here. The digital controllers are modelled as tasks that operate concurrently with any disturbances for the duration of the simulation. Both digital controllers are represented by instances of the user defined task shown in figure 3.11, although each operates on a unique sensor and valve combination. From an initial condition in which the system is at steady-state, a step increase in the volumetric flow of acid to the tank is introduced after 150 seconds. When the system approaches steady-state again 2000 seconds later, this disturbance is reversed. Variable trajectories resulting from this particular simulation are shown in figure 6.20. Although the pH controller can cope with the original disturbance, it is unable to deal satisfactorily with reversal of the disturbance. The valve stem position trajectory shown in figure 6.21 focusses on the control system's attempts to recover from the reversal of the original disturbance. It is interesting to note that relatively small variations in the stem position (on a scale 0 to 1) are causing such wide fluctuations in pH.

Improved performance could be achieved through more sophisticated control schemes, involving for example measurement transformation and/or feed forward control. Indeed, using the features of the simulation language described in earlier chapters it should be possible to develop schedules that automate the application of tuning algorithms to the control system of an entire process, regardless of whether this control system is analogue or digital.

```
SCHEDULE
  PARALLEL
    SEQUENCE
      CONTINUE FOR 150
      RESET Plant.Tank.Vol_Flow_Acid := 0.030 ; END
      CONTINUE FOR 2000
      RESET Plant.Tank.Vol_Flow_Acid := 0.025 ; END
    END
    Effluent_Control_System(Plant                 IS Effluent_Plant,
                            Termination_Condition IS Time > 3000   ) ;
  END
```

Figure 6.19: Schedule Describing Simulation of Effluent Tank

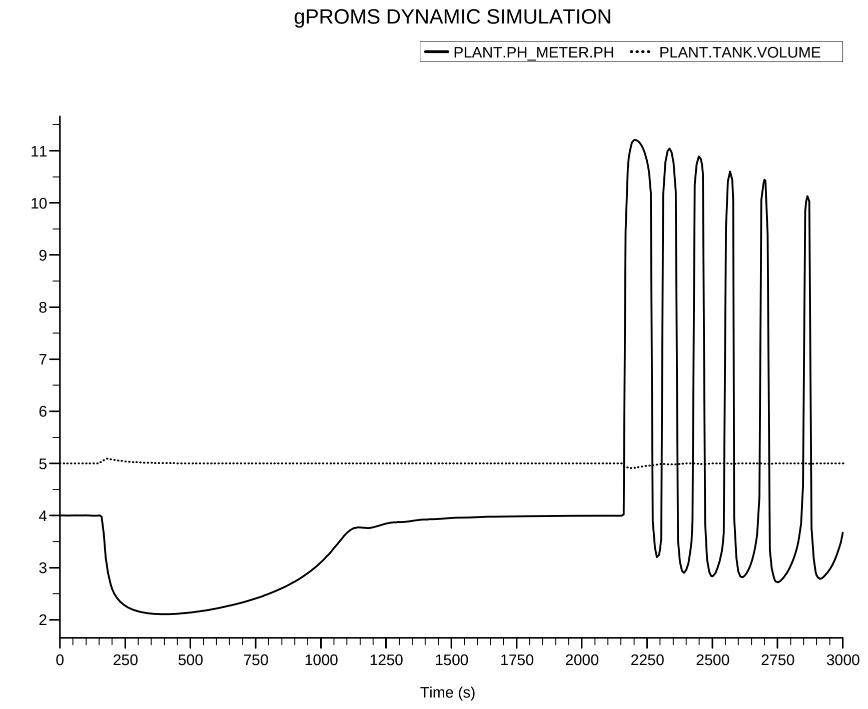

Figure 6.20: pH and Volume Trajectories for the Effluent Tank

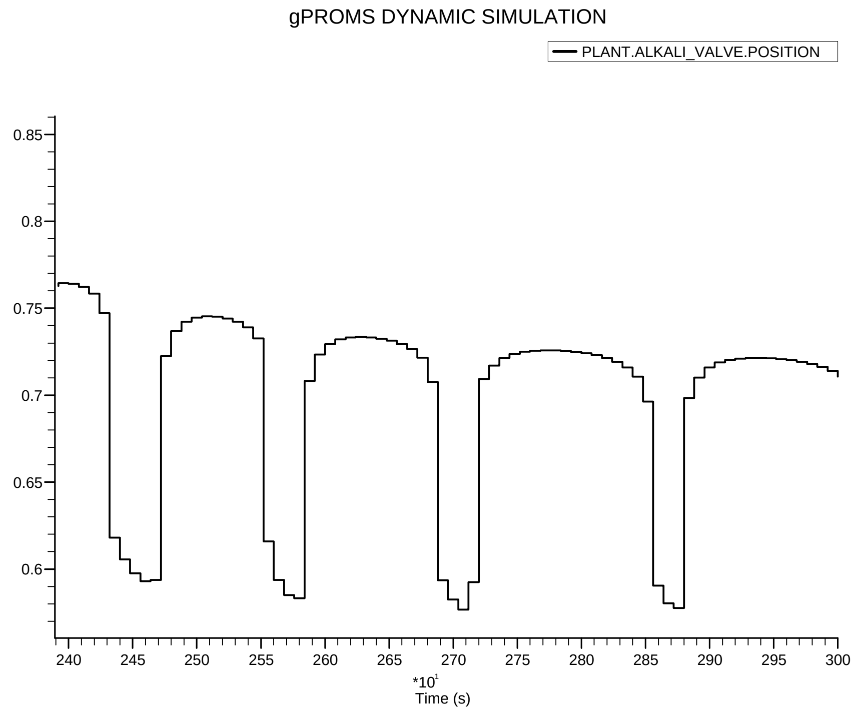

Figure 6.21: Stem Position Trajectory for the Alkali Valve

6.3 Periodic Processes

For a variety of reasons, an increasing number of unit operations, or even entire processes, are operated in a periodic (cyclic) manner. Examples include pressure and temperature swing adsorption, some modern effluent treatment processes, and multiple bed reactors in which the catalyst is periodically reactivated. The common feature of all these processes is the cyclic, and hence dynamic, nature of their operation. To assess the performance of these systems, the notion of 'cyclic steady-state' is introduced to indicate a point at which the trajectories of the process variables become virtually indistinguishable from one cycle to the next. Dynamic simulation is an invaluable tool for the analysis of processes in this category, such as pressure swing adsorption (Ward, 1990).

Periodic processes are by their nature combined discrete/continuous systems. On the one hand, the underlying continuous model is often as complex as those required for continuous processes, in many cases involving variables distributed in one or more spatial dimensions, and on the other, frequent control actions of considerable complexity are imposed in order to maintain the cyclic mode of operation.

Existing simulation tools for the analysis of periodic or cyclic processes have typically been developed with a particular technology in mind (e.g. ADSIM/SU (Prosys, 1989) for industrial adsorption processes). Although, in theory, the simulation packages specifically developed for batch and semi-continuous processes, BATCHES (Joglekar and Reklaitis, 1984) and UNIBATCH (Czulek, 1988), could also accommodate models of cyclic processes, in practice development of the detailed continuous models required would be difficult. On the other hand, the simulation language described in the preceding chapters is ideally suited to the modelling of periodic unit operations. Furthermore, such cyclic operations will often form part of a larger process that is operated predominantly in either the continuous or batch modes. A general-purpose combined simulation package, with its hierarchical model building capabilities, would greatly facilitate the analysis of the interactions of these cyclic unit operations with the rest of the process.

6.3.1 Pressure Swing Adsorption

The separation of gaseous components through selective adsorption on a solid phase is a well established technology. In recent years, renewed attention has been paid to the pressure swing adsorption (PSA) process, which employs a change in partial pressure, or *pressure swing*, to modify the quantity of gas adsorbed, and thereby achieve separation. This interest has been encouraged by increased understanding of the fundamental phenomena involved on the one hand, and the very tangible benefits accruing from the low energy consumption of the process on the other. Moreover, new demands on gas separation technology have promoted interest in the ability of PSA processes to handle low concentration feed stocks, or to produce high purity products.

In order to achieve separation, the sorbent columns that constitute a PSA process are repeatedly cycled between high and low pressures. This cyclic mode of operation makes PSA an ideal example for demonstrating the application of the prototype modelling package to periodic processes. The simulation presented here is based on a model reported in the literature that was utilised to establish the feasibility of employing PSA for simultaneous SO_2/NO_x removal and SO_2 recovery from flue gas (Kikkinides and Yang, 1991). The ease and speed with which this model was implemented within the framework of the prototype simulation package is a good illustration of the benefits that can be achieved through the use of general-purpose tools for combined discrete/continuous simulation.

The PSA process considered for flue gas desulphurisation employs the Skarstrom two column, four step configuration (Skarstrom, 1975). During a cycle, each column will pass through the following four elementary steps:

I. Repressurisation with feed.

II. Production with feed at high pressure.

III. Countercurrent blowdown.

IV. Countercurrent purge with product at low pressure.

Figure 6.22 illustrates how the two columns interact during a cycle. Initially, column A is at low pressure, and column B is at high pressure. The streams leaving the top of both columns are then shut off while column A is pressurised with feed, and column B is depressurised

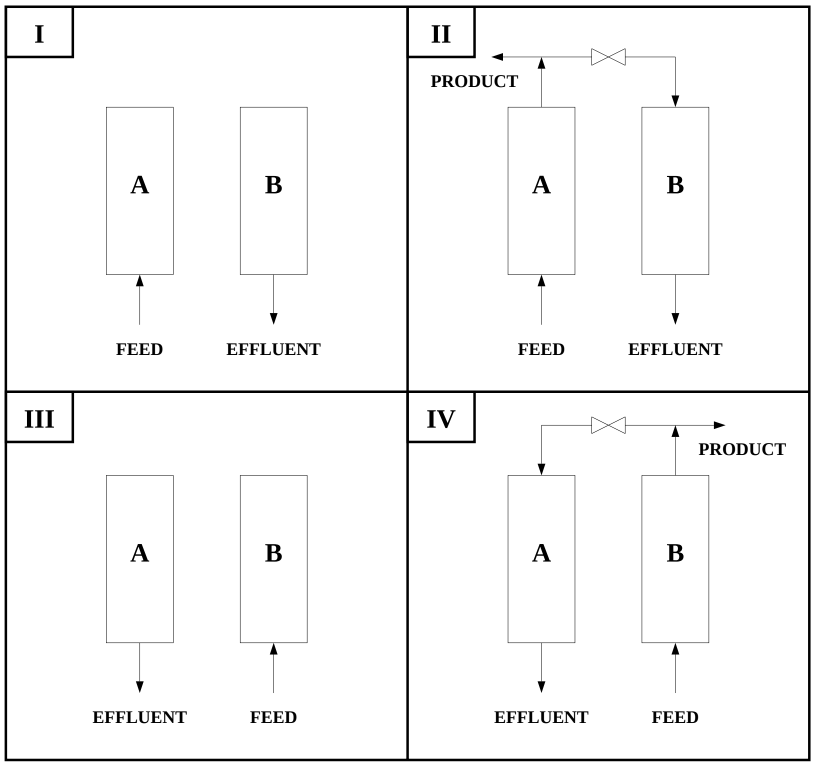

Figure 6.22: Skarstrom Two Column, Four Step Configuration

countercurrently. When column A has reached the required pressure, it begins to purify feed at high pressure while a small fraction of the product is diverted, via a pressure reduction valve, to column B for countercurrent purge at low pressure. The third step commences when column A has become saturated with the adsorbed components and SO_2 begins to 'break through' into the product stream. The streams leaving the top of both columns are again shut off and column A is depressurised countercurrently while column B is pressurised. Finally, during the fourth step a small fraction of the product from column B is used to purge countercurrently the adsorbed components from column A. At the end of this step, the sequence of operations is repeated. The original study was employed to determine if SO_2 enrichment in the effluent streams from this cycle was sufficient for elemental sulphur recovery by the Claus process.

The following assumptions (Kikkinides and Yang, 1991) were made in order to develop a model for the continuous time dependent behaviour of the adsorption columns:

- The thermal effects of adsorption and desorption are small, and isothermal operation can be assumed.

- The pressure drop in the bed is negligible, hence the pressure is uniform throughout the bed.
- Radial variations in velocity and concentration are negligible.
- Axial dispersion is negligible.
- The gas behaviour can be described by the perfect gas equation of state.

Table 6.3 contains a summary of the notation used in the equations that follow. Assuming that the Nth component is an inert (nitrogen) that is not adsorbed to any significant extent, the total mass balance can be written as:

$$\frac{\epsilon}{P}\frac{dP}{dt} + \frac{\partial u}{\partial z} + \frac{\rho_B RT}{P}\sum_{i=1}^{N-1}\frac{\partial q_i}{\partial t} = 0 \tag{6.1}$$

Similarly, the mass balance equations for each species can be written in terms of component mole fractions as:

$$\epsilon\frac{\partial y_i}{\partial t} + \frac{\epsilon y_i}{P}\frac{dP}{dt} + \frac{\partial (u y_i)}{\partial z} + \frac{\rho_B RT}{P}\frac{\partial q_i}{\partial t} = 0 \qquad \forall i = 1 \ldots N-1 \tag{6.2}$$

$$\sum_{i=1}^{N} y_i = 1 \tag{6.3}$$

Experiments on the sorbent considered in this study indicated that the equilibrium model, which assumes that the gaseous phase is in equilibrium with the adsorbed phase at all times, was justified. Hence, the concentration of each component in the solid phase is always equal to the equilibrium value, which is given by the loading ratio correlation (LRC):

$$q_i = q_i^* = \frac{q_{si} B_i P_i^{1/n_i}}{1 + \sum_{j=1}^{N-1} B_j P_j^{1/n_j}} \qquad \forall i = 1 \ldots N-1 \tag{6.4}$$

$$\text{where } P_j \equiv y_j P$$

The set of describing equations 6.1 to 6.4 therefore involves the following set of unknowns:

$$y_i(z,t) \quad \forall i = 1 \ldots N$$
$$q_i(z,t) \quad \forall i = 1 \ldots N-1$$
$$u(z,t)$$

B_i	Langmuir constant for component i, Pa$^{-1/n}$
L	Length of the bed, m
N	Number of components in feed stream
n_i	Exponent for component i in LRC
P	Bed pressure, Pa
P_L	Low pressure for cycle, Pa
P_H	High pressure for cycle, Pa
P_i	Partial pressure of component i, Pa
q_i	Moles adsorbed per gramme of solid, mol/g
q_i^*	Equilibrium moles adsorbed per gram of solid, mol/g
q_{si}	Saturated amount adsorbed, mol/g
R	Universal gas constant, Jmol^{-1}K^{-1}
T	Ambient temperature, K
t	Time, s
u	Superficial axial velocity, m/s
u_f	Feed velocity, m/s
u_{purge}	Purge velocity, m/s
y_i	Mole fraction of component i in gaseous phase
$y_{f,i}$	Mole fraction of component i in feed
$y_{p,i}$	Mole fraction of component i in purge
z	Axial position in bed, m
ϵ	Void fraction of bed
ρ_B	Density of the bed, kg/m^3

Table 6.3: Nomenclature for Model of Adsorption Column

where the temperature of the bed T is a given constant, and $P(t)$ is determined from a specification on dP/dt. In addition, appropriate boundary conditions for u and y_i are required.

Clearly, the differential variables, $y_i(z,t)$ and $q_i(z,t)$ $\forall i = 1 \ldots N-1$, cannot be assigned arbitrary initial values since they must also satisfy equation 6.4. It is therefore preferable to differentiate equation 6.4 with respect to time and use the resulting relationship to eliminate $\partial q_i/\partial t$ from equations 6.1 and 6.2. A full derivation of this differentiated form is given by Smith (1991).

Alternatively, the mass balance relations can be reformulated in terms of a control volume encompassing the two phases in equilibrium (Ponton and Gawthrop, 1991), thereby eliminating the variables $\partial q_i/\partial t$ from the model. Unfortunately, the specification of the bed pressure, combined with the assumption of negligible pressure drop in the bed, still increases the index of the resulting set of equations to two, and differentiation with respect to time in order to eliminate this problem re-introduces the variables $\partial q_i/\partial t$. However, replacement of the assumption of negligible pressure drop by a relationship such as the modified Sabri-Ergun equation (Ergun, 1952), relating the axial velocity at any point in the bed to the axial pressure drop, would result in a model of index one without consistent initialisation problems.

In order to employ the current implementation of the prototype modelling package to solve the equations detailed above, the partial differential equations that determine the column mass balances must be reduced to a set of ordinary differential equations with respect to time. This is accomplished, as in the original study, through a backward finite difference approximation based on a fixed spatial discretisation, although, considering the hyperbolic nature of the partial differential equations, an alternative approach such as the method of characteristics might prove to be more appropriate. A further complication arises from the reversals in flow direction that occur as each cycle progresses. While a column is undergoing steps I and II, gas will flow upwards through the column, whereas for steps III and IV this direction is reversed, and gas flows downwards through the column. To maintain stability, variant equations are utilised to ensure that the finite difference approximation is always backwards with respect to the direction of flow.

