WARWICK
THE UNIVERSITY OF WARWICK

A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:
http://wrap.warwick.ac.uk/106507

Copyright and reuse:
This thesis is made available online and is protected by original copyright.
Please scroll down to view the document itself.
Please refer to the repository record for this item for information to help you to cite it.
Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

warwick.ac.uk/lib-publications

THE BRITISH LIBRARY DOCUMENT SUPPLY CENTRE

TITLE Towards A Knowledge-Based Discrete Simulation Modelling Environment Using Prolog

AUTHOR Ali Ahmad

INSTITUTION and DATE University of Warwick, June 1989

Attention is drawn to the fact that the copyright of this thesis rests with its author.

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no information derived from it may be published without the author's prior written consent.

THE BRITISH LIBRARY
DOCUMENT SUPPLY CENTRE
Boston Spa, Wetherby
West Yorkshire
United Kingdom

cms 1 2 3 4 5 6

REDUCTION X 20

CRMCOA 3

Towards A Knowledge-Based Discrete Simulation Modelling Environment Using Prolog

by

Ali Ahmad

A thesis submitted for admission to the degree of Doctor of Philosophy

University of Warwick

School of Industrial and Business Studies

June 1989

TABLE OF CONTENTS

Title i

Table of contents ii

List of illustrations xii

Acknowledgements xiv

Summary xv

Introduction xvi

CHAPTER 1: PROBLEM SOLVING 1

INTRODUCTION 1

1.1. PROBLEM SOLVING IN GENERAL 1
1.1.1. PROBLEMATIC SITUATIONS 1
1.1.2. PROBLEM DESCRIPTION 2
1.1.3. THE SOLUTION 2
1.1.4. PROBLEM SOLVING 3
1.1.5. ABSTRACTION: REPRESENTATION IN FORMAL LANGUAGES 4
1.1.6. GENERAL FORMS 5
1.1.7. SOLUTION PROCEDURES 6
1.1.8. PROBLEM FORMULATION 7
1.1.9. METHODOLOGY IN PROBLEM SOLVING 7
1.1.10. THEORIES OF PROBLEM SOLVING 9
1.1.11. CLASSES OF PROBLEMS 9
1.1.12. PROBLEM SOLVING PARADIGMS 10

1.2. OPERATIONAL RESEARCH APPROACHES TO PROBLEM SOLVING 10
1.2.1. PROBLEM SOLVING WITHIN ORGANIZATIONAL DECISION MAKING 10
1.2.2. METHODOLOGY OF OPERATIONAL RESEARCH 11
1.2.3. TECHNIQUES OF OPERATIONAL RESEARCH 12
1.2.4. DECISION-MAKER AND THE ANALYST 12

1.3. THE USE OF COMPUTERS IN PROBLEM SOLVING 13
1.3.1. SOLUTION PROCEDURES APPLIED BY HUMANS 13
1.3.2. THE USE OF DEVICES 13
1.3.3. ANALOGUE ELECTRONIC DEVICES 13

1.3.4. DIGITAL COMPUTERS IN PROBLEM SOLVING 14
1.3.5. THE USE OF COMPUTERS IN OPERATIONAL RESEARCH 16
1.3.6. TOWARDS ARTIFICIAL INTELLIGENCE 19

1.4. ARTIFICIAL INTELLIGENCE APPROACHES TO PROBLEM SOLVING 20
1.4.1. PROBLEM SOLVING AND ARTIFICIAL INTELLIGENCE 21
1.4.2. THE FORMULATION OF PROBLEMS IN AI 21
1.4.3. PROBLEM-SOLVING PROCEDURES USED IN AI 21
1.4.4. KNOWLEDGE-BASED SYSTEMS (EXPERT SYSTEMS) 22

1.5. TOWARDS ADAPTING ARTIFICIAL INTELLIGENCE TECHNOLOGY IN MANAGEMENT SCIENCE / OPERATIONAL RESEARCH 22
1.5.1. THE EARLY HISTORY OF OR AND AI 22
1.5.2. THE POTENTIAL APPLICATIONS 23

1.6. CONCLUSION 24

CHAPTER 2: A REVIEW OF SIMULATION MODELLING 25

INTRODUCTION 25

2.1. ABOUT SIMULATION 25
2.1.1. DEFINITIONS OF SIMULATION 25
2.1.2. THE PLACE OF SIMULATION IN THE PROBLEM SOLVER'S TOOLBOX 26
2.1.3. THE DIMENSIONS OF SIMULATION 27
2.1.4. THE APPLICATION AREAS 28
2.1.5. PROBLEMS WITH THE USE OF SIMULATION 28

2.2. THE EXPERIMENTAL FRAME 28

2.3. SIMULATION MODELLING 31
2.3.1. SYSTEM DESCRIPTION 31
2.3.2. THE 'EXECUTABLE' SIMULATION MODEL 31

2.4. DYNAMIC BEHAVIOUR GENERATION 32
2.4.1. SIMULATION EXECUTIVES 33
2.4.2. THE REPRESENTATION SCHEME FOR THE SYSTEM'S STATE 33
2.4.3. APPROACHES TO DYNAMIC BEHAVIOUR GENERATION 34

2.5. SIMULATION SOFTWARE 42
2.5.1. SIMULATION LANGUAGES AND PACKAGES 42
2.5.2. SIMULATION PROGRAM GENERATORS 43
2.5.3. SIMULATION ENVIRONMENTS 43

2.6. THE INFLUENCES ON SIMULATION SOFTWARE OF DEVELOPMENTS IN COMPUTER SCIENCE 45
2.6.1. HARDWARE 45
2.6.2. THE AVAILABILITY OF NEW PROGRAMMING LANGUAGES 45
2.6.3. THE GRAMMAR FOR SIMULATION LANGUAGES 46
2.6.4. OBJECT ORIENTED PROGRAMMING 46
2.6.5. COMPUTER GRAPHICS 46
2.6.6. INTERACTIVE SOFTWARE 47
2.6.7. SOFTWARE ENGINEERING 47
2.6.8. DATA-BASE FACILITIES 48
2.6.9. FUNCTIONAL- AND LOGIC PROGRAMMING PARADIGMS 48

2.7. CURRENT TRENDS IN SIMULATION METHODOLOGY 49
2.7.1. TRENDS IN THE PRACTICE OF SIMULATION 49
2.7.2. EXPERIMENTAL-FRAME BASE AND MODEL BASE 50
2.7.3. INTERACTIVE MODEL DEVELOPMENT BY NON-EXPERTS 50
2.7.4. GRAPHICS WITHIN SIMULATION 50

2.8. SIMULATION AND ARTIFICIAL INTELLIGENCE 50
2.8.1. VIEWS ABOUT THE USE OF AI TECHNIQUES IN SIMULATION 51
2.8.2. THE IMPLEMENTATION OF ARTIFICIAL INTELLIGENCE TECHNOLOGY IN VARIOUS PHASES OF A SIMULATION STUDY 56

CHAPTER 3: KNOWLEDGE-BASED SYSTEMS- AND LOGIC PROGRAMMING PARADIGMS 59

INTRODUCTION 59

3.1. A REVIEW OF THE KNOWLEDGE BASED SYSTEMS PARADIGM 59
3.1.1. FROM GENERAL PROBLEM SOLVING TO KNOWLEDGE-BASED PROBLEM SOLVING 59
3.1.2. COMPUTER SYSTEMS FOR THE 1990s 61
3.1.3. TECHNICAL AND FUNCTIONAL CLASSIFICATIONS OF EXPERT SYSTEMS 61
3.1.4. KNOWLEDGE-BASED SYSTEMS ARCHITECTURE 62
3.1.5. KNOWLEDGE REPRESENTATION SCHEMES 62
3.1.6. THE INFERENCE ENGINE 62
3.1.7. THE CONTROL MECHANISMS FOR INFERENCE 65
3.1.8. THE HANDLING OF UNCERTAINTY WITHIN INFERENCE 65
3.1.9. THE USER INTERFACE AND EXPLANATION FACILITIES 66

3.1.10. THE KNOWLEDGE ENGINEER'S SOFTWARE TOOLS 67

3.2. A REVIEW OF THE LOGIC PROGRAMMING PARADIGM 68
3.2.1. AUTOMATION OF DEDUCTION IN FIRST ORDER PREDICATE LOGIC 68
3.2.2. THE CHARACTERISTIC FEATURES OF PROLOG 69
3.2.3. THE PROBLEM SOLVING INTERPRETATIONS OF A SET OF PROLOG CLAUSES 70
3.2.4. DATA-BASE INTERPRETATION OF PROLOG CLAUSES 70
3.2.5. COMPILER WRITING AND PROLOG 71
3.2.6. THE RELATIONSHIP BETWEEN PROLOG AND OBJECTS 71

3.3. THE REPORTED USE OF PROLOG IN SIMULATION 72

3.4. ARGUMENTS TO SUPPORT THE USE OF PROLOG AS THE IMPLEMENTATION LANGUAGE FOR THIS PROJECT 74
3.4.1. THE DISTINCTION BETWEEN KNOWLEDGE REPRESENTATION SCHEMES AND IMPLEMENTATION LANGUAGES 74
3.4.2. SUPPORTING ARGUMENTS 74

3.5. INITIAL CONJECTURES RELATING TO PROLOG AND SIMULATION LANGUAGE GRAMMARS 77

CHAPTER 4: A PROTOTYPE SIMULATION ENGINE WRITTEN IN PROLOG 78

INTRODUCTION 78

4.1. THE INITIAL WORK TOWARDS A SIMULATION ENGINE USING PASCAL 79
4.1.1. BACKGROUND 79
4.1.2. MOTIVATION 80
4.1.3. THE INITIAL WORK IN PASCAL 81

4.2. THE MOTIVATION FOR THE SHIFT TOWARDS LOGIC PROGRAMMING 82
4.2.1. TO EXPLORE THE FEASIBILITY OF USING LOGIC PROGRAMMING FOR SIMULATED BEHAVIOUR GENERATION 82
4.2.2. TO FACILITATE THE USE OF SIMULATION BASED AI PROBLEM SOLVING TECHNOLOGY 82
4.2.3. THE BUILT IN SYMBOLIC PROCESSING FEATURES IN PROLOG 83

4.3. BACKGROUND 84
4.3.1. EXPRESSING SIMULATION MODELS USING ALTERNATIVE FORMALISMS 84
4.3.2. A PROTOTYPE INTERACTIVE SIMULATION MODELLING ENVIRONMENT 84
4.3.3. A PROTOTYPE SIMULATION ENGINE WRITTEN IN PROLOG 85

4.4. OBJECTIVES 85

4.5. THE FIRST VERSION OF THE SIMULATION ENGINE (THREE PHASE ONLY) 86
4.5.1. THE DESIGN FEATURES 86
4.5.2. IMPLEMENTATION 89
4.5.3. AN EXAMPLE OF BEHAVIOUR GENERATION BY USING THE SIMULATION ENGINE 93

4.6. THE SECOND VERSION OF THE SIMULATION ENGINE 95
4.6.1. EXTENSIONS TO THE DESIGN SPECIFICATIONS 95
4.6.2. IMPLEMENTATION 95
4.6.3. AN EXAMPLE OF BEHAVIOUR GENERATION BY EXPRESSING THE 'LORRY' MODEL USING PROCESSES ONLY 98
4.6.4. AN EXAMPLE OF BEHAVIOUR GENERATION BY EXPRESSING THE 'LORRY' MODEL USING PROCESSES, EVENTS AND ACTIVITIES 99
4.6.5. A CONSOLIDATED VIEW OF THE SIMULATION ENGINE 99

4.7. THE DESIRABILITY OF THE PURELY DECLARATIVE SPECIFICATION OF SIMULATION MODELS 99
4.7.1. DISCUSSION 99
4.7.2. AN EXAMPLE 102

4.8. CONCLUSIONS AND FURTHER RESEARCH 102
4.8.1. CONCLUSIONS 102
4.8.2. FURTHER RESEARCH 103

ANNEXE 4A 105

4A.1. SIMULATION PROBLEM ('lorry') 105
4A.1.1. NATURAL LANGUAGE DESCRIPTION OF THE PROBLEM 105
4A.1.2. NATURAL LANGUAGE DESCRIPTION OF THE SYSTEM 105

ANNEXE 4B 108

4B.1. THE 'lorry' MODEL (THREE PHASE ONLY) 108
4B.1.1. MODEL ARTICULATION 108
4B.1.2. A TRACE OF THE SIMULATED BEHAVIOUR GENERATED BY THE SIMULATION ENGINE (THREE PHASE ONLY) 111

ANNEXE 4C 114

4C.1. THE 'lorry' MODEL (PROCESSES ONLY) 114
4C.1.1. MODEL ARTICULATION 114
4C.1.2. A TRACE OF THE SIMULATED BEHAVIOUR GENERATED BY THE SIMULATION ENGINE (PROCESSES ONLY) 115

ANNEXE 4D 119

4D.1. THE 'lorry' MODEL (A MIXTURE OF THREE PHASE AND PROCESSES) 119
4D.1.1. MODEL ARTICULATION 119
4D.1.2. A TRACE OF THE SIMULATED BEHAVIOUR GENERATED BY THE SIMULATION ENGINE (MIXED THREE PHASE AND PROCESSES) 119

ANNEXE 4E 123

4E.1. A DECLARATIVE ARTICULATION OF THE 'lorry' MODEL 123

4E.2. THE DOCUMENTATION FOR THE PREDICATES USED FOR DECLARATIVE ARTICULATION 123

4E.3. IMPLICATIONS IN RELATION TO MODEL GENERATION 125

CHAPTER 5: A PROTOTYPE KNOWLEDGE-BASED DISCRETE SIMULATION MODEL GENERATION FACILITY 126

INTRODUCTION 126

5.1. COMPUTER SUPPORT FOR CONSTRUCTING SIMULATION PROGRAMS 127
5.1.1. THE INTERACTIVE ENTRY OF MODEL COMPONENTS EXPRESSED IN A DIAGRAMMATIC FORMALISM 128
5.1.2. SIMULATION PROGRAM GENERATION IN ALTERNATE LANGUAGES 129
5.1.3. MODEL ENTRY AND OUTPUT USING ALTERNATE WORLD VIEWS 129
5.1.4. ASSISTANCE IN MODEL FORMULATION 130
5.1.5. KNOWLEDGE-BASED SIMULATION MODELLING 130
5.1.6. KNOWLEDGE-BASED SOFTWARE SPECIFICATION AND PROGRAM SYNTHESIS 131

5.2. MOTIVATION 132

5.3. OBJECTIVE 133

5.4. A SUB SET OF SIMULATION MODELS 133
5.4.1. CLASSES OF ENTITIES 134
5.4.2. QUEUES 134
5.4.3. ACTIVITY-SETS 134
5.4.4. RESOURCES 134

5.5. DESIGN ASPECTS 134
5.5.1. DESIGN PHILOSOPHY 134
5.5.2. A FORM FOR THE GENERIC SPECIFICATION OF SIMULATION MODELS 135
5.5.3. THE FORM OF THE EXECUTABLE MODEL 136

5.6. A PROTOTYPE KNOWLEDGE-BASED DISCRETE SIMULATION MODEL GENERATION FACILITY 136
5.6.1. AN OVERVIEW 136
5.6.2. THE REPRESENTATION OF THE KNOWLEDGE 136
5.6.3. THE METHOD EMPLOYED FOR MODEL BUILDING 141

5.7. EXAMPLES OF BUILDING SIMPLE MODELS 141
5.7.1. THE PARTIAL 'lorry' MODELS ('merchant' ONLY) 142
5.7.2. THE PARTIAL 'lorry' MODELS ('merchant' AND 'mcb' ONLY) 142
5.7.3. A COMPLETE VERSION OF THE 'lorry' MODEL 148
5.7.4. A HARBOUR MODEL 148
5.7.5. 'harbour-1' MODEL 149

5.8. EXTENSIONS TO ALLOW MORE COMPLEX MODELS 151
5.8.1. A LARGER MODEL 151
5.8.2. THE 'harbour-2' MODEL 152

5.9. CONCLUSIONS AND FUTURE DEVELOPMENT 156
5.9.1. CONCLUSIONS 156
5.9.2. FUTURE DEVELOPMENT 156

ANNEXE 5A 158

5A.1. THE PARTIAL 'lorry' MODEL (MERCHANT ONLY, ONE INSTANCE OF WEIGH BRIDGE) 158
5A.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL 158
5A.1.2. THE KNOWLEDGE-BASE 158
5A.1.3. THE EXECUTABLE MODEL AS GENERATED 159
5A.1.4. SNAPSHOTS OF THE WORKING MEMORY AT VARIOUS STAGES OF THE MODEL DEVELOPMENT 160

ANNEXE 5B 165

5B.1. THE PARTIAL 'lorry' MODEL (MERCHANT ONLY, TWO INSTANCES OF WEIGH-BRIDGE) 165
5B.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL 165
5B.1.2. THE KNOWLEDGE-BASE 165
5B.1.3. THE EXECUTABLE MODEL AS GENERATED 165
5B.1.4. THE CONTENTS OF THE WORKING MEMORY BEFORE THE OUTPUT OF THE EXECUTABLE MODEL 166

ANNEXE 5C 168

5C.1. THE PARTIAL 'lorry' MODEL (MERCHANT AND NCB ONLY, ONE INSTANCE OF WEIGH-BRIDGE, MIXED QUEUEING) 168
5C.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL 168
5C.1.2. THE KNOWLEDGE-BASE 168
5C.1.3. THE EXECUTABLE MODEL AS GENERATED 169
5C.1.4. THE CONTENTS OF THE WORKING MEMORY BEFORE THE OUTPUT OF THE EXECUTABLE MODEL 170

ANNEXE 5D 173

5D.1. THE PARTIAL 'lorry' MODEL (MERCHANT AND NCB ONLY, ONE INSTANCE OF WEIGH-BRIDGE, SEPARATE QUEUEING) 173
5D.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL 173
5D.1.2. THE KNOWLEDGE-BASE 173
5D.1.3. THE EXECUTABLE MODEL 173
5D.1.4. THE CONTENTS OF THE WORKING MEMORY BEFORE THE OUTPUT OF THE EXECUTABLE MODEL 174

ANNEXE 5E 178

5E.1. THE COMPLETE 'lorry' MODEL (MERCHANT, NCB AND TRAIN, ONE INSTANCE OF WEIGH-BRIDGE, ONE INSTANCE OF LOADER, SEPARATE QUEUEING) 178
5E.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL 178
5E.1.2. THE KNOWLEDGE-BASE 178
5E.1.3. THE EXECUTABLE MODEL AS GENERATED 179
5E.1.4. THE CONTENTS OF THE WORKING MEMORY BEFORE THE OUTPUT OF THE EXECUTABLE MODEL 181

ANNEXE 5F 184

5F.1. THE 'harbour-1' MODEL 184
5F.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL 184
5F.1.2. THE KNOWLEDGE BASE 184
5F.1.3. THE EXECUTABLE MODEL AS GENERATED 185

ANNEXE 5G 188

5G.1. THE 'harbour+lorry' MODEL 188
5G.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL 188
5G.1.2. THE KNOWLEDGE BASE 188
5G.1.3. THE EXECUTABLE MODEL AS GENERATED 190

ANNEXE 5H 194

5H.1. 'THE harbour-2' MODEL 194
5H.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL 194
5H.1.2. THE KNOWLEDGE-BASE 194
5H.1.3. THE EXECUTABLE MODEL AS GENERATED 195
5H.1.4. THE EXECUTABLE MODEL AFTER EDITING 196

CHAPTER 6: 'WISE' — A PROTOTYPE KNOWLEDGE-BASED DISCRETE SIMULATION MODELLING ENVIRONMENT 199

INTRODUCTION 199

6.1. MOTIVATION 200

6.2. THE CONCEPTUAL FRAMEWORK 200

6.3. INITIAL PROBLEMS 202

6.4. IMPLEMENTATION 202

6.5. AN EXAMPLE 204
6.5.1. THE KNOWLEDGE-BASE 204
6.5.2. THE INTERACTIVE DEFINITION OF THE 'lorry' MODEL 206
6.5.3. EXCEPTION HANDLING 213
6.5.4. THE USER INTERACTION DURING THE MODEL BUILDING 214

6.6. TOWARDS GENERALIZATIONS 218
6.6.1. THE SIMULATION METHODOLOGY KNOWLEDGE 218
6.6.2. CONDITIONAL BRANCHING 219
6.6.3. THE FORM OF THE EXECUTABLE MODEL 220
6.6.4. THE SUB-MODELS KNOWLEDGE BASE 220

6.7. CONCLUSIONS 224

CHAPTER 7: CONCLUSIONS AND FURTHER RESEARCH 225

7.1. CONCLUSIONS 225
7.1.1. CONCLUSIONS RELATED TO THE WORK ON THE SIMULATION ENGINE 225
7.1.2. CONCLUSIONS RELATED TO THE SIMULATION MODELLING ENVIRONMENT 226

7.2. FURTHER RESEARCH 228

REFERENCES 231

REFERENCES CITED WITHIN QUOTES 249

APPENDIX I 253

APPENDIX II 259

BIBLIOGRAPHY 265

LIST OF ILLUSTRATIONS

Figure 1.1. Summary of a package of methodological tools for systems modelling 18

Figure 1.2. Problem Solving can be regarded as the common area of interest between Operational Research and Artificial Intelligence 20

Figure 2.1. A framework for simulation 30

Figure 2.2. The simplified development of certain two-phase simulation languages 37

Figure 2.3. The simplified development of three-phase simulation languages 38

Figure 2.4. Simulation in the past was characterised by a lack of a unified methodology 41

Figure 3.1. A knowledge system and its environmental context 63

Figure 3.2. The building blocks of a knowledge system 64

Figure 4.1. An overview of the simulation engine environment 87

Figure 4.2. The organisation of the Arity/Prolog Database 88

Figure 4.3. A flow diagram for the 'four phase' mode of behaviour generation where the model can be expressed as a combination of events, activities and processes 97

Figure 4.4. A consolidated view of the simulation engine 100

Figure 4.5. The entity cycle diagram for the 'lorry' model 106

Figure 5.1. An overview of the prototype knowledge-based simulation model building environment 137

Figure 5.2. The entity cycle diagram for the partial 'lorry' model (merchant only, one weigh-bridge) 143

Figure 5.3. The entity cycle diagram for the partial 'lorry' model (merchant only, two weigh-bridges) 144

Figure 5.4. The entity cycle diagram for the partial 'lorry' model (merchant and ncb only, one weigh-bridge, mixed queuing) 146

Figure 5.5. The entity cycle diagram for the partial 'lorry' model (merchant and ncb only, one weigh-bridge, separate queuing) 147

Figure 5.6. The entity cycle diagram for the 'harbour_1' model 150

Figure 5.7. The scripts to provide an interaction between processes through message passing 153

Figure 5.8. The entity cycle diagram for the 'harbour_2' model 155

Figure 6.1. A Conceptual view of a simulation model at the generic level 201

Figure 6.2. An overview of 'WISE' (Warwick Intelligent Simulation Environment) 203

Figure 6.3. The entity cycle diagram for the sub-model 'lorry' 222

Figure 6.4. The Hierarchical structuring of a sub-models knowledge base 223

ACKNOWLEDGEMENTS

I feel greatly indebted to The British Council for their Fellowship Award to me which has enabled this project.

My first thanks should go to my Ph.D. supervisor, Dr. Robert Hurrion, for accepting to supervise this research, for his support during the project in the form of helpful discussions and comments on earlier drafts of this thesis, and for making available source code for his 'LEGO' system during early phases of this research.

I thank the following for their respective contribution in this project:

Prof. Rolfe Tomlinson for supporting my application to The British Council for extension in the duration of the Fellowship Award and also for permission to use a hard disk personal computer for this project for nearly a year.

Mr. Keith Halstead for making available the Pascal source code for an arithmetic statements interpreter.

The School of Industrial and Business Studies for allowing the use of an Olivetti M24 personal computer exclusively for this project and also for allowing use of photocopying, word processing and laser printing facilities within the School.

Lastly, I take this opportunity to offer my special thanks to my parents for their painstaking effort in bringing me up and for providing me with education and training. In many ways this thesis is as much their achievement as it is mine.

SUMMARY

The initial chapters of this thesis cover a survey of literature relating to problem solving, discrete simulation, knowledge-based systems and logic programming. The main emphasis in these chapters is on a review of the state of the art in the use of Artificial Intelligence methods in Operational Research in general and Discrete Simulation in particular.

One of the fundamental problems in discrete simulation is to mimic the operation of a system as a part of problem solving relating to the system. A number of methods of simulated behaviour generation exist which dictate the form in which a simulation model must be expressed. This thesis explores the possibility of employing logic programming paradigm for this purpose as it has been claimed to offer a number of advantages over procedural programming paradigm. As a result a prototype simulation engine has been implemented using Prolog which can generate simulated behaviour from an articulation of model using a three phase or process 'world views' (or a sensible mixture of these). The simulation engine approach can offer the advantage of building simulation models incrementally.

A new paradigm for computer software systems in the form of Knowledge-Based Systems has emerged from the research in the area of Artificial Intelligence. Use of this paradigm has been explored in the area of simulation model building. A feasible method of knowledge-based simulation model generation has been proposed and using this method a prototype knowledge-based simulation modelling environment has been implemented using Prolog. The knowledge based system paradigm has been seen to offer a number of advantages which include the possibility of representing both the application domain knowledge and the simulation methodology knowledge which can assist in the model definition as well as in the generation of executable code. These, in turn, may offer a greater amount of computer assistance in developing simulation models than would be possible otherwise.

The research aim is to make advances towards the goal of 'intelligent' simulation modelling environments. It consolidates the knowledge related to simulated behaviour generation methods using symbolic representation for the system state while permitting the use of alternate (and mixed) 'world views' for the model articulation. It further demonstrates that use of the knowledge-based systems paradigm for implementing a discrete simulation modelling environment is feasible and advantageous.

INTRODUCTION

The research described in this thesis was undertaken within the Operational Research/Systems Group of the School of Industrial and Business Studies under the supervision of Dr. R. D. Hurrion. The research started in October 1985 and this thesis marks its 'completion' in June 1989. The main concern of this thesis is Discrete Simulation Modelling.

The availability of inexpensive processing power in the form of micro computers has provided necessary encouragement for the use of discrete simulation for problem solving. Previously simulation has been regarded as a 'court of last resort', mainly because of its empirical nature, the amount of labour involved and the need for highly trained personnel for conducting these studies. This view has changed a great deal by the introduction of the visual interactive approach to Simulation which enables direct involvement of the decision maker in the simulation model development and also provides for interactive experimentation with these models. The use of an animated graphic trace provides a quicker verification of simulation models.

In the past, the conduct of simulation studies have been facilitated by special purpose software in the form of simulation languages and packages. The current trend in simulation software is to provide software tools for computer support in all phases of a simulation study and, with the help of suitable interfaces, to integrate such tools to provide integrated simulation environments. The main aim of such integration is to provide for the ease of conducting simulation studies and also to enable the conduct of simulation studies by non-simulation-experts [SHANNON, 86].

Concurrently with the ideas related to the development of integrated simulation environments have emerged two new paradigms related to computer software from research in the area of artificial intelligence. These are the Knowledge-Based Systems paradigm for software systems and the Logic Programming paradigm for computer programming. This thesis addresses the problems of implementing a prototype discrete simulation environment while using these new paradigms. Two areas of discrete simulation that have been concentrated upon are: (a) the generation of simulated behaviour from the articulation of a simulation model using alternate (or mixed) 'world views', and (b) simulation model

generation while using a knowledge-based systems framework. The logic programming language Prolog has been used for implementation throughout. The method employed for research has been mainly exploratory programming and prototype system implementation as is the case with most Artificial Intelligence related research.

Chapters 1 and 2 are intended to provide a perspective for the research described in later chapters and attempt a literature survey on Problem Solving and Discrete Simulation. These chapters undertake a review of the ideas related to the application of artificial intelligence techniques within expert problem solving and simulation. Chapter 3 looks somewhat more closely at the new paradigms of Knowledge-Based Systems and Logic Programming and argues in favour of exploring their use for providing 'intelligent' discrete simulation modelling environments.

Chapter 4 describes the research for devising a prototype simulation engine using Prolog as the implementation language. This simulation engine interprets the model code at run-time and is capable of driving simulated behaviour from articulation of model using three phase (i.e. events/activities) or process 'world views'. A sensible mixture of these two 'world views' is also supported. The need to support model articulation using alternative or multiple 'world views' was seen as a possible approach towards the goal of knowledge based simulation environments. The knowledge of the dynamic behaviour of systems in a given application domain can be more naturally captured as events, activities or processes [HURRION, 85]. The ability of a simulation engine to support multiple 'world views' would have direct relevance to the creation of a knowledge-based simulation modelling environment where the knowledge could be retrieved and assembled, without a 'world view' related transformation, into an executable simulation model directly.

Chapter 5 covers research for suitable knowledge representations to enable knowledge-based construction of simulation models. As a result of this research a prototype knowledge-based model builder has been implemented using Prolog. The working of this model builder has been demonstrated with the help of a number of examples.

Chapter 6 further develops the knowledge representations of chapter 5 to devise and implement a knowledge-based model acquisition system for interactively defining the simulation models. The later part of this chapter covers possible

generalisations which can be developed from the experiences gained from this research.

Chapter 7 concludes this thesis with the research findings of using Prolog in knowledge based environments and gives some ideas about further research.

A note on the citation of references. References cited within square brackets can be found in the section titled 'References'. Those references which have been cited by others within quotes have been enclosed in curly brackets and have been compiled in the section titled 'References Cited Within Quotes'.

A Bibliography at the end of this thesis represents the additional material consulted during this research.

CHAPTER 1: PROBLEM SOLVING

INTRODUCTION

This chapter aims to provide a perspective to the research described in this thesis. It takes a brief look at the problem solving activity at a general level while taking into account the role of previously accumulated application domain knowledge and problem solving methodology knowledge. The role of abstract knowledge and formal languages in relation to problem solving has been described and the use of Operational Research and Systems Analysis methodology and techniques in relation to managerial decision making has been briefly covered. The role of digital computers and computer programming languages for information processing during problem solving has been considered and Discrete Computer Simulation, as one of the Operational Research techniques, has been introduced.

More recent developments relating to problem solving computer systems (expert systems, knowledge-based systems), as have emerged from research in Artificial Intelligence, have been considered and different views relating to their relationship with Operational Research have been compiled. The relationships between artificial intelligence technology and discrete computer simulation have been taken up in more detail in chapter 2.

1.1. PROBLEM SOLVING IN GENERAL

1.1.1. PROBLEMATIC SITUATIONS

From what we already know about reality, and from what we currently observe we attempt to figure out if anything is going wrong or if any opportunities are being lost due to taking either a wrong action or not taking the right action of appropriate magnitude. If these 'symptoms' can be identified in a real life situation, it can be regarded as a problematic situation (e.g. [SIMON & DHPRSSITW, 87]).

1.1.2. PROBLEM DESCRIPTION

For the purpose of this thesis we shall take the view that a communicable description of reality which poses itself as a problem is the starting point of problem solving. Such a description shall be referred to as a problem description. It is only natural that the problem description is expressed in one of the 'natural' languages (e.g. Greek).

> "Regardless of the specific way in which a problem statement comes about ... it is conceptually useful to assume that there always exists such a statement (or its equivalent, from the point of view of information content and access) at the starting point of any problem-solving process. If the problem-solving activity requires a problem-acquisition stage whose end point is a problem statement that will govern the next stage of solution construction, it is useful to conceive of the situation as consisting of two well-defined problems: a problem-acquisition problem and a solution-construction problem."
> p 768;[AMAREL, 87]

A problem description therefore should indicate as to what information — directly observable, measurable or inferred — has led us to believe that we currently face a problematic situation, and the criterion which makes us believe so. The sorts of criteria which may be used include: our belief structure, utilization of resources, ecological reasons (e.g. pollution), better performance by competitors and so on.

The process of translating the problematic situation into problem description will be referred to as problem understanding.

1.1.3. THE SOLUTION

With reference to a problem description, a solution is a description of an achievable real life situation which we desire, envisage or hope for as a result of taking some form of corrective action. Generally, the specification of the corrective action is also considered as a part of the solution.

A solution is something we do not know directly and is something we seek to find. However, we know something about it. For example, when faced with the problem of planning a layout for a factory, we know the requirements which the layout must fulfil but we do not know the layout directly. If it is possible to describe the solution precisely by its characteristics then the problem is said to be well defined, otherwise, it is said to be ill defined. In case of ill-defined problems,

further exploratory work is generally needed to know more about the problem and the possible solutions that can be considered.

1.1.4. PROBLEM SOLVING

Once it has been possible to describe the problem and the various properties/features/characteristics/aspects of the solution we are looking for, then comes the task of determining a course of action about which we can claim that on implementation it will transform the current problematic situation into a situation which we presently describe and seek as a solution. This task we shall refer to as problem solving.

This requires bringing into play all the relevant pieces of knowledge we have about the situation under study and about the particular problem at hand in a suitable formation to bridge the gap between the problem description and the solution description. Particularly we must know what actions are applicable in the current situation and what effect each action or a sequence of actions will have on the situation and its successive developments.

The quality of knowledge we have therefore plays an important role in problem solving and the certainty with which each item of knowledge can be applied to the situation under study needs to be taken into account. We prefer our knowledge to be in as general a form as possible as this gives us the opportunity to apply it within the widest possible context. We also prefer our knowledge to be as 'fine-grained' as possible as this allows us to consider the problematic situations in greater detail. Sometimes when relevant pieces of knowledge are not available a resort has to be made to generate such unavailable knowledge by carrying out carefully controlled experiments.

It is therefore important that each experience of problem solving is well documented so that the knowledge is available for future problem solving situations. With an increasing body of knowledge it is a practical necessity that our knowledge base is partitioned for the convenience of learning and reference. It is out of this necessity that various disciplines of study have emerged. For example law, social sciences, physical sciences, engineering, technology, medical sciences, and so on, as they deal with problems of satisfying needs in the various dimensions of human life.

1.1.5. ABSTRACTION: REPRESENTATION IN FORMAL LANGUAGES

As noted earlier, knowledge about real life situations can be recorded using one of the natural languages. Words need to be chosen or invented to describe various elements of the situation and the way these are related to each other and the way they interact as a part of their behaviour. Natural languages suffer the limitation that their words do not always convey precise meaning and there is room for ambiguity. These are therefore not considered entirely suitable for describing our understanding of reality precisely with a view to problem solving. This is especially the case because we are always looking for procedural techniques for problem solving and, the problem description in a natural language is difficult to subject to known problem solving procedures, to say the least. These requirements on the use of knowledge have necessitated that the application domain knowledge is represented in a formal language to convey precise meanings whereas the problem solving knowledge takes the form of procedures to manipulate the knowledge thus represented (e.g. [NEWELL, 69]).

Formal languages are characterized by their finite vocabulary, precise meaning and precise rules of grammar for making legal constructs in those respective languages. It is not at all necessary that a formal language be restricted to words and sentences, it can consist of a set of symbols and rules (i.e. grammar) for combining these symbols to convey specific meanings. For example in Chemistry a set of symbols are used to represent elements, chemical reactions, the bondage of atoms in molecules and so on. Mathematics is another example of a formal language in which the quantitative relationships of real situations can be described. Various diagrammatic languages have been used to capture the system description such as entity cycle diagrams, petri-nets ...

The development of precise formal languages has also led to the development of a body of abstract knowledge which need not have any relationship with a specific real world situation (although abstract knowledge itself is a reality). Generally, the development of abstract knowledge begins with a set of definitions and axioms which intuitively we accept to be 'true'. For example with the introduction of simple ideas about a point, a straight line and a circle, the whole body of abstract knowledge called Euclidean Geometry has been developed. Abstract Knowledge is developed by using methods of logical inference and proof procedures to derive more abstract knowledge from the existing abstract knowledge. Many branches of mathematics like various algebras, Euclidean geometry, co-ordinate geometries,

differential calculus, integral calculus, complex mathematics, and so on, are examples of these bodies of abstract knowledge.

If it is possible to describe a real world problem in a language for which we have available a body of abstract knowledge, then such abstract knowledge (in the form of theorems, lemmas, ...) becomes directly applicable and can assist in arriving at a solution. For example, the language of geometry can be used to assist a land revenue department in assessing the tax for a piece of land and also it can assist an engineer to estimate forces in structural frames. Another example is linear programming. If it is possible to express the problematic situation as a linear objective function and a set of linear constraints, then it is possible to 'solve' the problem of optimizing the objective function. Such 'ready made' procedures based on abstract knowledge which transform the problem expressed in one form into a solution are referred to as problem solving techniques.

Not only does the abstract knowledge help us solve problems related to existing systems but it also assists us in the design of future systems. It is evident that by the use of abstract knowledge it has been possible to design and build systems of immense complexity. Examples are space missions, global communication networks, banking systems, and so on.

1.1.6. GENERAL FORMS

With the development of abstract knowledge in various languages it has been found to be convenient to describe this knowledge around some generalized forms. A general form provides a kind of template in which some of the aspects are made invariable whereas others may vary from one problem to another. For example in linear algebra we define a general form for a linear programming (LP) problem. This form requires an objective function and a set of linear constraints which constitutes the invariable part of the form. The part which can vary from one LP problem to another is the coefficients in the objective function, the number of variables, the number of constraints, the coefficients of the variables in the constraints.

An attempt is made to develop a general solution procedure related to each general form of the problem. For example a Simplex procedure is a general procedure for the solution of any problem which can be expressed in the general form for an LP problem. There may be more than one general solution procedure associated with a general form of a problem. Alternatively it may not always be

possible to develop a general solution procedure for a given general form. The claim for generality of a solution procedure related to a general form is backed by some form of proof that the application of the procedure to the problem description will always lead to the solution of the problem, provided one exists.

The knowledge of general forms and the related general solution procedures provide us with the necessary abstract conceptual framework with which we attempt to view problematic situations. If it is possible to express a problematic situation in a general form for which a general solution procedure is available then the problem is said to be well structured otherwise with the current state of our knowledge it is said to be ill structured.

Newell has described this activity in the following words:

> "We observe that on occasion expressions in some language are put forward that purport to state 'a problem.' In response a method (or algorithm) is advanced that claims to solve the problem. That is, if input data are given that meet all the specifications of the problem statement, the method produces another expression in the language that is the solution to the problem. If there is a challenge as to whether the method actually provides a general solution to the problem (i.e., for all admissible inputs), a proof may be forthcoming that it does. If there is a challenge to whether the problem statement is well defined, additional formalization of the problem statement may occur. In the extreme this can reach back to formalization of the language used to state the problem, until a formal logical calculus is used."
> p 361; [NEWELL, 69]

1.1.7. SOLUTION PROCEDURES

The general solution procedures which are backed by a proof are called algorithms. However, where adequate theory does not exist to provide the necessary proof but a procedure is intuitively known to provide an acceptable solution, the procedure is known as a heuristic. (Simon and Newell, 1958) have identified heuristics as appropriate for ill-structured problems and algorithms as appropriate to well-structured problems [FORDYCE & NS, 87].

(A) ALGORITHMS

(Raitman, 1964) has pointed out that the existence of an algorithm presumes: (1) an explicitly specifiable class of problems all of which may be solved by (2) the

program for the algorithm to (3) some well-defined criterion for solution, [FORDYCE & NS, 87].

(B) HEURISTICS

[Beltrami and Bodin, 1974] has defined heuristics as follows: 'Think of heuristic reasoning as meaning that one brings to bear as much intuition, and as many plausible arguments, as possible on problems which are either computationally intractable, or for which inadequate theory exists.' [FORDYCE & NS, 87].

1.1.8. PROBLEM FORMULATION

The formulation of a problem in a formal language represents our understanding of the problematic situation and our approach to its solution procedure. So far there is no general procedure for formulating problems and this area is regarded as an art rather than a science. A number of 'tactics' are used during the formulation phase which include simplifying the problem by the use of assumptions and introducing various levels of representation and interpretation.

The introduction of simplifying assumptions amounts to saying that the analyst knowingly solves a simpler problem (because he is limited either by the availability of technique(s) for solving the full problem or if available, their application is not cost effective). He can then amend the solution thus obtained in the light of the assumptions he has made (by making use of judgment or by additional computation).

Representation is used to describe the problem in terms of the theory on which a known problem-solving technique is based. A solution has to be interpreted (i.e. translated back from the formalism in which the problem was originally represented).

1.1.9. METHODOLOGY IN PROBLEM SOLVING

Having considered the role of abstract (theoretical) knowledge in problem solving we now turn our attention to the knowledge which relates to the generalizations of methods used within problem solving itself. These generalizations are applicable at the appropriate stages of both real world problem solving and of the development of a theory (i.e. abstract knowledge) since the development of a theory itself can be regarded as 'a problem'.

In another sense methodology also implies a criteria which has earned some credibility to be effective in dealing with problems and therefore should be complied with e.g. scientific method.

(A) METHODS WHICH CHARACTERIZE THE APPROACH TO PROBLEM SOLVING

(a) Top-Down and Bottom-Up Approaches

When we start the process of problem solving from our knowledge of the solution (i.e. goal) and work backwards through subgoals until we arrive at the existing situation, this approach is known as top-down. Alternatively when we proceed from the current problematic state and proceed towards the solution the approach is known as bottom-up. For more detail see p 7-8;[KOWALSKI, 79]

(b) Problem Reduction (Dealing with complexity)

While dealing with complex problematic situations (ones which do not lend themselves directly to be formulated within a general form for which we have a general solution procedure available) the problem needs to be divided into smaller more manageable sub-problems. This approach to problem solving is called problem reduction.

> "When we are confronted with a complex system and have decided upon a way of looking at this system, we may ask: how are its components to be identified? There is no unique answer to this question. Sometimes the answer is evident, sometimes it is a matter of taste, and at other times the selection of suitable components is a crucial point in the analysis."
> p 23;[BIRTWISTLE & DMN, 79].

(B) METHODS WHICH ALLOW US TO GENERATE MORE KNOWLEDGE FROM OUR EXISTING KNOWLEDGE

(a) Induction

Induction is inferring a general principle from a set of examples.

(b) Deduction

Deduction is inferring a conclusion (specific or general) from known facts and general principles.

(c) Abduction

Abduction is formulating a hypothesis about a law governing an observed phenomenon.

(C) METHODS WHICH RELATE TO ASCERTAINING OUR KNOWLEDGE

(a) Hypothesis Testing (Experimenting)

The knowledge which is generated needs to be tested before it can be accepted generally. This is done by performing experiments, whose outcome needs to be consistent with our expectations if the knowledge which we have generated is true.

1.1.10. THEORIES OF PROBLEM SOLVING

Having made use of abstraction in problem solving it is natural that some attention has been paid to develop an abstract theory for the problem solving phenomenon itself. [WICKELGREN, 74] describes elements of a theory of problem solving. Encyclopedic entry [AMAREL, 87] presents a formal abstract view of problem solving from the point of view of creating problem solving systems. [KOWALSKI, 79] compares the models of problem solving developed in cognitive psychology and artificial intelligence and argues in favour of logical inference as the general model for problem solving. [RICH, 83] presents a view of basic problem solving methods and representation schemes used in artificial intelligence and related solution procedures.

1.1.11. CLASSES OF PROBLEMS

When confronted with a problematic situation it is helpful to recognize it as one of a broad category of problems for which we have some accumulation of problem solving knowledge available. Apart from more usual classification — such as physical sciences, social sciences, and so on — a functional classification with a particular view to applicable problem solving methods has also evolved. For example a given problematic situation may be classified as a diagnostic problem, a design problem, a planning problem, a control problem and so on. Such classification is based on the nature of the problem and general similarities in their solution steps. For example, a doctor trying to diagnose an illness of a patient and an engineer trying to figure out what is wrong with a particular piece of machinery follow similar methods. Each may be following a binary search

strategy to narrow down the area in which the problem exists. Each may be hypothesizing and then making some tests to either confirm the hypothesis or rule a possibility out. This is so because both are working on a diagnostic problem. Such functional classification is a convenient way of classifying our problem solving knowledge at a functional level. Such a classification does not imply well defined general solution procedures, such as discussed previously, but only similarities of patterns of problem solving methods.

1.1.12. PROBLEM SOLVING PARADIGMS

The following problem solving paradigms are relevant to this thesis:

Scientific Method

Operational Research and Systems Analysis

Artificial Intelligence (Expert Systems)

1.2. OPERATIONAL RESEARCH APPROACHES TO PROBLEM SOLVING

1.2.1. PROBLEM SOLVING WITHIN ORGANIZATIONAL DECISION MAKING

Organizational decision making is characterized by an organizational structure with functional and hierarchical relationships, a communication/reporting protocol, operating procedures, organizational objectives and the environments within which the organization operates. Actions are taken in accordance with the decisions made by the decision makers within the organization. As an organization operates in real time, it is important to consider the times taken by the decision makers to sense the problematic situation, the time taken to decide upon a future course of action and the time taken to implement the decision. [TOMLINSON & D, 83] has proposed a feedback control model for strategic management which takes into account these time factors.

During the course of organizational decision making considerable knowledge is generated and the accumulation of such knowledge can be used in forecasting the future and then planning the organizational objectives, and course of actions accordingly. The forecast and the plans serve as a reference against which the actual performance is compared to sense if there is any cause to react. [COOKE & S, 84] provides a comprehensive coverage of managerial decision

making within an organizational context from a number of angles of views in addition to the modelling approach taken by Operational Research.

> "... generic definition of term 'model', given first by [Minsky, 65]
>
> 'An object 'A' is a model of an object 'B' for an observer 'C' if the observer can use 'A' to answer questions that interest him about 'B'.'"
> p 189;[OREN, 82]

The idea of using abstract models has existed in physical sciences for a long time but its application in decision making, initially to the war time problems, marked the beginnings of Operational Research [WHITE, 85]. Later on the ideas from Systems Analysis (the theory and methods developed to deal with the study of complex systems and problem solving therein) which again had their beginnings in the physical sciences were also introduced in the organizational decision making. These ideas were also employed later on for decision making within commercial organizations and governmental departments other than defence. Since then these ideas have grown into a discipline of study with its own set of methodologies and techniques to deal with decision making situations within the organizational context.

> "... OR has developed in two ways: first as an approach to aiding management decisions through modelling; second, by producing powerful standard models and methods to fit some well defined commonly encountered classes of decision."
> p 145;[COOKE & S, 84].

1.2.2. METHODOLOGY OF OPERATIONAL RESEARCH

(A) SCIENTIFIC METHOD AND OPERATIONAL RESEARCH

Initially, Operational Research and Systems Analysis was equivalent to the application of a scientific method to organizational decision problems. In recent years a large amount of experience has been gained from using this approach to organizational decision problems and consequently this view is changing. A number of papers in [Tomlinson & K (eds), 84] particularly concentrate on this issue, i.e. the evaluation of methodologies from past practices and their outcomes. More recently Simon has stated this issue in the following words:

> "The real world of human decisions is not a world of ideal gases, frictionless planes, or vacuums."
> [SIMON & DEPREETTH, 87]

(B) MULTI-DISCIPLINARY APPROACH IN OPERATIONAL RESEARCH

> " ... and, indeed, our whole process in Operational Research is a synthesis of various known, or at least accepted, hypotheses combined to produce a composite prediction." p 3;[WHITE, 85].

Operational research has drawn upon various disciplines for its problem solving knowledge. Such knowledge includes both the methodology and the theory on which various techniques have been based. These include linear algebra, probability theory, statistics, kinetic theory of gases, and so on. In effect the operational research approach tends to integrate and further generalize the problem solving knowledge already available in other disciplines.

(C) OPTIMIZATION

In Operational Research a great amount of effort has been spent on developing and employing optimization techniques to arrive at an optimum solution. Experience has shown that the meaning of optimum is only related to a particular formulation and is not absolutely related to reality. In recent years there has been more emphasis on achieving a 'satisfactory' solution rather than an 'optimum' solution.

1.2.3. TECHNIQUES OF OPERATIONAL RESEARCH

The techniques of operational research are generally classified as deterministic and stochastic. Deterministic techniques do not take into account the probabilistic variation in various elements of the model and mainly resort to mathematical forms of reasoning. Stochastic techniques on the other hand take into account the possible variability of model parameters and measurement precision. These techniques are mainly based on probability theory and statistical forms of reasoning. Further, a set of techniques is based on fuzzy forms of reasoning.

An application oriented also prevails. For example techniques related to project planning, scheduling, allocation, distribution, and so on

1.2.4. DECISION-MAKER AND THE ANALYST

Typically in an organization a decision maker (a line person, as opposed to staff) is charged with the responsibility of making decisions for taking actions, whereas an

analyst (a staff person) has the responsibility to provide the necessary support during the process of decision making.

It can be said that the decision maker on his own, or with the assistance of analyst brings about a declarative formulation for a decision problem. Whereas the role of the analyst is to make use of specialized problem solving knowledge to transcribe the declarative formulation into a procedural formulation, so as to provide a solution to the formulated problem.

1.3. THE USE OF COMPUTERS IN PROBLEM SOLVING

1.3.1. SOLUTION PROCEDURES APPLIED BY HUMANS

When solving a specific problem which has been formulated in one of the general forms for which a solution procedure is available, that solution procedure needs to be carried out to obtain the solution. For example, the solution procedure may involve drawing a geometrical construction to scale and then measuring off the solution (e.g. some problems in statics, dynamics, ...). For other problems it might involve performing some arithmetical operations on numbers which have a specific meaning for the problem in hand. For more complex and lengthy procedures some form of ready reckoners in the form of tables of values have been used. (e.g. logarithmic tables, trigonometric tables, ...).

1.3.2. THE USE OF DEVICES

To further ease the burden of calculation (with some sacrifice of accuracy) various devices have been invented which represent numerical quantities on logarithmic and other scales for manipulation. For example, in a slide rule these scales can be aligned and the result can be read off from the alignment. In specialized areas, such as weaving looms, musical instruments, it was also possible to store the procedure itself (e.g. a weaving pattern) which is acted upon by the machine at the time of weaving. In general, a person skilled in the use of these techniques is required to perform the procedures.

1.3.3. ANALOGUE ELECTRONIC DEVICES

Advancements in electronics have led to interesting developments in which it has been possible to represent various quantities and their relationships in terms of

their electronic analogues, as in abstraction their behaviour can be expressed by the same general form. This enables us to explore the behaviour of a system as a whole. This is in contrast with the problem reduction approach in which only a part of a larger system is explored at a time. This type of study was not possible previously because we were limited in our capability to integrate a large number of components and their interrelationships in one model.

1.3.4. DIGITAL COMPUTERS IN PROBLEM SOLVING

(A) GENERAL PROCEDURES TO EXPRESS GENERAL FORMS

The invention of the digital computer has opened up a multitude of exciting avenues in terms of our information processing capability for problem solving. Software has enabled us to build layers of abstraction on top of the basic primitive operations available at the machine level. In addition to the high speed of its operation it provides the necessary infra-structure for expressing general forms, which can be used directly for the solution of a problem complying with the requirements of the general form. These general forms are procedural in nature. For example a general procedure for solving polynomial equations can be written, which will accept the order of the equation and the values of the coefficients and then produce the solutions to the desired accuracy within a matter of seconds.

(B) PROCEDURAL LANGUAGES FOR COMPUTER PROGRAMMING

In order to express the general forms as procedures, procedural languages have been designed which provide the necessary elementary procedural constructs such as sequential processing, unconditional branching, conditional branching, different types of looping, which can be assembled serially in the form of a program. The development of computer languages and the study of the related grammars have attracted considerable attention and these are the major concern for research in computer science. There is a large variety within procedural languages available based on the general forms in which programs can be expressed. For example languages providing the facility to define one's own types for data and related operations, strongly typed languages, type-less languages, block structured languages, object oriented languages, a variety of subprogramming facilities and the way subprograms are invoked, different types of memory management models, co-routining facilities and so on.

(C) SYMBOLIC PROCESSING

Initially, in line with problem solving techniques, computer programming languages were designed to express problem solving procedures mainly of arithmetic nature e.g. FORTRAN. The design and implementation of languages themselves led to the ideas of symbolic processing. This type of processing has made it possible to write software which enables mathematical transformation operations to be performed on symbolic expressions (such as differentiation, integration, Laplace transformation and so on). The ability of symbolic processing reduces the need for coding the information as numbers (e.g. female = 0, male = 1).

(D) NON-PROCEDURAL LANGUAGES

More recently, work in the area of Artificial Intelligence has led to the development of declarative (i.e. non-procedural) languages. In contrast with the procedural languages where our problem solving knowledge must be translated into procedures, declarative languages permit the expression of this knowledge in the form of a set of facts and rules. The language facilities themselves proceed to select relevant pieces of declarative knowledge when working on the solution of a particular problem. The development of declarative language has enabled the implementation of operations used in logic such as matching and unification. Declarative languages bring computer based problem solving to a higher level i.e. the human readable description of problem solving knowledge can be directly acted upon by the computer system to solve a specific problem without going into the details of procedural coding, which is highly error prone and time consuming.

(E) COMPUTER PROGRAMMING AND PROGRAMMING ENVIRONMENTS

Computer programming has also developed mainly as an art. Various principles have evolved for efficient program development (e.g. structured programming techniques, top-down decomposition of large programs, documentation standards, and so on). Various support and productivity tools have also been created which help the programmers in the course of the development of software. There has been a trend towards combining languages, support and productivity tools into integrated programming environments. For larger software projects the discipline of software engineering has also evolved.

(F) DOMAIN SPECIFIC PROBLEM SOLVING SYSTEMS

In addition to the development of languages for expressing general forms as procedures, it has been possible to actually write such procedures for a given domain of problems and integrate them in the form of a problem solving system for that domain. Examples are maintenance management systems, project management systems, medical diagnostic systems, pattern recognition systems and so on.

1.3.5. THE USE OF COMPUTERS IN OPERATIONAL RESEARCH

(A) MECHANIZATION OF SOLUTION PROCEDURES

> "Initially, the complexity of the problems that OR could handle was severely limited by the computational tools that were available: paper, pencil, and desk calculators. Many real-world problems for which the OR tools were otherwise appropriate were ruled out by the limits on computation. These limits were dramatically altered by the modern digital computer, which began to find a civilian market about 1950. With computers available, if the sky was not quite the limit, the ceiling was now very high; and it became higher each year as computers became larger and more powerful and as researchers in OR addressed the problem of improving the computational efficiency of the tools."
> p 9;[SIMON, 87].

Numerical procedures, like evaluating integrals, solving differential equations numerically, numerical optimization techniques, have also been implemented, since such procedures already existed.

> "In addition to its invaluable contribution to computation, hence to the complexity of the problems that could be solved, the computer gave OR a new tool of analysis: simulation or modeling. Now, problems that were analytically intractable and not amenable to optimization could be approached by simulating system behavior numerically."
> p 9;[SIMON, 87].

The solution procedures which are not analytic in nature (e.g. procedures based on the representations of graph theory) have also been implemented.