The simulation presented here is a recreation of the base case of the original study. The initial condition considers both beds to be initially clean, effectively only containing pure nitrogen:

$$\begin{array}{lrcll}
\text{at } t=0 \text{ and } 0<z<L, & y_i & = & 10^{-6} & \forall i = 1 \ldots N-1 \\
\text{at } t=0 \text{ and } 0<z<L, & P & = & P_L & \text{Column A} \\
\text{at } t=0 \text{ and } 0<z<L, & P & = & P_H & \text{Column B}
\end{array}$$

The operation of the overall process is determined by the boundary conditions for each step. Note that the two columns are always two steps out of phase with each other; column B will effectively be at the beginning of step III when column A is at the beginning of step I. Considering an individual column, the boundary conditions for step I, pressurisation with feed, will be:

$$\begin{aligned} \text{at } z = 0, \quad y_i &= y_{f,i} \qquad\qquad \forall i = 1 \ldots N-1 \\ \text{at } z = L, \quad u &= 0 \\ \tfrac{dP}{dt} &= +1.17 \times 10^5/30 \end{aligned}$$

similarly, during step II, high pressure adsorption:

$$\begin{aligned} \text{at } z = 0, \quad y_i &= y_{f,i} \quad \forall i = 1 \ldots N-1 \\ \text{at } z = 0, \quad u &= u_f \\ \tfrac{dP}{dt} &= 0 \end{aligned}$$

and step III, countercurrent blowdown:

$$\begin{aligned} \text{at } z = L, \quad y_i &= y_{p,i} \qquad\qquad \forall i = 1 \ldots N-1 \\ \text{at } z = L, \quad u &= 0 \\ \tfrac{dP}{dt} &= -1.17 \times 10^5/30 \end{aligned}$$

Note that during step IV, countercurrent purge, the composition of the gas entering the column is equal to the composition of the product from the other column:

$$\begin{aligned} \text{at } z = L, \quad y_i &= y_{p,i} \quad \forall i = 1 \ldots N-1 \\ \text{at } z = L, \quad u &= u_{purge} \\ \tfrac{dP}{dt} &= 0 \end{aligned}$$

It is important to recognise that the spatial position of the boundary conditions depends on the current step of column operation. This effectively means that, at the end of each step, input equations involving a subset of the system variables may have to be replaced by an equal number of new input equations involving a different subset of the describing variables. The REPLACE task can therefore be usefully employed to implement the dynamic changes to boundary conditions.

The description of the control action sequence that drives the two columns through a complete cycle is, in fact, extremely simple using the simulation language described in earlier chapters. REPLACE tasks are employed to implement the changes to the boundary conditions for both columns that occur at the end of each step, and CONTINUE FOR tasks are employed to declare the (fixed) duration of each step, during which integration of the continuous model of overall process is advanced. A nine minute cycle was used, with the following distribution: steps I and III, 0.5 minutes; steps II and IV, 4.0 minutes.

In order to simulate more than one cycle, the sequence of control actions that model a complete cycle are merely enclosed within a WHILE control structure. The conditions that determine the number of cycles (iterations) will be specific to an individual simulation. Termination could, for example, be determined from a criterion that measures the approach to cyclic steady-state, or, if a fixed number of cycles are of interest, a local integer variable incremented at the end of each cycle could be used as a counter. This former criterion is usually based on the variation of time averaged quantities from one cycle to the next, for example the time averaged mole fraction of a component in the product stream:

$$\overline{y_i} = \frac{\displaystyle\int_{t_{II}}^{t_{III}} u(L,t)y_i(L,t)dt}{\displaystyle\int_{t_{II}}^{t_{III}} u(L,t)dt} \tag{6.5}$$

where t_{II} indicates the start of step II, and t_{III} indicates the start of step III. The integrals can be evaluated through simple differential equations, although their values should be initialised to zero by a REINITIAL task at the start of step II of each cycle. At the end of step II, the average mole fractions $\overline{y_i}$ can be evaluated and stored in local variables (see section 3.3.4).

The simulation of four cycles under the base case conditions are shown in figures 6.23 and 6.24. As noted in the original study (Kikkinides and Yang, 1991), cyclic steady-state is reached after about four cycles. The trajectories clearly demonstrate the wide variation in gaseous phase compositions from one step to the next as a consequence of the control actions imposed.

The continuous model of this process could be made more sophisticated through the introduction of a flowsheet involving on/off valves that determine the direction and magnitude of flows in the process from the pressure drop across the valve and the valve stem position. This would have the consequence of further simplifying the schedule that describes a complete cycle; all dynamic changes to the boundary conditions could be implemented by reusable tasks that simply open and close these valves. In any case, this example is a very good demonstration of the powerful discrete manipulations elemental tasks may impose on the underlying continuous model. Moreover, the insertion of the PSA process in a model of the overall power generation process actually producing the flue gas would be a relatively easy task. Such a composite model could, for example, be employed to develop or validate modifications to the PSA operating procedure that would cope with upsets in an upstream

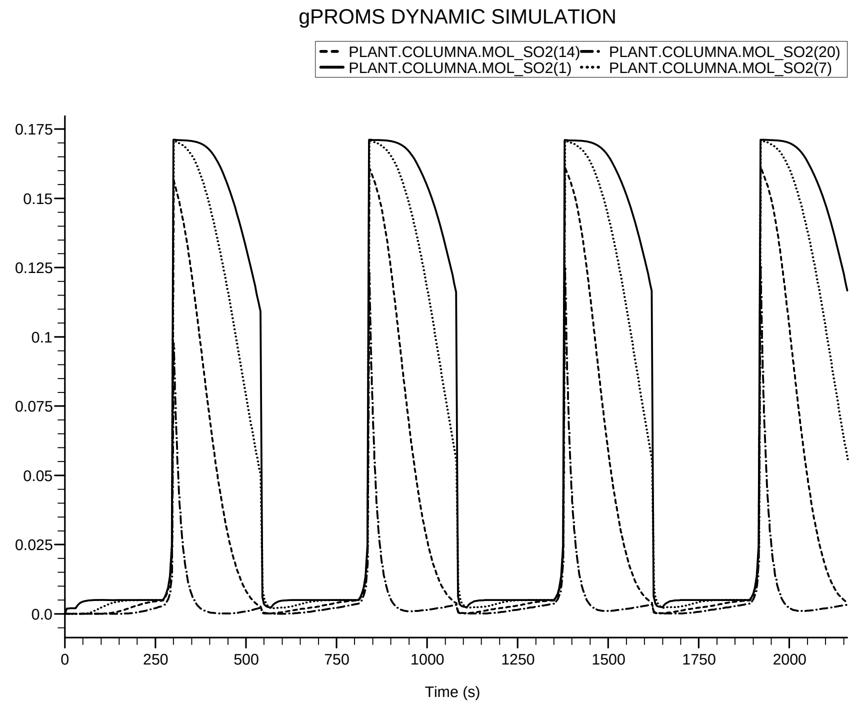

Figure 6.23: SO_2 Mole Fractions in the Gaseous Phase for Several Axial Positions

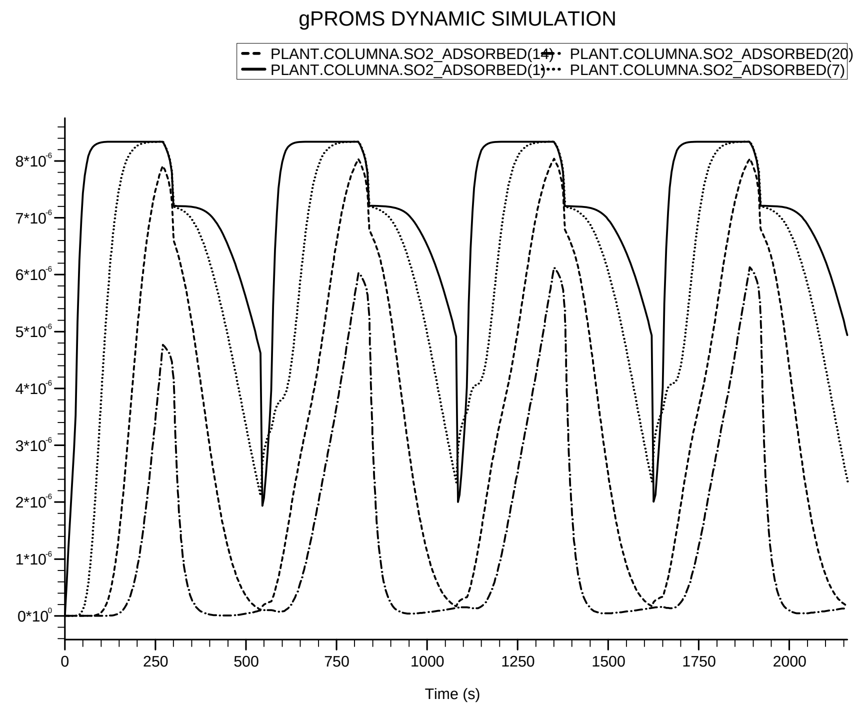

Figure 6.24: Moles of SO_2 Adsorbed per Gramme of Solid for Several Axial Positions

process.

6.4 Batch and Semi-Continuous Processes

The dynamic, often cyclic, mode in which batch and semi-continuous processes are operated dictates that dynamic simulation is the only practical and effective tool for the *detailed* analysis of this category of processes. As such, it complements analytically based preliminary design and production scheduling methodologies for batch and semi-continuous processes (Reklaitis, 1991). It can be employed, on the one hand, to validate the results produced by the simplified representations that must of necessity be utilised by these analytical methods, and, on the other, as the only recourse in circumstances in which the approximations adopted by available analytical techniques cannot be justified. Furthermore, the flexible, multiproduct or multipurpose nature of this mode of operation places special demands on any dynamic simulation tool.

A typical batch plant is composed of an inventory of process equipment connected together in a manner that promotes flexibility in the production routes employed. The equipment inventory may include storage vessels, reaction vessels, various separation unit operations, equipment to enable material transfers, and even packaging equipment. In addition, each item of equipment may be employed for several different operations during a production campaign. For example, a vessel fitted with a steam jacket and a mixing impeller may be utilised, at different points in time, for storage, blending, pre-heating, or even conversion of raw materials by reaction.

In general, batches of material enter the plant intermittently and are transferred from one item of equipment to another according to a prespecified production schedule. The features of this schedule usually dictate that only a subset of the total inventory of equipment is actually active at any point in time. Moreover, during an active phase, an item of equipment will be isolated from the rest of the plant, except when material is transferred to and from it through one of the built-in connections. Overall, this gives rise to the sporadic interaction of equipment that is otherwise operated independently.

The selection of the *material-oriented* approach discussed in chapter 3 by the developers of simulation packages for batch and semi-continuous processes, such as BATCHES

(Joglekar and Reklaitis, 1984) or UNIBATCH (Czulek, 1988), has been motivated by a combination of this intended role for dynamic simulation, and the characteristics of batch processes described above. This decision is especially justified in light of the realisation that the material-oriented approach is also that adopted by most batch plant design and scheduling methodologies, and the fact that the continuous models required for this type of study are relatively simple, particularly from the point of view of material transfers, in comparison to those routinely employed for the simulation of continuous processes. Moreover, if a simulation model concentrates on the batches of material moving through a plant, and regards processing equipment merely as a resource that is intermittently brought on and off line, dynamic changes to the process topology and the set of active equipment can be exploited to minimise the dimensionality of the continuous model solved at any point in time.

There are, however, many applications of dynamic simulation to batch and semi-continuous processes for which the *equipment-oriented* approach normally adopted for the simulation of continuous processes is probably more appropriate. These applications are typically those where a much more sophisticated continuous model of the process is required than those employed for the validation of production schedules, and include many also advocated for continuous processes. For example:

- Regulatory control system design and model based control.
- Computer based operator training.
- Safety and environmental studies.
- Detailed operating procedure synthesis and validation.
- Post commissioning experimentation.

Dynamic simulation is also an extremely important detailed design tool for complex batch unit operations such as batch distillation columns, where steady-state process analysis techniques obviously provide few insights.

Furthermore, an equipment-oriented approach is particularly suitable for applications in which the integrity of individual batches is not preserved throughout the process (e.g. where mixing or splitting of batches is allowed), and also those in which the same plant is composed of both batch and continuous sections.

Adoption of this latter approach dictates that the inventory of equipment in a batch process, and all the interconnecting piping and ancillary equipment, be modelled as a single flowsheet, in a manner identical to that of a continuous process. A schedule of task entities may then be applied to this continuous model of the overall process in order to emulate the control actions required by the batch mode of operation. Very detailed modelling of the batch process, particularly of the piping and ancillary equipment employed to achieve material transfers, is thereby possible. However, if a significant proportion of the process equipment and piping is inactive at any point in time, but is included in the continuous model as a consequence of this latter approach, the computational burden associated with the solution of the simulation model could be increased unnecessarily. On the other hand, if, for example, heat loss to the surroundings or side reactions significantly alter the state of any material left in an item of equipment during an inactive phase, it becomes unjustifiable to drop this item of equipment from the overall continuous model anyway.

This section is devoted to the demonstration of how the equipment-oriented approach to the modelling of batch and semi-continuous processes may be implemented by the simulation language presented in the preceding chapters.

6.4.1 Batch Reactor

The hypothetical batch plant shown in figure 6.25 is considered. The flowsheet consists of a batch reactor, upstream storage tanks for the reactants and a cleaning fluid, and downstream stream storage tanks for the product and the waste produced by each cleaning cycle.

The well-stirred batch reactor is employed to undertake the following consecutive homogeneous liquid phase reactions:

$$A + B \longrightarrow C$$

$$C + B \longrightarrow D$$

The desired product is component C, and the reactor must be operated in a manner that maximises its yield. Moreover, high temperatures favour the undesired reaction, so the operating policy must balance cycle length against achievable yield.

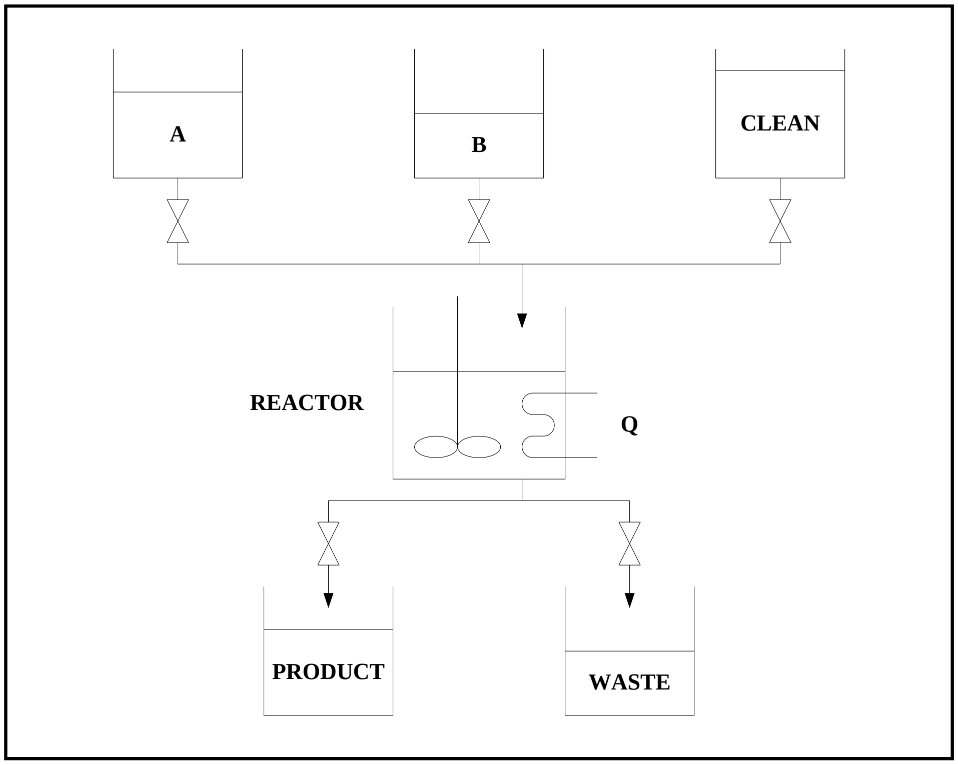

Figure 6.25: Hypothetical Batch Plant

A purely batch operation scheme is infeasible due to the highly exothermic nature of both reactions. Even mixing equal quantities of reactants A and B at ambient temperature would lead to the rapid release of large quantities of heat. A semi-continuous mode of operation is therefore necessary. As components A and C compete for reactant B, but only the reaction of A is desirable, component A must be provided in excess and component B supplied in a manner that limits the rate of reaction. Two possible modes of operation are described in more detail below.

6.4.1.1 Controlled Cooling Load

In this mode of operation, a constant flowrate of B is supplied to the reactor, and the cooling load is adjusted by a PI-controller in order to maintain a constant reactor temperature. A single cycle of operation will be composed of the following sequence of elementary processing steps:

1. Charge the reactor with A.

2. Set flow of B to a small constant value.
3. Wait until the vessel temperature has risen to the set point.
4. Increase the flow of B and close the temperature control loop.
5. Wait until the yield of C has reached a maximum.
6. Stop the flow of B.
7. Open the temperature control loop.
8. Wait until the temperature has dropped to 25^{o}C.
9. Empty the reactor to the product storage tank.

The time trajectories of the overall fractional yield, the conversion, and the selectivity resulting from a simulation of this cycle are shown in figure 6.26.