(B) COMPUTER SIMULATION

The Computer has not only provided the means of implementing the solution procedures already discovered theoretically, but also has provided the facilities

for modelling itself. With the suitable representation of the system's components and their interaction with each other during the course of its operation, it is possible to build a computer model of the system under study. Such a model can be 'run' to mimic the behaviour of the system over time. Although running the simulation model itself does not provide the solution to the problems related to the system, it provides a source of information about the performance of the system, which otherwise may not have been available or may have required years to obtain from the study of the actual system. With suitable alterations in the parameters of the simulation model a whole performance space can be generated by running the model after each alteration. Having obtained this information from simulation runs, the problem solving is then a matter of making a choice.

A validated computer simulation model provides a suitable test ground for testing hypotheses about the system without affecting the actual system. Such experimentation results from our formulation of meta-models about the system and the computer simulation model is parameterised/formulated in response to the specific demands of the meta model formulated. For extensive discussion on experimentation with simulation models see [KLEIJNEN, 87].

Also, Fig. 1.1. (from [KLIR, 79]) shows the use of simulation modelling within the context of problem solving using a system theoretic approach.

(a) The Types of problems for which simulation is a suitable tool

> "[Karplus, 1976] has shown that simulation can only solve three (interrelated) fundamental types of problems. These are:
>
> a. Assuming that knowledge is available on a system and its input vector, compute its resulting output characteristics (Analysis/Prediction)
>
> b. Assuming knowledge of input vector and resulting output characteristics, establish the nature of the system (Synthesis/Identification)
>
> c. Assuming knowledge of the system and the (desired) output characteristics, compute the corresponding (necessary) input vector (Management/Control).
>
> (Note that the term vector is used here for a set of time-trajectories)."
> p 7;[KLIAS, 80]

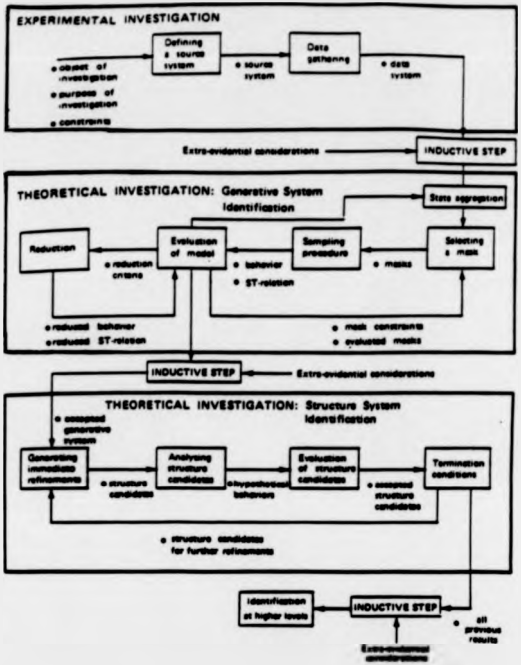


Figure 1.1. Summary of a package of methodological tools for systems modelling (from [KLIR, 79];p 13)

"The nature and fundamental aspects of simulation also depend to some extent on the purpose of the exercise. Those who work in the field of Artificial Intelligence often speak in this context about working modes. These modes do not only describe the way in which the computer is used, but also the relation between this use and human activity. Three working modes can be recognized (Weizenbaum 1976):

Simulation Mode:	understanding/designing/planning by imitation
Performance Mode:	designing/controlling based on whatever (goal-oriented) principle one can discover
Theory Mode:	designing (abstract) theories based on analogies (practical entities being mere models of theories).

"It can be stated that, just as the inventions of the early pioneers in technology, the origin of simulation and modelling hails from imitation of apparent reality."
p 7;[ELIAS, 80].

1.3.6. TOWARDS ARTIFICIAL INTELLIGENCE

The role of the computer in problem solving has extended from the mechanization of solution procedures and simulation, to the very processes by which theory is developed namely mechanical theorem proving [BUNDY, 83]. Although thinking about discovering general procedures for theorem proving is not new, the digital computer has provided a fresh impetus to this line of research and as a result theorem proving systems have been developed.

Further, a new logic programming paradigm has also evolved, which has logical operations like matching, unification, and deduction as its primitives.

"One of the key software advances to come out of expert systems is non-procedural or declarative programming (Schor, 1986); (Hong, 1986). It is the inference mechanism (IM) concept which makes this possible

"Non-procedural programming is writing a program by just stating the appropriate set of if/then statements without concern for execution order or which if/then statements are applicable to different situations. A controlling function determines which statements to use and their order. Although this method is not workable across all programming situations, it is often applicable to expert system situations."
p 70;[FORDYCE & NS, 87]

"By using predicate logic statements, we can write rules that are more general than those already described. This type of rule building is called logic programming [Dahl, 1983]. Prolog [Clocksin and Mellish, 1984] is the most popular language for building these types of rules. But it is straightforward to build such structures in other languages like LISP and APL2 [Brown et al., 1986]."
p 71;[FORDYCE & NS, 87]

1.4. ARTIFICIAL INTELLIGENCE APPROACHES TO PROBLEM SOLVING

Artificial intelligence is an umbrella term covering the study of the following with a view to developing computational models for these:

- Natural language understanding
- human perceptions of hearing, vision.
- human capability of learning
- human capability of problem solving and related reasoning processes.

Fig. 1.2 shows that problem solving is the common area of interest between operational research and artificial intelligence.

[Shapiro (ed), 87] provides a comprehensive collection of knowledge about artificial intelligence in the form of an encyclopedia.

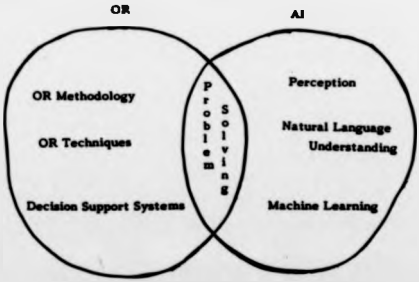


Figure 1.2. Problem Solving can be regarded as the common area of interest between Operational Research and Artificial Intelligence.

1.4.1. PROBLEM SOLVING AND ARTIFICIAL INTELLIGENCE

> "Problem solving is the central phenomenon studied in AI ... A major goal of AI is to develop and study problem-solving systems that are computationally efficient and that are effective over a broad range of problems."
> p 767;[AMAREL, 87]

The ability to construct reasoning systems has motivated even more general representations of problems together with more general (possibly weaker) solution procedures. In one sense it means that we can adopt a problem oriented approach to problem solving rather than a technique oriented approach which has prevailed.

> "... artificial intelligence is the application of methods of heuristic search to the solution of complex problems that:
>
> (a) defy the mathematics of optimization,
>
> (b) contain non-quantifiable components,
>
> (c) involve large knowledge bases (including knowledge expressed in natural language),
>
> (d) incorporate the discovery and design of alternatives of choice, and
>
> (e) admit ill-specified goals and constraints."
> p 11;[SIMON, 87].

1.4.2. THE FORMULATION OF PROBLEMS IN AI

The principal form used within AI for expressing a problem is by representing it as a state space. Such a state space can be generated from an articulation of the problem at the time of solving the problem or it can be made available in the form of a knowledge-base (e.g. production rules) or a combination of the two.

1.4.3. PROBLEM-SOLVING PROCEDURES USED IN AI

Corresponding to the representation of problems as a state space, the solution method principally consists of searching this space to arrive at the solution to the problem in hand. State space may be searched by using a 'blind' search or by employing some form of heuristic method to make the search more efficient.

1.4.4. KNOWLEDGE-BASED SYSTEMS (EXPERT SYSTEMS)

During the early AI research the emphasis was on the generalized representations of problems and the generalized solution procedures. Attempts at the incorporation of domain knowledge in problem solving met with greater success and this gave rise to what is now known as Intelligent Knowledge Based Systems or more popularly Expert Systems.

Expert systems have been seen to be useful and economical as reported by [FORDYCE & NS, 87]:

> "Expert systems have value because
>
> 1. they capture, refine, package, and distribute expertise, and
>
> 2. they solve problems whose complexity (reasoning) exceeds human ability, or the required scope exceeds any individual's."
> p 67;[FORDYCE & NS, 87]

> "Expert systems ... are economical because of four characteristics: they are
>
> 1. relatively simple to create
> 2. self documenting
> 3. capable of significant levels of adaptation to new situations, and
> 4. can explain how they arrived at their recommendation.
> p 75;[FORDYCE & NS, 87]

1.5. TOWARDS ADAPTING ARTIFICIAL INTELLIGENCE TECHNOLOGY IN MANAGEMENT SCIENCE / OPERATIONAL RESEARCH

In Simon's words:

> "Instead of differentiating between OR and AI, we need to confuse, blend, and synthesize them as much as possible. We need to build our professional institutions and organizations to use them together, supporting, reinforcing, and extending each other."
> p 11;[SIMON, 87].

1.5.1. THE EARLY HISTORY OF OR AND AI

[SIMON, 87] has reviewed the early history of both Operational Research and Artificial Intelligence. [FORDYCE & NS, 87] has identified that intelligent

systems have a strong but less publicised root in the Management Science and Operational Research field. Also in the words of Simon:

> "After about 1960, AI and OR went their separate ways; whole new generations of scientists trained in each of these disciplines were largely unacquainted with the techniques provided by the other. Only when AI builders of expert systems began, about 1980, to invade the field that had traditionally been occupied by OR did each group begin to be aware again of the existence of the other."
> p 10;[SIMON, 87].

As noted earlier in this chapter, Operational Research has a history of adapting and enhancing tools from other disciplines [WHITE, 85] and so the developments in the area of Artificial Intelligence can not be ignored [FORDYCE & NS, 87].

> "... We should aspire to increase the impact of MS/OR by incorporating the AI kit of tools that can be applied to ill-structured, knowledge-rich, non-quantitative decision domains that characterize the work of top management and that characterize the great policy decisions that face our society."
> p 8;[SIMON, 87]

1.5.2. THE POTENTIAL APPLICATIONS

[SIMON, 87] has identified some of the specific classes of managerial decision problems in which artificial intelligence technology can be applied advantageously. These include: (a) the approach to problem solving, i.e. being able to take a problem oriented point of view as opposed to a technique oriented view, thus making use of both algorithmic and heuristic problem solving knowledge, (b) the symbolic processing nature of AI tools can assist the MS/OR scientist in building effective decision aids for top management, where a large proportion of the information dealt with is non-quantitative (e.g. English sentences), and (c) the AI technology of heuristic search is well suited to addressing design problems which would include generating decision choices for which the technology did not exist before and where the generation of decision choices depends mainly on the creativity of the human brain. And finally:

> "Ultimately this domain knowledge will enable the computer to 'understand' the data and problem well enough to reduce the flow of information to the decision maker to a structured series of available alternatives with an explanation of their consequences."
> p 75;[FORDYCE & NS, 87]

1.6. CONCLUSION

Operational Researchers and Management Scientists need to take an active interest in the emerging Artificial Intelligence technology with a view to its utilisation in the designing of more powerful computer systems for managerial problem solving and decision support.

END OF CHAPTER 1

CHAPTER 2: A REVIEW OF SIMULATION MODELLING

INTRODUCTION

This chapter continues with the theme of the previous chapter but narrows its focus on the aspects of discrete simulation modelling which relate to the research described in the later chapters. The initial sections attempt to provide the reader with a scenario of discrete simulation modelling with particular emphasis on the simulated behaviour generation methods. The practice of computer simulation is naturally influenced by the developments in the area of computer science. Such influences have been given a brief coverage and current trends in simulation software have been described. Finally, a review of the literature which relates simulation with artificial intelligence has been undertaken, as it is the main concern of this thesis. It must be added that most of this literature became available only when most of the implementation work described in chapters 4, 5 and 6 had been completed and is included in the review for the sake of completeness.

2.1. ABOUT SIMULATION

2.1.1. DEFINITIONS OF SIMULATION

In 1979 Pritsker compiled 22 definitions of simulation by various authors, [PRITSKER, 79], and there has been more since then. This shows the multiplicity of views about simulation and its evolving nature as the practice of simulation is becoming more prevalent, due to of computer processing power becoming cheaper, and the advancement in simulation methodology and available simulation software.

One reason for many of the definitions of simulation is that it is used in many subject areas:

> "... simulation is common practice in many fields of science, which all claim ownership rights to this speciality. To mention but a few: cybernetics, general systems science, engineering (especially control-),

management science, operations research, econometrics and - last but not least - physics."
p 5;[ELIAS, 80]

It is instructive to have a look at some of these. In mathematics, for example, simulation covers methods for the evaluation of integrals and the solution of differential equations by numerical means. In engineering, simulation covers the finite element techniques for the modelling solids under stress and the flow of fluids; the simulation of physical systems or their abstract formulations e.g. control systems, flight simulators, communication networks, distributed computer systems. Other areas include simulation of global weather conditions, world trade modelling, battlefield simulations, and so on.

2.1.2. THE PLACE OF SIMULATION IN THE PROBLEM SOLVER'S TOOLBOX

Since computer simulation has taken many forms in various disciplines, each has its own domain specific view about it and this somewhat tends to obscure the fundamental nature of its approach. Simulation has been used for solving problems which cannot be formulated in one of the general forms for which an algorithmic solution procedure is available, in the form of an algorithm or a heuristic. We have an option either to develop the theory well enough to be able to devise a general procedure for its solution or to resort to the computation intensive technique of simulation. An example is the evaluation of an integral which does not lend itself to being solved by the use of a formula. The simulation approach to the evaluation of the integral proceeds with the evaluation of the area under the curve in a piecewise fashion using an interval small enough that the accuracy is at an acceptable level and large enough that the computation costs are also at an acceptable level. This idea of proceeding from an initial state of the 'system' (i.e. the function to be integrated) and generating successive 'system states' in subsequent intervals for data collection (cumulative area) is fundamental to simulation. This feature characterizes simulation from other forms of modelling.

Unlike analytical techniques the information obtained from a simulation run is not of general applicability because the results are specific to the initial conditions and the terminating limit. Some level of generalization is, however, possible from the results of many simulation runs, each with different parameters, if the results (or any transformation of them) depict any consistent recognisable pattern. Whereas algorithmic or heuristic procedures are applicable to the aggregate

quantities in the given decision situations (e.g. monthly production), simulation requires an understanding of the mechanics of the operations in order to produce a valid simulation model for the problem in hand.

Due to these aspects of using simulation and the fact that simulation is a computation intensive technique, it has been labelled as 'a technique of last resort':

> "The reason for this experimental approach, as opposed to more 'scientific' analytical techniques, is the complexity of the situation at hand. Operational researchers sometimes refer to simulation as the technique of last resort (to be used when all else fails)."
> p3;[VAUCHER, 85].

However, with the availability of inexpensive processing power in the form of the computer (microcomputers) alongwith the advancement in simulation software, this view is changing.

With the more recent systems approach to management problem solving the use of simulation has further been emphasised. When all the relevant components of a system are put together in one model the complexity of the model gets well beyond the limits which analytical techniques can handle.

> "Therefore the choice for systemic management approaches is a necessary one. Maybe the most important facet of this approach is the setting of a course only after rational evaluation of the effects of alternative courses. This implies the simulation connection."
> p 5;[ELIAS, 80]

2.1.3. THE DIMENSIONS OF SIMULATION

[SISSON, 69] has identified the following dimensions of simulation:

Static — Dynamic
Aggregate — Detailed
Physical — Behavioural
Computer — Human
Recursive — Quasi-equilibrium
Continuous — Discrete
Size of Time Quanta
Deterministic — Stochastic.

This thesis will be concerned with the type of simulation which is Dynamic, Detailed, Physical, Computer, Recursive and Discrete. The dimensions not

mentioned are considered to be not directly related to this thesis (this, however, should not be taken to mean that they are not important).

2.1.4. THE APPLICATION AREAS

The following classification of papers in [UKSC, 84] provides a representative sample of the application areas where simulation has been used:

Simulation for Policy and Planning
Biology and Medicine
Control Systems Simulation
Manufacturing Systems Simulation
Simulation in Education
Simulation of Electronic and Computer Systems
Real Time Applications

2.1.5. PROBLEMS WITH THE USE OF SIMULATION

Highly skilled and trained experts are needed to conduct simulation studies which makes simulation an expensive tool:

> "Today, in order to use simulation correctly and intelligently, the practitioner is required to have expertise in a number of different fields. This generally means separate courses in probability, statistics, design of experiments, modeling, computer programming and a simulation language. This translates to about 720 hours of formal classroom instruction plus another 1440 hours of outside study (more than 1 man-year of effort) and that is only to gain the basic tools. In order to really become proficient, the practitioner must then go out and gain real world practical experience (hopefully under the tutelage of an expert)."
> p 152;[SHANNON, 86].

2.2. THE EXPERIMENTAL FRAME

From a more practical point of view:

> "Simulation is experimentation with dynamic models. Simulation of systems necessitates (1) a model which can be conceived as a pair of parametric model and a relevant parameter set, (2) experimental conditions, and (3) a behaviour generator. During a simulation run, the behavior generator drives the (model, parameter set) pair under given experimental conditions to generate model behaviour which can be trajectory behavior or structural behavior"
> p 3;[OREN, 86].

As a computer simulation model is constructed within an overall framework for solving specific decision problems related to the system under study, fig. 2.1. (from [SOL, 86]), the simulation model needs to be general/generic enough that it is capable of accepting changes in the decision parameters and then producing the system behaviour accordingly. In this sense the information generated from running a simulation model is used to define a search space to be explored by the analyst, to arrive at a requisite solution to the problems in hand or to carry out further experimentation. Such searchs can be performed intuitively, by using heuristic methods or be based on meta models as noted in chapter 1. The search strategy needs to be specified in advance of the construction of the simulation model. This is also referred to as experimental frame.

> "If we think of the modelling process as part of the overall process of decision making, then the objectives derive from requests by decision makers for model with which to assess the efficacy of proposed policies. ('Decision maker' and 'policy' are intended here in a broad sense to include engineer, designer, manager, etc., and implementation, design, tactic, strategy, etc., respectively.) Such objectives are supposed to be formulated as series of questions regarding a real system or its components and ultimately to be formulated as experimental frames. An experimental frame is a specification of the kind of data a model should produce in order to answer the questions of interest. The concept however must be meaningful both for real system as well as model experimentation since in principle the same data could be obtained from the real system (although there are many reasons why models are preferred in practice)."
> pp 29-30; [ZEIGLER, 80].

There can be a number of reasons why the experimentation is preferred on a simulation model rather than on the real system itself and most textbooks on simulation provide a list of these (e.g. [PIDD, 84], [LAW & K, 82], [FISHMAN, 78]).

A simulation model needs to be validated and its credibility established before any confidence can be placed on the information it generates and the experimental frame plays a major role in this.

> "It should be noted that the experimental frame is key concept in model assessment since model validity is properly formulated as a relation involving a model, a real system, and a frame in which the behavioral data of the two are compared."
> p 31;[ZEIGLER, 80].

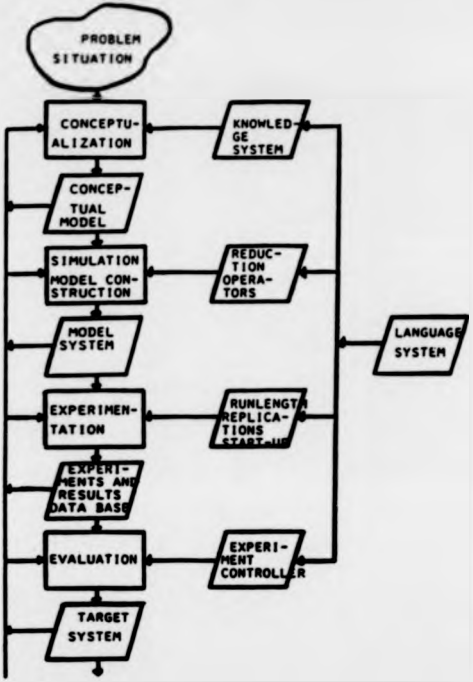


Figure 2.1. "A framework for simulation" (from [SOL, 86] p 359).

2.3. SIMULATION MODELLING

2.3.1. SYSTEM DESCRIPTION

Depending on the decision problem in hand and the system under study, an analyst decides on the aspects of the system which are relevant and must be incorporated in the model. At the first level of abstraction these are put together in the form of a system description. Such a description needs to be in a formal language to precisely capture the system components and their static and dynamic interrelationships. Such a description may be narrative or diagrammatic or a mixture of the two. A number of formal languages are in use for this purpose. Examples are entity/activity cycle diagrams [CLEMENTSON, 80], petri-nets [D'ANGELO, 83], queueing networks, system theoretic representations [ZEIGLER, 84]. System description languages have also been developed which serve as pseudo-code as well as documentation. Examples include process oriented simulation model specification and documentation language [FRANKOWSKI & F, 80], DELTA description language [HOLBAEK-HANSSEN & HN, 77].

The purposes served by a system description are:

- human communication
- documentation of the system
- a basis for programming and verification

Most of the time, the system description is declarative in nature and does not provide for behaviour generation directly, as time is not explicitly represented. It would be most desirable if it were possible to generate the system behaviour directly from the system description using one of the above formalisms.

2.3.2. THE 'EXECUTABLE' SIMULATION MODEL

System behaviour generation by the computer however, requires the model to be in a form acceptable to the computer system in use. The system description alongwith our approach to behaviour generation therefore must be 'coded', using one of the computer languages available, before simulation can proceed. The

language software (compilers/interpreters) available provide the necessary facilities for conversion of the coded model into directly executable form and for run time support.

For the convenience of reference, we shall refer to the model expressed in a computer language as 'executable'. This implies that the model can be 'run' directly from the command level of the operating system, after the language software (e.g. a compiler) has converted it into a directly executable form.

The form that an 'executable' simulation model takes depends upon the available language/software facilities on the computer in use. Some possibilities are general purpose high level procedural languages (e.g. FORTRAN, Pascal, C, ...), simulation packages, simulation languages, or a combination of these. More recently, non-procedural languages (like PROLOG) have also been used for simulation purposes [FUTO & S, 82].

2.4. DYNAMIC BEHAVIOUR GENERATION

> "... but it is not the language in which the simulation is described that is of primary importance: it is the system which handles the statements of this language to transform a set of serial program steps into a simulation of a parallel-acting real system."
> p 74:[TOCHER, 69].

The author does not agree with the first part of this statement and takes the view that elegance and 'naturalness' (e.g. referential transparency [ROBERTSON, 86]) supported by a simulation language would have significant effect on the cost of a simulation study. The second part of the statement, however, is the topic of this section.

With some experience of coding computer simulation models it is not difficult to recognise that some features are common in all simulation models (e.g. mechanism for time advance) and others can be formalized (e.g. approaches to behaviour generation). It is natural that attempts have been made to code these common aspects of simulation models in a general form, so that these do not have to be coded for every simulation model. The result of such attempts have been a number of simulation packages (i.e. suites of routines in general purpose high level languages) and a number of simulation languages.

> "The central problem of discrete system simulation is scheduling the execution in correct chronological sequence of sections of program which represent the occurrence of

> random phenomena. It would be nonsensical to have to write and debug quite complicated routines to handle this dynamic sequencing every time we implement a simulation model. Hence most workers are more than happy to utilise a well-proven commercially available simulation programming system."
> p 179;[DAVIES, 79]

2.4.1. SIMULATION EXECUTIVES

An essential part of every simulation language and simulation package is what is known as the simulation executive [TOCHER, 69] which goes through successive phases repeatedly to generate the behaviour of the system in the simulated time. For example while using Hand or Computer Simulation Package (HOCUS) [HILL, 71] the person who performs the hand simulation is the simulation executive. His/her role is replaced by a software component within the simulation language or a simulation package.

2.4.2. THE REPRESENTATION SCHEME FOR THE SYSTEM'S STATE

The model components need to be represented for programming the model for 'execution' by the computer. It is an intermediate level where the requirements of the programming language available and the modelling requirements of the simulation study need to both be met in an efficient manner. On the modelling end, various items of the model are abstracted as entities, sets, queues, queue disciplines, etc. All these are then represented in terms of the programming language facilities. For example an entity can be represented as a Pascal RECORD or a FORTRAN integer and so on, depending on the implementation language used.

Two main representations used in implementations of simulation software are: (a) the use of state variables, and (b) set theoretic representations [TOCHER, 69]. A conceptual term 'entity', which may have attributes, is used to denote the elements which 'flow' through the system, whereas the system itself is represented by queues and operations, both of which are mapped onto sets. The simulation software provides the necessary primitives for the various possible actions by which the state of the system may be changed (e.g. adding an entity to a set). It also allows for the inspection of the current state of an entity, queue, operation, and so on. These representations provide for the coding of the static features of the system.

2.4.3. APPROACHES TO DYNAMIC BEHAVIOUR GENERATION

The issue of dynamic behaviour generation relates both to the form in which the dynamic behaviour is coded and the phases in which the simulation executive 'fires' the relevant parts of the code. There is considerable confusion in the terminology used to describe these. [TOCHER, 69] has elaborated on this confusion.

The following sections consider behaviour generation within the context of two distinct programming paradigms namely procedural and object oriented.

(A) DYNAMIC BEHAVIOUR GENERATION WHILE USING PROCEDURAL PROGRAMMING PARADIGM

From the point of view of behaviour generation the simulation executives can be broadly classified in two categories as Two Phase or Three Phase systems. The two phase systems can be further classified according to the way the dynamic behaviour is coded, i.e. activity based or event based. A brief review has been undertaken in the following:

(a) Two Phase Activity Type

Phase I. The time advance is determined from what are described as time cells. Time cells are associated with entities in the system or with other system state variables. The values of the time cells represent the remaining time for which the associated entity(ies) or system state variable(s) will remain in their respective current state(s). The time advance is determined by the minimum value of the time cells, and all the time cells are modified accordingly.

Phase II. All the possible changes of the system states are scanned. The time cells at value zero are regarded as representing a particular system condition and are treated at par with the other conditional system states which lead to a change of the system state. If the system's state is altered, the scanning is repeated to explore the possibility of any further changes, until no further changes in the system state are possible, when Phase I is re-entered.

Such simulation executives therefore require the dynamic behaviour of the system to be coded as what is known as 'activities', which specify the conditions which must be met before the state of the system is changed and the actions for

changing the system state. The actions to be performed at the completion of an activity are part of the specification of the activity and are not separately coded.

For an example of an activity type coding of a simulation program using Extended Control and Simulation Language (ECSL) see [CLEMENTSON, 80].

(b) Two Phase Event Type

Phase I. The time advance is determined from a sorted list of time/event/entity triples. The entries in this list result from the explicit scheduling of what is termed as events. These events typically represent the completion of activities (whose completion time can be predicted when activities are started and therefore can be scheduled). The events are also known by the name of bound activities (only to add confusion to the terminology).

Phase II. The state of the system is altered by the execution of what is known as event routines. From within the event routine further possible changes are explored which can be made as a result of changes already made by the current event. Multiple events scheduled for the same simulated time and their respective conditional changes of state are handled one after the other. When no more changes can possibly be made then Phase I (i.e. time advance) is re-entered.

Such simulation software therefore requires the dynamic behaviour of the system to be coded as event routines. The event routines allow for the coding of the dynamic behaviour of a class of entities in a general form.

(c) Three Phase

Three phase simulation systems concentrate on processing all the scheduled events for a given time before the state of the system is scanned for making further conditional changes, as a result of changes already made during the execution of the event routines.

Phase I (also called A phase). Time is advanced in this phase as described in the two phase event type systems.

Phase II (also called B phase). All the events scheduled for the current simulated time are executed through what is known as event routines.

Phase III (also called C phase). The state of the system is explored through specific routines for possible conditional changes of the system's state. When no further changes are possible then Phase I is re-entered.

While using a three phase approach to behaviour generation the dynamic behaviour of the system is coded as a set of 'event routines' (also called B routines) and a set of 'activity routines' (also called C routines). An event typically represents the completion of an operation in the real system thus releasing entities and resources, whereas an activity typically represents the start of an operation to engage the available entities and resources. In activity routines the conditions are known as activity test heads, and the actual actions for changing the system's state are referred to as the activity bodies [TOCHER, 69].

The three phase approach is favoured more in the UK, whereas the two phase approach is preferred in the USA. A recent exchange of ideas on this can be seen in [O'KEEFE, 86b] and [HOOPER, 86]. Fig 2.2 (from [MILLS, 86]) shows the simplified development of two phase languages whereas fig. 2.3 (also from [MILLS, 86]) depicts the simplified development of three phase languages.

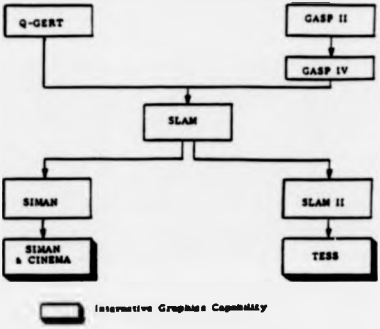


Figure 2.2. "The Simplified development of certain two-phase simulation languages" (from [MILLS, 86] P 214).

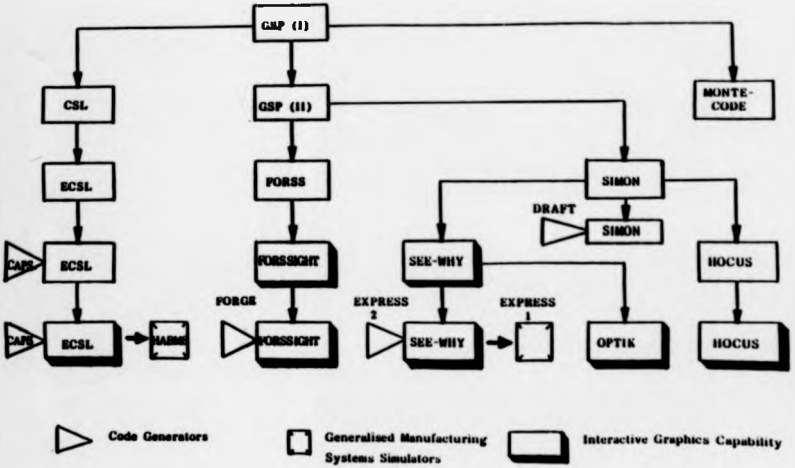


Figure 2.3. "The Simplified development of three-phase simulation languages" (from [MILLS, 86] P233).

(B) DYNAMIC BEHAVIOUR GENERATION WHILE USING OBJECT ORIENTED PROGRAMMING PARADIGM

Object oriented programming has been given more importance in recent years, although it was conceived in the form of the SIMULA language in the 1960's [STEFIK & B, 86].

(a) The Process View of Simulation

An alternate form of an executable simulation model is the description of a process for each class of entities in the system. A process represents the life-cycle of an entity in the system, as viewed from the entity's own angle [FRANTA, 77], [FRANKOWSKI & F, 80]. As such an executable simulation model coded in this form is nearer to the human understanding than the one expressed as a set of actions to be performed by the simulation executive in the form of events and/or activity routines. This implies less effort in coding the simulation model in the first place and easier maintainability of the code, as there is an element of self documentation in the code itself. The process form of describing an executable simulation model therefore represents a 'higher level' specification (i.e. closer to human understanding as well as machine executable).

> "The process approach comes closest to modelling the reality of interacting entities and gives very modular model specifications."
> p 4;[VAUCHER, 85].

The simulation executive capable of generating the behaviour for a simulation model expressed in the process form makes use of what is called a 'process interaction' approach to behaviour generation. This approach is in turn based on computer science concepts of co-routining.

(C) ALTERNATE WORLD VIEWS

Although the issues of behaviour generation have been considered within the context of two programming paradigms in which these have evolved, these are by no means restricted to the respective programming paradigms. A simulation executive using either of these 'world views' can be implemented using either of the programming paradigms. More recently, simulation software provides the flexibility of coding models using alternate world views or even a mixture of these

in one model, for example SLAM [PEGDEN & P, 79]. [HURRION, 85] has shown that it is possible that the process form of a simulation model can be used to generate the system's behaviour while using a three phase simulation executive.

> "Several types of simulation use different modelling and computational paradigms some of which have fundamental similarities. for example, simulation uses both global computation as well as local processing paradigms. The interface of model modules in the first case can be specified as coupling, including nested couplings. In local processing, the interface of model modules can be handled either by scheduling (as in event or process interaction) or by message passing (as in object oriented modelling). It appears that a powerful modelling formalism could capture the common elements of different paradigms to represent objects and their interfaces to provide a methodologically sound basis to conceive complex systems that can be modelled and simulated. Definitely, the time is ripe for multi-formalism model modules in simulation."
> p 7;[OREN, 86].

(D) NEED FOR A UNIFIED WORLD VIEW

> "There are now at least thirty different simulation tools available to assist the planning of everything from a post office layout to a complex manufacturing system. This variety of packages, ranging from a few hundred to many thousands of pounds in initial cost, has served mainly to confuse the prospective purchaser of either the actual software or the services using it. Each supplier, of course, maintains that his is the best package for your application."
> p 226;[MILLS, 86]

Figure 2.4 from [SCHMIDT, 88] depicts the present state of simulation software.

Figure 2.4. "Simulation in the past was characterised by a lack of a unified methodology" (from SCHMIDT, 88] p 40).

There is a need to reconcile the various approaches to bring about what can be described as 'a unified world view' by closing the gap between the various world views in existance presently:

> "The 'closing of the gap' in fact only depends on two changes:
>
> a. Finding, enumerating and standardizing the different modelling formalisms, so as to be able to provide the right interfaces to allow transit from one formalism to the other
>
> b. Leaving behind us the concept that a language, in which we can formulate models for simulation on a computer, would also cater for conventional algorithmic (if you wish: procedural) computer programming. In other words: making the transition from SPLs (Simulation Programming Languages) to MOLs (Model Oriented Languages)."
> p 9;[ELIAS, 80]

2.5. SIMULATION SOFTWARE

2.5.1. SIMULATION LANGUAGES AND PACKAGES

Having considered the basic ideas related to behaviour generation and the form a simulation model must take for it to be acceptable to a particular simulation language, a brief look at the technology of simulation software is appropriate:

> "Each particular simulation programming language (SPL) provides the skeleton of a program, principally the time advance mechanism, together with a series of routines and data structures which we may use to add the flesh to describe the characteristics of our own models. The SPL provides high-level concepts to help us articulate the unique features of our model, but at the same time it imposes a rigid structure within which to define the dynamic behaviour of the elements of the model. There are in fact some half dozen different skeletal structures, each of which dictates apparently quite different forms of (user-supplied) flesh."
> p 180;[DAVIES, 79]

Every simulation package and simulation language subscribes to a 'world view', in terms of a formalism for the system description and the particular behaviour generation approach it employs, which in turn, has a major influence on the form in which the program is expressed.

[NANCE, 84] has reviewed the history of the way simulation software has evolved over the past decades. [DAVIES, 79] provides an over view of the internal structure of a number of different simulation languages. For a well categorized bibliography of simulation software see [SHUB, 80].

2.5.2. SIMULATION PROGRAM GENERATORS

Coding a simulation model from a system description is a major step in simulation modelling. Manual programming is highly error prone and expensive. To ease this burden items of software have been developed under the name of simulation program generators. These make use of the computer science ideas of automatic programming coupled with the formalism for the system description (e.g. activity cycle diagram) and the syntax and semantics of a particular simulation language (or package) to generate a simulation program in that language. The system is generally described to a simulation program generator through an interactive session with it. Examples include CAPS [CLEMENTSON, 80] and DRAFT [MATHEWSON, 85].

2.5.3. SIMULATION ENVIRONMENTS

Simulation software has evolved well beyond providing the specification facilities for the behaviour generation aspects of a simulation model. Such software has been further developed to allow for the specification of an experimental frame, computer assisted model development, automatic statistics gathering, analysis of results, graphic display of results, and so on. More specialised simulation software, which relate to simulations within a specific application domain (e.g. manufacturing), provide for sub-model library features, where the behaviour of different types of machinery can be stored to be retrieved and parameterized for building specific models (e.g. WITNESS [ISTEL, 86]).

> "... the distinction of model generation and model referencing is as follows: as the term implies, in model generation, models are generated with or without the assistance of a computer. In model referecing, already existing model which were stored in model files, are retrieved. As a combination of the two concepts, one can consider, for example, the case where a retrieved component model can be considered with or without modification and can be coupled with a newly generated component model."
> p 190;[OREN, 82]

The current trend in the design of simulation environments can be summarised by:

- Computer support throughout a simulation study

- Comprehensive and integrated simulation environments

(A) COMPUTER ASSISTANCE IN ALL PHASES OF SIMULATION

> "... there are four basic groups of possibilities for computer-aided modelling systems. They are: (1) model generation and/or referencing, (2) model acceptability, (3) model processing, and (4) behaviour processing."
> p 189;[OREN, 82]

The CASM simulation environment is an example of a research project which aims to provide computer assistance in simulation modelling [PAUL, 88].

(B) COMPREHENSIVE AND INTEGRATED

> "Perhaps the most descriptive keyword for characterizing the envisioned simulation practice of the future is 'comprehensive'. That is, practitioners will be doing the same things they do now, except that, individually and collectively, they will be able to carry out more of these activities, to greater effect, in the time and money available."
> p 25;[ZEIGLER, 80].

> "The sine qua non of advanced simulation methodology is the objective of providing comprehensive and integrated assistance in all aspects of the modelling and simulation process."
> p 25;[ZEIGLER, 80].

> "Self contained workstations with integrated development environments will make a large impact on the shape of programming in the 90s. Exploratory prototyping of complex systems, an acceptance of the inevitability of change in system specifications, and a shift in emphasis from writing new to modifying existing programs will further increase the attraction of late binding, object orientation and powerful program management tools for browsing, design, testing, modification, instrumentation and optimization of code. ***Simulation laboratories*** hosted on personal workstations with desktop-based programming environments and graphics support may well prove a major breakthrough in terms of effectiveness and user acceptance."
> p 47;[KREUTZER, 88].

2.6. THE INFLUENCES ON SIMULATION SOFTWARE OF DEVELOPMENTS IN COMPUTER SCIENCE

> "By exploring the fundamental concepts which are common to simulation and other relevant fields (such as artificial intelligence, cybernetics, general system theory, and computer science) specialized in the knowledge representation, knowledge generation, knowledge processing, and knowledge assimilation, we may facilitate their synergies and symbiosis."
> p 3;[OREN, 86].

In the following subsections a brief review has been undertaken to establish the relevance of developments in computer science, and their effect on simulation methodologies and its software. In a later section the influence of developments in the field of artificial intelligence have been reviewed.

2.6.1. HARDWARE

Developments in the area of computer hardware, especially the availability of microcomputers as personal machines, have provided the necessary motivation for the use of simulation modelling in preference to other forms of modelling, although other forms of modelling have also been implemented on microcomputers.

Another area in hardware technology which is already making significant impact on research in simulation is that of parallel processing (e.g. [KOWALIK, 86]).

2.6.2. THE AVAILABILITY OF NEW PROGRAMMING LANGUAGES

A great majority of simulation systems have been implemented in general purpose high level computer languages. When FORTRAN was the only available high level language, both simulation software and simulation models had to be coded in it. The implication is that all the features of the system must be described in terms of the limited variety of data structures offered by FORTRAN and its subprogramming facilities. The advances in general purpose computer language concepts has enabled simulation software writers to implement their software in more elegant forms by devising data structures and their related operations specifically for this purpose. For example a greater variety of data types are provided by languages like Pascal, Ada, Modula, C, ... and more elegant subprogramming facilities, in terms of the way subprograms can be invoked, provide for richer higher level constructs in which to express simulation concepts.

Dynamic memory management concepts in these languages allow for the dynamic generation of entities during the simulation run which was not possible in FORTRAN. As an example, [WALES & L, 86] and [DOWNES & B, 84] investigate the suitability of the use of the ADA programming language for simulation.

2.6.3. THE GRAMMAR FOR SIMULATION LANGUAGES

Although the specification of grammars for programming languages is one of principal concern to computer scientists, an attempt at the formalization of the simulation language CSSL has been commented on by Oren as follows:

> "In the 60s, even the most basic concepts of computer science such as language specification and grammars, did not have a strong influence on simulation. For example, the much acclaimed and influential CSSL definition published in 1967 [Strauss et al, 67] had 126 rules expressed in BNF (Backus-Naur Form) and 41 syntactic errors [Oren, 75]" p 3;[OREN, 86].

[BONGULIELMI & C, 84] has provided a view on the usefulness of deterministic grammars for simulation languages and has recommended group LL(1) languages for the construction of simulation software.

2.6.4. OBJECT ORIENTED PROGRAMMING

As noted earlier, object oriented programming paradigm, [STEFIK & B, 86] and ideas about co-routining, support the process view of simulation. The ideas of having data structures and related operations as one unit, as objects, allows data abstraction which in turn leads to referential transparency thus enabling more elegant code at a higher level. SIMULA [BIRTWISTLE & DMN, 79], DEMOS [BIRTWISTLE, 81], SMALLTALK [GOLDBERG & R, 83], ROSS [KLAHR, 86], BLOBS [MIDDLETON & Z, 86] provide examples of object oriented languages, which can be employed for simulation related use.

2.6.5. COMPUTER GRAPHICS

The role which computer graphics has played in the area of simulation deserves a special mention, as it has more or less revolutionized the way simulation is used. The animated graphic trace of the execution of a simulation model and the ability to interrupt and interact with the model while it is running, has given the decision maker greater involvement in both visual verification/validation of the simulation

model and in experimentation with it. [HURRION, 76] has used the term 'visual interactive simulation' for this approach to simulation.

These ideas have been further developed [SECKER, 77], [BROWN, 78], [RUBENS, 79], [WITHERS, 81], [FISHER, 82] and also have found their applications in the development of decision support system generators [MOREIRA da SILVA, 82].

2.6.6. INTERACTIVE SOFTWARE

Considerable research in man-machine interfaces has brought about systems which are easier to use (i.e. user friendly). These ideas have found their way into the design of simulation software:

> "In such a scheme, users interact with the computer system through interfaces which enable them to initiate or engage in activities. The sequencing of activities may be partly fixed and partly open to users' control. An activity is executed by one or more processors (in conjunction with the user) and acts upon one or more data bases ... In executing an activity, information is stored in the bases. The information so generated is accessible to the user through the interfaces."
> p 29;[ZEIGLER, 80].

2.6.7. SOFTWARE ENGINEERING

Looking from the software engineering point of view a simulation program is a piece of software, which requires construction and maintenance. Various techniques developed in the area of software engineering are therefore relevant. [SHEPPARD, 83] covers the application of software engineering to simulation. Of particular relevance to simulation are the ideas related to Fast Prototyping and Executable Specifications.

> "Desirable features of modelling techniques are modularity and abstraction. One should be able to specify independently (as far as possible) each part of the system and one should not be forced to consider implementation details."
> p 3;[VAUCHER, 85].

> "It is much easier to design and debug a specification than an implementation."
> p 11;[ROBERTSON, 86].

2.6.8. DATA-BASE FACILITIES

The use of data base concepts in simulation software have been described in [STANDRIDGE & P, 82]. These include the use of data bases of model elements and data bases of results or intermediate results. Also in the words of Zeigler:

> "Indeed, methodological research in modelling and simulation methodology has much in common with developments in the areas of software engineering and in data base/information systems design.
>
> "On the one hand simulation programs and tools constitute a class of software (with some quite distinctive characteristics) and should therefore benefit from the concepts and tools being developed in structured programming (e.g. top down design; [Dykstra, 1976]), data structuring (e.g. abstract data types; [Liskova and Zilles, 1974]) and data base design (e.g. data model concepts; [Nijssen, 1977]).
>
> "On the other hand, modelling and simulation methodology is playing an increasingly important role in software and data base design methodologies. Indeed there is an intimate interreliance of design and modelling methodologies on each other and their activities are also partially analogous (with some essential differences; see editor's introduction to Methodology in systems Modelling and Simulation, [Zeigler et. al.(eds), 1979]. It is not surprising then that computer based software design systems [Teichrow and Hershey, 1976]; [Matrin, 1978]; [Yeh, 1977]) resemble the simulation support systems to be described below in architectural philosophy, if not in facilities provided. Likewise data models developed in the data base field resemble the schemes for data representation provided by simulation languages (indeed a data base management system built upon SIMSCRIPT principles is nearing completion (Markowitz, personal communication)].
>
> "As developments proceed in these areas, points of commonality can be shared to mutual advantage ([Sanguinetti, 1979]; [Cuttler, 1980]; [Beauchamp and Field, 1979]; [Ryan, 1979]; [Rzevski, 1980])."
> p 28;[ZEIGLER, 80].

2.6.9. FUNCTIONAL- AND LOGIC PROGRAMMING PARADIGMS

The functional programming language LISP and the logic programming language PROLOG have also been employed for simulation work. [BIRTWISTLE & K, 86] reviews the possibilities of the use of LISP for simulation related work. This thesis is mainly concerned with the use of the logic programming paradigm for

providing computer simulation environments. The use of PROLOG for simulation will be reviewed in chapter 3.

2.7. CURRENT TRENDS IN SIMULATION METHODOLOGY

In this section the current trends in the development of simulation methodology have been reviewed. The following section covers the application of techniques of artificial intelligence in simulation methodology and simulation environments.

2.7.1. TRENDS IN THE PRACTICE OF SIMULATION

Zeigler has identified the following trends in the practice of simulation:

> "1. Trend toward independence of model specification from procedural and machine required specification (providing the user with model oriented languages which separate him to an increasing extent from implementation detail).
>
> 2. Trend towards modularity of functional elements in simulation programming (providing clear segmentation of parts of a program devoted to distinct tasks e.g., structure declarations, process descriptions, experimental control, etc.)
>
> 3. Trend toward flexibility in modelling formalism (providing the user with a wider range of formalisms in which to describe components of his model).
>
> 4. Trend toward tools which provide specialized support of modelling and simulation activities (e.g. statistics, graphics and optimization packages).
>
> 5. Trend toward the integration of such tools (e.g., the linking of standard simulation languages to statistical and optimization modules).
>
> 6. Trend toward increased interactivity (providing the user with greater immediacy in perceiving the status of computer activity and controlling its direction).
>
> 7. Trend toward integration and comprehensiveness in specific application domains (e.g., aircraft design, econometrics models, etc.).
>
> 8. Trend toward incorporation into larger contexts (employing modelling and simulation modules as parts of larger decision support systems)."
> p 28;[ZEIGLER, 80].

2.7.2. EXPERIMENTAL-FRAME BASE AND MODEL BASE

> "The support system should maintain a base of previously defined experimental frames and help to locate a new frame among them. A model base of previously developed model should also be maintained which should be referenceable from the frames base. That is, knowing which resident frames are similar to the new frame should provide an entree to the existing models which are relevant to it. Such models, after adaptation and simplification, should serve as components to be interfaced to form a model to which the new frame is applicable."
> p 31; [ZEIGLER, 80].

2.7.3. INTERACTIVE MODEL DEVELOPMENT BY NON-EXPERTS

In the words of Shannon:

> "... is the desire to make using the system as easy as possible and to build into the modeling system most of the decisions that are now made by the simulation expert. ... The goal of development of expert simulation systems is to make it possible for engineers, scientists and managers to do simulation studies correctly and easily without such elaborate training."
> p 152;[SHANNON, 86].

2.7.4. GRAPHICS WITHIN SIMULATION

While reviewing the state of the art [SHANNON, 86] has viewed the use of graphics in simulation to fall into three categories:

- to facilitate model construction and debugging,
- to provide interactive control during the running of the simulation, and
- to display and help in the understanding of the simulation results.

It further goes on to suggest that graphics could also be used for the specification (definition) of the system through the use of icons.

2.8. SIMULATION AND ARTIFICIAL INTELLIGENCE

Artificial intelligence is not only a computer science development, it also cuts across a number of other disciplines including philosophy and psychology. Its influence on the practice of simulation therefore needs to be considered separately from other developments within computer science.

> "AI can be viewed from two angles. The scientific approach aims at understanding the mechanisms of human intelligence, the computer being used to provide a simulation to verify theories about intelligence. On the other hand, the engineering approach attempts to endow a computer with the intellectual capabilities of people."
> p 701;[DOUKIDIS, 87].

In recent years, following the success of early expert systems, the simulation community has shown considerable interest in Artificial Intelligence (AI) and its technology. A number of conferences have taken place which were specifically devoted to this topic. These include [Holmes (ed), 85], [Birtwistle (ed), 85], [Luker & A (eds), 86], [Kerckhoffs & VZ (eds), 86], [Luker & B (eds), 87], [Henson (ed), 88]. In addition to these conferences, a book titled "Modelling and Simulation Methodology in the Artificial Intelligence Era" has been published [Elzas & OZ (eds), 86]. The number of papers published in the professional journals which are devoted to this topic, provide further evidence of the active interest which the simulation community has been and is taking in artificial intelligence technology. The following comments from [SHANNON & MA, 85] are representative:

> "... If these claims are even half true, then AI is bound to have a profound effect upon the art and science of simulation."
> p 275;[SHANNON & MA, 85].

> "Unless a lot of people are wrong, the technology being developed in the AI field is going to significantly affect computers, software, problem solving, and management. If this is true, then it is obvious that it will also affect the art and science of simulation. It appears that profound changes may be inevitable as a result of artificial intelligence and simulation professionals must not be intimidated by these changes. Simulationists should think about how they can benefit from AI's potentials."
> p 276;[SHANNON & MA, 85].

2.8.1. VIEWS ABOUT THE USE OF AI TECHNIQUES IN SIMULATION

The following sub sections cover the reported views which relate artificial intelligence and simulation at a conceptual level.

(A) AI AND SIMULATION HAVE A RECIPROCAL NEED FOR EACH OTHER

> "If AI has a need for simulation, operational researchers using discrete simulation have a reciprocal need for AI. One of the major limitations of traditional simulation is the inability to model intelligent behaviour [Evans, 1984].

> Development of worthwhile simulations has proved difficult in a number of domains where some element of autonomous decision-making is part of the system - for instance, battle management.
> p 713;[O'KEEFE & R, 87].

> "Thus, since both fields can benefit from each other in a significant way, the marriage is inevitable."
> p 280;[SHANNON & MA, 85].

> "The interchange and coupling between these two disciplines will, undoubtedly, continue to provide a foundation allowing significant improvement in the design and construction of sophisticated 'real-world' systems."
> from Preface in [Henson (ed), 88].

(B) THE IDENTIFICATION OF POTENTIAL AREAS FOR THE APPLICATION OF AI TECHNOLOGY

The following areas have been identified as having potential scope for the application of artificial intelligence technology:

> "... one can readily see that the main developments can be expected in:
> => improved *dialogue* facilities with Modelling & Simulation systems, especially those that have model bases,
> => *Modelling & Simulation consultation* systems, that provide advice on *how* to use *which* models for *what* purposes,
> => *symbolic model manipulation*, for comparing model formulation, searching for certain variables in simulation programs, extracting steady state (analytical) model from simulation models, etc.
> => *model search* and *model inferencing* in/from assemblies of model components in model bases,
> => automated simulation program generation from structured model- and experimental-frame specifications."
> p 73;[ELIAS, 86].

[O'KEEFE & R, 87] have viewed that the application of AI methods and AI software tools for constructing simulations have resulted in the extension of existing simulation concepts by the introduction of: Knowledge-based simulation, Goal-directed simulation, Abstraction, Introspection, and Qualitative simulation. In addition, other AI ideas applicable to simulation include: Intelligent front-ends, Access-oriented programming, Temporal reasoning, Computing environments.

(C) COMMON GROUND AND CONTRAST

[SHANNON & MA, 85] has identified five dimensions in which AI and simulation can be contrasted. These can be summarised as: (a) the way the model is constructed and run, (b) the separation of the knowledge-base and the control structure, (c) the nature of the data base, (d) the characteristics related to processing, and (e) the way expert simulation systems would be used namely: 'user as tutor', 'user as client' and 'user as pupil'.

Problem solving using simulation and AI approaches regard are very similar (mainly search). The association of the two therefore is a natural one, as simulation was already making use of generate and test methods (manually) for solving problems, while using simulation models to test the alternatives. The availability of AI technology with knowledge-based heuristic search techniques fills a long sought after gap in the ability to produce powerful mechanized problem-solving system involving simulation. Simulation, therefore, can be regarded as the most appropriate area in Operational Research to start a cross fertilization with AI.

> "Researchers and practitioners in the field of Simulation and those in Artificial Intelligence (AI) have had to face quite similar problems in creating models of complex and sometimes partially understood systems. To a large extent, solutions have been developed independently in each area leading to techniques and software tools which differ markedly in terminology but often overlap in terms of concepts. The recent stress on knowledge representation in AI has emphasized a common ground, modelling of reality, but each group maintains a slightly different emphasis: dynamic behaviour for simulationists and logical inference for AI workers. In the paper, modelling tools and practice in both areas are contrasted and useful areas of cross-fertilization are suggested."
> p 1;Abstract [VAUCHER, 85].

"... that both simulation and AI are concerned with modelling reality and that there is much similarity of purpose in the need to represent objects, their attributes and their interrelations. However, when considering the dynamics of programs in each area, the similarity is much less clear. To the first approximation,
=> Simulation considers the evolution of systems through time,
=> Artificial intelligence considers proof of system properties.

"Actually, there is common ground. In both, progress is achieved via a series of transitions subject to preconditions, events in one case and deductions in the other. Much of the modeller's art resides in the specification of suitable set of <precondition, transition> pairs."
p 5;[VAUCHER, 85].

[DOUKIDIS, 87] has viewed the production rule system, from artificial intelligence, to be conceptually similar to the three phase behaviour generation method of simulation.

(D) THE RELATION WITH DECISION SUPPORT

[MOSER, 86] has outlined the integration of artificial intelligence and simulation in a comprehensive decision-support system and has reported an experimental implementation of this named EXSYS. Also:

"Application of the results of artificial intelligence research is making possible the next advancement in the provision of management decision support. These new systems will be able to generate (as well as analyze) solution alternatives to problem situations."
p 276;[SHANNON & MA, 85].

"... the use of simulation methodology in conjunction with decision analysis based on Artificial Intelligence is an area with almost limitless potential. This is one stage advanced from decision support, and is definitely a long-term concept."
p 231; [MILLS, 86].

(E) SIMULATION AS A KNOWLEDGE GENERATION TECHNIQUE

"Perceived from a higher and abstract point of view, simulation is a form of knowledge generation, based on three types of knowledge which are (1) descriptive knowledge, (2) intentional knowledge, and (3) knowledge processing knowledge."
p 3;[OREN, 86].

The knowledge engineer therefore has a need to know more about simulation as a source of knowledge.

The statistical knowledge generated by running a simulation can be used to induce rules about the performance of the system being modelled. Rule induction from statistical data is described in [MINGERS, 87]. Such rules can be made use of during experimentation with the model to determine the need for further experimentation. Also such rules can supplement the existing knowledge about the system which in turn can be utilised for problem solving.

(F) THE MODELLING OF INTELLIGENT ENTITIES IN A SIMULATION MODEL

As early as 1969 [SISSON, 69] described the ideas about the simulation of: group behaviour, mass behaviour, individual behaviour and mentioned artificial intelligence in this context. The need for modelling intelligent entities e.g. managers in business type simulations or generals in battlefield type simulations, ..., has long been felt, especially when approaching the problem from a systems point of view. The necessary impetus however, has been provided by research in and availability of AI techniques, whose fundamental objective is to develop computer systems which simulate the intelligent behaviour of individuals.