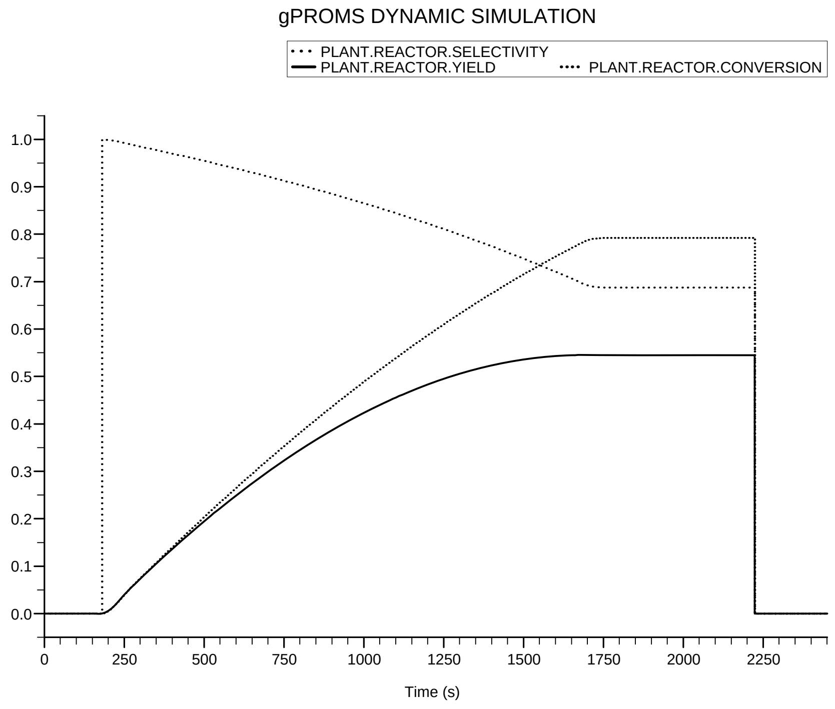

Figure 6.26: Yield, Conversion, and Selectivity for Controlled Cooling Load

6.4.1.2 Discrete Charges of Feed

An interesting mode of operation arises when the feed is added in a series of finite charges. This is the simplest mode because no regulatory temperature control is required. When a charge of B is supplied to the vessel, the concentration of B rises, leading to increased reaction rates and a temperature rise. The next charge is added when the temperature has dropped to a predefined value. A single cycle of operation will be composed of the following sequence of elementary processing steps:

1. Charge the reactor with A.
2. Set flow of B to a small constant value.
3. Wait until the vessel temperature has risen to the set point.
4. Turn on the cooling water and supply a charge of B.
5. Wait until the temperature has dropped to 42^{o}C.
6. Supply a charge of B.
7. Repeat the previous two steps until the desired conversion has been achieved.
8. Wait until the temperature has dropped to 25^{o}C.
9. Empty the reactor to the product storage tank.

The time trajectories of the overall fractional yield, the conversion, and the selectivity resulting from a simulation of this cycle are shown in figure 6.27.

6.4.1.3 Cyclic Operation of the Entire Plant

Finally, it is possible to build a model of the entire flowsheet shown in 6.25 in order to simulate several cycles of operation. In this case, it is assumed that a cleaning operation is necessary after each batch. After being emptied, the reactor is filled with cleaning fluid. When the concentration of the product compound has dropped below a specified value, the supply is stopped and the resulting fluid is pumped to the waste tank.

All the external actions required to model the reaction cycle and the cleaning can be expressed as a schedule of tasks which is then applied to the flowsheet. The trajectories of

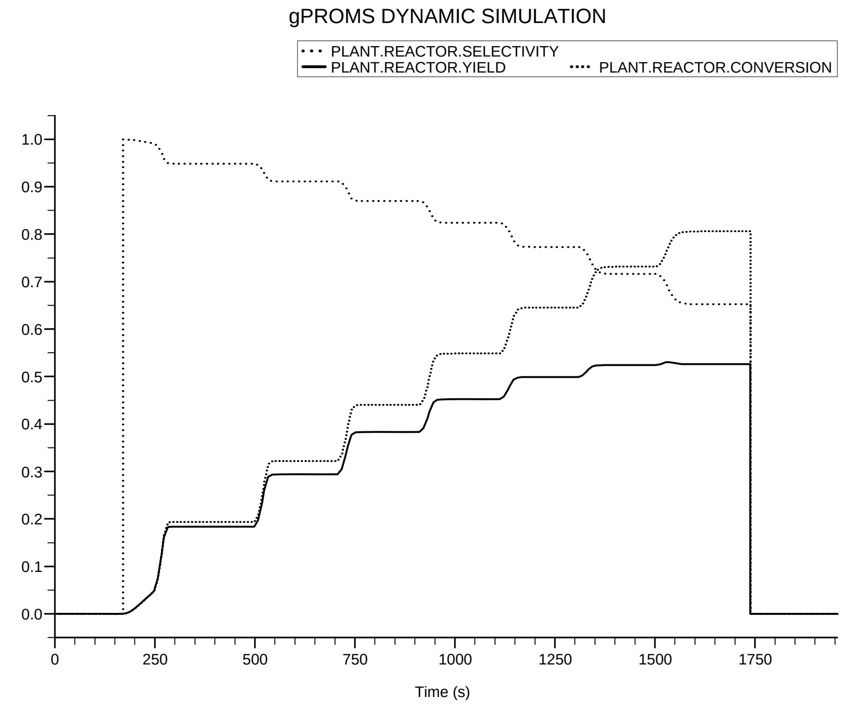

Figure 6.27: Yield, Conversion, and Selectivity for Discrete Charges of Feed

the molar holdups in the batch reactor over three cycles are shown in figure 6.28. Component 5 is the cleaning fluid.

This model could be augmented by the addition of some form of downstream separation unit that periodically recycles unreacted reactants back to the upstream storage vessels. It is intended to use an existing gPROMS model of a batch distillation column for this purpose (Jourda, 1992).

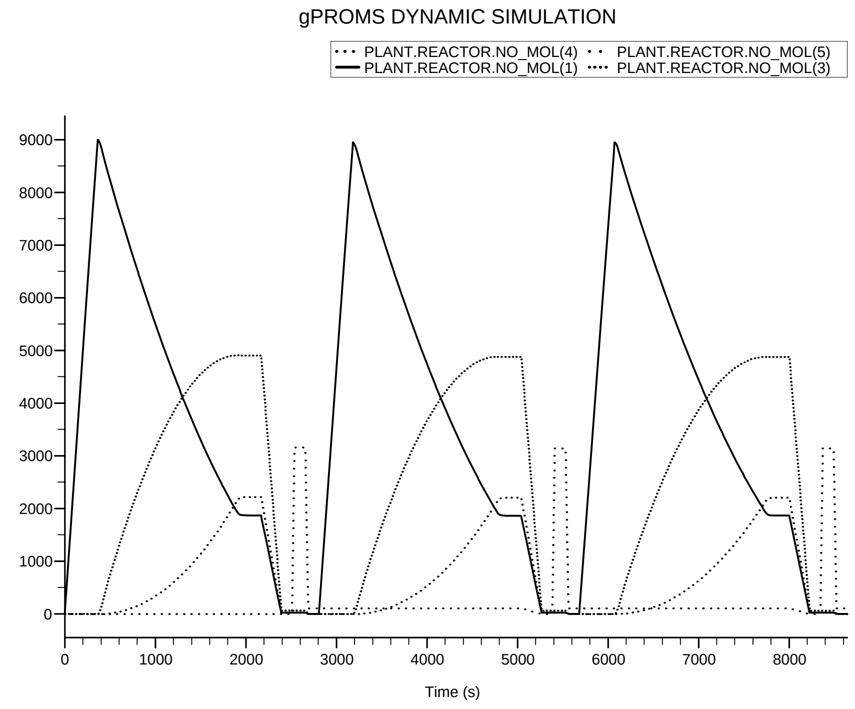

Figure 6.28: Molar Holdups in the Batch Reactor

Chapter 7

Conclusions and Recommendations

7.1 Conclusions

For a variety of reasons, currently available general-purpose software packages for the dynamic simulation of industrial processing systems are still not meeting many of the demands of potential industrial users. The fact that the engineer is still unable to pose an important class of problems in a cost-effective manner is an important contributor to this deficiency. It is our opinion that much of this difficulty has arisen from the failure to recognise that the analysis of the time dependent behaviour of the majority of industrial processing systems requires the solution of a combined discrete/continuous simulation problem as opposed to the continuous simulation problem solved by most of the above packages.

As recently argued by Marquardt (1991), future general-purpose simulation packages should support the analysis of arbitrarily operated processes in a unified framework. On the basis of the above discussion we believe that this new generation of packages should support combined discrete/continuous process simulation. The design and implementation of this new type of simulation package was therefore considered, with particular emphasis on the representational methodologies required to facilitate the specification of this class of problem.

No other process modelling package designed to encompass such a broad range of problems has yet been reported. However, the approach described in this thesis can be compared with that of the more modern general-purpose combined simulation languages, such as SYSMOD (Smart and Baker, 1984), and combined simulation packages specifically designed for the analysis of batch processes, such as BATCHES (Joglekar and Reklaitis, 1984).

The viewpoint conventionally adopted by the general-purpose combined simulation languages requires a system to be decomposed into a continuous and a discrete subsystem which are then allowed to interact as equals during the course of a simulation experiment.

Most industrial processing systems, however, belong to a special class of combined system that is composed of a fixed dimension continuous subsystem involved in the production of some material. The fundamental physical behaviour of this plant is frequently subject to discrete changes. It therefore seems more natural to decompose a processing system into the underlying combined discrete/continuous physical behaviour of the plant, and the external actions imposed on it by its environment under a particular set of circumstances.

In light of this alternative decomposition, it follows that the underlying physical behaviour should be considered to be subordinate to the external actions, as opposed to being equal. The external actions can then drive the plant through the desired changes by direct manipulation of its characteristics.

The material-oriented approach adopted by simulation packages such as BATCHES has also been compared at length with the equipment-oriented approach described above. In conclusion, it is argued that the latter approach is much more suitable for a package designed to encompass the entire range of processing system operation. However, it is also important to recognise that the material-oriented approach is probably more convenient for a limited class of applications to batch processes.

The facilities provided to describe the underlying physical behaviour of a processing system incorporate much of recent work reported in the literature, particularly relating to model complexity and model reuse, and enhance it further in certain cases. These enhancements are summarised at the end of chapter 2. Furthermore, the recognition that the initial condition of a set of DAEs can be expressed in the most general terms possible by the requisite number of nonlinear equations places even greater flexibility in the hands of the engineer.

The recognition that the external actions can be modelled in terms of discrete manipulations of the underlying continuous model has led to the development of a set of general-purpose primitives that provide considerable flexibility in the scope of these manipulations, albeit to a continuous model of fixed dimensionality. To the author's knowledge, facilities with such flexibility do not exist in any other simulation language. Moreover, this feature promotes the reusability of the continuous model through the elimination of any requirement to anticipate the external actions that may be imposed on it.

Furthermore, recent work relating to model complexity and model reuse has been complemented with effective mechanisms for managing the complexity and promoting the reuse of the declarations of external actions.

In addition, it has been demonstrated that a practical implementation of these ideas in the form of a prototype process modelling package is possible. This experience illustrates, on the one hand, the distinctive software architecture required by such a package, and on the other, the relative ease with which such a system may be developed within the scope of a single Ph.D project if an automated compiler construction tool and a modern programming language are employed.

Finally, it has been possible to employ this prototype to demonstrate the usefulness and necessity of the application of combined discrete/continuous simulation to the entire range of processing system operation, from purely continuous to batch.

The tangible product of the research work is therefore the first general-purpose combined discrete/continuous process simulation package suitable for application to the entire range of processing system operation.

7.2 Recommendations for Future Research

A key concept in gPROMS is that of the process entity combining a continuous model entity with a set of task entities describing external actions. Currently, each gPROMS simulation can involve only a single process entity. The idea of multiple interacting process entities as a useful extension has been touched upon in section 4.4 but requires further consideration and eventual implementation.

The complexity of any activity involving dynamic simulation makes the design of a more sophisticated user interface for the gPROMS package an issue of key importance. Recently, important contributions have been made by Stephanopoulos *et al.* (1987), Bar and Zeitz (1990), and Westerberg *et al.* (1991). Attention should be focused on three aspects of this interface:

- Graphical and textual problem construction tools to aid the correct formulation of process models through an understanding of the fundamental structuring concepts of the simulation language, and the exploitation of recent work relating to model specification based on physical mechanisms (see, for example, Stephanopoulos *et al.*

(1990a) and Vázquez-Román (1992)).

- Diagnostic and reporting tools that exploit the wealth of symbolic and structural information made available as a consequence of holding in memory detailed information concerning the continuous time dependent behaviour of a system throughout a simulation experiment (Piela, 1989).
- Tools for multiple problem management that facilitate exploitation of the hierarchical mechanisms employed to build simulation models.

A number of issues of both theoretical and practical interest can be identified as potential areas for longer term research. These are discussed briefly in the following text.

7.2.1 Modelling of Distributed Parameter Systems

In chapter 1, it was observed that a processing system that contains variables distributed in one or more spatial dimensions gives rise to a mathematical model composed of a mixed set of partial differential, ordinary differential and algebraic equations, or even a mixed set of partial integro-differential and algebraic equations. Within the context of currently available process modelling systems, this class of problems can only be handled by a fixed manual discretisation of the distributed variables and equations (see, for example, section 6.3.1).

An extension of the modelling language to encompass the declaration of both distributed and lumped parameter variables and equations within the same model entity would therefore seem to be extremely desirable. This would free the engineer to concentrate on the correct formulation of the process model, rather than becoming concerned with the spatial approximations required by the particular numerical method employed to solve the resulting set of equations. However, the user may still be required to provide some guidance as to the most appropriate numerical method(s) for a particular distributed parameter system.

Two further issues of considerable interest can be immediately identified: the specification of boundary and initial conditions in a completely general manner, and the use of discrete changes to the functional form of distributed equations and their boundary conditions in order to model phenomena such as physico-chemical discontinuities and control actions.

7.2.2 Dynamic Alteration to the Static Characteristics of Model Entities

Oren and Ziegler (1979) have partitioned the structure of a dynamic model into *static* and *dynamic* characteristics. The static characteristics of the model entities as presented in chapter 2 include:

- The component models that make up the overall model.
- The time invariant parameters.
- The set of variables which describe the time dependent behaviour of the system.

The dynamic characteristics refer to the relationships between static characteristics that determine the time dependent behaviour of the system, in this case the DAEs (or PDAEs).

Combined discrete/continuous simulation requires the ability to vary these characteristics with time. Language structures that facilitate the declaration of variation in the equation attributes of a model entity have already been discussed at length in chapters 2 and 3.

It is, however, also possible to perceive the need for language structures that facilitate variation of the *static* characteristics of a model entity as well. A limited form of this has already implemented by the UNDEFINED construct (see section 2.3.5.4), which essentially enables a subset of variables, and a corresponding number of equations, to be dropped from the overall system model during certain well defined periods. Furthermore, during the simulation of a batch process, the ability to alter the static characteristics of a model entity could ensure that only active items of process equipment are included in the continuous model of a system at any given point in time; a schedule of tasks could dynamically activate and deactivate component models of the overall system model as items of process equipment were brought on and off line.

7.2.3 Multipurpose Equipment

An industrial processing plant is essentially composed of an inventory of process equipment which may be engaged in the production of one or more materials. Continuous process simulation packages have conventionally merged an item of process equipment and the model that describes its continuous time dependent behaviour into a single entity. This view is consistent with the observation that the operations for which an item of equipment may be employed will not vary to a significant extent during the lifetime of a continuous plant. It is, however, inconsistent with the requirements of many combined discrete/continuous simulations, where the model employed to describe the continuous time dependent behaviour of an item of equipment may to a certain degree depend on the operation for which it is currently being utilised. For instance, in the context of multipurpose batch processes, an individual item of equipment may be employed for a broad range of operations during the progress of a production campaign, each of which might be described by a different continuous model. Moreover, recent work (Stephanopoulos *et al.*, 1990b) has pointed out that there is not even a unique mapping between a model and an item of equipment in a continuous process, because the model employed will usually evolve and expand as a design project progresses.

In recognition of this, it is argued that a certain class of problems will require an additional functional element in order to complete the simulation description – the inventory of *equipment entities* available. This inventory would represent a time invariant resource with which simulations of process operations may be performed. Obviously, an equipment entity will keep track of whether it is active or inactive, and will prevent any attempts to employ it for two different operations at the same time. An attempt to activate an already active equipment entity can either be interpreted as an abnormal termination condition, or a condition under which activation is delayed until the desired equipment entity becomes available. In this latter case, priorities could be established for situations in which operations compete for the same equipment entity.

This discussion also raises the issue of the suitability of an item of equipment for a particular operation. For example, the vessel fitted with a steam jacket and impeller described in section 6.4 may be suitable for all the operations listed there, but is patently unsuitable for a batch distillation operation. A model entity will dictate the continuous time dependent behaviour of an equipment entity during a particular operation, so some constraints on the range of model entities that may describe an operation should be imposed by the physical nature of the equipment. The careful development of model entities in inheritance hierarchies could again be exploited to establish these constraints. The declaration of any equipment entity would be accompanied by a specification of a template model entity, which dictates the broad class of model entities that may describe the operations in which the equipment is employed. Any operation applied to a equipment entity must then be described by a descendant of this template model.

A template model will usually be similar to the root model of an inheritance hierarchy (if it is not in fact one): it may not contain sufficient information for it to be actually used for simulation, but will be used to describe the basic characteristics of the continuous behaviour of the operations it is possible perform within the equipment entity. The declaration of an equipment entity could also include the assignment of numerical values to any of the parameter attributes of the template model, which would then be automatically passed to any operation that employs the equipment entity. These values would typically relate to physical characteristics of the vessel concerned, such as size or geometry, and will take precedence over values from any other source.

Finally, one side effect of the same item of equipment being used by a number of successive operations interspersed by periods of inactivity, is that the final state of each operation may be associated with the equipment entity and automatically transferred as the initial condition for the subsequent operation. This, for example, would enable the final molar holdup of material from a batch reaction operation to become the initial condition for the next operation performed in the same vessel, even if the two operations were separated by a significant period of inactivity. When two consecutive operations use different model entities to describe the behaviour of the same item of equipment, the information transferred would correspond to that declared in the closest common ancestor by inheritance of the two model entities.

7.2.4 Broadening the Range of Activities

A discussion in chapter 4 has already emphasised the broad range of activities for which a model formalism based on DAEs or PDAEs can be employed. The introduction of process entities to encapsulate the description of these additional activities is an obvious area of interest.