> "More importantly, many systems include the presence of a decision-maker who has considerable control over what happens in the system - for example, a production controller in a production system, or a general in a battlefield. To simulate such a system, either the simulation must have access to the decision-maker to make the decisions where necessary, or the decision-maker must be modelled. The former approach has been very successfully employed within visual interactive simulation (VIS) [Hall and O'Keefe, 1987]. The latter approach is untested, requiring the use of AI methods in simulation."
> p 713;[O'KEEFE & H, 87].

> "Often simple approximations are used instead of modelling behaviour; for example, the path of activities that a customer takes in a service simulation (for example, a simulation of a shop) is modelled by probabilities determined by prior observation and sampling, rather than by modelling the decision mechanisms of the customer. The latter approach may allow for the modelling of aspects of the system that are typically ignored or are difficult to model when using the former method - for instance, adaptive behaviour, where the activity attempted next is determined by some perception of the present state of the system. Simple decision rules (for instance, always join the

> shortest queue) are frequently inadequate."
> p 713;[O'KEEFE & M, 87].

> "An intelligent agent differs in that its goal structure is represented rather than its behavior structure. By representing only its goal structure, a number of significant advantages are gained. ..."
> p 11;[ROBERTSON, 86].

(G) HIGH LEVEL MODEL SPECIFICATION

Using the knowledge based framework for simulation environments it should be possible to afford a higher level of specification of the problem, the experimental frame and the required model than is possible within the conventional programming environments, e.g. [ROBERTSON, 86], [MUETZELFELDT & RUB, 87].

2.8.2. THE IMPLEMENTATION OF ARTIFICIAL INTELLIGENCE TECHNOLOGY IN VARIOUS PHASES OF A SIMULATION STUDY

The following sections document the research implementations of AI technology in the various phases of simulation.

(A) SIMULATION SOFTWARE AND ENVIRONMENTS

ROSS and KBS are amongst the first implementations making use of artificial intelligence technology:

> "ROSS is one of the first languages that attempts to marry artificial intelligence (AI) methods with simulation technology. We have found that the marriage benefits both parties..."
> p 1;[McARTHUR & KM, 84]

> "The KBS approach is similar to another artificial intelligence simulation system, ROSS. Both KBS and ROSS are object oriented modeling systems that contain attribute and behavioral descriptions and provide interactive access and display."
> p 26;[REDDY & FHM, 86]]

[ADELSBERGER & PSW, 86] has reviewed rule based object oriented simulation systems and has covered a description and comparison of Simula, Smalltalk, Ross, KBS, ORIENT84/K, Smallworld and Omega.

[DOUKIDIS, 87] has reported the following software implementations which make use of AI techniques in simulation. These implementations form part of an over all computer aided simulation modelling environment CASM [PAUL, 88].

> SPIF: Simulation Problem Intelligent Formulator
> uses a natural language interface to develop a logic model of the system under study.
>
> SIPDES: A Simulation Program Debugger using Expert Systems
> helps the user to discover the location of run-time and logical errors in simulation programs and proposes possible solutions.
>
> ASPES: A Skeletal Pascal Expert System
> It is an expert system to which the user can add Pascal code according to the particular application.

[FLITMAN & H, 87] has reported an expert controller for the control of experimentation with simulation models.

[MUETZELFELDT & RUB, 87] has reported ECO, a system for computer-aided construction of simulation programs for ecological modelling.

[KHOSHNEVIS & C, 87] has reported an automated simulation modelling system, EZSIM, based on AI techniques.

[AHMAD & H, 88] has reported a simulation model generation system using a Prolog model base. This paper has been included as appendix I for convenience of reference. This system has been considered in more detail in chapter 5.

[RUIZ-MIER & T, 87] has reported an experimental network simulation environment, SIMYON, which aims to employ a hybrid methodology unifying the concepts of Object-Oriented programming, Logic Programming and the discrete event approach to system modelling.

[HADDOCK, 87] has reported an intelligent front end (a simulation generator) which allows the user to communicate with the system using only the application domain terminology.

[MURRAY & S, 88] has reported an implementation of a knowledge-based simulation model specification system using the generalised knowledge of queueing systems and the knowledge of SIMAN as the target language.

(B) ADVISORY AND SUPPORT SYSTEMS

A further group of implementations can be classified as advisory or support systems. The main purpose of these systems is that the person conducting a simulation study can interact with them, instead of a human simulation expert, for advice or instructions. As such these can be regarded as important constituents of an intelligent simulation environment.

[TAYLOR & H, 88] has reported "An expert advisor for simulation experimental design and analysis."

[SARGENT & R, 88] has reported "An experimental advisory system for operational validity."

[HILL & R, 87] has reported a simulation support system which helps users of INSIGHT simulation software to locate those logical errors in their models which lead to the exhaustion of the available memory.

END OF CHAPTER 2

CHAPTER 3: KNOWLEDGE-BASED SYSTEMS- AND LOGIC PROGRAMMING PARADIGMS

INTRODUCTION

Chapters 1 and 2 focused on a review of the basic ideas relating to managerial problem solving and the use of discrete simulation as one of the problem solving techniques. These chapters also covered the different views relating to the application of artificial intelligence concepts and techniques within problem solving and, in particular, within simulation modelling. This chapter concentrates on the emerging knowledge-based systems and logic programming paradigms at a technical level and argues in favour of using Prolog for implementing the research described in chapters 4, 5, and 6.

3.1. A REVIEW OF THE KNOWLEDGE BASED SYSTEMS PARADIGM

3.1.1. FROM GENERAL PROBLEM SOLVING TO KNOWLEDGE-BASED PROBLEM SOLVING

AI research aims to establish computational approaches to model human cognitive processes [SIMON, 81]. AI researchers have spent nearly two decades (1956-1976) pursuing a line of research related to discovering generalized representations for problems and the related general solution methods. At the end of this period these researchers have concluded that this line of research had not proved extremely fruitful. Using this approach only 'toy' problems could be undertaken and so-called weak general methods broke down when attempted on non-trivial problems (either in terms of time required for the solution or the memory space required). The importance of involving the problem domain knowledge in the process of solution was realized and has been used to advantage.

> "Although computers have many advantages over humans, including speed and consistency, these cannot compensate for ignorance."
> p 288;[HAYS-ROTH, 87]

> "Early on, artificial intelligence researchers discovered that intelligent behaviour is not so much due to the methods of reasoning, as it is dependent on the knowledge one has to reason with."
> p 4;[HUNT, 86].

This has given rise to a branch of research known under the name of Intelligent Knowledge Based Systems or under the more popular name of Expert Systems. Since these systems involve domain knowledge, are designed to capture the expertise in a specific restricted problem domain and have the capability of solving a specific set of problems related to that domain. A number of early successes in expert systems are reported and these include DENDRAL, MYCIN, R1, PROSPECTOR [Shapiro (ed), 87]. As a result of these early successes, the interest in the area of expert systems has intensified and the number of experimental expert systems reported has grown tremendously. At the same time a number of expert systems have found their way into the commercial market as well. [WALKER & M, 86] has reported 1025 internationally identified expert systems.

The basic idea of these systems is to make the domain knowledge available in such a form that it can be retrieved and incorporated in the construction of the solution, in response to a problem posed to the system. The method of constructing the solution is kept separate from the domain knowledge (usually known as the inference engine) and this feature characterizes expert systems paradigm from other types of computer based problem solving in which problem solving knowledge is integrated with the domain knowledge or a representation of it (e.g. in a FORTRAN program). Knowledge representation in this way is however not straightforward:

> "... knowledge representation is one of the most active areas of research in artificial intelligence today. The needed knowledge is not easy to represent, nor is the best representation obvious for a given task."
> p 4;[HUNT, 86].

The problem solving knowledge is built into an inference engine which proceeds with the interpretation of the problem when it is posed and then proceeds with the solution construction process by interpreting and applying suitably represented knowledge from the knowledge base made available to it. The interpretation and selection of a particular item of knowledge depends upon the current state of the development of the solution.

The reader wishing to explore the area of knowledge based systems in greater detail may consult the following references: [Shapiro (ed), 87] extensively and comprehensively covers the field of artificial intelligence as it has developed so far. [JACKSON, 86] and [JOHNSON & K, 85] provide a suitable introduction to expert systems. [WALKER & M, 86] provide a recent assessment of expert systems technology and its applications. [Fox (ed), 84] also provides a review of

developments in the expert systems area. A regular series of technical conferences related to expert systems are organized by the British Computer Society Specialist Group on Expert Systems, [ESTC, 84] and [ESTC, 85]. Another series of workshops on expert systems and their applications is held yearly, [ESAPP, 85] and [ESAPP, 86].

3.1.2. COMPUTER SYSTEMS FOR THE 1990s

The confidence in AI technology has grown enough to plan for its implementation in the computer systems for the 1990s. [BROOKING, 84] has reviewed the developments related to the next generation of computers in Japan (the Fifth Generation project), in the UK (the Alvey Directorate initiative), in Europe (ESPRIT) and in the USA (DARPA and other). Alvey Directorate in the UK and ICOT in Japan are the sources of further periodic information in this regard.

3.1.3. TECHNICAL AND FUNCTIONAL CLASSIFICATIONS OF EXPERT SYSTEMS

> "... knowledge systems differ from conventional programs in the way they're organized, the way they incorporate knowledge, the way they execute and the impression they create through their interactions."
> p 288;[HAYS-ROTH, 87]

A technical classification can be based on the problem solving model employed by the expert systems:

> "The most common way to classify expert systems is by the type of problem-solving model they employ. A problem-solving model [Nii, 1986] is a scheme for organizing reasoning steps and domain knowledge to construct a solution to the problem. AI research and debate has focused and continues to focus on problem-solving models [Chandrasekaran, 1986]."
> p 67;[FORDYCE & NG, 87]

Also the expert systems can be viewed from a functional classification

> "[Bychener, 1985] breaks the present application areas for expert systems into three categories: diagnosis, design, and planning"
> p 74;[FORDYCE & NG, 87]

3.1.4. KNOWLEDGE-BASED SYSTEMS ARCHITECTURE

[JOHNSON & K, 85] has reviewed the architectures of a number of early expert systems in some detail. A chapter is devoted to each of MYCIN, PROSPECTOR, PIP, and so on. The review covers the knowledge representation schemes used, the inferencing techniques employed and the explanation facilities offered.

Figures 3.1 and 3.2 (both from [HAYS-ROTH, 87]) present an overview of the architecture of a knowledge system and its building blocks.

After the initial experience of devising the elements of the various knowledge based systems architecture, attention has been paid to the construction of higher level building blocks which can perform the generic functions needed for the performance of an expert system. [CHANDRASEKARAN, 86] represents an effort in this line.

3.1.5. KNOWLEDGE REPRESENTATION SCHEMES

A number of knowledge representation schemes have been devised and experimented with by the researchers working in the area of artificial intelligence / expert systems [Shapiro (ed), 87]. The most common are:

Production Rules

Frames

Semantic Networks

First Order Predicate Calculus

3.1.6. THE INFERENCE ENGINE

An inference engine incorporates a particular problem solving method and is related directly to the knowledge representation scheme used. For example, with the production rule type knowledge representation, two approaches to problem that are solving used are forward chaining or backward chaining, which define a state space which is then heuristically searched to arrive at a solution. [RICH, 83] covers the topic of heuristic search. [KOWALSKI, 79] covers the topic of search when the knowledge is represented as first order predicate logic notation.

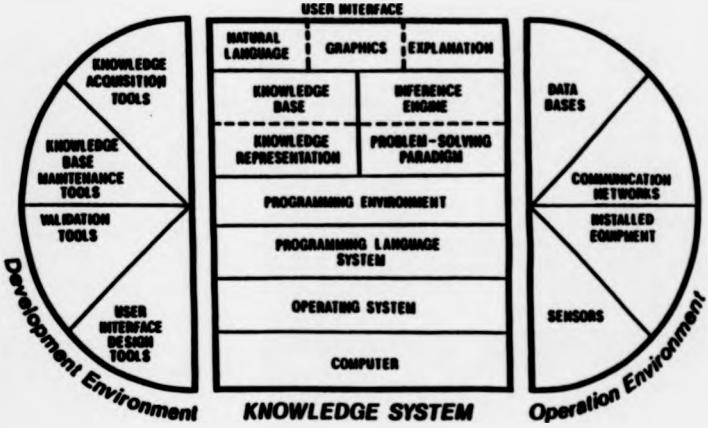


Figure 3.1. "A knowledge system and its environmental context" (from [HAYS-ROTH, 87] p 291).

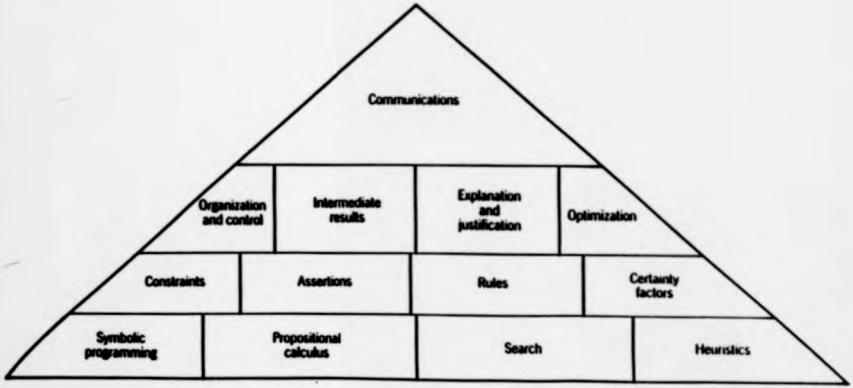


Figure 3.2. "The Building blocks of a knowledge system" (from [HAYS-ROTH, 87] p 289).

In the most general terms, an inference engine works on what is known as a 'recognize and act' cycle, i.e. to recognize the current state of the development of the solution and then act accordingly to make further progress in the solution construction. This type of problem solving has been termed as the pattern directed inference method. The 'act' part is mainly to retrieve from the knowledge-base the needed piece of knowledge that is most appropriate to the current state of development of the solution, to further the progress towards the solution. The knowledge retrieved is incorporated in the solution constructed so far and the cycle repeated. Some forms of domain-independent conflict resolution methods are built into the inference engine to decide if more than one item of knowledge is applicable, in the current cycle, to determine which one should be preferred and applied.

3.1.7. THE CONTROL MECHANISMS FOR INFERENCE

> "Control mechanisms, metarules, or metaknowledge help direct the inference mechanism in its rule selection to improve performance and resolve or prioritize conflicting instructions (Cromarty, 1985)"
> p 70;[FORDYCE & NG, 87]

Both conflict resolution and the search for a solution can be made more efficient by the use of what is known as meta-knowledge. Such knowledge supplements the domain independent built in conflict resolution methods with the domain-specific knowledge. The availability of meta-knowledge tends to make the search for a solution more efficient and also restrains the computer from appearing 'un-intelligent' or following redundant steps.

[REICHGELT & V, 86] cover the criteria for choosing representation languages and control regimes for expert systems.

3.1.8. THE HANDLING OF UNCERTAINTY WITHIN INFERENCE

A further and important source of complexity in inferencing techniques comes from the way uncertainty about the facts, related to the specific problem being solved, are treated and incorporated in the solution construction process. This handling of uncertainty is further accentuated when the knowledge in the knowledge base also carries associated uncertainty values. This in turn leads to

solutions which, of necessity, must have uncertainty (or certainty) value(s) associated with them.

> "In the situation where certainty values are assigned to variable values, two additional items are needed in the inference engine. One is a mechanism for combining certainties. The second is a certainty threshold for firing a rule. [Expert Systems Development Environment/VM Reference Manual, 1985]"
> p 69;[FORDYCE & NS, 87]

Of particular relevance here are methods known as truth maintenance and dependency directed backtracking, which deal with whether to proceed with the next 'recognise and act' cycle or undo previous solution steps in order to take a different course [HUNT, 86].

> "Many people devote considerable effort to the task of improving the certainty factor technology. To a large extent this may prove fruitless."
> p 289;[HAYS-ROTH, 87]

3.1.9. THE USER INTERFACE AND EXPLANATION FACILITIES

User interaction is used during the initial problem acquisition phase and also during the solution construction phase. When the inference engine recognizes a state during the solution construction where an item of knowledge in the knowledge base can only be applied if further information about the specific problem being solved is made available, then it employs user interaction through some form of user interface. In response to the computer needing further information the user can question why this information is required. The expert system then explains the context of the question, i.e. the current inference step, to satisfy the user.

Also, in response to the expert system delivering a conclusion the user may ask for an explanation as to how the conclusion has been arrived at. The expert system has then to show the inferencing steps it has taken to arrive at the conclusion.

> "The capability of self-explanation has been demonstrated to have significant effect on the faith the decision maker has in the system's recommendations. This in turn contributes to the degree to which the system will be used as an ally."
> p 75;[FORDYCE & NS, 87]

3.1.10. THE KNOWLEDGE ENGINEER'S SOFTWARE TOOLS

[WALKER & M, 86] has listed 168 tools for building expert systems. These have been subdivided into (a) tools for symbolic processors and workstations (46), (b) tools for personal computers (89), (c) tools for mainframes (8), and (d) experimental and inhouse tools (25). The numbers in brackets represent the number of tools in each category. These tools comprise of artificial intelligence programming languages, knowledge representation languages, and expert systems shells.

(A) AI PROGRAMMING LANGUAGES

> "There are three general families of languages used for artificial intelligence programming: (1) functional applications languages such as LISP, (2) logic programming languages such as PROLOG, and (3) object-oriented languages such as SMALLTALK and ACTOR."
> p 42:[WALKER & M, 86].

These languages were developed in response to the need for 'higher level' languages than the various procedural languages available. Further, these languages embody some of the primitives used in AI work (e.g. list processing, search, theorem-proving) which makes articulation of the AI programs much less tedious than if this had to be done in a procedural language. Although these languages are known as artificial intelligence languages, it would be a misconception if every program written in these is regarded as an 'intelligent' program. The above mentioned three types of languages can be regarded to represent three distinct programming paradigms, even though these have emerged from research in artificial intelligence.

> "There is open controversy regarding which language is best for artificial intelligence and multiprocessor programming. LISP, a functional programming language, has always been most popular in the United States. PROLOG, a logic programming language, was chosen for the Japanese fifth generation project and is gaining some support in North America. Only the next few years will determine which language becomes dominant or if both will remain equally popular in different programming circles."
> p 44:[WALKER & M, 86].

(B) KNOWLEDGE ENGINEERING SOFTWARE TOOLS

A number of software tools have been developed which provide computer assistance in the various phases of the development of an expert system. These

include knowledge representation languages e.g. KRL, OPS5; knowledge engineering environments e.g. KEE; expert system shells e.g. SAVOIR. These tools vary greatly in their architecture and capabilities. Some of these tools can only run on special purpose hardware e.g. LISP machine. As noted earlier, [WALKER & M, 86] include brief technical details on (and also prices for most of) 168 internationally identified software tools for expert systems development. In general, the tools which can be applied for the development of a comprehensive expert system tend to be very expensive.

3.2. A REVIEW OF THE LOGIC PROGRAMMING PARADIGM

The study of logic provides the forms for valid reasoning. There are many types of logics in use but we shall be mainly concerned with symbolic logic and in particular with First Order Predicate Calculus [KOWALSKI, 79].

> "Logic programming can be defined broadly as the use of symbolic logic for the explicit representation of problems and their associated knowledge bases, together with the use of controlled logical inference for the effective solution of those problems."
> p 544;[KOWALSKI & M, 87].

> "Prolog is a *practical* and *efficient* implementation of many aspects of 'intelligent' program execution such as non-determinism, parallelism, and pattern-directed procedure call. Prolog provides a uniform data structure, called *term*, out of which all data, as well as Prolog programs are constructed."
> p VIII;[CLOCKSIN & M, 84].

The reader wishing to explore the relationships of Logic Programming with other computer programming paradigms, and its applications, should refer to the encyclopedic entry [KOWALSKI & M, 87] which provides a coverage of these.

3.2.1. AUTOMATION OF DEDUCTION IN FIRST ORDER PREDICATE LOGIC

More recently it has been possible to develop procedures to automate deduction, provided the domain knowledge is expressed using first order predicate calculus notation and a query is posed using the same notation. These procedures are built around the symbol processing operation of matching and logical operation of unification used as primitives. These procedures have theoretical foundations

based on Robinson's principle of Resolution for theorem proving. As a result a computer language Prolog has been developed [COLMERAUER, 85] which employs first order predicate calculus notation of atoms and terms to represent knowledge as facts and rules. During the course of deduction in response to a query, unification bindings of variables, in the query, provide the specific values which represent solution(s) to the problem posed.

3.2.2. THE CHARACTERISTIC FEATURES OF PROLOG

Prolog offers only one uniform symbolic structure (a term) in which everything is expressed. The procedural interpretation of a set of Prolog clauses is implicit and is built into the language software (i.e. Prolog interpreter) rather than required to be coded with the program itself. This supports what is known as non-determinism. In procedural languages all the possibilities must be explicitly enumerated and coded. These are 'executed' according to the specific set of data supplied at the run time whereas the non-determinism in logic programming permits the same set of clauses to perform differently in response to the different data supplied at the query time. Further, recursion is strongly supported in Prolog which provides a high level specification feature.

The syntactic simplicity of only one type of symbolic data structure alongwith the non-determinism in the specification of programs, combined with recursion leads to very compact code which is free from the clutter of procedural details (the 'flow' of control) and allows the programmer to concentrate only on the symbolic representation of the concepts in the problem domain (arguments) and their interrelationships (predicates). These features make Prolog a very high level elegant declarative language as compared with the other general purpose high level computer languages available.

An introductory tutorial on Prolog is provided by [DAVIS, 85]. The paper by Alain Colmerauer, who has done pioneering work in implementing the first Prolog interpreter, provides a sound introduction to the language [COLMERAUER, 85]. For a long time [CLOCKSIN & M, 84] has been the only book available on Prolog. A more recently published book [STERLING & S, 86] provides advanced Prolog programming techniques and also includes a wealth of example programs for a number of applications. [POE & NPS, 84] provide an extensive KWIC (Key Word In Context) bibliography on Prolog.

3.2.3. THE PROBLEM SOLVING INTERPRETATIONS OF A SET OF PROLOG CLAUSES

A set of Prolog clauses have dual declarative/procedural interpretations, the latter is based on the resolution principle (or one of its variants) employed. In the current implementations of Prolog the procedural interpretation must take into account the sequencing of the clauses and the sequencing of the terms in the body of clauses, which in turn affect the order in which the variables can be bound and also defines the basis for backtracking. In this sense, the current implementations of Prolog can not be regarded as 100 per cent 'pure' logical programming, which requires procedural interpretation to be independent of the sequencing of the terms in the body of clauses or of clauses themselves.

> "However the exigencies of making Prolog into an efficient programming language have led to the use of control mechanisms which effect the declarative reading. As indicated by [Robinson, 1983] the CUT is the GOTO of logic programming, with well documented effects on the declarative semantics of the program statements. ..."
> p 141;[MORROW, 84]

The logic programming research community is therefore actively engaged in research to discover methods of bringing Prolog closer to the ideal logic programming language, thus permitting truely declarative semantics.

3.2.4. DATA-BASE INTERPRETATION OF PROLOG CLAUSES

A set of Prolog clauses can not only be interpreted as a computer program, but also as a data base on which complex information retrieval queries can be made. In many ways the Prolog data base is superior to the current relational data model, in terms of supporting complex queries and in the maintenance of consistency and integrity of the data base. These constraints themselves can be specified as Prolog clauses. The data base aspects of Prolog are directly related to the knowledge engineering work, in that a knowledge base is a data base containing generic knowledge items on which pattern directed complex retrieval operations are required to be made during inferencing. Further, the use of variables in Prolog terms allows for the generic/general specification of knowledge which can be particularized at the time of retrieval.

3.2.5. COMPILER WRITING AND PROLOG

> "The most popular approach to parsing in Prolog is definite clause grammars or DCGs. DCGs are a generalization of context-free grammars that are executable, because they are a notational variant of a class of Prolog programs."
> p 256;[STERLING & S, 86].

As a matter of fact the notational symbols for definite clause grammar are either usually built into Prolog interpreters or can easily be defined as operators to allow for the grammars to be specified directly in that notation. The importance of using Prolog for compiler writing comes from the fact that "the specification is the implementation" (p 121;[WARREN, 80]) as the rest is taken care of by the nondeterministic nature of Prolog program execution.

> "Parsing with DCGs ... is a perfect illustration of Prolog programming using nondeterministic programming and difference-lists."
> p 256;[STERLING & S, 86].

> "... it should come as no surprise that the Prolog compiler is itself written in Prolog, using the very principles which are the subject of this paper."
> p 124;[WARREN, 80].

The relevance of Prolog to parsing with reference to a particular class of grammars should hardly come as a surprise, because Prolog itself has been evolved during research into natural language understanding.

3.2.6. THE RELATIONSHIP BETWEEN PROLOG AND OBJECTS

It is generally agreed that the first implementation of an object-oriented programming language was SIMULA by Dahl and Nygaard in 1966 [OOPW, 86]. There is a considerable renewal of interest in object oriented programming because of its recent use in the implementation of artificial intelligence programs.

> "An emerging trend is the increased use of object-oriented programming to ease the creation of large exploratory programs."
> p 43;[WALKER & M, 86].

> "The history of ideas (related to object oriented programming) has some additional threads including work on message passing as in ACTORS, by Lieberman in 1981, and multiple inheritance as in FLAVOURS, by Weinreb and Moon in 1981. It is also related to a line of work in AI on theory of frames by Minsky in 1975, and their implementation in knowledge representation languages such as KRL by Bobrow and

> Winograd in 1977, KEE by Fikes and Kehler in 1985, FRL by Goldstein and Roberts in 1977, and UNITS by Stefik in 1978."
> p 44;[WALKER & M, 86]

The following excerpt from Abstract to [McCABE, 86] explains the need to relate object-oriented programming to logic programming:

> "... In particular we examine the class template structure of object oriented programming languages and relate it to logic programming. We shall see that there is indeed a natural relationship: one which can contribute both to the practice of logic programming and to object oriented programming. The inheritance analogy suggests a solution to the problem of how to build large programs."
> p 1;[McCABE, 86].

The relationship between object oriented programming and logic programming is important for simulation related work, because object oriented programming directly supports the process view of the simulation program specification, whereas logic programming affords a higher level of specification. If it is possible to create an automatic translation system which translates the logic specification of a system in an executable object oriented simulation program then the advantage can be gained of both programming paradigms. Efficient implementations of object oriented programming languages are becoming available (e.g. C++ [STROUSTRUP, 86]). This avenue of research therefore seems worth exploring.

3.3. THE REPORTED USE OF PROLOG IN SIMULATION

A number of implementations of simulation software using Prolog have been reported and these are mentioned in the following. It should be stated that most of these implementations (other than those reported by Futo et al) are concurrent with the work reported in chapters 4, 5 and 6 of this thesis. These systems have been included here for the sake of completeness.

M-PROLOG and T-PROLOG [FUTO & S, 82] are among the first reported applications of logic programming within simulation. These have been achieved by extending the Prolog language with simulation related predicates, as it has been argued that Prolog on its own does not provide facilities directly related to simulation [FUTO & G, 87]. In these implementations a simplistic view of simulation has been taken and there is no explicit representation of either time or

the system state. [ADELSBERGER, 84] has provided a further example of T-PROLOG (a bank robbery) and reviewed the use of Prolog as a simulation language.

TS-PROLOG [FUTO & G, 87] is an extension of T-PROLOG which provides for high level hierarchical specification of the system. TS-PROLOG has also been described as being suitable for combined discrete/continuous modelling requirements [FUTO, 85]. A further application of TS-PROLOG for the simulation of an insulin administration problem has been reported in [FUTO & GD, 86].

Another simulation system TC-PROLOG has been reported [FUTO & PS, 86] and its application to compute optimal insulin infusion profiles has been described [FUTO & P, 86].

T-PROLOG, TS-PROLOG and TC-PROLOG support the process interaction approach to behaviour generation and allow for backtracking in simulated time.

T-CP [CLEARY & GU, 85] is an extension of Concurrent Prolog and serves as a simulation language. In this respect it is similar to TS-PROLOG. It does not subscribe to a particular simulation world view and supports a declarative specification of the simulation model and allows for limited backtracking in the simulated time. Its usefulness in relation to simulating concurrent Prolog programs has been identified:

> "We have also observed during the development of T-CP that it can be a very useful tool for understanding, debugging, and testing Concurrent Prolog (CP) programs. The incorporation of simulation time provides additional clues to the operation of CP programs. ..."
> p 12;[CLEARY & GU, 85].

SIMPOOPS [VAUCHER & L, 87] has been described as a simulation system built around POOPS, which claims to combine the best features of both logic- and object-oriented programming.

LOPPS [RADIYA & S, 87] is a Prolog simulation system which takes into account the three major behaviour generation world views and provides for model specification using event scheduling, activity scanning or process interaction world views.

[FLITMAN & H, 87] has reported an expert system implemented in Prolog for controlling the experimentation with simulation models written in a procedural language (FORTRAN).

[TAYLOR & H, 88] has reported WES, an advisory expert system implemented in Prolog for experimentation with simulation models .

PROSS [O'KEEFE & R, 87] is a simulation system similar to GPSS and has been implemented using HC-Prolog.

3.4. ARGUMENTS TO SUPPORT THE USE OF PROLOG AS THE IMPLEMENTATION LANGUAGE FOR THIS PROJECT

3.4.1. THE DISTINCTION BETWEEN KNOWLEDGE REPRESENTATION SCHEMES AND IMPLEMENTATION LANGUAGES

> "Logic programming advocates have been split in whether one should consider such programming as knowledge representation or higher level programming."
> p 141;[BOBROW, 84].

> "Hence we will distinguish between Knowledge Representation Schemes and Knowledge Representation Languages -- ***schemes*** look towards human understanding and the methodologies for knowledge elicitation etc., ***languages*** are implementations of schemes and hence questions of control and efficiency arise. One finds that certain languages have been directly influenced by a scheme. A point that we wish to make is that Predicate Calculus can be thought of as both a scheme and as the basis for a language. When conceived as a language Predicate Calculus may be used, as indeed any language can, to implement any of the schemes. The argument for treating Logic Programming as a basis for work in expert systems, is one to do with having a high level language with a clear semantics. One can accept, or reject, these arguments independently of accepting, or rejecting, that Predicate Calculus is the only scheme in which to consider knowledge representation."
> p 4;[JOHNSON & H, 85].

3.4.2. SUPPORTING ARGUMENTS

(A) ARGUMENT AGAINST THE USE OF A SHELL

Although in principle it is possible to explore the new ground of the use of Artificial Intelligence in simulation by using an expert system package or a shell, e.g. [O'KEEFE, 86a], however, such packages are currently experimental in nature or extremely expensive if well developed. While using such packages one is constrained by the design and the implementation details of the package e.g. [AESSP, 85]. The choice of a general purpose language is therefore an obvious

one. Using a general purpose language leads to transportable source code, especially if the language has been standardized. The use of a general purpose language also provides for more freedom in exploring various ideas, than would be possible using a package. Further, a language has a sound theoretical basis whereas a package is a realization of a scheme which might have only a transitory existence.

(B) THE SELECTION BETWEEN LISP AND PROLOG

As noted earlier, the two general purpose languages developed and used in AI research are LISP and Prolog. Prolog being a later development than LISP remains much less explored for its use in simulation and simulation environments, whereas LISP has been previously considered for simulation work [BIRTWISTLE & K, 86]. Parallel processor machines appear to be the hardware of tomorrow and Prolog is more suitable for such machines than LISP. This is evident by Japan's decision to use Prolog for its Fifth Generation Computer Project.

(C) PROLOG HAS A NUMBER OF HIGH LEVEL PROGRAMMING FEATURES

The choice is further substantiated by the fact that as noted earlier first order predicate logic is in itself a knowledge representation formalism having well defined inferencing (problem solving) methods [KOWALSKI, 79]. Further, Prolog implementations provide built-in theorem-prover and depth-first search facilities. These provide very high level constructs for implementing knowledge-based types of systems. It is an opportunity which can not be ignored.

(D) PROLOG AND THE RESEARCH STRATEGY

The fact that Predicate Logic is both a knowledge representation scheme and a language has been exploited in devising a strategy for this research. Instead of applying the more frequently used knowledge representation schemes like production rules or frames it was considered preferable to initially start with the basic simulation concepts and implement these in Prolog. The experience thus gained can then be made use of, in visualizing and evolving the higher level knowledge representations which are specific to simulation and possibly also to the problem domain. This can be regarded as an application of the problem oriented approach.

(E) ADVANTAGES ANTICIPATED FROM THE USE OF PROLOG IN SIMULATION

In the simulation area, different types of formalisms are used for the system description and its coding into an executable simulation model (chapter 2). It is this coding process (i.e. accurately transcribing the model from one formalism into another) which is error prone and time consuming and is therefore expensive. The need to communicate with the decision maker during this phase also makes it further expensive. These difficulties are the major factors which discourage the use of simulation for problem solving. In order to overcome these difficulties, the declarative cum procedural nature of Prolog can be explored. The declarative aspects of Prolog can possibly be exploited for the system description whereas the procedural aspects can be used for the behaviour generation from such a description (or from a derivative of it, obtained by an automatic transformation). If this can be shown to be feasible it would be a substantial advancement in the simulation modelling area, as it would by-pass the need to transcribe the coding of a simulation model from one formalism into another. These ideas also link in with the software engineering concepts of executable specifications and rapid prototyping.

This process does not need to stop at the specification of the executable model. It should in principle be possible to gather the statistics by running the simulation model, also in the first order predicate calculus notation, to serve as a database of the system's performance under different decision conditions. Such a data base can be supplemented with other system's knowledge to proceed with the problem solving proper. Further, if this can be integrated into an over all framework, then a knowledge based problem solving environment can be created, which can generate knowledge through automatically building simulation models and running them, to obtain the needed knowledge to proceed with problem solving. Using this approach the user need only specify the problem and the rest is handled by the problem solving system.

Prolog has thus been envisaged as having the potential to provide the necessary expression facilities in which all types of knowledge (the system description, behaviour generation, the system's performance, problem solving) can be expressed. Kowalski has expressed this view rather strongly:

"There is only one language suitable for representing information - whether declarative or procedural - and that is first-order predicate logic. There is only one intelligent way to process information - and that is by applying deductive inference methods." [SIGART70, 80]
quoted from p 92;[JACKSON, 86]

3.5. INITIAL CONJECTURES RELATING TO PROLOG AND SIMULATION LANGUAGE GRAMMARS

[BONGULIELMI & C, 84] has described the usefulness of deterministic grammars for simulation languages. [DAVIES, 79] has identified the basic unit of a simulation model for the purpose of simulation program generation and has formulated a grammar for it using BNF (Backus Naur Form) notation. [RADIYA & S, 87] has also formulated BNF like representations for event scheduling, activity scanning and process interaction world views. The relevance of Prolog with the language grammars has been noted earlier in this chapter.

This idea can possibly be exploited for expressing parts of the problem, using languages most suited to the individual parts. An example of research towards this goal is represented by the JADE project [UNGER & DCB, 86] in the use of multiple languages along with a uniform communication protocol to develop JADE distributed software prototyping and simulation environment. Similar principles are also evident in the use of a blackboard model for expert system, where multiple expert systems cooperate to solve the problem through intercommunication [NII, 86a], [NII, 86b].

Provided that the grammars for the respective languages can be formalised as definite clause grammars, these would constitute the implementation of the system in conjunction with Prolog [WARREN, 80]. The need for writing compilers for these languages can therefore be by-passed, thus permitting a very high level of specification of the problem solving system or even an assembly of the system itself in response to a problem.

END OF CHAPTER 3

CHAPTER 4: A PROTOTYPE SIMULATION ENGINE WRITTEN IN PROLOG

INTRODUCTION

The earlier part of this chapter describes the preliminary work carried out towards developing a simulation facility, which could generate simulation behaviour by interpreting the simulation program at the run time. This facility was seen as providing the base layer for further research in knowledge-based discrete simulation environments. This preliminary work was implemented in the programming language Pascal. It was envisaged that it should be possible to develop a knowledge-base in the form of a library of software modules, which represented the behaviour of the individual components in a particular application domain (e.g. in manufacturing domains: conveyors, robots, machining centres and the like). Such a knowledge-base could then be experimented with, to explore the ways of knowledge-based construction of discrete simulation models. A simulation engine, which would interpret the model code at the run time, was seen as a useful research tool as well as a useful part of a simulation model development environment.

In the mean time, a study of the recent developments in artificial intelligence and knowledge based systems (chapters 2 and 3) suggested that these provide a more advanced framework for simulation research, as compared with the use of a Pascal based simulation engine. The use of a logic programming paradigm was seen as particularly attractive as it provided for the specification of programs at a higher level than Pascal. These developments provided the necessary motivation to enhance the scope of this research towards exploring the feasibility of creating an 'intelligent' simulation modelling environment. The use of Pascal for this project was therefore discontinued, and it was decided to use the declarative (non-procedural) language Prolog for the implementation while using a Knowledge-Based Systems framework.

The rest of the chapter is devoted to a description of the research carried out in developing a prototype simulation engine, which was implemented in Prolog. The simulation engine enabled simulated behaviour generation directly from a declarative articulation of the simulation model as a set of Prolog clauses, while using the three phase approach to behaviour generation. The simulation engine was then further extended to also support the articulation of simulation models

using the process view. This extension permitted a mode of specification of the simulation models, where parts of the simulation model could be expressed using the three phase 'world view', whereas the other parts could be expressed using the process 'world view'. This capability of the articulation of the simulation models using a mixture of world views was considered to be particularly relevant from the view point of developing a knowledge based framework for the simulation environment. The behaviour of the various components in a system can be captured more naturally as events (e.g. a break-down of machinery), activities (e.g. a machining operation) or process (e.g. the production process for a component) [HURRION, 85]. A simulation engine which could generate simulated behaviour from an articulation using a mixture of world views was seen as simplifying both the knowledge engineering problems as well as the simulation model construction method.

A prototype knowledge-based simulation model construction system was developed and is the topic of chapter 5, whereas implementation of a knowledge-based interactive model acquisition system is described in chapter 6. Together the three systems constitute a prototype knowledge-based simulation modelling environment.

4.1. THE INITIAL WORK TOWARDS A SIMULATION ENGINE USING PASCAL

4.1.1. BACKGROUND

(A) AN EXISTING SIMULATION FACILITY (MICROSIM)

A visual interactive simulation package of FORTRAN subroutines (MICROSIM) written by Dr. Robert Hurrion was made available in compiled form (i.e. a set of object modules). This simulation package is used on microcomputers during the teaching of simulation courses within the Warwick Business School. MICROSIM subscribes to set representation for the system state together with an event scheduling framework for the time advance. It provides the necessary routines for defining the systems components and for altering and inspecting the system's state, in terms of suitable set operations. It further provides routines for the generation of random numbers and for data collection, during the experimentation with simulation models. The graphic facilities provide for an animated trace of the execution of the simulation model and the graphic display of the results in the form of histograms. The interactive facilities provided by the package include the ability to interrupt the execution of the model to inspect the current state of the

model elements, and provision of suitable user supplied routines to alter the model parameters (e.g. resource levels).

Initially simple simulation models were developed using MICROSIM to gain familiarity with the package.

(B) AN EXISTING INTERPRETER

Mr. Keith Halstead in the Computing Services Unit of Warwick University has developed an interpreter using Pascal, which interprets arithmetic assignment statements. On successful parsing it outputs the statement in prefix notation along with its evaluation, if the variables used in the input statement have previously been assigned values during the current session. If the statement contained errors, parsing is aborted and the error is reported. It is understood that this interpreter was developed within the context of a larger interactive system, where the user could be prompted to provide an arithmetic expression, which was interpreted for immediate evaluation.

Mr. Halstead kindly agreed to make available the Pascal source code for this interpreter for this project.

4.1.2. MOTIVATION

The main motivation at the time was to research into the problems involved in developing an existing visual interactive package of simulation routines into an advanced simulation environment (albeit, a research prototype). An interpreter facility which interpreted the model at the run time was considered to be a necessary component of this envisaged simulation research environment. The initial step was to develop a simulation engine to facilitate programming of the simulation models by using an interpreted approach, to save the turnaround time for compiling the model and linking it with the packaged routines.

The use of Pascal for writing simulation software is considered to be a step further from the use of FORTRAN which has prevailed in the past. The ability to create special purpose data structures alongwith the dynamic memory allocation used in Pascal are distinct advantages for simulation work. Better readability of the Pascal code and the 'goto less' block structure of Pascal provide for better software maintenance.

There has been considerable interest in the UK in the use of Pascal for coding simulation models. This is indicated by the Computer Aided Simulation Modelling Project (CASM Project) at the London School of Economics and Political Science [PAUL, 88]. Also, O'Keefe had previously reported research in the interactive simulation modelling environment by using a Pascal based interpreter built in conjunction with a three phase simulation system by Crookes [O'KEEFE, 84]. It was felt that there was room for exploring different (more advanced) directions in this line of research, to develop and experiment with a more integrated and comprehensive simulation modelling environment than has been reported (e.g. the provision of model-base facilities, the provision for the articulation of the simulation model using alternative formalisms within one environment, and the like).

4.1.3. THE INITIAL WORK IN PASCAL

(A) A PRELIMINARY EXERCISE IN MIXING PASCAL WITH FORTRAN

As a preliminary exercise, a shell was written in Pascal which enabled the writing of simulation programs in Pascal, which could be linked to the MICROSIM modules (which have been compiled from FORTRAN source) alongwith the shell itself. Using this facility much more readable simulation programs could be written in Pascal, as compared with the same models written in FORTRAN. The mixing of the languages at the object code level was possible by using Pascal and FORTRAN compilers from the same company (Prospero Software) which uses a uniform format for the object modules produced by the two compilers. Some initial experimentation with the two compilers and a study of the behaviour of the mixed code was however necessary.

(B) MAKING THE INITIAL STEPS TOWARDS INTERPRETED BEHAVIOUR GENERATION.

An insight was gained into the interpreter technology, after a study of the Pascal code for the arithmetic statements interpreter. As a result, it was possible to make extensions to the existing interpreter code to provide the capability of interpreting a call to a MICROSIM routine and then actually calling the respective routine to perform its operation. Making use of the previous experience of mixing Pascal with FORTRAN, it was possible to link the extended interpreter with the MICROSIM object modules to give an elementary form of the envisaged simulation engine.

It was at this point that the decision was made to explore the possibility of developing an 'intelligent' simulation modelling environment by making use of the logic programming paradigm within the knowledge based systems framework. The reasons for this shift are the subject of the next section.

4.2. THE MOTIVATION FOR THE SHIFT TOWARDS LOGIC PROGRAMMING

4.2.1. TO EXPLORE THE FEASIBILITY OF USING LOGIC PROGRAMMING FOR SIMULATED BEHAVIOUR GENERATION

As noted in chapter 2, in the past the computer simulation community has been keen to upgrade their techniques in response to the developments related to computer languages and programming paradigms. A progression path can be seen from the use of high level procedural languages (e.g. FORTRAN) through strongly typed languages (Pascal, Ada, C), object oriented languages (SIMULA, ROSS) to the functional programming language (LISP).

Since the advent of Prolog (around 1972) there has been considerable interest in its use in many areas which had previously employed procedural languages (chapter 3). [COELHO, 83] has discussed its relevance and usefulness for developing decision support systems and has also provided a list of other areas in which Prolog has found applications. [BHARATH, 86] discusses the use of Logic Programming in Management Science and Operational Research.

At Warwick Business School [FLITMAN, 86] has reported a simulation engine written in Prolog and also a simulation control expert system also written in Prolog. [TAYLOR & H, 88] have reported the use of Prolog to develop an expert simulation experimentation advisor for experimentation with the simulation models written using the MICROSIM simulation package.

4.2.2. TO FACILITATE THE USE OF SIMULATION BASED AI PROBLEM SOLVING TECHNOLOGY

Exploring the use of logic programming (Prolog) within simulation for behaviour generation and within other simulation related areas (e.g. model building, model validation, experimentation with simulation models) is a research issue in its own right. The fact that a large proportion of AI work has been implemented by using either LISP or PROLOG provided a further motivation for the use of Prolog for this project. It was hoped that the use of Prolog for this project would prepare the ground to facilitate the future use of AI problem solving technology in collaboration with discrete simulation modelling.

4.2.3. THE BUILT IN SYMBOLIC PROCESSING FEATURES IN PROLOG

While using a simulation language or a simulation package the symbolic representations in a simulation program are further converted at compile time into the internal representations used by the particular simulation software. For example, the MICROSIM package represented the state of the system by using an array of integers, and each symbol for a set or an entity is assigned an integer value. Further, the attributes of an entity must themselves be referred to by an integer (e.g. first attribute, second attribute and so on) and can only assume numerical values. Such representations derived from the symbolic description of a simulation program make the query and interaction with the model at run time less intelligible, and require more knowledge of the implementation details used by the simulation software, which may or may not be available.

The symbolic processing capabilities of Prolog provided an incentive for attempting an implementation of the representation of the system state directly from the symbols used in the simulation program. Further, these capabilities were envisaged to be directly related to the original ideas of interpreting the simulation program at the run time. After having an experience of the amount of Pascal code required (over 30 pages) to interpret just an arithmetic statement, these high level symbolic processing features of Prolog provided an opportunity which was very difficult to ignore.

At this time the considerations related to the efficiency of running the simulation model were set aside, and the research explorations concentrated only on establishing the technical feasibility of a symbolic representation for the system state, which could permit behaviour generation directly.

Logic programming being a comparatively new programming paradigm and Prolog being a new language, considerable time and effort had to be spent to gain familiarity with these.

4.3. BACKGROUND

4.3.1. EXPRESSING SIMULATION MODELS USING ALTERNATIVE FORMALISMS

The ideas of having options for expressing models using alternative and possibly multiple formalisms have been introduced in [DAVIES, 79] and [PEGDEN & P, 79]. Davies has reported an interactive discrete event simulation modelling environment, which provides for the model description using either event, the

three phase or the life-cycle diagram world views. The model generator receives the information thus supplied, in a world-view independent form termed as 'descriptive units'. From these descriptive units a simulation program can be generated which subscribes to the two phase or the three phase model of behaviour generation. Thus it is possible to interactively enter a simulation model using three phase events and activities and get a simulation program for a two phase simulation language.

[PEGDEN & P, 79] goes a step further and provides the capability to express the model using either a process, event or state variable and allows for either a discrete or continuous simulation capability.

4.3.2. A PROTOTYPE INTERACTIVE SIMULATION MODELLING ENVIRONMENT

Dr. Robert Hurrion has developed a prototype simulation modelling environment using SEE WHY [FIDDY & EH, 81] and has given it the code name LEGO [HURRION, 85]. This modelling environment provides for the interactive entry of sections of a simulation model using a particular behaviour generation world view, and it was possible to use a mixture of these (e.g. process, events, activities). These parts of the model could be saved in the form of a library for later use by parameterization and the assembly into larger simulation models, which could be run using the SEE WHY package. This work has shown that it is possible to generate the system's behaviour from a process type articulation of the executable simulation model while the simulation executive is operating in the three phase mode, thus permitting a sensible and meaningful mixing of the world views when articulating an executable model.

[HURRION, 85] also provides a list of advantages to be gained from having the option to express a simulation model as a mixture of processes, events and activities. The major advantage among these is that parts of the simulation model

can be expressed using the world view most suited to the individual part. Also the model can be incrementally augmented by including further processes in an existing model thus building on previous work.

4.3.3. A PROTOTYPE SIMULATION ENGINE WRITTEN IN PROLOG

Flitman has reported writing a simulation engine in Prolog [FLITMAN, 86]. The engine was reported to have been based on a three phase mode of behaviour generation. The system state, however, is not based on set representation but on a mixture of lists and clauses in the Prolog database. As is usual for a research prototype, simple options for data recording and interaction with the model have been implemented. It is reported that during interaction with the model it is possible to alter the logical structure of the model (in addition to more usual changes, e.g. resource levels) which was regarded as a novel feature. Flitman concludes that:

> "The results of this research have indicated that PROLOG is well suited both theoretically and practically as a simulation language. The natural partition by the PROLOG system of simulation logic and problem characteristics means that the simulation engine is very simple to use. Writing simulation programs is reduced to writing a few facts about the problem (and not the mode of solution)."
> p 103;[FLITMAN, 86].

Among the limitations of his simulation engine in Prolog, Flitman mentions the limited arithmetic capability of Prolog and the slow speed of the execution of the simulation, which could be overcome by using a faster implementation of Prolog having extended arithmetic capability.

Flitman's simulation engine did not provide for the facility of alternative world views for expressing simulation models.

4.4. OBJECTIVES

The objective was set to write a simulation behaviour generation facility in Prolog, which will accept a simulation model expressed using either process, event or activity world views or a mixture of these and generate the behaviour of the system. In effect this was to consolidate the past research within one item of software, while using the logic programming paradigm for implementation. This facility will be referred to as the simulation engine.

It was decided to write the simulation engine only to generate the dynamic behaviour of a model using deterministic time values. The facilities, like the

generation of random numbers, sampling from distributions, statistics collection, visual interactive facilities, were decided to be left for future development if the current research proved fruitful. The engine was meant to be a research prototype rather than a full simulation facility. [CROOKES, 82] has reported 137 simulation packages already in existence.

The main purpose of this exercise was to explore afresh the problems of implementing a simulation engine in a non-procedural programming language and to determine if there were any benefits of such a facility.

4.5. THE FIRST VERSION OF THE SIMULATION ENGINE (THREE PHASE ONLY)

As an initial step an implementation was attempted following the three phase articulation of the dynamic behaviour. This was later extended to allow for the expression of the dynamic behaviour as processes, and as a sensible mixture of the three phase and process specification. Fig. 4.1. shows an overview of the use of the simulation engine in a diagrammatic form. In this section an exposition of the three phase only version will be undertaken. Section 4.6. will cover the extensions.

4.5.1. THE DESIGN FEATURES

(A) THE CHOICE OF THE SET REPRESENTATION FOR THE SYSTEM STATE

The particular Prolog interpreter used during this project (Arity/Prolog) provided a number of built-in predicates known as data-base predicates. Using these predicates Prolog terms could be stored and retrieved using appropriate keys. The interpreter itself uses these predicates for storing the clauses in its program data-base, a Prolog clause being a term with ':-' (the neck symbol) as its functor. Three distinct forms of data-base storage available, were (a) a chain of terms stored under one key (fig. 4.2), (b) terms in a b-tree, where the tree has a name and the terms can be stored under different keys. Conceptually these keys form part of a balanced tree structure whose leaf nodes provide the storage for the Prolog terms, and (c) terms stored in hash tables under sort keys.

The availability of the data base primitives for the storage of Prolog terms and the availability of the built-in predicates for the data base operations like

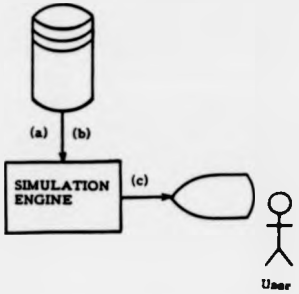


Figure 4.1. An overview of the simulation engine environment.

(a) indicates model statics
(b) indicates model dynamics
(c) indicates simulation trace

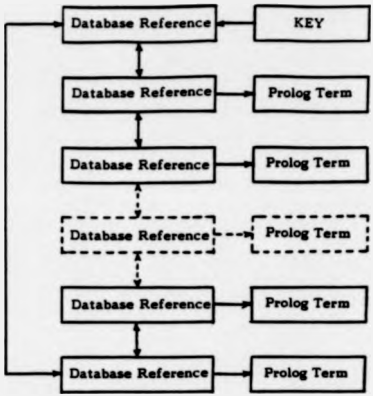


Figure 4.2. The Organisation of The Arity/Prolog Database. (The arrows indicate the direction in which other elements of the database can be accessed)

inserting, deleting, retrieving terms provided the necessary motivation to attempt a generalized implementation of a set representation for the system state, while using set operations available in MICROSIM as a guideline. A summary list of the facilities provided by MICROSIM can be seen in Appendix II.

(B) MEMORY ALLOCATION MODEL

In line with the implementation of MICROSIM in FORTRAN, it was decided to generate a finite number of entities in the beginning, rather than to use the dynamic generation of entities as the simulation proceeds.

(C) THE FORM OF ARTICULATION OF THE SIMULATION MODELS

The following general sections of a simulation program are more or less dictated by the decision to adopt a set representation for the system state and the three phase approach to behaviour generation.

(i) Model statics. The definitions of sets (for queues, activity-sets), classes of entities, resources and the like.

(ii) Model dynamics. The dynamic behaviour coded as a set of events and activities.

(iii) Start-up events. The scheduling of the first arrivals which allow the behaviour generation to begin.

(D) TRACE FACILITIES

An event by event trace on the screen was regarded as adequate, and any graphical animated trace on the screen was left for future development.

4.5.2. IMPLEMENTATION

(A) REPRESENTATIONS FOR THE MODEL STATICS

(a) Sets

Queues and service points are represented as sets, which in their turn are the names of the respective keys under which the Prolog terms are to be stored. All set names are stored likewise in a master set.

The following MICROSIM statement for the definition of a set has been implemented using the following syntax:

```
vset('set_name'(1,2,3,4,5,6,7)).
```

where 'set_name' is the name under which the set is to be defined. The arguments (numbers 1 to 7) at present are dummies. In MICROSIM these are the screen display attributes associated with the set being defined. The dummy arguments hold the place for possible future developments related to graphics.

The following set operation/inspection primitives were implemented:

```
vaddfi
      To add an entity before the first element in a set

vaddla
      To add an entity after the last element in a set

vload
      To add specified entities in a set

vdelete
      To remove an entity from a set

vbehead
      To remove the first element of a set

vbetail
      To remove the last element of a set

move
      To move an element from one set to another

vempty
      To remove all elements from a set

is_empty
      To backtrack if the set is not empty.

is_not_empty
      To backtrack if the set is empty

head_tail
      Returns both first and last elements of a set.
```

(b) Resources

In MICROSIM resources are not specially declared. It was thought convenient to have a separate representation for resources (also referred to in the simulation literature as service facilities) as this tends to make the execution of simulation

faster, because of the reduced overhead of set operations. Conceptually a resource has two possible states: busy and idle. This was implemented as a section of the database with the name of the resource as a key and the number of resources as the number of terms (atoms) as 'busy' or 'idle'

The following syntax has been used for declaring a resource:

```
vresource('resource_name', <number>)
```

The following operation/inspection primitives related to resources have been implemented:

```
set_busy

set_idle

get_idle
```

(c) Entity Classes

Classes of entities are declared by using the following syntax:

```
vclass('class_name'(1,2), <attributes>, <number>).
```

The arguments 1 and 2 are dummy graphic arguments for future use. <attributes> is an integer specifying the number of attributes each entity may have and <number> specifies the size of the class.

The classes of the entities have been implemented as an entry for the class name stored under a key for the master class, where the names of all the classes declared are stored. The attributes have been stored as terms in a B-tree with the index number of the entity in the class as the key.