The extension of the modelling environment to encompass optimisation problems of both a steady-state and a dynamic nature is extremely interesting. An optimisation problem would be posed in terms of an objective function and constraints, rather than the fully determined set of constraints, initial condition, and quasi-explicit sequence of discrete changes required for dynamic simulation. A modelling language with which dynamic optimisation problems can be posed is considered particularly interesting – the concept of a schedule could, for example, be exploited to describe point constraints. Moreover, drastic changes to the static characteristics of a model may be required for mixed integer nonlinear optimisation.

Appendix A

The DECLARE Block

Basic variable and stream types are declared within a DECLARE block and are made available globally to all model entity declarations. This is quite similar to the DECLARE section of SpeedUp (Prosys, 1991). More than one DECLARE block may exist, the only restriction being that a basic type must be declared before it can be employed by a model entity declaration. DECLARE blocks containing declarations of commonly used types are suitable for storage in a library, which can then be imported and reused in new applications.

It may be desirable to include a DECLARE block within the declaration of a model entity. The basic types declared here would override or specialise the globally declared types, but would only be made available to the model entity concerned. A proper consideration of this issue lies beyond the scope of this document.

A DECLARE block is divided into a series of optional *sections*, outlined below. Each section contains declarations of basic types in a particular category.

1.1 The TYPE Section

Within the TYPE section a list of *variable types* may be declared. These are the basic types for the variables that describe the time dependent behaviour of a system, and are all refinements of the simple real type. A variable type declaration includes the following information:

- An identifier by which the type may be referred to globally.
- A default real value for instances of the type. This value will be used as the initial guess for any iterative calculation involving instances of the type, unless it is overridden by individual instances or a better guess is available from a previous calculation.
- Upper and lower bounds on the value of instances of the type. Any calculation involving instances of the type must give results that lie within these bounds. These bounds

```
DECLARE

  TYPE
    Flowrate    =   1.0 : 0 : 1E3   UNIT = "Kmol/s"
    Temperature = 298.0 : 0 : 1000  UNIT = "K"

  STREAM
    MainStream IS Flowrate, Temperature, Pressure, Enthalpy_Flow

END
```

Figure A.1: Simple DECLARE block.

ensure that the results of a calculation are physically meaningful; they can also be used to select the desired solution in situations where multiple solutions to a problem exist. Again, these bounds may be overridden by individual instances of the type.

- An optional units declaration. Whenever instances of the type are reported, the values will be accompanied by these units.

An example is the declaration of variable types for flowrate and temperature quantities shown in figure A.1. The variable type concept should be extended to include a declaration of the dimensionality of the quantities represented by instances of the type (in terms of the fundamental physical dimensions mass, length, time, temperature, charge, and moles). With this information, it is possible to automatically check the dimensional consistency of equations during the translation of a problem description (Piela, 1989).

1.2 The STREAM Section

Stream types provide a template for the stream attributes of a model entity. A stream type is essentially a record with a series of variable type fields, so a stream type declaration includes the following information:

- An identifier by which the type may be referred to globally.
- A list of variable types for the fields of the stream type. An instance of the stream type will contain a subset of the system variables with the listed types.

Note that, unlike for example SpeedUp (Prosys, 1991), a stream type declaration contains no information concerning the component mixtures transferred by the stream. A list of stream types may be declared within a STREAM section. The stream type declaration shown in figure A.1 might be used to represent a process stream.

The language definition includes one built-in variable type referred to by the identifier AnyType. The type conformance normally required for the subset of the variable attributes contained within a stream attribute is relaxed for fields of this type, which may therefore contain a variable attribute of any type. This relaxation is typically useful in situations where a stream attribute represents a measurement or control signal which must be matched with different physical quantities in different instances of a model.

Appendix B

Yacc Input File for the Current Implementation of gPROMS

```
%token Identifier    1 ILITERAL     2 RLITERAL     3 STRLITERAL  4
%token '^'           5 '+'          6 '-'          7 '*'          8 '/'    9
%token '$'          10 ';'         11 '='         12 '('         13 ')'   14
%token ':='         15 ':'         16 ','         17 '.'         18 '>'   19
%token '<'          20 '>='        21 '<='        22 '<>'        23 '['   24
%token ']'          25
%token AND          26 ARRAY       27 AS          28
%token ASSIGN       29 CASE        30 CONNECTION 31
%token CONTINUE     32 DECLARE     33 DEFAULT     34
%token DIV          35 DO          36 ELSE        37
%token END          38 EQUATION    39 FALSE       40
%token FOR          41 GLOBAL_TIME 42 IF          43
%token INHERITS     44 INITIAL     45 INPUT       46
%token INTEGER      47 IS          48 LOGICAL     49
%token MOD          50 MODEL       51 NOT         52
%token OF           53 OLD         54 OR          55
%token PARALLEL     56 PARAMETER   57 PRESET      58
%token PROCESS      59 REAL        60 REINITIAL   61
%token REPLACE      62 RESET       63 SCHEDULE    64
%token SELECTOR     65 SEQUENCE    66 SET         67
%token STEADY_STATE 68 STEP        69 STREAM      70
%token SWITCH       71 TASK        72 THEN        73
%token TIME         74 TO          75 TRUE        76
%token TYPE         77 UNIT        78 UNTIL       79
%token VARIABLE     80 WHEN        81 WHILE       82
%token WITH         83 WITHIN      84
%%
/*   GRAMMAR DEFINITION */

Simulation       : Block
                 | Simulation Block
                 ;

/* DEFINITION OF A BLOCK */

Block            : DeclareBlock
                 | ModelBlock
                 | TaskBlock
                 | ProcessBlock
                 ;

/*   SYNTAX FOR A DECLARE BLOCK */
```

```
DeclareBlock    : DECLARE VarTypeSection StrmTypeSection END ;

/*     SYNTAX FOR VARIABLE TYPE DECLARATIONS  */

VarTypeSection  : TYPE VarTypeDec
                | VarTypeSection VarTypeDec
                ;

VarTypeDec      : Identifier '=' Real ':' Real ':' Real OptionalUnits
                  { AddVarType($1,$3,$5,$7,$8) } ;

OptionalUnits   : UNIT '=' Units { AssignSemanticRecord($$,$3) }
                | Empty
                ;

Units           : STRLITERAL
                | Identifier
                ;

/* SYNTAX FOR STREAM TYPE DECLARATIONS */

StrmTypeSection : STREAM StrmTypeList
                | Empty
                ;

StrmTypeList    : StreamTypeDec
                | StrmTypeList StreamTypeDec
                ;

StreamTypeDec   : Identifier IS IdentifierList { AddStreamType($1,$3) } ;


/*      SYNTAX FOR A MODEL TYPE DECLARATION */

ModelBlock      : MODEL Identifier { CreateModel($2) } Parent
                  ParameterSection UnitSection  VariableSection
                  StreamSection SelectorSection SetSection
                  EquationSection  END
                  { TerminateModel($9,$10) }
                ;

Parent          : INHERITS Identifier { SetParent($2) }
                | Empty
                ;

/*      PARAMETER SECTION */

ParameterSection : PARAMETER ParameterList
                 |
                 ;
```

```
ParameterList   : ParameterDec
                | ParameterList ParameterDec
                ;

ParameterDec    : IdentifierList AS ParameterType
                  { ProcessEntity($1,$3) } ;

ParameterType   : BasicParameter
                | ARRAY '(' ExpList ')' OF BasicParameter
                  { CreateArray($3,$6,$$) }
                ;

BasicParameter  : BasicType DEFAULT SetValues
                  { CreatePmtr($1,Default,$3,$$)  }
                | BasicType
                  { CreatePmtr($1,NotAssigned,$1,$$) }
                ;

BasicType       : REAL
                | INTEGER
                | LOGICAL
                ;

/*          UNIT SECTION */

UnitSection     : UNIT UnitList
                | Empty
                ;

UnitList        : UnitDec
                | UnitList UnitDec
                ;

UnitDec         : IdentifierList AS UnitType { ProcessEntity($1,$3) } ;

UnitType        : BasicUnit
                | ARRAY '(' ExpList ')' OF BasicUnit
                  { CreateArray($3,$6,$$) }
                ;

BasicUnit       : Identifier { CreateUnit($1,$$) } ;


/*          SET  SECTION  */

SetSection      : SET  { BeginParaSetSection } SetList
                       { AssignSemanticRecord($$,$3) }
                | Empty
                ;

SetList         : SetDec  { StartAssignList($1,$$) }
```

```
                | SetList SetDec  { NextAssignment($1,$2,$$) }
                ;

SetDec          : PathName  ':='  SetValues ';'  { Assignment($1,$3,$$) }
                ;


/*          VARIABLE SECTION */

VariableSection : VARIABLE VariableList
                | Empty
                ;

VariableList    : VariableDec
                | VariableList VariableDec
                ;

VariableDec     : IdentifierList AS VariableType { ProcessEntity($1,$3) } ;

VariableType    : BasicVariable
                | ARRAY '(' ExpList ')' OF BasicVariable
                  { CreateArray($3,$6,$$) }
                ;

BasicVariable   : Identifier { CreateVar($1,$$) } ;

/*          STREAM SECTION */

StreamSection   : STREAM StreamList
                | Empty
                ;

StreamList      : StreamDec
                | StreamList StreamDec
                ;

StreamDec       : Identifier IS PathName { ProcessStreamMerge($1,$3) }
                | Identifier ':' PathNameList AS StreamType
                  { ProcessStream($1,$3,$5) }
                ;

StreamType      : BasicStream
                | ARRAY '(' ExpList ')' OF BasicStream
                  { CreateArray($3,$6,$$) }
                ;

BasicStream     : Identifier { CreateStream($1,$$)  }
                | CONNECTION { CreateConnection($$) }
                ;

/*          SELECTOR SECTION */
```

```
SelectorSection : SELECTOR SelectorList
                | Empty
                ;

SelectorList    : SelectorDec
                | SelectorList SelectorDec
                ;

SelectorDec     : IdentifierList AS SelectorType { ProcessEntity($1,$3) } ;

SelectorType    : BasicSelector
                | ARRAY '(' ExpList ')' OF BasicSelector
                  { CreateArray($3,$6,$$) }
                ;

BasicSelector   : '(' IdentifierList ')' { CreateSelector($2,FALSE,$1,$$) }
                | '(' IdentifierList ')' DEFAULT Identifier
                  { CreateSelector($2,TRUE,$5,$$) }
                ;

/*          EQUATION SECTION */

EquationSection : EQUATION EquationList { AssignSemanticRecord($$,$2) }
                | Empty
                ;

/*     SYNTAX FOR A TASK TYPE DECLARATION */

TaskBlock       : TASK Identifier { CreateTask($2) } TaskParaSection
                  ScheduleSection END { TerminateTask($5) } ;

/*       PARAMETER SECTION */

TaskParaSection : PARAMETER TaskParaList
                | Empty
                ;

TaskParaList    : TaskParameter
                | TaskParaList TaskParameter
                ;

TaskParameter   : IdentifierList AS TaskParamType
                  { ProcessEntity($1,$3) }
                ;

TaskParamType   : MODEL Identifier { CreateUnit($2,$$) } ;

/*       SCHEDULE SECTION */

ScheduleSection : SCHEDULE Schedule { AssignSemanticRecord($$,$2) } ;
```

```
/*     SYNTAX FOR A PROCESS TYPE DECLARATION */

ProcessBlock    : PROCESS Identifier { CreateProcess($2) }
                  ParameterSection IntUnit SetSection
                  EquationSection AssignSection PresetSection
                  IntSelector IntInitial IntSchedule END
                  { TerminateProcess($6,$7,$8,$9,$10,$12) }
                ;

/*       INT UNIT SECTION  */
IntUnit         : UNIT UnitList  ;

/*       ASSIGN SECTION */

AssignSection   : ASSIGN { BeginSetSection } AssignWithList
                  { AssignSemanticRecord($$,$3) }
                | Empty
                ;

/*       PRESET SECTION */

PresetSection   : PRESET PresetWithList { AssignSemanticRecord($$,$2) }
                | Empty
                ;

PresetWithList  : PresetWithin                { StartAssignList($1,$$)   }
                | PresetWithList PresetWithin { NextAssignment($1,$2,$$) }
                ;

PresetList      : Preset            { StartAssignList($1,$$)   }
                | PresetList Preset { NextAssignment($1,$2,$$) }
                ;

Preset          : Name ':=' Real ';' { PresetVariable($1,$3,$1,$1,$$) }
                | Name ':=' OptReal ':' OptReal ':' OptReal ';'
                  { PresetVariable($1,$3,$5,$7,$$) }
                | FOR Identifier ':=' SimpleExp TO SimpleExp OptStep DO
                  { InsertCounter($2) } PresetList END
                  { ForAssignment($2,$4,$6,$7,$10,$$) }
                | PresetWithin
                ;

PresetWithin    : WITHIN PathName { PushPathName($2,$$) } DO PresetList END
                  { WithinAssignment($3,$5,$$) }
                ;

OptReal         : Real
                | Empty
                ;
```

```
/*      SELECTOR SECTION */

IntSelector     : SELECTOR { BeginSelectorSection } AssignmentList
                  { AssignSemanticRecord($$,$3) }
                | Empty
                ;

/*      INITIAL SECTION */

IntInitial      : INITIAL EquationList { StoreInitials($2) }
                | INITIAL STEADY_STATE { SteadyState       }
                | Empty
                ;

/*      SCHEDULE SECTION */

IntSchedule     : ScheduleSection
                | Empty
                ;

/*  UTILITY SYNTAX CONSTRUCTS */

/*    A LIST OF IDENTIFIERS */

IdentifierList  : Identifier { StartIdList(FALSE,$1,$1,$$) }
                | IdentifierList ',' Identifier
                  { NextId($1,FALSE,$3,$3,$$)   }
                ;

/*    A LIST OF PATHNAMES */

PathNameList    : PathName                  { StartPathNameList($1,$$) }
                | PathNameList ',' PathName { NextPathName($1,$3,$$)   }
                ;

/*    A LIST OF EQUATIONS */

EquationList    : Equation              { StartEqtnList($1,$$) }
                | EquationList Equation { NextEqtn($1,$2,$$)   }
                ;

Equation        : SimpleEquation  ';' { SimpleEquation($1,$$) }
                | WithinEquation
                | ForEquation
                | IfEquation
                | CaseEquation
                ;

/*    STRUCTURED EQUATIONS */
```

```
WithinEquation  : WITHIN PathName { PushPathName($2,$$) } DO
                  EquationList END
                  { WithinEquation($3,$5,$$) }
                ;

ForEquation     : FOR Identifier ':=' SimpleExp TO SimpleExp OptStep DO
                  { InsertCounter($2) } EquationList END
                  { ForEquation($2,$4,$6,$7,$10,$$) }
                ;

OptStep         : STEP SimpleExp { AssignSemanticRecord($$,$2) }
                | Empty
                ;

IfEquation      : IF Expression THEN EquationList ELSE EquationList END
                  { IfEquation($2,$4,$6,$$) }
                ;

CaseEquation    : CASE PathName OF CaseList END
                  { CaseEquation($2,$4,$$) }
                ;

CaseList        : CaseClause          { StartClauseList($1,$$) }
                | CaseList CaseClause { NextClause($1,$2,$$)   }
                ;

CaseClause      : WHEN PathName ':' EquationList OptSwitchList
                  { CaseClause($2,$4,$5,$$) }
                ;

OptSwitchList   : SwitchList
                | Empty
                ;

SwitchList      : Switch            { StartEqtnList($1,$$) }
                | SwitchList Switch { NextEqtn($1,$2,$$)   }
                ;

Switch          : SWITCH TO PathName IF Expression ';'
                  { SwitchEquation($3,$5,$$) } ;

/*    A SIMPLE EQUATION */

SimpleEquation  : SimpleExp Equality SimpleExp
                  { StartExpList(Single,$1,$1,$2) ;
                    NextExp($2,Single,$3,$3,$$)   }
                | SimpleEquation Equality SimpleExp
                  { NextExp($1,Single,$3,$3,$$)   }
                ;

/*    A LIST OF ASSIGNMENTS */
```

```
AssignWithList  : AssignWithin                { StartAssignList($1,$$)   }
                | AssignWithList AssignWithin { NextAssignment($1,$2,$$) }
                ;

AssignmentList  : AssignEntity                { StartAssignList($1,$$)   }
                | AssignmentList AssignEntity { NextAssignment($1,$2,$$) }
                ;

AssignEntity    : PathName ':=' SetValues ';' { Assignment($1,$3,$$) }
                | AssignWithin
                | AssignFor
                ;

AssignWithin    : WITHIN PathName { PushPathName($2,$$) } DO
                  AssignmentList END
                  { WithinAssignment($3,$5,$$) }
                ;

AssignFor       : FOR Identifier ':=' SimpleExp TO SimpleExp OptStep DO
                  { InsertCounter($2) } AssignmentList END
                  { ForAssignment($2,$4,$6,$7,$10,$$) }
                ;

/*    A SCHEDULE OF TASKS */

ScheduleList    : Schedule              { StartScheduleList($1,$$) }
                | ScheduleList Schedule { NextSchedule($1,$2,$$)   }
                ;

Schedule        : TaskInstance
                | SequenceTask
                | ParallelTask
                | WhileTask
                | IfTask
                | ContinueTask
                | ResetTask
                | ReinitialTask
                | ReplaceTask
                ;

TaskInstance    : Identifier '(' ParaAssignList ')' ';'
                  { TaskInstance($1,$3,$$) }
                | Identifier ';'
                  { TaskInstance($1,$2,$$) }
                ;

SequenceTask    : SEQUENCE ScheduleList END { SequenceTask($2,$$) } ;

ParallelTask    : PARALLEL ScheduleList END { ParallelTask($2,$$) } ;
```

```
WhileTask       : WHILE Expression DO Schedule END
                  { WhileDoTask($2,$4,$$) }
                ;

IfTask          : IF Expression THEN Schedule END
                  { IfThenTask($2,$4,$1,$$) }
                | IF Expression THEN Schedule ELSE Schedule END
                  { IfThenTask($2,$4,$6,$$) }
                ;

ContinueTask    : CONTINUE FOR SimpleExp    { ContinueFor($3,$$)   }
                | CONTINUE UNTIL Expression { ContinueUntil($3,$$) }
                | CONTINUE FOR SimpleExp LogicalOp UNTIL Expression
                  { ContinueLogical($3,$4,$6,$$) }
                ;

ResetTask       : RESET { BeginResetTask } AssignmentList END
                  { ResetTask($3,$$) } ;

ReinitialTask   : REINITIAL PathNameList WITH { OLDOkay } EquationList END
                  { ReinitialTask($2,$5,$$) } ;

ReplaceTask     : REPLACE PathNameList WITH { BeginResetTask }
                  AssignmentList END { ReplaceTask($2,$5,$$) }
                ;

/*    LISTS OF PARAMETER VALUE ASSIGNMENTS */

ParaAssignList  : Identifier IS SetValues
                  { StartIdList(FALSE,$1,$3,$$) }
                | ParaAssignList ',' Identifier IS SetValues
                  { NextId($1,FALSE,$3,$5,$$)   }
                ;

/*    LISTS OF EXPRESSIONS */

SetValues       : '[' ExpList ']' { AssignSemanticRecord($$,$2)   }
                | Expression      { StartExpList(Single,$1,$1,$$) }
                ;

ExpList         : Expression
                  { StartExpList(Single,$1,$1,$$) }
                | Empty
                  { StartExpList(Blank,$1,$1,$$)  }
                | SimpleExp ':' SimpleExp
                  { StartExpList(Bounds,$1,$3,$$) }
                | ExpList ',' Expression
                  { NextExp($1,Single,$3,$3,$$)   }
                | ExpList ',' Empty
                  { NextExp($1,Blank,$3,$3,$$)    }
                | ExpList ',' SimpleExp ':' SimpleExp
```

```
                  { NextExp($1,Bounds,$3,$5,$$)   }
                ;

/*    AN EXPRESSION */

Expression      : Relation
                | Expression LogicalOp Relation { Reduce($2,$1,$3,$$) }
                ;

Relation        : SimpleExp
                | Relation RelationOp SimpleExp { Reduce($2,$1,$3,$$) }
                ;

SimpleExp       : UnaryTerm
                | SimpleExp AddOp Term { Reduce($2,$1,$3,$$) }
                ;

UnaryTerm       : UnaryOp Term { EvalUnary($1,$2,$$) }
                | Term
                ;

Term            : Factor
                | Term  MultOp  Factor { Reduce($2,$1,$3,$$) }
                ;

Factor          : Primary Power Primary { Reduce($2,$1,$3,$$) }
                | NOT Primary           { EvalUnary($1,$2,$$) }
                | Primary
                ;

Primary         : ILITERAL               { PushIntConst($1,$$)         }
                | RLITERAL               { PushRealConst($1,$$)        }
                | Logical                { PushLogConst($1,$$)         }
                | TIME                   { PushTime($$)                }
                | GLOBAL_TIME            { PushGlobalTime($$)          }
                | Name                   { PushVariable($1,$$)         }
                | '(' Expression ')'     { AssignSemanticRecord($$,$2) }
                | OLD '(' Expression ')' { PushOld($3,$$)              }
                ;

/* A PATHNAME */

Name            : PathName
                | '$' Identifier IdSuffix { StartIdList(TRUE,$2,$3,$$) }
                | PathName '.' '$' Identifier IdSuffix
                  { NextId($1,TRUE,$4,$5,$$) }
                ;

PathName        : Identifier IdSuffix
                  { StartIdList(FALSE,$1,$2,$$) }
                | PathName '.' Identifier IdSuffix
```

```
                 { NextId($1,FALSE,$3,$4,$$)   }
                ;

IdSuffix        : '(' ExpList ')' { AssignSemanticRecord($$,$2) }
                | Empty
                ;

/*  REAL, INTEGER OR BOOLEAN VALUES */

Real            : RealValue
                | UnaryOp RealValue { ProcessReal($1,$2,$$) }
                ;

Logical         : TRUE
                | FALSE
                ;

RealValue       : RLITERAL
                | ILITERAL
                ;

/*    OPERATOR DEFINITIONS */

LogicalOp       : AND  { PushRator($1,$$) }
                | OR   { PushRator($1,$$) }
                ;

RelationOp      : '='  { PushRator($1,$$) }
                | '<>' { PushRator($1,$$) }
                | '<'  { PushRator($1,$$) }
                | '<=' { PushRator($1,$$) }
                | '>'  { PushRator($1,$$) }
                | '>=' { PushRator($1,$$) }
                ;

AddOp           : '+'  { PushRator($1,$$) }
                | '-'  { PushRator($1,$$) }
                ;

MultOp          : '*'  { PushRator($1,$$) }
                | '/'  { PushRator($1,$$) }
                | DIV  { PushRator($1,$$) }
                | MOD  { PushRator($1,$$) }
                ;

Power           : '^' { PushRator($1,$$) } ;

UnaryOp         : '+'
                | '-'
                ;
```

```
Equality        : '='
                | IS
                ;

/*    THE EMPTY SET */

Empty   : { $$.Form := Empty } ;
```

Appendix C

gPROMS Input File for the Evaporator Pilot Plant

```
#
#==================================================================
#
# Model to simulate the Evaporator Pilot Plant at Imperial College
#
#==================================================================
#

DECLARE

TYPE
  Concentration      = 0.050 : -1E-9 :   1.001    UNIT = "kg/kg"
  Mass_Rate          =   1.0 :  -1E9 :   1E9      UNIT = "kg/sec"
  Temperature        =    25 :     0 :   200      UNIT = "degC"
  Length             =   0.1 : -1E-3 :   100      UNIT = "m"
  Enthalpy           =   100 :     0 :  5000      UNIT = "kJ/kg"
  Volume             =     5 : -1E-3 :    10      UNIT = "m3"
  Pressure           = 1.013 :   0.0 :    10      UNIT = "kg/cm2"
  Enthalpy_Flow      =   200 :  -1E5 :   1E5      UNIT = "kJ/sec"
  Area               =   0.1 :     0 :   100      UNIT = "m2"
  Mass_Holdup        =  12E3 : -1E-3 :  50E3      UNIT = "kg"
  Int_Energy         =   100 :  -1E7 :  10E7      UNIT = "kJ"
  Density            =  1000 :     0 :  2000      UNIT = "kg/m3"
  Fraction           =   0.5 :     0 :   1.0001
  Positive           =   0.5 : -1E-4 :   1E9
  NoType             =   1.0 :  -1E9 :   1E9

STREAM
  Mainstream_CT IS Mass_Rate, Concentration, Temperature
  Mainstream    IS Mass_Rate, Concentration, Temperature, Pressure
  Reflux_CT     IS Mass_Rate, Mass_Rate, Concentration, Temperature
  Reflux        IS Mass_Rate, Mass_Rate, Concentration, Temperature,
                   Pressure
  Vapour        IS Mass_Rate, Temperature
  Utility       IS Mass_Rate, Pressure

END

#
#===================================================================
#
# Model of a flow meter
#
```

```
#==================================================================
#

MODEL Flow_Meter

VARIABLE
  Flow                                              AS Mass_Rate
  Concentration                                     AS Concentration
  Temperature                                       AS Temperature
  Pressure                                          AS Pressure
  X1, X2                                            AS Pressure

STREAM
  Inlet  : Flow, Concentration,
           Temperature, Pressure                    AS Mainstream
  Output : Flow, Concentration,
           Temperature, Pressure                    AS Mainstream
  Signal : Flow                                     AS CONNECTION

END # Flow_Meter

#
#===================================================================
#
# Model of a control valve
#
#===================================================================
#

MODEL Control_Valve

VARIABLE
  Position                                           AS Fraction
  Flow                                               AS Mass_Rate
  Concentration                                      AS Concentration
  Press_In, Press_Out                                AS Pressure
  Temperature                                        AS Temperature
  Delta_P, Valve_Constant, Control_Action            AS Notype

STREAM
  Inlet  : Flow, Concentration,
           Temperature, Press_In                     AS Mainstream
  Output : Flow, Concentration,
           Temperature, Press_Out                    AS Mainstream
  Action : Control_Action                            AS CONNECTION

EQUATION

  # Clip signal from controller
  IF Control_Action > 1 THEN
    Position = 1 ;
  ELSE
    IF Control_Action < 0 THEN
      Position = 0 ;
    ELSE
      Position = Control_Action ;
    END
  END #if

  # Flowrate / Delta P relationship
  Flow = Position*Valve_Constant*SGN(Delta_P)*SQRT(ABS(Delta_P)) ;

  # Delta_P definition
  Delta_P = Press_In - Press_Out ;

END # Control_Valve

#
#=====================================================================
#
# Model of a proportional controller
#
#=====================================================================
#

MODEL Proportional_Integral_Controller

VARIABLE
  Setpoint, Measured_Variable, Gain                     AS Notype
  Integral_Error, Error, Reset_Time                     AS Notype
  Control_Action, Bias, Steady_Bias                     AS Notype

STREAM
  Action  : Control_Action                              AS CONNECTION
  Reading : Measured_Variable                           AS CONNECTION

EQUATION

  # Control_Action
  Control_Action = Bias + Gain*(Error + Integral_Error/Reset_Time) ;

  # Integral Error definition
  $Integral_Error = Error ;

  # Error definition
  Error = Setpoint - Measured_Variable ;

END # Proportional_Integral_Controller

#
#=====================================================================
#
```

```
# Model of a pump
#
#======================================================================
#

MODEL Pump

VARIABLE
  Pump_Status                                         AS Fraction
  Flow                                                AS Mass_Rate
  Press_In, Press_Out                                 AS Pressure
  Delta_P                                             AS Positive
  Temperature                                         AS Temperature
  Concentration                                       AS Concentration
  Parameter_1, Parameter_2                            AS NoType

STREAM
  Inlet  : Flow, Concentration,
           Temperature, Press_In                      AS Mainstream
  Output : Flow, Concentration,
           Temperature, Press_Out                     AS Mainstream

EQUATION

  # Pump Characteristic
  IF Pump_Status > 0.5 THEN
    Flow = Parameter_1 - Parameter_2*Delta_P^4 ;
  ELSE
    Delta_P = 0.0 ;
  END

  # Delta P definition
  Delta_P = Press_Out - Press_In ;

END # Pump

#
#=====================================================================
#
# Model of a flow control loop
#
#=====================================================================
#

MODEL Flow_Control

UNIT
  Valve                 AS Control_Valve
  Controller            AS Proportional_Integral_Controller
  Pump                  AS Pump
  Sensor                AS Flow_Meter
```

```
STREAM
  Inlet  IS Pump.Inlet
  Output IS Sensor.Output

EQUATION
  Pump.Output       IS Valve.Inlet        ;
  Valve.Output      IS Sensor.Inlet       ;
  Sensor.Signal     IS Controller.Reading ;
  Controller.Action IS Valve.Action       ;

END # Flow_Control

#
#=====================================================================
#
# Model to simulate a feed tank
#
#=====================================================================
#

MODEL Tank_Feed

PARAMETER
  Press_Atm                                        AS REAL

VARIABLE
  Flow_Bottom, Flow_Refill                         AS Mass_Rate
  Conc_Bulk, Conc_Refill, Conc_Bottom              AS Concentration
  Enth_Bulk, Enth_Refill, Enth_Bottom              AS Enthalpy
  Temp_Bulk, Temp_Refill, Temp_Bottom              AS Temperature
  Area                                             AS Area
  Liquid_Height                                    AS Length
  Heat_Loss                                        AS Enthalpy_Flow
  Holdup, Total_Holdup                             AS Mass_Holdup
  Total_U_Holdup                                   AS Int_Energy
  Density                                          AS Density
  Press                                            AS Pressure

STREAM
  # Inlet streams
  Inlet_Refill : Flow_Refill,
                 Conc_Refill, Temp_Refill          AS Mainstream_CT
  Bottom       : Flow_Bottom,
                 Conc_Bottom, Temp_Bottom, Press    AS Mainstream

EQUATION

  # Hydrostatic pressure
  Press = Press_Atm + 9.81*Density*Liquid_Height/1E5 ;
  # Material Balances
  $Total_Holdup = Flow_Refill - Flow_Bottom ;
  $Holdup = Flow_Refill*Conc_Refill - Flow_Bottom*Conc_Bottom ;

  # Conc Bulk definition
  Conc_Bulk*Total_Holdup = Holdup ;

  # Total_Holdup / Height relationship
  Total_Holdup = Area*Liquid_Height*Density ;

  # Energy Balance
  $Total_U_Holdup = Flow_Refill*Enth_Refill
                  - Flow_Bottom*Enth_Bottom - Heat_Loss ;

  # Total U Holdup definition
  Total_U_Holdup = Total_Holdup*Enth_Bulk ; # pv term ?????????

  # PHYSICAL PROPERTIES

  # Specific enthalpies
  Enth_Bulk = 0.654 + 4.188*Temp_Bulk
            + (368.3 - 4.26*Temp_Bulk)*Conc_Bulk ;
  Enth_Bottom = 0.654 + 4.188*Temp_Bottom
              + (368.3 - 4.26*Temp_Bottom)*Conc_Bottom ;
  Enth_Refill = 0.654 + 4.188*Temp_Refill
               + (368.3 - 4.26*Temp_Refill)*Conc_Refill ;

  # Density
  Density = (681.7 - 1.549*Temp_Bulk + 0.00778*Temp_Bulk^2)*Conc_Bulk
          + 1000.66 - 0.09748*Temp_Bulk - 0.003313*Temp_Bulk^2 ;

END

#
#====================================================================
#
# Model to simulate the product tank - with built in level control
#
#====================================================================
#

MODEL Tank_Product

VARIABLE
  Flow_In, Flow_Out                                AS Mass_Rate
  Temp_In, Temp_Bulk, Temp_Ambient                 AS Temperature
  Enth_In, Enth_Bulk                               AS Enthalpy
  Conc_In, Conc_Bulk                               AS Concentration
  Area, HT_Area                                    AS Area
  Height_Tank, Setpoint                            AS Length
  Heat_Loss                                        AS Enthalpy_Flow
  Density                                          AS Density
  Cv, Pump_Product, HT_Coeff                       AS Notype
  Holdup, Total_Holdup                             AS Mass_Holdup
  Total_U_Holdup                                   AS Int_Energy
  Position, Action, Gain, Error, Bias              AS Notype

STREAM
  # Inlet stream
  Inlet        : Flow_In, Conc_In, Temp_In         AS Mainstream_CT
  # Output stream
  Output       : Flow_Out, Conc_Bulk, Temp_Bulk    AS Mainstream_CT

EQUATION

  # Material Balances
  $Total_Holdup = Flow_In - Flow_Out ;
  $Holdup = Flow_In*Conc_In - Flow_Out*Conc_Bulk ;

  # Conc bulk definition
  Holdup = Conc_Bulk*Total_Holdup ;

  # Total Holdup / Level relationship
  Total_Holdup = Density*Height_Tank*Area ;

  # Flow Out control
  IF Pump_Product >= 1.0 THEN
    Flow_Out = Cv*Position ;
  ELSE
    Flow_Out = 0.0 ;
  END # if

  # Energy Balance
  $Total_U_Holdup = Flow_In*Enth_In -
                              Flow_Out*Enth_Bulk - Heat_Loss ;

  # Total U Holdup definition
  Total_U_Holdup = Total_Holdup*Enth_Bulk ;

  # Heat Transfer coefficient
  HT_Coeff = 0.0443*ABS(Temp_Bulk - Temp_Ambient) + 9.577 ;

  # Heat loss
  Heat_Loss = HT_Coeff*HT_Area*(Temp_Bulk - Temp_Ambient)/1000 ;

  # Define error
  Error = Height_Tank - Setpoint ;

  # Calculate control action
  Action = Bias + Gain*Error ;

  # Limit maximum control action
  IF Action >= 1 THEN
    Position = 1.0 ;
  ELSE
    IF Action >= 0.0 THEN
      Position = Action ;
    ELSE
      Position = 0.0 ;
    END # if
  END # if

  # PHYSICAL PROPERTIES

  # Specific enthalpies
  Enth_In = 0.654 + 4.188*Temp_In
          + (368.3 - 4.26*Temp_In)*Conc_In ;
  Enth_Bulk = 0.654 + 4.188*Temp_Bulk
            + (368.3 - 4.26*Temp_Bulk)*Conc_Bulk ;

  # Density
  Density = 1000.66 - 0.09748*Temp_Bulk - 0.003313*Temp_Bulk^2 +
       (681.7 - 1.549*Temp_Bulk + 0.00778*Temp_Bulk^2)*Conc_Bulk ;

END

#
#=================================================================
#
# Model to simulate the calandria
#
#=================================================================
#

MODEL Calandria_D

PARAMETER
  Press_Atm                              AS REAL

VARIABLE
  OverFlow_Flow,
  Flow_Steam, Flow_Below, Flow_Vapour    AS Mass_Rate
  Liquid_Volume                          AS Volume
  Liquid_Height, Height_Liquid_Max       AS Length
  Total_Holdup, Solute_Holdup            AS Mass_Holdup
  Feed_Density,
  Bulk_Density, OverFlow_Density         AS Density
  Press_Top, Press_Below                 AS Pressure
  Temp_Bulk, Temp_Below, Temp_OverFlow   AS Temperature
  Conc_Bulk, Conc_Below, Conc_OverFlow   AS Concentration
  Enth_Below, Enth_Bulk, Enth_OverFlow   AS Enthalpy
  Total_U_Holdup                         AS Int_Energy
  UA                                     AS NoType
  Delta_T, Temp_Steam, Temp_Boil         AS Temperature
  Enth_Vapour                            AS Enthalpy
  Heat, Heat_Loss                        AS Enthalpy_Flow
  Temp_Ambient                           AS Temperature
  HT_Coeff                               AS Notype
  HT_Area                                AS Area

STREAM
  Below    : Flow_Below, Conc_Below,
             Temp_Below, Press_Below     AS Mainstream
  OverFlow : OverFlow_Flow,
             Conc_OverFlow,
             Temp_OverFlow, Press_Top    AS MainStream
  Vapour   : Flow_Vapour                 AS CONNECTION

SELECTOR
  Flow_Flag                              AS (Full,Not_Full)

EQUATION

  # Total Mass Balance
  $Total_Holdup = Flow_Below - Flow_Vapour - OverFlow_Flow ;

  # Solute balance
  $Solute_Holdup = Flow_Below*Conc_Below
                 - OverFlow_Flow*Conc_OverFlow ;

  # Energy balance
  $Total_U_Holdup = Flow_Below*Enth_Below
                  - OverFlow_Flow*Enth_OverFlow
                  - Flow_Vapour*Enth_Vapour
                  + Heat - Heat_Loss ;

  # Total_Holdup / Volume relationship
  Total_Holdup = Liquid_Volume*Bulk_Density ;

  # Volume / Height relationship
  Liquid_Volume = Liquid_Height*35.47E-4 ;

  # Hydrostatic pressre head
  Press_Below = Press_Top + Bulk_Density*Liquid_Height*9.81/1E5 ;

  # Determine overflow out the top. To avoid a problem of high index,
  # a controller with a very high gain is put on the overflow.
  CASE Flow_Flag OF
    WHEN Not_Full : OverFlow_Flow = 0 ;
                    SWITCH TO Full
                    IF Liquid_Height > Height_Liquid_Max ;
    WHEN Full     : OverFlow_Flow =
                      1E5*(Liquid_Height - Height_Liquid_Max) ;
                    SWITCH TO Not_Full
                    IF Press_Top <= Press_Atm ;
  END # case

  # Vapour flow. Controller to avoid a problem of high index.
  IF Temp_Bulk > Temp_Boil THEN
  Flow_Vapour = 1E5*(Temp_Bulk - Temp_Boil) ;
  ELSE
  Flow_Vapour = 0.0 ;
  END

  # Heat Transfer coefficient
  HT_Coeff = 0.0443*ABS(Temp_Bulk - Temp_Ambient) + 9.577 ;

  # Heat loss
  Heat_Loss = HT_Coeff*HT_Area*(Temp_Bulk - Temp_Ambient)/1000 ;

  # Boiling point temperature
  Temp_Boil = 20.16*Conc_Bulk + 99.975 ;

  # Bulk concentration and bulk temperature definitions
  IF Total_Holdup > 0 THEN
  Solute_Holdup = Conc_Bulk*Total_Holdup ;
  Total_U_Holdup = Total_Holdup*Enth_Bulk ;
  ELSE
  Conc_Bulk = 0.0 ;
  Temp_Bulk = Temp_Ambient ;
  END

  # Delta T definition
  Delta_T = ABS(Temp_Steam - (Temp_Bulk + Temp_Below)/2.0) ;

  # UA coeff definition
  IF Delta_T > 40 THEN
  UA = 0.7266 - 0.006*Delta_T ;
  ELSE
  IF Delta_T >= 29.5 THEN
    UA = 0.5 ;
  ELSE
    IF Delta_T >= 22.71 THEN
      UA = 3.427 - 0.098*Delta_T ;
    ELSE
      UA = 1.2 ;
    END # if
  END # if
  END # if

  # Heat from steam conditional equations
  IF Flow_Steam <= 0 THEN
  Heat = 0.0 ;
  ELSE
  IF (Temp_Steam <= Temp_Bulk) OR (Total_Holdup <= 0) THEN
```

```
      Heat = 0.0 ;
    ELSE
      Heat = UA*Delta_T ;
    END # if
  END # if

  # PHYSICAL PROPERTIES

  # Densities
  Feed_Density =
       (681.7 - 1.549*Temp_Below + 0.00778*Temp_Below^2)*Conc_Below
            + 1000.66 - 0.09748*Temp_Below - 0.003313*Temp_Below^2 ;
  Bulk_Density =
       (681.7 - 1.549*Temp_Bulk + 0.00778*Temp_Bulk^2)*Conc_Bulk
            + 1000.66 - 0.09748*Temp_Bulk - 0.003313*Temp_Bulk^2 ;
  OverFlow_Density = 1000.66 + Conc_OverFlow*(681.7 -
            1.549*Temp_OverFlow + 0.00778*Temp_OverFlow^2)
                - 0.09748*Temp_OverFlow - 0.003313*Temp_OverFlow^2 ;

  # Specific Enthalpies
  Enth_Bulk     = 0.654 + 4.188*Temp_Bulk
                  + (368.3 - 4.26*Temp_Bulk)*Conc_Bulk ;
  Enth_Below    = 0.654 + 4.188*Temp_Below
                  + (368.3 - 4.26*Temp_Below)*Conc_Below ;
  Enth_Vapour   = 2.008*Temp_Bulk + 2475.4 ;
  Enth_OverFlow = 0.654 + 4.188*Temp_OverFlow
                  + (368.3 - 4.26*Temp_OverFlow)*Conc_OverFlow ;

END

#
#=================================================================
#
# Model of a vertical tube
#
#=================================================================
#

MODEL Vertical_Tube

PARAMETER
  Press_Atm                                 AS REAL

VARIABLE
  Feed_Flow, Flow_Vapour,OverFlow_Flow      AS Mass_Rate
  Liquid_Volume                             AS Volume
  Liquid_Height                             AS Length
  Total_Holdup, Solute_Holdup               AS Mass_Holdup
  Feed_Density, Bulk_Density                AS Density
  Press_Bottom                              AS Pressure
  Temp_Feed, Temp_Bulk                      AS Temperature
  Conc_Feed, Conc_Bulk                     AS Concentration
  Enth_Feed, Enth_Bulk                     AS Enthalpy
  Total_U_Holdup                           AS Int_Energy
  Heat_Loss                                AS Enthalpy_Flow
  HT_Coeff                                 AS NoType
  HT_Area                                  AS Area
  Temp_Ambient                             AS Temperature
  Height_Liquid_Max                        AS Length

STREAM
  Feed       : Feed_Flow, Conc_Feed,
               Temp_Feed, Press_Bottom     AS MainStream
  Vapour_In  : Flow_Vapour                 AS CONNECTION
  OverFlow   : OverFlow_Flow, Flow_Vapour,
               Conc_Bulk, Temp_Bulk        AS Reflux_CT

EQUATION

  # Overall Mass Balance
  $Total_Holdup = Feed_Flow - OverFlow_Flow ;

  # Total Holdup / Volume relationship
  Total_Holdup = Liquid_Volume*Bulk_Density ;

  # Volume / Height relationship
  Liquid_Volume = Liquid_Height*28.27E-4 ;

  # Hydrostatic pressure head
  Press_Bottom = Press_Atm + Bulk_Density*Liquid_Height*9.81/1E5 ;

  IF Liquid_Height > Height_Liquid_Max THEN
    OverFlow_Flow = 1E3*(Liquid_Height - Height_Liquid_Max) ;
  ELSE
    OverFlow_Flow = 0 ;
  END

  # Solute balance
  $Solute_Holdup = Feed_Flow*Conc_Feed - OverFlow_Flow*Conc_Bulk ;

  # Energy Balance
  $Total_U_Holdup = Feed_Flow*Enth_Feed -
                           OverFlow_Flow*Enth_Bulk - Heat_Loss ;

  # Bulk concentration and bulk temperature definitions
  IF Total_Holdup > 0 THEN
    Solute_Holdup = Conc_Bulk*Total_Holdup ;
    Total_U_Holdup = Total_Holdup*Enth_Bulk ;
  ELSE
    Conc_Bulk = 0.0 ;
    Temp_Bulk = Temp_Ambient ;
  END
  # Heat Transfer coefficient
  HT_Coeff = 0.0443*ABS(Temp_Bulk - Temp_Ambient) + 9.577 ;

  # Heat loss
  Heat_Loss = HT_Coeff*HT_Area*(Temp_Bulk - Temp_Ambient)/1000 ;

  # PHYSICAL PROPERTIES

  # Densities
  Feed_Density =
        (681.7 - 1.549*Temp_Feed + 0.00778*Temp_Feed^2)*Conc_Feed
              + 1000.66 - 0.09748*Temp_Feed - 0.003313*Temp_Feed^2 ;
  Bulk_Density =
        (681.7 - 1.549*Temp_Bulk + 0.00778*Temp_Bulk^2)*Conc_Bulk
              + 1000.66 - 0.09748*Temp_Bulk - 0.003313*Temp_Bulk^2 ;

  # Specific Enthalpies
  Enth_Bulk = 0.654 + 4.188*Temp_Bulk
            + (368.3 - 4.26*Temp_Bulk)*Conc_Bulk ;
  Enth_Feed = 0.654 + 4.188*Temp_Feed
            + (368.3 - 4.26*Temp_Feed)*Conc_Feed ;

END

#
#=================================================================
#
# Model of a splitter
#
#=================================================================
#

MODEL Split

PARAMETER
  Press_Atm                                     AS REAL
  Pipe_Height                                   AS REAL

VARIABLE
  Flow_Vapour,
  Flow_Tube, Flow_Junct, Product_Flow           AS Mass_Rate
  Liquid_Volume                                 AS Volume
  Liquid_Height, Height_Liquid_Max              AS Length
  Total_Holdup, Solute_Holdup                   AS Mass_Holdup
  Feed_Density, Density_Tube, Bulk_Density      AS Density
  Press_Bottom                                  AS Pressure
  Temp_Junct, Temp_Bulk, Temp_Tube              AS Temperature
  Conc_Junct, Conc_Bulk, Conc_Tube              AS Concentration
  Enth_Junct, Enth_Bulk, Enth_Tube              AS Enthalpy
  Total_U_Holdup                                AS Int_Energy
  Heat_Loss                                       AS Enthalpy_Flow
  HT_Coeff                                        AS Notype
  HT_Area                                         AS Area