The following operation/inspection primitives related to the attributes of the entities have been implemented:

```
get_att

set_att
```

(B) REPRESENTATIONS FOR THE MODEL DYNAMICS

(a) Events

The set of events are stored as a part of the program database as a set of Prolog clauses with the predicate 'event' and arity 2. For example the following event clause:

```
event(end_service(Entity), actions) :-
        move(Entity, set_1, set_2),
        set_idle(machine).
```

means that when the event named 'end_service' is processed the Entity on which the event has been scheduled is to be moved from 'set_1' to 'set_2', and the resource called machine is to be set to its idle state from its current busy state.

(b) Activities

Each activity is represented by two clauses with the predicate 'activity' and arity 2. As an example the following two Prolog clauses specify an activity:

```
activity(start_service, conditions) :-
        is_not_empty(queue),
        get_idle(machine).

activity(start_service(Entity), actions) :-
        head_tail(queue, Entity, _),
        move(Entity, queue, set_1),
        set_busy(machine),
        schedule(end_service, Entity, 5.0).
```

The first 'activity' clause represents the conditions which must be satisfied before the activity 'start_service' can begin. In the example above it checks that the queue is not empty and the resource 'machine' is in an idle state. The second 'activity' clause represents the actions which the simulation executive is required to perform to alter the state of the model when the conditions in the first clause are met. In the example above it identifies the head entity in the queue into the Prolog variable "Entity" and then moves the specific entity from 'queue' to 'set_1' and sets the resource 'machine' to the state 'busy' and the completion of the activity is scheduled.

(C) REPRESENTATIONS FOR THE TIME SET AND TIME ADVANCE MECHANISM

The time set which represents the future scheduled events has been represented as a b-tree. The event name and the entity on which the event is scheduled is stored

under the time value which is used as a key. The property of b-trees that enablesinformation to be retrieved in the sort order of keys has been made use of, and this saves the coding required for the insertion of a scheduled event at its chronological position in the time set. In an earlier attempt, however, the ordinary Prolog database representation for the time set was also successfully implemented.

4.5.3. AN EXAMPLE OF BEHAVIOUR GENERATION USING THE SIMULATION ENGINE

An example model of a coal depot [LEC_NOTES, 85] was used to test the simulation engine. A description of this model has been included in Annexe 4A along with an entity cycle diagram for the model. This model shall be referred to as the 'lorry' model.

Annexe 4B presents the articulation of the 'lorry' model using the three phase world view. The articulation consists of the declaration of:

> Model statics: A set of imperative Prolog clauses (these clauses are obeyed immediataly when 'consulted' and do not become part of the Prolog database) declare sets, classes of entities, resources, and so on,
>
> Model dynamics: The dynamic behaviour of the system is specified as a set of Prolog clauses, which become part of the Prolog database, and is referred to at the run time. Two sets of clauses with predicates 'event' and 'activity', as explained above, specify the dynamic behaviour of the system using the three phase world view, and,
>
> Start up events: These imperative clauses specify the scheduling of the first arrivals of the entities from each class.

The three sets of Prolog clauses are presented to the simulation engine through Prolog's 'reconsult' predicate when the simulation engine itself has been 'reconsulted' and initialized. The start-up events need to be presented last as these refer to both events and entity declarations. The clauses declaring the model statics and the model dynamics may be presented to the simulation engine in any order, as there are no references from one to the other.

When the complete specification of the model has been presented to the simulation engine, the model can be run by entering the command 'simulate' to produce a simulation trace on the screen. A sample of a simulation trace for the 'lorry' model as articulated has also been included in Annexe 4B.

The user's typing of 'simulate' calls the simulation engine predicate of the same name as a goal, which provides for the simulated behaviour generation. This predicate has been reproduced in the following:

```
simulate :-
        ctr_set(0,0),
        time(time(0,0,0,0)),
        repeat,
        [!      nl,
                gc(full),
                advance_time,
                ctr_inc(0, _),
                do_scanning
        !],
        fail.
```

An Arity/Prolog counter is used to count the number of time advances (i.e. event number) and is initially set to zero using the 'ctr_set(0,0)' subgoal in the body of the clause. The real time clock of the computer is also set to zero by using the 'time' subgoal. The event number and the real time clock reading is produced as a part of the simulation trace.

After initialising the counter and the real time clock, the simulation engine gets into what can be described as a 'repeat-fail loop', as appears next in the body of the 'simulate' clause. Within this loop:

'nl' gives a blank line on the screen,

'gc(full)' is an Arity/Prolog's system predicate which recovers memory space released by erasing terms in the database,

'advance_time' subgoal advances the simulated time by retrieving the next scheduled event from the time set and updating the clock. It also invokes the b-phase which alters the system state by calling the respective 'event' clause and producing a line of the trace on the screen,

'ctr_inc' increments the event counter by 1, and finally,

'do_scanning' provides for a scan of all the 'activity' clauses that have been previously made available to the simulation engine as part of the model dynamics. During scanning the 'conditions' part of the 'activity' clauses are attempted first and if one succeeds then its 'actions' part is called, which alters the system state and produces a line of the trace on the screen. The scanning is repeated until no further 'activity' can be started.

Backtracking occurs when the last subgoal 'fail' is encountered. An Arity/Prolog control structure called snips (shown by the symbols [! and !]) has been used to specify that the subgoals within '[!' and '!]' are skipped on backtracking and the subgoal 'repeat' is attempted next. The subgoal 'repeat' always succeeds on backtracking which begins the next simulation cycle.

4.6. THE SECOND VERSION OF THE SIMULATION ENGINE

4.6.1. EXTENSIONS TO THE DESIGN SPECIFICATIONS

In the second version of the simulation engine the design specifications were extended to have the ability of expressing the model dynamics purely as a set of processes and also as a combination of processes, events and activities.

4.6.2. IMPLEMENTATION

(A) REPRESENTATIONS FOR THE MODEL STATICS

The representations related to the model statics were kept unaltered, because the same set of static declarations were needed to serve both the three phase and process forms of articulation of the model's dynamics. This was particularly so, when the simulation engine was also needed to support the articulation of the model's dynamics using a mixture of the two world views.

(B) REPRESENTATIONS FOR THE MODEL DYNAMICS

A process for an entity represents its life cycle in the system, expressed in terms of the model's statics, as viewed from the entity's own angle of view. It was decided to express the process of an entity as the body of a Prolog clause. The head of the clause identifies the name of the process using a suitable predicate, whereas the body of the clause represents the steps which an entity goes through while it is associated with this process. These steps are expressed as Prolog terms separated by commas, as in the usual Prolog clause notation. Internally, however, this clause is broken down and the individual terms, which represent individual steps, are recorded in the Prolog database with the process name as the key while keeping the sequence. In this way each step is uniquely identified and can be accessed through the process name or through the database reference to it.

Another data structure was defined in the form of a b-tree, which represented the 'system attributes' of each entity defined in the model. These system attributes pointed to the appropriate process step through which the entity was currently passing provided that it had previously been 'introduced' to the process by the scheduling of the following system event

```
introduce(Entity, Process)
```

The following process steps were implemented:

```
gen_next(Process)

wait_until(head_of(Queue))

wait_until(idle(Resource))

wait_until(message_found(Message)

seize(Resource)

release(Resource)

leave_message(Message)

remove_message(Message)

move(Set_1, Set_2)

hold(Duration)

exit_system(Set_1, Set_2)
```

(C) THE METHOD OF BEHAVIOUR GENERATION AND RELATED REPRESENTATIONS

A system event 'activate' was defined and implemented. When this event is processed a system attribute associated with the entity on which the event had been previously scheduled is set to 'active'. It is possible to scan through these system attributes to locate all the entities in the model which are active at any given time. The system events (i.e. 'introduce' and 'activate') are scheduled in the same way as the user supplied events and are kept in the same time set. When an entity attempts to execute the 'hold(Duration)' process step in the course of progressing through its process steps, the simulation executive 'de-activates' the entity and at the same time an 'activate' event is scheduled for this entity after 'Duration'. In this way the entity is temporarily stopped from advancing through its process steps for the simulated period of 'Duration', because it is not in the 'active' state.

The following 'four phase' model of behaviour generation has been adopted from [HURRION, 85] (fig. 4.3) for implementation in Prolog.

> Phase 1: Advance time to nearest scheduled event (either a system event or a user supplied event).
>
> Phase 2: Alter the state of the model by performing the actions specified by the event.

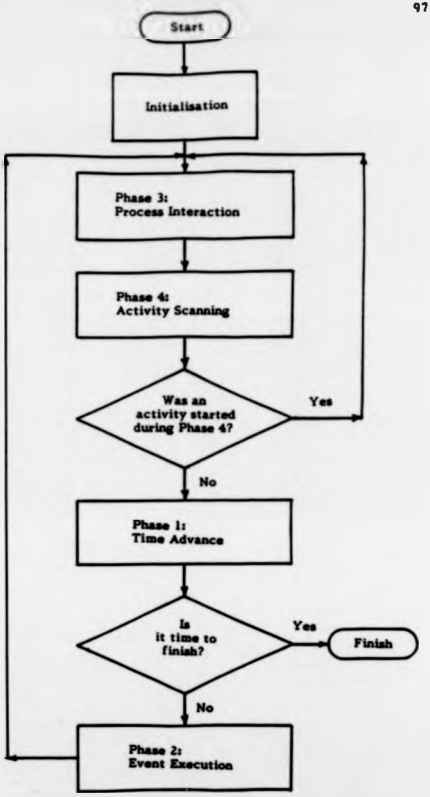


Figure 4.3. A flow diagram for the 'four phase' mode of behaviour generation, where the model can be expressed as a combination of events, activities and processes (adapted from [HURRION, 85]).

Phase 3: Scan the entities which are currently 'active' and attempt to advance each through its respective process steps as far as possible. Re-scan if any entity could be advanced. Enter the fourth phase if no active entity could be advanced through its process on a re-scan.

Phase 4: Scan the user supplied 'activities' for possible starting. If any activity could be started then leave the further scanning of the activities and start phase 3 again. If no activity could be started in a complete scan then go to phase 1.

4.6.3. AN EXAMPLE OF BEHAVIOUR GENERATION BY EXPRESSING THE 'LORRY' MODEL USING PROCESSES ONLY

The 'lorry' model, previously programmed by expressing the dynamic behaviour by using events and activities only, has been re-programmed using processes only. This articulation of the model and the resulting trace from the simulation engine has been included in Annexe 4C. The same segments of the trace as in the events/activities version have been included to facilitate comparison.

There are two observations which can be made by comparing the simulation traces from the two versions. First the process model is slower in its execution as compared with the events/activities version. This is apparent from the trace that in the events/activities version the simulated time 152.51 (i.e. the time of arrival of the 21st merchant) took 2 minutes and 8.9 seconds whereas the corresponding time value for the process version is 12 minutes and 11.2 seconds. The reasons are understandable because the search space in the case of the events/activities version is finite (i.e. the number of activity clauses supplied by the user) whereas the search space in the case of the process version is dependent on the number of entities which are currently active. If in the model the queues are building up (due to say faster arrivals and slower service time) the (real) time between processing two successive events would increase.

A second observation is that the two traces are not identical. In the events/activities version it was possible to express the priority rule that the coal board lorries have a priority over the merchant lorries in the use of the weigh bridge and the lorries waiting to weigh-in have priority over those waiting to weigh-out. This was done by ordering the one argument 'activity' clauses, as these define the search pattern for starting activities. In the process version further articulation is necessary to translate the priority rule into the search pattern.

4.6.4. AN EXAMPLE OF BEHAVIOUR GENERATION BY EXPRESSING THE 'LORRY' MODEL USING PROCESSES, EVENTS AND ACTIVITIES

As a further example the same model dynamics code for the previous two versions is combined together to show that it is possible to use a mixture of processes, events and activities for the articulation of the model's dynamics. Through the start-up events the merchant entity is introduced to the merchant process, whereas the first arrivals of the ncb and the train entities have been scheduled through their respective arrival events. Although all the code for all the processes, events and activities is present, the simulation engine 'drives' the model sensibly. As noted previously further articulation is however necessary to express priority.

4.6.5. A CONSOLIDATED VIEW OF THE SIMULATION ENGINE

To sum up, Fig 4.4 provides a consolidated view of the operation of the simulation engine as described in the preceding sections.

4.7. THE DESIRABILITY OF THE PURELY DECLARATIVE SPECIFICATION OF SIMULATION MODELS

4.7.1. DISCUSSION

It must be stated that the forms used for the articulation of a simulation model are not fully declarative, which is somewhat against the spirit of logic programming. In the events/activities case the model must be viewed as 'event' and 'activity' clauses to express the model in terms of actions to be performed by the simulation executive. In this sense the articulation is highly procedural as it specifies the sequence of actions to be performed within the framework of the three phase behaviour generation method. The process form is somewhat less procedural in the sense that it does not have to specify the actions to be taken by the simulation executive, but still it is a sequence of steps through which an entity must go through during its life cycle in the system.

Implementing a change in a model expressed in a procedural formulation involves going through an error prone and time consuming (therefore expensive) debugging phase, followed by verification and possibly revalidation. This is so because the logical structuring of the model, the behaviour generation approach and the operational decision rules are interwoven into the procedural code. Such

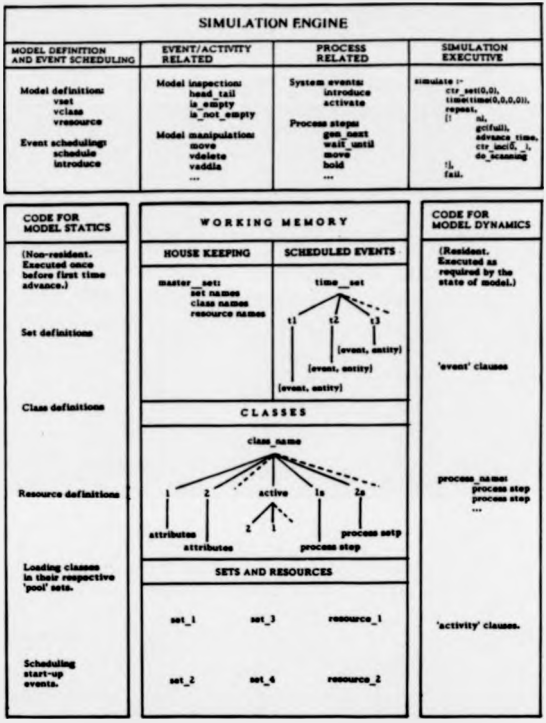


Figure 4.4. A consolidated view of the simulation engine.

difficulties related to the modification of the code for a simulation model during experimentation tend to limit the use of simulation for problem solving [e.g. McARTHUR & KN, 84].

Logic programming being a declarative programming paradigm can be seen as offering features which could provide the ability to separate the control (behaviour generation method) from the model code. This can have a number of advantages both in the overall simulation study and in programming and modifying the model specification.

The model in declarative form is less susceptible to programming errors for the simple reason that there is less to code. A declarative logic specification of the model would consist of a set of symbols representing the model components which are related to each other through a set of predicates and expressed as clauses. A further set of clauses can specify the operational rules for investigating the system's behaviour. A number of such sets of operating rules can be prepared and simply 'plugged' in for various experiments without having a need to alter the rest of the model code. Further, since the declarative specification is a statement of the problem at a higher level, some form of knowledge based model synthesis and verification system could further enhance the programmer's productivity and the reliability of the code produced, thus making a simulation study cost effective.

The ability to systematically change a simulation model during experimentation is of the essence for a successful simulation study. Model specification in a declarative form should lend itself to the automation of experimentation with the simulation model and also present the ability to explore the possibilities in which the logical structure of the model can be altered. The procedural specification of a model does not easily lend itself to making structural changes to the model and the scope of experimentation has to be limited to changing some of the model parameters even if some degree of automation can be achieved for experimentation.

The use of the declarative form also opens a way to the use of knowledge based problem solving techniques developed in the area of artificial intelligence. Using a knowledge based systems framework for both the model building and the experimentation with it, one should only need to specify the objectives of the study and the rest could be left to the computer system. The computer system should be able to generate the appropriate model requested based on an

experimental frame, test the model to see how far it meets the requisite performance measures and proceed to make 'intelligent' (heuristic) alterations in the model's specifications to carry out the next 'generate and test' cycle, until the model which satisfies the original requirements is obtained.

The work by [DAVIS, 79] has already shown that it is possible to automatically transform the logic specifications of a program into the program in a specified general purpose high level procedural language. This approach should in principle be extendible to specialized simulation programs. The advantage of this approach would be to get all the benefits of a compact high level logic specification, as described above, together with the efficiency of running a simulation program in a procedural language or possibly in a simulation language.

4.7.2. AN EXAMPLE

In order to provide a concrete example, a possible declarative articulation of the 'lorry' model as a set of Prolog clauses has been presented in Annexe 4E. The Annexe also includes the documentation for the predicates used to define the model.

Although this declarative form of the 'lorry' model is derived from the entity cycle diagram of the model, it should in principle be possible to derive the entity cycle diagram from this description (assuming a complete description). The point is that using a suitable set of predicates, the logic specification can serve the dual purposes of the system description and of the behaviour generation using the appropriate simulation engine which is essentially in line with the spirit of logic programming.

As should be clear from the declarative description of the 'lorry' model, it has not been 'coded' with reference to a particular behaviour generation approach (i.e. a world view) this makes the behaviour generation transparent and leaves the simulation model in a much more communicable form, which is nearer the level of human (decision maker's) understanding.

4.8. CONCLUSIONS AND FURTHER RESEARCH

4.8.1. CONCLUSIONS

The work described in this chapter has built upon the existing body of knowledge related to the methods of simulated behaviour generation (the three phase method, process interaction), the related forms for the articulation of the model's dynamics (events/activities, processes), and a representation scheme for the system's state (set representation). This work has demonstrated that it is technically feasible to unify these ideas by writing a (prototype) generalized simulation engine using the logic programming paradigm and ideas of symbolic processing which permit the articulation of the model as Prolog clauses, while providing the option to choose between alternative world views or even a (sensible) mixture of these for expressing the model. The simulation engine provides the capability to generate behaviour directly from a suitable articulation of the model without taking any further conversion steps (e.g. compilation, linking).

Purely declarative forms of the articulation of a model are desirable. Future research should therefore aim at writing a 'simulation engine' which could accept the declarative specification of a model (for example, based on queuing network terminology or on system theory terminology) and is able to generate an appropriate internal representation of the model, for behaviour generation purposes, and be able to generate simulated behaviour directly. Some success in this regard has been reported by [FUTO & G, 87] although the process view of simulation has been adopted and the approach has been to extend the Prolog interpreter rather than write a generalised simulation facility using it.

4.8.2. FURTHER RESEARCH

The simulation engine is useful in that it is possible to generate behaviour directly from the model even though the model runs slower than if it were compiled and linked. This should not be seen as a shortcoming, particularly in pedagogical environments. The benefit of a shorter turnaround time could outweigh the slower execution in terms of saving time which is consumed by the user waiting for the model to be compiled and linked only to discover that a small error in transcribing the model logic will force him to go through this time consuming cycle again. In future, however, parallel processing hardware should improve the execution speed.

In a real simulation study the simulation engine concept is still useful. Using the ideas presented in this chapter it should in principle be possible to write a simulation engine which will accept models written in one of the prevalent simulation languages. Such a simulation engine could then be used during the model development stages, and when the model is fully developed and validated it can be compiled for running efficiently during experimentation.

(A) EFFICIENCY RELATED

Further work, however, is necessary to address the problems related to the efficiency of running the simulation model. There is an obvious trade-off between the analyst's time required to explicitly code the method of behaviour generation by expressing the model as a set of events and activities and the speed of execution attained by the simulation engine. If the analyst saves his time by not explicitly coding the behaviour generation method into the model by expressing the model as a set of processes, then the simulation engine would require more time in generating the behaviour through the same length of simulated time. Expressing the model in terms of events/activities suitably breaks down the behaviour generation problem for the simulation engine, whereas using process articulation the method of behaviour generation is not explicitly supplied and the simulation engine has to do more work to generate behaviour.

The further work can be on the lines of developing heuristic search strategies during the scanning phase of the behaviour generation, and experimenting with devising representations to eliminate or constrain any unnecessary searches. These heuristic search strategies must relate to the further language constructs for expressing the priorities and other operating rules.

(B) THE FACILITIES FOR EXPRESSING PRIORITIES AND OTHER OPERATIONAL RULES

As noted earlier, the operation of the simulation engine is unconstrained with regards to following any priority rules when the model is expressed (fully or partially) using the process form. Further work is required to develop language constructs to express the priority and other operational rules which are 'obeyed' by the simulation engine at the run time.

END OF CHAPTER 4

ANNEXE 4A

4A.1. SIMULATION PROBLEM ('lorry')

This problem has been adapted from an example problem from [LEC_NOTES, 85]. Since the facilities for random variate generation within the simulation engine have been left for future development, all stochastic time durations have been changed to constant values.

4A.1.1. NATURAL LANGUAGE DESCRIPTION OF THE PROBLEM

The problem concerns a coal depot where merchants are complaining that their vehicles are experiencing delays. Suggestions for improving the service at the depot are that either an extra weigh-bridge be installed or an extra mechanical loader be provided. The advantages of these alternatives can be assessed by simulating three situations :-

(i) The Present Situation - one loader, one weigh-bridge

(ii) An extra weighbridge - one loader, two weigh-bridges

(iii) An extra loader - two loaders, one weigh-bridge.

4A.1.2. NATURAL LANGUAGE DESCRIPTION OF THE SYSTEM

The present situation is as follows :-

There is one weighbridge which is used by all vehicles to weigh in and to weigh out. There is also one mechanical loader to load coal into the merchants' lorries.

The merchants' lorries arrive with an interarrival time of 7.27 minutes. They weigh in (2.42 minutes) and run to the coal stocking area, where the loader loads them (5.84 minutes). They then run to the weighbridge and weigh out (3.17 minutes).

The Coal Board lorries arrive regularly every 12.14 minutes. They do other work on the site, and are not loaded with coal. After weighing in (2.22 minutes), they spend 22 minutes working in the depot, and then weigh out (2.93 minutes).

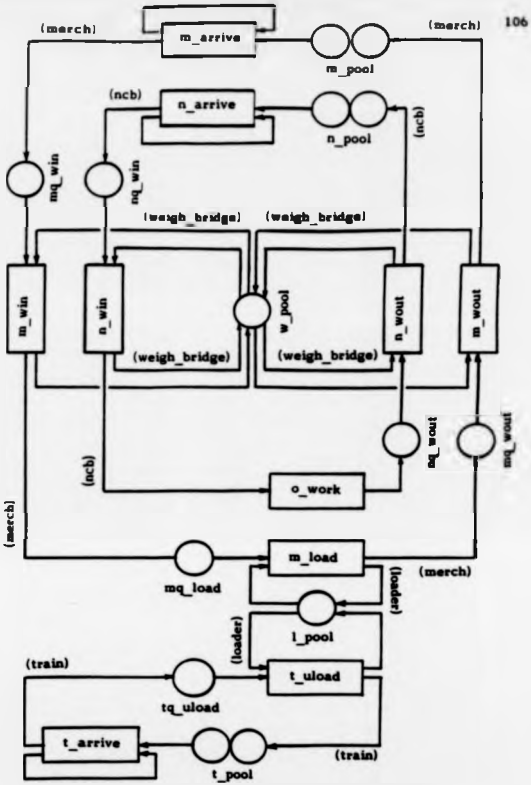


Figure 4.5. The entity cycle diagram for the 'lorry' model.

The Coal Board lorries get priority over the merchants' lorries at the weighbridge, but any lorry arriving to weigh in has priority over any lorry waiting to weigh out.

The loader is usually available for loading lorries. It has to be taken off this work however when a train arrives at the depot requiring to be unloaded. The inter-arrival times for the trains is 13.40 minutes. The time taken to unload a train is 17.50 minutes. If the loader is busy when the train arrives, it is allowed to finish loading the lorry it is working on, but after that the train unloading takes priority over other waiting lorries.

An entity cycle diagram for this model has been included in fig 4.5.