  Temp_Ambient                                    AS Temperature

STREAM
  # Inlet streams
  Inlet_Tube     : Flow_Tube, Flow_Vapour,
                   Conc_Tube, Temp_Tube       AS Reflux_CT
  # Output streams
  Output_Product : Product_Flow,
                   Conc_Bulk, Temp_Bulk       AS MainStream_CT
  Output_Vapour  : Flow_Vapour, Temp_Bulk     AS Vapour
  Junction       : Flow_Junct, Conc_Junct,
                   Temp_Junct, Press_Bottom   AS Mainstream

SET
  Pipe_Height := 3.5 ;

EQUATION

  # Overall Mass Balance
  $Total_Holdup = Flow_Tube + Flow_Junct - Product_Flow ;

  # Total Holdup / Volume relationship
  Total_Holdup = Liquid_Volume*Bulk_Density ;

  # Liquid_Height / Volume relationship
  IF Liquid_Height > Pipe_Height THEN
    Liquid_Volume = 0.007854*Pipe_Height +
                        0.1217*(Liquid_Height - Pipe_Height) ;
  ELSE
    Liquid_Volume = 0.007854*Liquid_Height ;
  END # if

  # Hydrostatic pressure head
  Press_Bottom = Press_Atm + Bulk_Density*Liquid_Height*9.81/1E5 ;

  IF Liquid_Height >= Height_Liquid_Max THEN
    Product_Flow = 1E3*(Liquid_Height - Height_Liquid_Max) ;
  ELSE
    Product_Flow = 0 ;
  END

  # Solute balance
  $Solute_Holdup = Flow_Junct*Conc_Junct
                   + Flow_Tube*Conc_Tube
                   - Conc_Bulk*Product_Flow ;

  # Energy Balance
  $Total_U_Holdup = Flow_Junct*Enth_Junct + Flow_Tube*Enth_Tube
```

```
                              - Enth_Bulk*Product_Flow - Heat_Loss ;

  # Bulk concentration and bulk temperature definitions
  IF Total_Holdup > 0 THEN
    Solute_Holdup = Conc_Bulk*Total_Holdup ;
    Total_U_Holdup = Total_Holdup*Enth_Bulk ;
  ELSE
    Conc_Bulk = 0.0 ;
    Temp_Bulk = Temp_Ambient ;
  END

  # Heat Transfer coefficient
  HT_Coeff = 0.0443*ABS(Temp_Bulk - Temp_Ambient) + 9.577 ;

  # Heat loss
  Heat_Loss = HT_Coeff*HT_Area*(Temp_Bulk - Temp_Ambient)/1000 ;

  # PHYSICAL PROPERTIES

  # Densities
  Feed_Density =
       (681.7 - 1.549*Temp_Junct + 0.00778*Temp_Junct^2)*Conc_Junct
            + 1000.66 - 0.09748*Temp_Junct - 0.003313*Temp_Junct^2 ;
  Bulk_Density =
       (681.7 - 1.549*Temp_Bulk + 0.00778*Temp_Bulk^2)*Conc_Bulk
            + 1000.66 - 0.09748*Temp_Bulk - 0.003313*Temp_Bulk^2 ;
  Density_Tube =
       (681.7 - 1.549*Temp_Tube + 0.00778*Temp_Tube^2)*Conc_Tube
            + 1000.66 - 0.09748*Temp_Tube - 0.003313*Temp_Tube^2 ;

  # Specific Enthalpies
  Enth_Bulk = 0.654 + 4.188*Temp_Bulk
            + (368.3 - 4.26*Temp_Bulk)*Conc_Bulk ;
  Enth_Tube = 0.654 + 4.188*Temp_Tube
            + (368.3 - 4.26*Temp_Tube)*Conc_Tube ;
  Enth_Junct = 0.654 + 4.188*Temp_Junct
             + (368.3 - 4.26*Temp_Junct)*Conc_Junct ;

END

#
#=====================================================================
#
# Model of the mixing junction
#
#=====================================================================
#

MODEL Mixer

VARIABLE
```

```
  Flow_Feed, Flow_Cala                              AS Mass_Rate
  Flow_Split                                        AS Mass_Rate
  Flow_Recycle_Set, Flow_Feed_Set                   AS Mass_Rate
  Temp_Bulk, Temp_Ambient                           AS Temperature
  Temp_Cala, Temp_Split, Temp_Feed                  AS Temperature
  Enth_Bulk, Enth_Feed, Enth_Split, Enth_Cala       AS Enthalpy
  Conc_Bulk, Conc_Feed, Conc_Cala, Conc_Split       AS Concentration
  Press_Cala, Press_Split, Press_Mixer, Press_CM    AS Pressure
  Feed_Pump, Recycle_Pump, Manual_Valve             AS Notype
  Density_Feed, Density_Bulk, Density_Cala          AS Density
  Density_Split                                     AS Density
  Total_Holdup, Solute_Holdup                       AS Mass_Holdup
  Total_U_Holdup                                    AS Int_Energy
  Heat_Loss                                         AS Enthalpy_Flow
  HT_Coeff                                          AS Notype
  HT_Area                                           AS Area
  Volume                                            AS Volume

STREAM
  # Inlet streams
  Feed   : Flow_Feed, Conc_Feed,
           Temp_Feed, Press_Mixer                   AS Mainstream
  Caland : Flow_Cala, Conc_Cala,
           Temp_Cala, Press_Cala                    AS Mainstream
  Split  : Flow_Split, Conc_Split,
           Temp_Split, Press_Split                  AS Mainstream

EQUATION

  # Solute mass balance
  $Solute_Holdup = Flow_Feed*Conc_Feed - Flow_Cala*Conc_Cala
                 - Flow_Split*Conc_Split ;

  # Total Mass Balance
  Total_Holdup = Density_Bulk*Volume ;

  # Conc Bulk definition
  Conc_Bulk*Total_Holdup = Solute_Holdup ;

  # Energy Balance
  $Total_U_Holdup = Flow_Feed*Enth_Feed - Flow_Cala*Enth_Cala
                  - Flow_Split*Enth_Split ;

  # Total U Holdup definition
  Total_U_Holdup = Total_Holdup*Enth_Bulk ;

  # Overall volume flow balance
  Flow_Feed/Density_Feed =
          Flow_Cala/Density_Cala + Flow_Split/Density_Split ;

  # Flow / Pressure relationships !!!
  Press_mixer = Press_CM = Press_Split ;

  # Press CM definition
  Press_CM = Press_Cala + 0.6*Density_Cala*9.81/1E5 ;

  # Heat Transfer coefficient
  HT_Coeff = 0.0443*ABS(Temp_Bulk - Temp_Ambient) + 9.577 ;

  # Heat loss
  Heat_Loss = HT_Coeff*HT_Area*(Temp_Bulk - Temp_Ambient)/1000 ;

  # PHYSICAL PROPERTIES - later to be replaced by procedures

  # Specific enthalpies
  Enth_Feed     = 0.654 + 4.188*Temp_Feed
                + (368.3 - 4.26*Temp_Feed)*Conc_Feed ;
  Enth_Split    = 0.654 + 4.188*Temp_Split
                + (368.3 - 4.26*Temp_Split)*Conc_Split ;
  Enth_Cala     = 0.654 + 4.188*Temp_Cala
                + (368.3 - 4.26*Temp_Cala)*Conc_Cala ;
  Enth_Bulk     = 0.654 + 4.188*Temp_Bulk
                + (368.3 - 4.26*Temp_Bulk)*Conc_Bulk ;

  # Density
  Density_Feed  =
       (681.7 - 1.549*Temp_Feed + 0.00778*Temp_Feed^2)*Conc_Feed
             + 1000.66 - 0.09748*Temp_Feed - 0.003313*Temp_Feed^2 ;
  Density_Split =
   1000.66 - 0.09748*Temp_Split - 0.003313*Temp_Split^2
    + (681.7 - 1.549*Temp_Split + 0.00778*Temp_Split^2)*Conc_Split ;
  Density_Cala  =
      (681.7 - 1.549*Temp_Cala + 0.00778*Temp_Cala^2)*Conc_Cala
             + 1000.66 - 0.09748*Temp_Cala - 0.003313*Temp_Cala^2 ;
  Density_Bulk  =
      (681.7 - 1.549*Temp_Bulk + 0.00778*Temp_Bulk^2)*Conc_Bulk
             + 1000.66 - 0.09748*Temp_Bulk - 0.003313*Temp_Bulk^2 ;

END

#
#====================================================================
#
# Model of the entire plant
#
#====================================================================
#

MODEL Complete_Plant_Flowsheet

PARAMETER
  Press_Atm                              AS REAL
```

```
UNIT
  Tube                                  AS Vertical_Tube
  Feed_Tank                             AS Tank_Feed
  Product_Tank                          AS Tank_Product
  Junction                              AS Mixer
  Calandria                             AS Calandria_D
  Splitter                              AS Split
  Feed_Control, Recycle_Control         AS Flow_Control

SELECTOR
  Cala_Junct_Flag, Cala_Tube_Flag,
  Split_Junct_Flag, Feed_Junct_Flag     AS (Positive, Negative)

SET
  Press_Atm := 1.01325 ;

EQUATION

  # Intensive properties for Calandria overflow.
  CASE Cala_Junct_Flag OF
    WHEN Positive : WITHIN Junction DO
                      Conc_Cala = Conc_Bulk ;
                      Temp_Cala = Temp_Bulk ;
                    END # within
                    SWITCH TO Negative
                    IF Junction.Flow_Cala <= -1E-4 ;
    WHEN Negative : WITHIN Calandria DO
                      Conc_Below = Conc_Bulk ;
                      Temp_Below = Temp_Bulk ;
                    END # within
                    SWITCH TO Positive
                    IF Junction.Flow_Cala >= 1E-4 ;
  END # case

  CASE Cala_Tube_Flag OF
    WHEN Positive : WITHIN Calandria DO
                      Conc_OverFlow = Conc_Bulk ;
                      Temp_OverFlow = Temp_Bulk ;
                    END # within
                    SWITCH TO Negative
                    IF Calandria.OverFlow_Flow <= -1E-4 ;
    WHEN Negative  : WITHIN Tube DO
                      Conc_Feed = Conc_Bulk ;
                      Temp_Feed = Temp_Bulk ;
                    END # within
                    SWITCH TO Positive
                    IF Calandria.OverFlow_Flow >= 1E-4 ;
  END # case

  CASE Split_Junct_Flag OF
```

```
    WHEN Positive : WITHIN Junction DO
                      Conc_Split = Conc_Bulk ;
                      Temp_Split = Temp_Bulk ;
                    END # within
                    SWITCH TO Negative
                    IF Junction.Flow_Split <= -1E-4 ;
    WHEN Negative : WITHIN Splitter DO
                      Conc_Junct = Conc_Bulk ;
                      Temp_Junct = Temp_Bulk ;
                    END # within
                    SWITCH TO Positive
                    IF Junction.Flow_Split >= 1E-4 ;
  END # case

  CASE Feed_Junct_Flag OF
    WHEN Positive : WITHIN Feed_Tank DO
                      Conc_Bottom = Conc_Bulk ;
                      Temp_Bottom = Temp_Bulk ;
                    END # within
                    SWITCH TO Negative
                    IF Junction.Flow_Feed <= -1E-4 ;
    WHEN Negative : WITHIN Junction DO
                      Conc_Feed = Conc_Bulk ;
                      Temp_Feed = Temp_Bulk ;
                    END # within
                    SWITCH TO Positive
                    IF Junction.Flow_Feed >= 1E-4 ;
  END # case

  # Streams
  Feed_Tank.Bottom       IS Feed_Control.Inlet      ;
  Feed_Control.Output    IS Junction.Feed           ;
  Junction.Caland        IS Recycle_Control.Inlet   ;
  Recycle_Control.Output IS Calandria.Below         ;
  Tube.Feed              IS Calandria.Overflow      ;
  Tube.Vapour_In         IS Calandria.Vapour        ;
  Splitter.Inlet_Tube    IS Tube.OverFlow           ;
  Splitter.Junction      IS Junction.Split          ;
  Product_Tank.Inlet     IS Splitter.Output_Product ;

END

#
#=====================================================================
#
# Task to start a pump
#
#=====================================================================
#

TASK Start_Pump
```

```
PARAMETER
  Pump                        AS MODEL Pump

SCHEDULE
  RESET Pump.Pump_Status := 1.0 ; END
END # Start_Pump

#
#=====================================================================
#
# Task to stop a pump
#
#=====================================================================
#

TASK Stop_Pump

PARAMETER
  Pump                        AS MODEL Pump

SCHEDULE
  RESET Pump.Pump_Status := 0.0 ; END
END # Stop_Pump

#
#=====================================================================
#
# Task to close a control loop
#
#=====================================================================
#

TASK Close_Loop

PARAMETER
  Controller             AS MODEL Proportional_Integral_Controller

SCHEDULE
  PARALLEL
    # closes control loop
    REPLACE
      Controller.Control_Action
    WITH
      Controller.Bias := Controller.Steady_Bias ;
    END
    # reinitializes integral error
    REINITIAL
      Controller.Integral_Error
    WITH
      Controller.Integral_Error = 0.0 ;
    END
  END
END # Close_Loop

#
#=====================================================================
#
# Task to open a control loop
#
#=====================================================================
#

TASK Open_Loop

PARAMETER
  Controller              AS MODEL Proportional_Integral_Controller

SCHEDULE
  REPLACE
    Controller.Bias
  WITH
    Controller.Control_Action := 1.0 ;
  END
END # Open_Loop

#
#=====================================================================
#
# Task to start-up the complete plant
#
#=====================================================================
#

TASK Start_Up_Pilot_Plant

PARAMETER
  Plant                         AS MODEL Complete_Plant_Flowsheet

SCHEDULE
  SEQUENCE
    PARALLEL
      Start_Pump(Pump IS Plant.Feed_Control.Pump) ;
      Close_Loop(Controller IS Plant.Feed_Control.Controller) ;
    END
    CONTINUE UNTIL Plant.Splitter.Liquid_Height >
                                  Plant.Splitter.Height_Liquid_Max
    RESET Plant.Product_Tank.Pump_Product := 1.0 ; END
    CONTINUE FOR 20
    PARALLEL
      Start_Pump(Pump IS Plant.Recycle_Control.Pump) ;
      Close_Loop(Controller IS Plant.Recycle_Control.Controller) ;
    END
    CONTINUE FOR 20
    RESET Plant.Calandria.Flow_Steam := 1.0 ; END
    CONTINUE UNTIL Plant.Splitter.Temp_Bulk > 99.0
  END
END # Start_Up_Pilot_Plant

#
#=====================================================================
#
# Task to shut-down the complete plant
#
#=====================================================================
#

TASK Shut_Down_Pilot_Plant

PARAMETER
  Plant                         AS MODEL Complete_Plant_Flowsheet

SCHEDULE
  SEQUENCE
    RESET Plant.Calandria.Flow_Steam := 0.0 ; END
    CONTINUE FOR 20
    RESET Plant.Product_Tank.Pump_Product := 0.0 ; END
    CONTINUE UNTIL Plant.Calandria.Temp_Bulk < 60.0
    PARALLEL
      Stop_Pump(Pump IS Plant.Feed_Control.Pump) ;
      Stop_Pump(Pump IS Plant.Recycle_Control.Pump) ;
      Open_Loop(Controller IS Plant.Recycle_Control.Controller) ;
      Open_Loop(Controller IS Plant.Feed_Control.Controller) ;
    END
  END
END # Shut_Down_Pilot_Plant

#
#=====================================================================
#
# Process to test the model of the complete plant
#
#=====================================================================
#

PROCESS Whole_Plant_Model_Test

  UNIT
    Plant AS Complete_Plant_Flowsheet

  ASSIGN
    WITHIN Plant DO
      WITHIN Feed_Tank DO
        Flow_Refill                 :=     0.0         ;
        Conc_Refill                 :=     0.08        ;
        Temp_Refill                 :=    23.0         ;
        Heat_Loss                   :=     0.0         ;
        Area                        :=     0.8659      ;
      END # Feed_Tank
      WITHIN Junction DO
        Volume                      :=     0.001       ;
        Temp_Ambient                :=    23.0         ;
        HT_Area                     :=     0.067       ;
        Recycle_Pump                :=     0.0         ;
        Manual_Valve                :=     0.0         ;
        Feed_Pump                   :=     0.0         ;
        Flow_Recycle_Set            :=     2.0         ;
        Flow_Feed_Set               :=     0.02        ;
      END # Junction
      WITHIN Product_Tank DO
        Area                        :=     0.049       ;
        Setpoint                    :=     0.09        ;
        Cv                          :=     0.05        ;
        Temp_Ambient                :=    23.0         ;
        HT_Area                     :=     0.38        ;
        Pump_Product                :=     0.0         ;
        Bias                        :=     0.5         ;
        Gain                        :=     8.0         ;
      END # Product_Tank
      WITHIN Calandria DO
        Height_Liquid_Max           :=     1.12        ;
        Temp_Ambient                :=    23.0         ;
        HT_Area                     :=     0.592       ;
        Flow_Steam                  :=     0.0         ;
        Temp_Steam                  :=   125.0         ;
      END # Calandria
      WITHIN Splitter DO
         Height_Liquid_Max          :=     4.0         ;
        Temp_Ambient                :=    23.0         ;
        HT_Area                     :=     1.72        ;
      END # Splitter
      WITHIN Tube DO
        Height_Liquid_Max           :=     2.35        ;
        Temp_Ambient                :=    23.0         ;
        HT_Area                     :=     0.75        ;
      END # Tube
      WITHIN Feed_Control DO
        WITHIN Controller DO
          Gain                      :=     0.01        ;
          Reset_Time                :=      5          ;
          Setpoint                  :=     0.02        ;
          Control_Action            :=     0.0         ;
          Steady_Bias               :=     0.23        ;
        END # Controller
```

```
      WITHIN Pump DO
        Pump_Status              :=     0.0       ;
        Parameter_1              :=   100.0E-3    ;
        Parameter_2              :=     0.16E-3   ;
      END # Pump
      WITHIN Valve DO
        Valve_Constant           :=     0.04      ;
      END # Valve
    END # Feed
    WITHIN Recycle_Control DO
      WITHIN Controller DO
        Gain                     :=     1.0       ;
        Reset_Time               :=    50.0       ;
        Setpoint                 :=     2.0       ;
        Control_Action           :=     1.0       ;
        Steady_Bias              :=     0.9       ;
      END # Controller
      WITHIN Pump DO
        Pump_Status              :=     0.0       ;
        Parameter_1              :=  6250.0E-3    ;
        Parameter_2              :=    10.0E-3    ;
      END # Pump
      WITHIN Valve DO
        Valve_Constant           :=     1.10      ;
      END # Valve
    END # Recycle
  END # Plant

PRESET
  WITHIN Plant DO
    WITHIN Feed_Tank DO
      Enth_Bulk                  :=   135.17      ;
      Enth_Refill                :=   144.89      ;
      Density                    :=  1030.46      ;
    END # Feed Tank
    WITHIN Junction DO
      Flow_Cala                  :=     0.0       ;
      Flow_Split                 :=     0.0       ;
      Density_Bulk               :=  1050.9       ;
      Enth_Bulk                  :=   144.89      ;
      Enth_Feed                  :=   135.17      ;
      Enth_Split                 :=   144.89      ;
      Enth_Cala                  :=   144.89      ;
    END # Junction
    WITHIN Calandria DO
      Flow_Vapour                := 0.0            ;
      OverFlow_Flow              := 0.0            ;
      Delta_T                    := 100            ;
    END # Calandria
    WITHIN Tube DO
      Solute_Holdup              :=     0.0       ;
      Total_Holdup               :=     0.0        ;
      Total_U_Holdup             :=     0.0        ;
      Conc_Bulk                  :=     0.0        ;
    END # Tube
  END # Within

SELECTOR
  WITHIN Plant DO
    WITHIN Calandria DO
      Flow_Flag      := Not_Full ;
    END
    Split_Junct_Flag := Positive ;
    Cala_Junct_Flag  := Positive ;
    Cala_Tube_Flag   := Positive ;
    Feed_Junct_Flag  := Positive ;
  END

INITIAL
  WITHIN Plant DO
    WITHIN Calandria DO
      Temp_Bulk                   =    23.0        ;
      Conc_Bulk                   =     0.080      ;
      Liquid_Height               =     0.5        ;
    END # Calandria
    WITHIN Feed_Tank DO
      Temp_Bulk                   =    23.0        ;
      Conc_Bulk                   =     0.055      ;
      Liquid_Height               =     1.5        ;
    END # Feed_Tank
    WITHIN Splitter DO
      Temp_Bulk                   =    23.0        ;
      Conc_Bulk                   =     0.080      ;
      Liquid_Height               =     1.1        ;
    END # Splitter
    WITHIN Tube DO
      Solute_Holdup               =     0.0        ;
      Total_Holdup                =     0.0        ;
      Total_U_Holdup              =     0.0        ;
    END # Tube
    WITHIN Product_Tank DO
      Temp_Bulk                   =    23.0        ;
      Conc_Bulk                   =     0.080      ;
      Height_Tank                 =     0.05       ;
    END # Product_Tank
    WITHIN Junction DO
      Temp_Bulk                   =    23.0        ;
      Conc_Bulk                   =     0.080      ;
    END # Junction
    WITHIN Feed_Control.Controller DO
      Integral_Error              =     0.0        ;
    END # Feed_Control.Controller
```

```
      WITHIN Recycle_Control.Controller DO
        Integral_Error              =     0.0        ;
      END # Recycle_Control.Controller
    END # Within

  SCHEDULE
    SEQUENCE
      CONTINUE FOR 100
      # Apply the start-up procedure
      Start_Up_Pilot_Plant(Plant IS Plant) ;
      # Run the plant for a while
      CONTINUE FOR 5000
      # Apply the shut-down procedure
      Shut_Down_Pilot_Plant(Plant IS Plant) ;
      # Let the plant drain
      CONTINUE UNTIL Time > 21000
    END
END
```

Appendix D

Modelling Equations for the Equilibrium Flash Drum

A dynamic model for continuous time dependent behaviour of an equilibrium flash drum, according to the assumptions described in chapter 6, is detailed here. At any point in time, the model may be in one of three states, corresponding to whether the vessel contains both liquid and vapour phases, subcooled liquid, or superheated vapour. Table D.1 details the notation employed.

The invariant equations will be dealt with first. The total component mass balances yield the following differential equations:

$$\frac{d\sigma_i}{dt} = F_i - Vx_i - Ly_i \qquad \forall i = 1 \ldots N \tag{D.1}$$

Similarly, a total energy balance yields the differential equation:

$$\frac{dU}{dt} = H_{in} - Vh_v - Lh_l + Q \tag{D.2}$$

And, the fact that the volume of the contents of the vessel must equal the volume of the vessel yields:

$$\mathcal{V} = \frac{\sigma_v}{\rho_v} + \frac{\sigma_l}{\rho_l} \tag{D.3}$$

With this constraint, the minimal set of extensive properties (σ_i, U), completely define the state of the system. All other quantities can therefore be derived from these properties via the auxiliary algebraic relationships:

$$\sigma_i = \sigma_v y_i + \sigma_l x_i \qquad \forall i = 1 \ldots N \tag{D.4}$$

$$U = \sigma_v u_v + \sigma_l u_l \tag{D.5}$$

$$h_v = u_v + \frac{P}{\rho_v} \tag{D.6}$$

$$h_l = u_l + \frac{P}{\rho_l} \tag{D.7}$$

$$h_v = \Phi(y_i, T, P) \tag{D.8}$$

F_i	Molar feed of component i
H_{in}	Enthalpy flow of feed stream
h_l	Molar enthalpy of liquid phase
h_v	Molar enthalpy of vapour phase
L	Total molar flow of liquid from vessel
k_i	Vapour-liquid distribution co-efficient for component i
N	Number of components in feed stream
P	Pressure
P_l	Downstream pressure for liquid stream
P_v	Downstream pressure for vapour stream
Q	Total heat flow to the vessel
T	Temperature
t	Time
U	Total internal energy holdup
u_l	Molar internal energy of liquid phase
u_v	Molar internal energy of vapour phase
V	Total molar flow of vapour from vessel
$\mathcal{V}$	Volume of vessel (time invariant)
x_i	Mole fraction of component i in the liquid phase
y_i	Mole fraction of component i in the vapour phase
ρ_l	Molar density of liquid phase
ρ_v	Molar density of vapour phase
σ_i	Molar holdup of component i
σ_l	Total molar holdup in liquid phase
σ_v	Total molar holdup in vapour phase
$\Phi(\ldots)$	Generic function of its arguments

Table D.1: Nomenclature for Model of Equilibrium Flash Drum

$$h_l = \Phi(x_i, T, P) \tag{D.9}$$

$$\rho_v = \Phi(y_i, T, P) \tag{D.10}$$

$$\rho_l = \Phi(x_i, T) \tag{D.11}$$

$$k_i = \Phi(x_i, y_i, T, P) \qquad \forall i = 1 \ldots N \tag{D.12}$$

The total liquid and vapour flows from the vessel can be determined from flow/pressure relationships, which could be declared in downstream units such as valves or pumps as opposed to the flash drum itself:

$$V = \Phi(P, P_l, \sigma_L) \tag{D.13}$$

$$L = \Phi(P, P_v, \sigma_L) \tag{D.14}$$

Conditional equations may be employed in these relationships to model devices that prevent the wrong phase flowing in a particular stream.

It is important to recognise that, provided correct values are assigned to the liquid and vapour phase mole fractions, x_i and y_i, all the above equations remain well behaved in all three phase regimes, although some inefficiency is introduced when the physical properties of a phase not present are evaluated.[1]

Finally, the variant equations specific to each phase regime must be specified. In the two-phase regime we have:

$$y_i = k_i x_i \qquad \forall i = 1 \ldots N \tag{D.15}$$

$$\sum_{i=1}^{N} y_i = \sum_{i=1}^{N} x_i = 1 \tag{D.16}$$

In the liquid regime we have:

$$\sigma_v = 0 \tag{D.17}$$

$$\sum_{i=1}^{N} x_i = 1 \tag{D.18}$$

and y_i are undefined. This can be declared explicitly through use of the UNDEFINED construct (see section 2.3.5.4) or y_i can be arbitrarily set equal to x_i. In the vapour regime we have:

$$\sigma_l = 0 \tag{D.19}$$

$$\sum_{i=1}^{N} y_i = 1 \tag{D.20}$$

and x_i are undefined or arbitrarily set equal to y_i. The set of variant equations can therefore be expressed with the CASE equation shown in figure D.1. Of course, extra equations to define the bubble and dew temperatures that appear in the logical expressions must also be introduced.

[1]Alternatively, the equations that determine the physical properties of each phase could be added to the set of variant equations.

```
CASE Phase OF
  WHEN Liquid    : Vapour_Holdup = 0 ;
                   SIGMA(X) = 1 ;
                   UNDEFINED(Y) ;
                   SWITCH TO Two_Phase IF Temp > Bub_Temp ;
  WHEN Two_Phase : Y = K_Value*X ;
                   SIGMA(X) = 1 ;
                   SIGMA(Y) = 1 ;
                   SWITCH TO Liquid IF Vapour_Holdup <= 0 ;
                   SWITCH TO Vapour IF Liquid_Holdup <= 0 ;
  WHEN Vapour    : Liquid_Holdup = 0 ;
                   SIGMA(Y) = 1 ;
                   UNDEFINED(X) ;
                   SWITCH TO Two_Phase IF Temp < Dew_Temp ;
END # case
```

Figure D.1: Extract from the Model of the Flash Drum

References

M. Andersson. An object-oriented modelling environment. In *Proceedings European Simulation Multiconference*, Rome, Italy, 1989.

M. Andersson. *Omola - An Object-Oriented Language for Model Representation.* PhD thesis, Lund Institute of Technology, 1990.

M. Andersson. Discrete event modelling and simulation in Omola. In *IEEE Symposium on Computer-Aided Control System Design*, Napa, California, 1992.

M. R. Aylott, J. W. Ponton, and D. H. Lott. Development of a dynamic flowsheeting program. In *Proceedings 2nd International Symposium on Process Systems Engineering*, pages 55–66, Cambridge, England, 1985.

R. Bachmann, L. Brull, Th. Mrziglod, and V. Pallaske. On methods for reducing the index of differential algebraic equations. *Computers and Chemical Engineering*, 14:1271–1273, 1990.

M. Bar and M. Zeitz. A knowledge-based flowsheet-oriented user interface for a dynamic process simulator. *Computers and Chemical Engineering*, 14:1275–1280, 1990.

R. A. Bernal. Development of a dynamic model for a pilot plant evaporator. Master's thesis, Imperial College, 1986.

G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA BEGIN.* Chartwell-Bratt, 1979.

K. E. Brenan, S. L. Campbell, and L. R. Petzold. *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations.* North-Holland, 1989.

F. E. Cellier and A. P. Bongulielmi. The COSY simulation language. In L. Dekker, G. Savastano, and G. C. Vansteenkiste, editors, *Simulation of Systems '79*, pages 271–281. North-Holland, 1980.

F. E. Cellier. *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools.* PhD thesis, Swiss Federal Institute of Technology Zurich, 1979.

F. E. Cellier. Combined discrete/continuous system simulation languages – usefullness, experiences and future development. In B. P. Ziegler, M. S. Elzas, G. J. Klir, and T. I. Oren, editors, *Methodology in Systems Modelling and Simulation*, pages 201–220. North-Holland, 1979.

F. E. Cellier. Combined continuous/discrete simulation applications, techniques, and tools. In J. Wilson, J. Henriken, and S. Roberts, editors, *Proceedings of the 1986 Winter Simulation Conference*, pages 24–33, 1986.

Y. Chung and A. W. Westerberg. A proposed numerical algorithm for solving nonlinear index problems. *Ind. Eng. Chem. Res.*, 29:1234–1239, 1990.

S. M. Clark and K. Kuriyan. BATCHES – simulation software for managing semicontinuous and batch processes. In *Proceedings AIChE National Meeting*, April 1989.

B. J. Cott. *An Integrated Computer-Aided Production Management System for Batch Chemical Process.* PhD thesis, University of London, 1989.

A. J. Czulek. An experimental simulator for batch chemical processes. *Computers and Chemical Engineering*, 12:253–259, 1988.

H. Elmquist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Lund Institute of Technology, 1978.

S. Ergun. Fluid flow through packed columns. *Chemical Engineering Progress*, 48:87–94, 1952.

S. F. Evans and P. Wylie. A plant simulator for the THORP nuclear fuel reprocessing plant at Sellafield. In *Dynamic Simulation in the Process Industries*, UMIST Manchester, 1990. IChemE.

D. A. Fahrland. Combined discrete event continuous systems simulation. *Simulation*, 14:61–72, 1970.

M. J. Ferney. Process simulators for safety. *Plant/Operations Progress*, 11:133–135, 1991.

C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings, 1988.

W. M. Fruit, G. V. Reklaitis, and J. M. Woods. Simulation of multiproduct batch chemical processes. *The Chemical Engineering Journal*, 8:199–211, 1974.

R. Gani, E. L. Sørensen, and J. Perregaard. Design and analysis of chemical processes through DYNSIM. *Ind. Eng. Chem. Res.*, 31:244–254, 1992.

C. W. Gear. Simultaneous numerical solution of differential-algebraic equations. *IEEE Transactions on Circuit Theory*, CT–18:89–95, 1971.

D. G. Golden and J. D. Schoeffler. GSL – a combined continuous and discrete simulation language. *Simulation*, 20:1–8, 1973.

D. Gritsis, C. C. Pantelides, and R. W. H. Sargent. The dynamic simulation of transient systems described by index two differential-algebraic equations. In *Proceedings 3rd International Symposium on Process Systems Engineering*, Sydney, Australia, 1988.

J. L. Hay, R. E. Crosbie, and R. I. Chaplin. Integration routines for systems with discontinuities. *The Computer Journal*, 17:275–278, 1974.

K. Helsgaun. DISCO - a SIMULA based language for continuous combined and discrete simulation. *Simulation*, 35:1–12, July 1980.

J. C. Heydweiller, R. F. Sincovec, and L. T. Fan. Dynamic simulation of chemical processes described by distributed and lumped parameter models. *Computers and Chemical Engineering*, 1:125–131, 1977.

P. Holl, W. Marquardt, and E. D. Gilles. Diva - a powerful tool for dynamic process simulation. *Computers and Chemical Engineering*, 12:421–426, 1988.

J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addsion Wesley, 1979.

D. Hutton, A. Struthers, and J. W. Ponton. Knowledge based flowsheeting. In *Proceedings 1991 IChemE Research Event*, pages 127–130, Queen's College, Cambridge, England, 1991.

INMOS. *OCCAM Programming Manual.* Prentice–Hall, 1984.

R. B. Jarvis and C. C. Pantelides. Numerical methods for differential-algebraic equations – new problems and new solutions. In *Proceedings AIChemE Annual Meeting*, Los Angeles, USA, 1991.

R. B. Jarvis and C. C. Pantelides. DASOLV a differential-algebraic equation solver. Technical report, Imperial College, 1992.

G. S. Joglekar and G. V. Reklaitis. A simulator for batch and semi-continuous processes. *Computers and Chemical Engineering*, 8:315–327, 1984.

S. C. Johnson. Yacc – yet another compiler compiler. Technical Report C.S. #32, Bell Telephone Laboratories, 1975.

L. E. Jourda. Dynamic modelling and simulation of batch distillation columns using gproms. Technical report, Imperial College, April 1992.

S. C. Kassianides. *An Integrated System for Computer Based Training of Process Operators.* PhD thesis, University of London, 1991.

D. L. Kettenis. COSMOS: A simulation language for continuous, discrete and combined models. *Simulation*, 58:32–41, 1992.

E. S. Kikkinides and R. T. Yang. Simultaneous SO_2/NO_x removal and SO_2 recovery from flue gas by presure swing adsorption. *Ind. Eng. Chem. Res.*, 30:1981–1989, 1991.

K. N. King. *Modula–2 A Complete Guide.* D. C. Heath and Company, 1988.

W. Kreutzer. *System Simulation Programming Styles and Languages.* Addison-Wesley, 1986.

S. Macchietto, M. Matzopoulos, and G. Stuart. On-line simulation and optimization as an aid to plant operation – the easy way. In G. V. Reklaitis and H. D. Spriggs, editors, *Foundations of Computer Aided Process Operations*, pages 669–677. Cache–Elsevier, 1987.

S. Mani, S. K. Shoor, and H. S. Pedersen. Experience with a simulator for training ammonia plant operators. *Plant/Operations Progress*, 10:6–10, 1990.

W. Marquardt. Dynamic process simulation - recent progress and future challenges. In *Proceedings CPC IV*, Texas, USA, 1991.

S. E. Mattsson and M. Andersson. A kernel for system representation. In *Proceedings 11th IFAC World Congress*, Tallin, USSR, 1990.

S. E. Mattsson. On modelling and differential/algebraic systems. *Simulation*, pages 24–32, January 1989.

L. M. McLoughlin. The Pascal and Modula–2 versions of Yacc. Technical report, Westfield College, London, 1981.

B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

E. E. L. Mitchell and J. S. Gauthier. Advanced continuous simulation language (ACSL). *Simulation*, 26:72–78, 1976.

B. Nilsson. *Structured Modelling of Chemical Processes - An Object-Oriented Approach.* PhD thesis, Lund Institute of Technology, 1989.

B. Nilsson. Structured modelling of chemical processes with control systems. In *Proceedings AIChE Annual Meeting*, 1989.

T. I. Oren and B. P. Ziegler. Concepts for advanced simulation methodologies. *Simulation*, 32:69–82, March 1979.

T. I. Oren. Software for simulation of combined continuous and discrete systems: A state-of-the-art review. *Simulation*, 28:33–45, 1977.

C. C. Pantelides, D. Gritsis, K. R. Morison, and R. W. H. Sargent. The mathematical modelling of transient systems using differential-algebraic equations. *Computers and Chemical Engineering*, 12:449–454, 1988.

C. C. Pantelides. The consistent initialisation of differential/algebraic systems. *SIAM J. Sci. Stat. Comput.*, 9:213–231, 1988.

C. C. Pantelides. SpeedUp – recent advances in process simulation. *Computers and Chemical Engineering*, 12:745–755, 1988.

C. C. Pantelides. *Symbolic and Numerical Techniques for the Solution of Large Systems of Nonlinear Algebraic Equations.* PhD thesis, University of London, 1988.

R. Parakrama. Improving batch chemical processes. *The Chemical Engineer*, pages 24–25, September 1985.

J. G. Pearce. Computer simulation of multi-state systems. In *Proceedings UK Simulation Council Conference on Computer Simulation*, pages 484–493. IPC Science and Tech Press, 1978.

C. D. Pegden. *Introduction to SIMAN.* Systems Modelling Corporation, 1982.

J. D. Perkins and G. W. Barton. Modelling and simulation in process operation. In G. V. Reklaitis and H. D. Spriggs, editors, *Foundations of Computer Aided Process Operations*, pages 287–316. Cache–Elsevier, 1987.

J. D. Perkins and R. W. H. Sargent. SPEEDUP: A computer program for steady-state and dynamic simulation and design of chemical processes. In *AIChE Symposium Series No. 78*, 1982.

J. D. Perkins. Dynamic simulation of chemical plants: Why and how? In *Proceedings Multi-Stream '85*, London, England, 1985.

J. D. Perkins. Survey of existing systems for the dynamic simulation of industrial processes. *Modeling, Identification and Control*, 7:71–81, 1986.

F. A. Perris. The growing importance of dynamic simulation for process engineers. In *Dynamic Simulation in the Process Industries*, UMIST Manchester, 1990. IChemE.

C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.

L. Petzold. A description of DASSL: a differential-algebraic equation solver. In *Proceedings 10th IMACS World Congress*, Montreal, Canada, 1982.

P. C. Piela, T. G. Epperly, K. M. Westerberg, and A. W. Westerberg. ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language. *Computers and Chemical Engineering*, 15:53–72, 1991.

P. C. Piela. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis.* PhD thesis, Carnegie–Mellon University, 1989.

K. G. Pipilis. *Higher Order Moving Finite Element Methods for Systems Described by Partial Differential-Algebraic Equations.* PhD thesis, University of London, 1990.

J. W. Ponton and P. J. Gawthrop. Systematic construction of dynamic models for phase equilibrium. *Computers and Chemical Engineering*, 15:803–808, 1991.

A. J. Preston and M. Berzins. Algorithms for the location of discontinuities in dynamic simulation problems. *Computers and Chemical Engineering*, 15:701–713, 1991.

A. A. B. Pritsker and R. E. Hurst. GASP IV: a combined continuous-discrete fortran-based simulation language. *Simulation*, 21:65–70, 1973.

A. A. B. Pritsker and P. J. Kiviat. *Simulation with GASP II.* Prentice-Hall, 1969.

A. A. B. Pritsker. *Introduction to Simulation and SLAM II.* John Wiley, 1986.

Prosys. *The ADSIM/SU User Manual Version 5.1.* Prosys Technology Ltd., 1989.

Prosys. *SpeedUp User Manual Issue 5.3.1.* Prosys Technology Ltd., 1991.

G. V. Reklaitis. Perspectives on scheduling and planning of process operations. In *Proceedings 4th International Symposium on Process Systems Engineering*, volume III, 1991.

H. P. Rosenof and A. Ghosh. *Batch Process Automation Theory and Practice.* Van Nostrand Reinhold, New York, 1987.

R. P. Rozo. Development of a real-time simulation for a forced recirculation evaporator. Master's thesis, Imperial College, 1987.

P. Sawyer. Simulate to succeed. *The Chemical Engineer*, pages 28–30, March 1992.

C. F. Shewchuk and W. Morton. A new dynamic simulator for on-line applications. In *Proceedings 1990 TAPPI Engineering Conference*, volume 2, pages 595–603, 1990.

T. Shinohara. A block structured approach to dynamic process simulation. In G. V. Reklaitis and H. D. Spriggs, editors, *Foundations of Computer Aided Process Operations*, pages 317–331. Cache–Elsevier, 1987.

R. J. W. Sim. CADSIM – user's guide and reference manual. Technical report, Imperial College, 1975.

C. W. Skarstrom. In N. Li, editor, *Recent Developments in Separation Science*, volume II, page 95. C.R.C. Press, 1975.

P. J. Smart and N. J. C. Baker. SYSMOD – an environment for modular simulation. In *Proceedings Summer Computer Simulation Conference*, pages 77–82. North-Holland, 1984.

G. J. Smith and W. Morton. Dynamic simulation using an equation-oriented flowsheeting package. *Computers and Chemical Engineering*, 12:469–473, 1988.

E. M. B. Smith. Simulation of combined/discrete continuous chemical processes using gPROMS. Technical report, Imperial College, 1991.

H. Speckhart and W. H. Green. *A Guide to Using CSMP*. Prentice-Hall, 1976.

G. Stephanopoulos, J. Johnston, T. Kriticos, R. Lakshmanan, M. Mavrovouniotis, and C. Siletti. DESIGN-KIT: An object-oriented environment for process engineering. *Computers and Chemical Engineering*, 11:655–674, 1987.

G. Stephanopoulos, G. Henning, and H. Leone. MODEL.LA. a modeling language for process engineering - I.The formal framework. *Computers and Chemical Engineering*, 14:813–846, 1990.

G. Stephanopoulos, G. Henning, and H. Leone. MODEL.LA. a modeling language for process engineering - II. Multifaceted modeling of processing systems. *Computers and Chemical Engineering*, 14:847–869, 1990.

J. C. Strauss. The SCi continuous system simulation language (CSSL). *Simulation*, 9:281–303, 1967.

R. Vázquez-Román. *Computer Aids for Process Model–Building*. PhD thesis, University of London, 1992.

P. S. Ward. The simulation of industrial adsorption processes. In *Dynamic Simulation in the Process Industries*, UMIST Manchester, 1990. IChemE.

A. W. Westerberg and D. R. Benjamin. Thoughts on a future equation-oriented flowsheeting system. *Computers and Chemical Engineering*, 9:517–526, 1985.

A. W. Westerberg, P. C. Piela, R. D. McKelvey, and T. G. Epperly. The ASCEND modeling environment and its implications. In *Proceedings 4th International Symposium on Process Systems Engineering*, volume I, 1991.

N. Wirth. *Algorithms and Data Structures*. Prentice-Hall, 1986.

N. Wirth. *Programming in Modula–2*. Springer-Verlag, 4th edition, 1988.

R. K. Wood, R. K. M. Thambynayagam, R. G. Noble, and D. J. Sebastian. DPS - a digital simulation language for the process industries. *Simulation*, pages 221–233, May 1984.

J. J. Zaher and A. W. Westerberg. Conditional modeling in an equation-based modeling environment. In *Proceedings AIChemE Annual Meeting*, Los Angeles, USA, 1991.