END OF ANNEXE 4A

ANNEXE 4B

4B.1. THE 'larry' MODEL (THREE PHASE ONLY)

4B.1.1. MODEL ARTICULATION

(A) MODEL STATICS

```
:- vset(m_pool[1,2,3,4,5,6,7]).
:- vset(mq_win[1,2,3,4,5,6,7]).
:- vset(m_win[1,2,3,4,5,6,7]).
:- vset(mq_load[1,2,3,4,5,6,7]).
:- vset(m_load[1,2,3,4,5,6,7]).
:- vset(mq_wout[1,2,3,4,5,6,7]).
:- vset(m_wout[1,2,3,4,5,6,7]).
:- vset(n_pool[1,2,3,4,5,6,7]).
:- vset(nq_win[1,2,3,4,5,6,7]).
:- vset(n_win[1,2,3,4,5,6,7]).
:- vset(o_work[1,2,3,4,5,6,7]).
:- vset(nq_wout[1,2,3,4,5,6,7]).
:- vset(n_wout[1,2,3,4,5,6,7]).
:- vset(t_pool[1,2,3,4,5,6,7]).
:- vset(tq_uload[1,2,3,4,5,6,7]).
:- vset(t_uload[1,2,3,4,5,6,7]).
:- nl, write($sets defined$).
:- nl.

:- vclass(march(1, 2), 0, 50).
:- vclass(ncb(1, 2), 0, 50).
:- vclass(train(1, 2), 0, 15).
:- write($classes defined$).
:- nl.

:- vresource(weigh_bridge, 1).
:- vresource(loader, 2).
:- write($resources defined$).
:- nl.

:- write($loading classes in pools$).
:- nl.
:- vload(march, 1, 50, m_pool).
:- vload(ncb, 1, 50, n_pool).
:- vload(train, 1, 15, t_pool).
```

(B) MODEL DYNAMICS

(a) Activities

```
activity(n_win).
activity(m_win).
activity(n_wout).
activity(m_wout).
activity(t_uload).
activity(m_load).
```

```
activity(m_win, conditions] :-
    is_not_empty(mq_win),
    get_idle(weigh_bridge],
    !.
activity(m_win(Entity], actions] :-
    head_tail(mq_win, Entity, _],
    move(Entity, mq_win, m_win],
    set_busy(weigh_bridge],
    schedule(m_win_end, Entity, 2.42],
    !.

activity(m_load, conditions] :-
    is_not_empty(mq_load),
    get_idle(loader],
    !.
activity(m_load(Entity], actions] :-
    head_tail(mq_load, Entity, _],
    move(Entity, mq_load, m_load],
    set_busy(loader],
    schedule(m_load_end, Entity, 5.84],
    !.


activity(m_wout, conditions] :-
    is_not_empty(mq_wout),
    get_idle(weigh_bridge),
    !.
activity(m_wout(Entity], actions] :-
    head_tail(mq_wout, Entity, _],
    move(Entity, mq_wout, m_wout],
    set_busy(weigh_bridge],
    schedule(m_wout_end, Entity, 3.17],
    !.

activity(n_win,conditions] :-
    is_not_empty(nq_win),
    get_idle(weigh_bridge),
    !.
activity(n_win(Entity], actions] :-
    head_tail(nq_win, Entity, _],
    move(Entity, nq_win, n_win],
    set_busy(weigh_bridge],
    schedule(n_win_end, Entity, 2.22],
    !.


activity(n_wout, conditions] :-
    is_not_empty(nq_wout),
    get_idle(weigh_bridge),
    !.
activity(n_wout(Entity], actions] :-
    head_tail(nq_wout, Entity, _],
    move(Entity, nq_wout, n_wout],
    set_busy(weigh_bridge],
    schedule(n_wout_end, Entity, 2.93],
    !.

activity(t_uload, conditions] :-
    is_not_empty(tq_uload],
    get_idle(loader],
    !.
activity(t_uload(Entity], actions] :-
    head_tail(tq_uload, Entity, _],
    move(Entity, tq_uload, t_uload],
    set_busy(loader],
    schedule(t_uload_end, Entity, 17.50],
    !.
```

(b) Events

```
event(m_arrive).
event(m_win_end).
event(m_load_end).
event(m_wout_end).
event(n_arrive).
event(n_win_end).
event(o_work_end).
event(n_wout_end).
event(t_arrive).
event(t_uload_end).

event(m_arrive(Entity), actions) :-
    move(Entity, m_pool, mq_win),
    head_tail(m_pool, Head, _),
    schedule(m_arrive, Head, 7.27),
    !.

event(m_win_end(Entity), actions) :-
    move(Entity, m_win, mq_load),
    set_idle(weigh_bridge),
    !.

event(m_load_end(Entity), actions) :-
    move(Entity, m_load, mq_wout),
    set_idle(loader),
    !.

event(m_wout_end(Entity), actions) :-
    move(Entity, m_wout, m_pool),
    set_idle(weigh_bridge),
    !.

event(n_arrive(Entity), actions) :-
    move(Entity, n_pool, nq_win),
    head_tail(n_pool, Head, _),
    schedule(n_arrive, Head, 12.14),
    !.

event(n_win_end(Entity), actions) :-
    move(Entity, n_win, o_work),
    set_idle(weigh_bridge),
    schedule(o_work_end, Entity, 22.00),
    !.

event(o_work_end(Entity), actions) :-
    move(Entity, o_work, nq_wout),
    !.

event(n_wout_end(Entity), actions) :-
    move(Entity, n_wout, n_pool),
    set_idle(weigh_bridge),
    !.

event(t_arrive(Entity), actions) :-
    move(Entity, t_pool, tq_uload),
    head_tail(t_pool, Head, _),
    schedule(t_arrive, Head, 13.40),
    !.

event(t_uload_end(Entity), actions) :-
    move(Entity, t_uload, t_pool),
    set_idle(loader),
    !.
```

(C) START-UP EVENTS

```
:- schedule(m_arrive, merch(1), 7.11).
:- schedule(n_arrive, ncb(1), 12.22).
:- schedule(t_arrive, train(1), 13.33).
:- write('initial events scheduled').
:- nl.
```

4B.1.2. A TRACE OF THE SIMULATED BEHAVIOUR GENERATED BY THE SIMULATION ENGINE (THREE PHASE ONLY)

The following trace was obtained on the computer screen when the simulation engine generated the behaviour from the above articulation of the 'lorry' modal. The numbers on the left are the simulated time, whereas the numbers within the brackets following the word 'time' on the right are the real time clock readings. These time values follow the format (hh,mm,ss,nn) where hh is hours, mm is minutes, ss is seconds and nn is hundredths of a second. The rest is self explanatory. At the end when the simulation run is interrupted the simulation engine gets into the query mode and the state of the system can be viewed by using the query commands.

```
                                        event_no(0) / time(0,0,0,50)
7.11  m_arrive(merch(1))
        >>>>> started activity(m_win(merch(1)))

                                        event_no(1) / time(0,0,1,0)
9.53  m_win_end(merch(1))
        >>>>> started activity(m_load(merch(1)))

                                        event_no(2) / time(0,0,1,70)
12.22  n_arrive(ncb(1))
        >>>>> started activity(n_win(ncb(1)))

                                        event_no(3) / time(0,0,2,50)
13.33  t_arrive(train(1))
        >>>>> started activity(t_uload(train(1)))

                                        event_no(4) / time(0,0,4,50)
14.38  m_arrive(merch(2))

                                        event_no(5) / time(0,0,4,80)
14.44  n_win_end(ncb(1))
        >>>>> started activity(m_win(merch(2)))

                                        event_no(6) / time(0,0,5,50)
15.37  m_load_end(merch(1))

                                        event_no(7) / time(0,0,5,90)
16.86  m_win_end(merch(2))
        >>>>> started activity(m_wout(merch(1)))
        >>>>> started activity(m_load(merch(2)))
```

```
                                        event_no(8) / time(0,0,8,20)
20.03  m_wout_end(march(1))

                                        event_no(9) / time(0,0,8,60)
21.65  m_arrive(march(3))
         >>>>> started activity(m_win(march(3)))

                                        event_no(10) / time(0,0,9,30)
22.7  m_load_end(march(2))

                                        event_no(11) / time(0,0,10,80)
24.07  m_win_end(march(3))
         >>>>> started activity(m_wout(march(2)))
         >>>>> started activity(m_load(march(3)))

                                        event_no(12) / time(0,0,11,90)
24.36  n_arrive(ncb(2))

                                        event_no(13) / time(0,0,12,30)
26.73  t_arrive(train(2))

...
...
...

                                        event_no(132) / time(0,2,8,90)
152.51  m_arrive(march(21))

                                        event_no(133) / time(0,2,9,30)
152.73  t_uload_end(train(10))
         >>>>> started activity(m_load(march(18)))

                                        event_no(134) / time(0,2,10,20)
154.94  n_wout_end(ncb(10))
         >>>>> started activity(m_win(march(21)))

                                        event_no(135) / time(0,2,11,0)
157.36  m_win_end(march(21))
         >>>>> started activity(m_wout(march(13)))

                                        event_no(136) / time(0,2,13,10)
157.9  n_arrive(ncb(13))

                                        event_no(137) / time(0,2,13,40)
158.57  m_load_end(march(18))
         >>>>> started activity(m_load(march(19)))

                                        event_no(138) / time(0,2,14,30)
159.78  m_arrive(march(22))
```

```
                                                    event_no(139) / time(0,2,15,90)
160.53  m_wout_end(merch(13))
          >>>>> started activity(n_win(ncb(13)))

                                                    event_no(140) / time(0,2,16,70)
160.73  t_arrive(train(12))

...
...
...
Enter query (enter "help." for help):
```

END OF ANNEXE 4B

ANNEXE 4C

4C.1. THE 'larry' MODEL (PROCESSES ONLY)

4C.1.1. MODEL ARTICULATION

(A) MODEL STATICS

This section of the model is exactly the same as in the previous articulation using the three phase approach and is therefore not reproduced here.

(B) MODEL DYNAMICS

```
process(march_process) :-
    move(m_pool, mq_win),
    gen_next(march_process, 7.27, m_pool),
    wait_until(head_of(mq_win)),
    wait_until(idle(weigh_bridge)),
    move(mq_win, m_win),
    seize(weigh_bridge),
    hold(2.42),
    release(weigh_bridge),
    move(m_win, mq_load),
    wait_until(head_of(mq_load)),
    wait_until(idle(loader)),
    move(mq_load, m_load),
    seize(loader),
    hold(5.84),
    release(loader),
    move(m_load, mq_wout),
    wait_until(head_of(mq_wout)),
    wait_until(idle(weigh_bridge)),
    move(mq_wout, m_wout),
    seize(weigh_bridge),
    hold(3.17),
    release(weigh_bridge),
    exit_system(m_wout, m_pool).


process(ncb_process) :-
    move(n_pool, nq_win),
    gen_next(ncb_process, 12.14, n_pool),
    wait_until(head_of(nq_win)),
    wait_until(idle(weigh_bridge)),
    move(nq_win, n_win),
    seize(weigh_bridge),
    hold(2.22),
    release(weigh_bridge),
```

```
        move(n_win, o_work),
        hold(22.8),
        move(o_work, nq_wout),
        wait_until(head_of(nq_wout)),
        wait_until(idle(weigh_bridge)),
        move(nq_wout, n_wout),
        seize(weigh_bridge),
        hold(2.93),
        release(weigh_bridge),
        exit_system(n_wout, n_pool).

    process(train_process) :-
        move(t_pool, tq_uload),
        gen_next(train_process, 13.40, t_pool),
        wait_until(head_of(tq_uload)),
        wait_until(idle(loader)),
        move(tq_uload, t_uload),
        seize(loader),
        hold(17.50),
        release(loader),
        exit_system(t_uload, t_pool).
```

(C) START-UP EVENTS

```
    :- introduce(merch(1), merch_process, 7.11).
    :- introduce(ncb(1), ncb_process, 12.22).
    :- introduce(train(1), train_process, 13.33).
    :- write('Initial events scheduled'), nl.
```

4C.1.2. A TRACE OF THE SIMULATED BEHAVIOUR GENERATED BY THE SIMULATION ENGINE (PROCESSES ONLY)

The note for the trace in section A4.2.2. applies here.

```
                                        event_no(0) / time(0,0,0,60)
7.11  sys_event(introduce(merch(1),merch_process))
        merch(1) executed move(m_pool,mq_win)
        merch(1) executed gen_next(merch_process,7.27,m_pool)
        merch(1) executed wait_until(head_of(mq_win))
        merch(1) executed wait_until(idle(weigh_bridge))
        merch(1) executed move(mq_win,m_win)
        merch(1) executed seize(weigh_bridge)
        merch(1) executed hold(2.42)

                                        event_no(1) / time(0,0,4,60)
9.53  sys_event(activate(merch(1)))
        merch(1) executed release(weigh_bridge)
        merch(1) executed move(m_win,mq_load)
        merch(1) executed wait_until(head_of(mq_load))
        merch(1) executed wait_until(idle(loader))
        merch(1) executed move(mq_load,m_load)
        merch(1) executed seize(loader)
        merch(1) executed hold(5.84)
```

```
                                                    event_no(2) / time(0,0,7,50)
12.22  sys_event(introduce(ncb(1),ncb_process))
         ncb(1) executed move(n_pool,nq_win)
         ncb(1) executed gen_next(ncb_process,12.14,n_pool)
         ncb(1) executed wait_until(head_of(nq_win))
         ncb(1) executed wait_until(idle(weigh_bridge))
         ncb(1) executed move(nq_win,n_win)
         ncb(1) executed seize(weigh_bridge)
         ncb(1) executed hold(2.22)


                                                    event_no(3) / time(0,0,12,0)
13.33  sys_event(introduce(train(1),train_process))
         train(1) executed move(t_pool,tq_uload)
         train(1) executed gen_next(train_process,13.4,t_pool)
         train(1) executed wait_until(head_of(tq_uload))
         train(1) executed wait_until(idle(loader))
         train(1) executed move(tq_uload,t_uload)
         train(1) executed seize(loader)
         train(1) executed hold(17.5)


                                                    event_no(4) / time(0,0,16,50)
14.38  sys_event(introduce(march(2),march_process))
         march(2) executed move(m_pool,mq_win)
         march(2) executed gen_next(march_process,7.27,m_pool)
         march(2) executed wait_until(head_of(mq_win))


                                                    event_no(5) / time(0,0,19,50)
14.44  sys_event(activate(ncb(1)))
         ncb(1) executed release(weigh_bridge)
         ncb(1) executed move(n_win,o_work)
         ncb(1) executed hold(32.0)
         march(2) executed wait_until(idle(weigh_bridge))
         march(2) executed move(mq_win,m_win)
         march(2) executed seize(weigh_bridge)
         march(2) executed hold(2.42)


                                                    event_no(6) / time(0,0,24,70)
15.37  sys_event(activate(march(1)))
         march(1) executed release(loader)
         march(1) executed move(m_load,mq_wout)
         march(1) executed wait_until(head_of(mq_wout))


                                                    event_no(7) / time(0,0,26,50)
16.86  sys_event(activate(march(2)))
         march(2) executed release(weigh_bridge)
         march(2) executed move(m_win,mq_load)
         march(2) executed wait_until(head_of(mq_load))
         march(2) executed wait_until(idle(loader))
         march(2) executed move(mq_load,m_load)
         march(2) executed seize(loader)
         march(2) executed hold(5.84)
         march(1) executed wait_until(idle(weigh_bridge))
         march(1) executed move(mq_wout,m_wout)
         march(1) executed seize(weigh_bridge)
         march(1) executed hold(3.17)


                                                    event_no(8) / time(0,0,36,60)
20.03  sys_event(activate(march(1)))
         march(1) executed release(weigh_bridge)
         march(1) executed exit_system(m_wout,m_pool)
```

```
                                        event_no(9) / time(0,0,38,0)
21.65  sys_event(introduce(march(3),march_process))
          march(3) executed move(m_pool,mq_win)
          march(3) executed gen_next(march_process,7.27,m_pool)
          march(3) executed wait_until(head_of(mq_win))
          march(3) executed wait_until(idle(weigh_bridge))
          march(3) executed move(mq_win,m_win)
          march(3) executed seize(weigh_bridge)
          march(3) executed hold(2.42)


                                        event_no(10) / time(0,0,42,10)
22.7   sys_event(activate(march(2)))
          march(2) executed release(loader)
          march(2) executed move(m_load,mq_wout)
          march(2) executed wait_until(head_of(mq_wout))


                                        event_no(11) / time(0,0,43,70)
24.07  sys_event(activate(march(3)))
          march(3) executed release(weigh_bridge)
          march(3) executed move(m_win,mq_load)
          march(3) executed wait_until(head_of(mq_load))
          march(3) executed wait_until(idle(loader))
          march(3) executed move(mq_load,m_load)
          march(3) executed seize(loader)
          march(3) executed hold(5.84)
          march(2) executed wait_until(idle(weigh_bridge))
          march(2) executed move(mq_wout,m_wout)
          march(2) executed seize(weigh_bridge)
          march(2) executed hold(3.17)


                                        event_no(12) / time(0,0,51,10)
24.36  sys_event(introduce(ncb(2),ncb_process))
          ncb(2) executed move(n_pool,nq_win)
          ncb(2) executed gen_next(ncb_process,12.14,n_pool)
          ncb(2) executed wait_until(head_of(nq_win))


                                        event_no(13) / time(0,0,53,0)
26.73  sys_event(introduce(train(2),train_process))
          train(2) executed move(t_pool,tq_uload)
          train(2) executed gen_next(train_process,11.4,t_pool)
          train(2) executed wait_until(head_of(tq_uload))

...
...
...
                                        event_no(131) / time(0,12,11,20)
152.51  sys_event(introduce(march(21),march_process))
          march(21) executed move(m_pool,mq_win)
          march(21) executed gen_next(march_process,7.27,m_pool)


                                        event_no(132) / time(0,12,17,0)
153.51  sys_event(activate(march(14)))
          march(14) executed release(weigh_bridge)
          march(14) executed exit_system(m_wout,m_pool)
          ncb(10) executed wait_until(idle(weigh_bridge))
          ncb(10) executed move(nq_wout,n_wout)
          ncb(10) executed seize(weigh_bridge)
          ncb(10) executed hold(2.93)
```

```
                                                    event_no(133) / time(0,13,27,0)
155.88  sys_event(activate(train(10)))
          train(10) executed release(loader)
          train(10) executed exit_system(t_uload,t_pool)
          march(18) executed wait_until(idle(loader))
          march(18) executed move(mq_load,m_load)
          march(18) executed seize(loader)
          march(18) executed hold(5.84)


                                                    event_no(134) / time(0,13,38,60)
156.44  sys_event(activate(ncb(10)))
          ncb(10) executed release(weigh_bridge)
          ncb(10) executed exit_system(n_wout,n_pool)
          march(19) executed wait_until(idle(weigh_bridge))
          march(19) executed move(mq_win,m_win)
          march(19) executed seize(weigh_bridge)
          march(19) executed hold(1.42)
          march(20) executed wait_until(head_of(mq_win))


                                                    event_no(135) / time(0,13,50,10)
157.9   sys_event(introduce(ncb(11),ncb_process))
          ncb(11) executed move(n_pool,nq_win)
          ncb(11) executed gen_next(ncb_process,12.14,n_pool)
          ncb(11) executed wait_until(head_of(nq_win))


                                                    event_no(136) / time(0,13,54,70)
158.86  sys_event(activate(march(19)))
          march(19) executed release(weigh_bridge)
          march(19) executed move(m_win,mq_load)
          march(19) executed wait_until(head_of(mq_load))
          ncb(11) executed wait_until(idle(weigh_bridge))
          ncb(11) executed move(nq_win,n_win)
          ncb(11) executed seize(weigh_bridge)
          ncb(11) executed hold(2.22)


                                                    event_no(137) / time(0,13,1,80)
159.18  sys_event(activate(ncb(11)))
          ncb(11) executed move(o_work,nq_wout)
          ncb(11) executed wait_until(head_of(nq_wout))


                                                    event_no(138) / time(0,13,9,60)
159.78  sys_event(introduce(march(22),march_process))
          march(22) executed move(m_pool,mq_win)
          march(22) executed gen_next(march_process,7.27,m_pool)


                                                    event_no(139) / time(0,13,15,10)
160.73  sys_event(introduce(train(12),train_process))
          train(12) executed move(t_pool,tq_uload)
          train(12) executed gen_next(train_process,13.4,t_pool)
          train(12) executed wait_until(head_of(tq_uload))

...
...
...
Enter query (enter "help." for help):
```

END OF ANNEXE 4C

ANNEXE 4D

4D.1. THE 'lorry' MODEL (A MIXTURE OF THREE PHASE AND PROCESSES)

4D.1.1. MODEL ARTICULATION

(A) MODEL STATICS

This part of the model remains unchanged.

(B) MODEL DYNAMICS

This part of the model is a sum of the model dynamics code for the previous two cases.

(C) START-UP EVENTS

Here the entity 'march' has been introduced to the 'march_process' and the first arrivals for the entities 'nch' and 'train' have been scheduled with reference to their respective arrival events. All the code for the model's dynamics in the previous two cases (i.e. events and activities only, and processes only), however, is present and can be activated by a different combination for the scheduling of the start-up events.

```
:- introduce(march(1), march_process, 7.11).
:- schedule(n_arrive, nch(1), 12.22).
:- schedule(t_arrive, train(1), 13.33).
:- write('$initial events scheduled$').
:- nl.
```

4D.1.2. A TRACE OF THE SIMULATED BEHAVIOUR GENERATED BY THE SIMULATION ENGINE (MIXED THREE PHASE AND PROCESSES)

The note for the trace in section A4.2.2. applies here.

```
                                          event_no(0) / time(0,0,0,60)
7.11  sys_event(introduce(march(1),march_process))
        march(1) executed move(m_pool,mq_win)
        march(1) executed gen_next(march_process,7.27,m_pool)
        march(1) executed wait_until(head_of(mq_win))
        march(1) executed wait_until(idle[weigh_bridge])
        march(1) executed move(mq_win,m_win)
        march(1) executed seize(weigh_bridge)
        march(1) executed hold(2.42)
```

```
                                        event_no(1) / time(0,0,4,70)
9.53  sys_event(activate(march(1)))
        march(1) executed release(weigh_bridge)
        march(1) executed move(m_win,mq_load)
        march(1) executed wait_until(head_of(mq_load))
        march(1) executed wait_until(idle(loader))
        march(1) executed move(mq_load,m_load)
        march(1) executed seize(loader)
        march(1) executed hold(5.84)


                                        event_no(2) / time(0,0,7,50)
12.22  n_arrive(ncb(1))
        >>>>> started activity(n_win(ncb(1)))


                                        event_no(3) / time(0,0,9,60)
13.33  t_arrive(train(1))
        >>>>> started activity(t_uload(train(1)))


                                        event_no(4) / time(0,0,10,40)
14.38  sys_event(introduce(march(2),march_process))
        march(2) executed move(m_pool,mq_win)
        march(2) executed gen_next(march_process,7.27,m_pool)
        march(2) executed wait_until(head_of(mq_win))


                                        event_no(5) / time(0,0,13,70)
14.44  n_win_end(ncb(1))
        march(2) executed wait_until(idle(weigh_bridge))
        march(2) executed move(mq_win,m_win)
        march(2) executed seize(weigh_bridge)
        march(2) executed hold(2.42)


                                        event_no(6) / time(0,0,15,70)
15.37  sys_event(activate(march(1)))
        march(1) executed release(loader)
        march(1) executed move(m_load,mq_wout)
        march(1) executed wait_until(head_of(mq_wout))


                                        event_no(7) / time(0,0,18,80)
16.86  sys_event(activate(march(2)))
        march(2) executed release(weigh_bridge)
        march(2) executed move(m_win,mq_load)
        march(2) executed wait_until(head_of(mq_load))
        march(2) executed wait_until(idle(loader))
        march(2) executed move(mq_load,m_load)
        march(2) executed seize(loader)
        march(2) executed hold(5.84)
        march(1) executed wait_until(idle(weigh_bridge))
        march(1) executed move(mq_wout,m_wout)
        march(1) executed seize(weigh_bridge)
        march(1) executed hold(3.17)


                                        event_no(8) / time(0,0,26,80)
20.03  sys_event(activate(march(1)))
        march(1) executed release(weigh_bridge)
        march(1) executed exit_system(m_wout,m_pool)
```

```
                                        event_no(9) / time(0,0,28,20)
21.65  sys_event(introduce(march(3),march_process))
          march(3) executed move(m_pool,mq_win)
          march(3) executed gen_next(march_process,7.27,m_pool)
          march(3) executed wait_until(head_of(mq_win))
          march(3) executed wait_until(idle(weigh_bridge))
          march(3) executed move(mq_win,m_win)
          march(3) executed seize(weigh_bridge)
          march(3) executed hold(2.42)


                                        event_no(10) / time(0,0,32,20)
22.7   sys_event(activate(march(2)))
          march(2) executed release(loader)
          march(2) executed move(m_load,mq_wout)
          march(2) executed wait_until(head_of(mq_wout))


                                        event_no(11) / time(0,0,35,10)
24.07  sys_event(activate(march(3)))
          march(3) executed release(weigh_bridge)
          march(3) executed move(m_win,mq_load)
          march(3) executed wait_until(head_of(mq_load))
          march(3) executed wait_until(idle(loader))
          march(3) executed move(mq_load,m_load)
          march(3) executed seize(loader)
          march(3) executed hold(5.84)
          march(2) executed wait_until(idle(weigh_bridge))
          march(2) executed move(mq_wout,m_wout)
          march(2) executed seize(weigh_bridge)
          march(2) executed hold(3.17)


                                        event_no(12) / time(0,0,41,40)
24.36  n_arrive(ncb(2))


                                        event_no(13) / time(0,0,41,80)
26.73  t_arrive(train(2))

...
...
...


                                        event_no(132) / time(0,6,36,90)
152.51  sys_event(introduce(march(21),march_process))
          march(21) executed move(m_pool,mq_win)
          march(21) executed gen_next(march_process,7.27,m_pool)
          march(21) executed wait_until(head_of(mq_win))


                                        event_no(133) / time(0,6,40,60)
153.45  n_win_end(ncb(12))
          march(21) executed wait_until(idle(weigh_bridge))
          march(21) executed move(mq_win,m_win)
          march(21) executed seize(weigh_bridge)
          march(21) executed hold(2.42)


                                        event_no(134) / time(0,6,43,80)
155.33  sys_event(activate(march(19)))
          march(19) executed release(loader)
          march(19) executed move(m_load,mq_wout)
          march(19) executed wait_until(head_of(mq_wout))
          march(20) executed wait_until(idle(loader))
          march(20) executed move(mq_load,m_load)
          march(20) executed seize(loader)
          march(20) executed hold(5.84)
```

```
                                                event_no(135) / time(0,6,48,90)
155.87  sys_event(activate(march(21)))
          march(21) executed release(weigh_bridge)
          march(21) executed move(m_win,mq_load)
          march(21) executed wait_until(head_of(mq_load))
          march(19) executed wait_until(idle(weigh_bridge))
          march(19) executed move(mq_wout,m_wout)
          march(19) executed seize(weigh_bridge)
          march(19) executed hold(3.17)


                                                event_no(136) / time(0,6,55,30)
157.9   n_arrive(ncb(13))


                                                event_no(137) / time(0,6,55,90)
158.57  t_uload_end(train(10))
          march(21) executed wait_until(idle(loader))
          march(21) executed move(mq_load,m_load)
          march(21) executed seize(loader)
          march(21) executed hold(5.84)


                                                event_no(138) / time(0,6,59,0)
159.04  sys_event(activate(march(19)))
          march(19) executed release(weigh_bridge)
          march(19) executed exit_system(m_wout,m_pool)
          >>>>> started activity(n_win(ncb(13)))


                                                event_no(139) / time(0,7,0,90)
159.78  sys_event(introduce(march(22),march_process))
          march(22) executed move(m_pool,mq_win)
          march(22) executed gen_next(march_process,7.27,m_pool)
          march(22) executed wait_until(head_of(mq_win))


                                                event_no(140) / time(0,7,4,30)
160.73  t_arrive(train(12))

***
***
***

Enter query (enter "help." for help):
```

END OF ANNEXE 4D

ANNEXE 4E

4E.1. A DECLARATIVE ARTICULATION OF THE 'larry' MODEL

The following set of Prolog clauses 'declare' the model described in Annexe 4A. The following section elaborates on the predicates used.

```
is_inf(m_pool, march, 7.27).
is_inf(n_pool, nch, 12.14).
is_inf(t_pool, train, 13.40).

is_queue(mq_win, march, m_win).
is_queue(mq_load, march, m_load).
is_queue(mq_wout, march, m_wout).
is_queue(nq_win, nch, n_win).
is_queue(nq_wout, nch, n_wout).
is_queue(tq_uload, train, t_uload).

resource_level(weigh_bridge, 1).
resource_level(loader, 2).

is_activity(m_win, [march, weigh_bridge], 2.42).
is_activity(m_load, [march, loader], 5.84).
is_activity(m_wout, [march, weigh_bridge], 3.17).
is_activity(n_win, [nch, weigh_bridge], 2.22).
is_activity(o_work, [nch], 22.0).
is_activity(n_wout, [nch, weigh_bridge], 2.93).
is_activity(t_uload, [train, loader], 17.50).

priority(weigh_bridge, [n_win, m_win, n_wout, m_wout]).
priority(loader, [t_uload, m_load]).

entity_cycle(march, [m_win, m_load, m_wout]).
entity_cycle(nch, [n_win, o_work, n_wout]).
entity_cycle(train, [t_uload]).
```

4E.2. THE DOCUMENTATION FOR THE PREDICATES USED FOR DECLARATIVE ARTICULATION

The predicates used for the declarative model articulation in the previous section have been documented in the following:

(i) Predicate: **is_inf / 3**

```
is_inf(Pool_set, Entity, Arrival_distribution).
```

The 'is_inf' predicate specifies the 'pool sets' (or infinite queues) in which a particular class of entities is initially loaded. Entities 'arrive' from their

respective infinite queues according to the 'Arrival_distribution' and return there when they have completed their cycle in the system.

(ii) Predicate: is_queue / 3

```
is_queue(Queue_name, Entity_class, Activity_name).
```

The 'is_queue' predicate declares the presence of a queue in the model. The first argument specifies the name of the queue, the second argument specifies the entity class which makes use of this queue to wait for an activity which is specified by the third argument.

(iii) Predicate: resource_level / 2

```
resource_level(Resource, Level).
```

The 'resource_level' predicate declares the presence of a resource in the model and specifies its numbers. The number of identical resources implies the possible maximum simultaneous occurrences of an activity using the resource as one of the co-operating entities.

(iv) Predicate: is_activity / 3

```
is_activity(Activity_name,List_of_Participants,
     Activity_duration).
```

The 'is_activity' predicate declares the presence of a distinctly identifiable activity in the model. The first argument 'Activity_name' specifies a name for the activity. The second argument 'List_of_Participants' specifies the requisite co-operating entities without which the activity can not proceed. The third argument specifies the duration of the activity.

(v) Predicate: priority / 2

```
priority(Resource, Priority_list).
```

The 'priority' predicate specifies the priority which an activity has over the use of a resource. The 'Priority_list' specifies the activities in a sequence from high to low priority.

(vi) Predicate entity_cycle / 2

```
entity_cycle(Entity_name, Entity_cycle).
```

The 'entity_cycle' predicate provides a sequence of activities which an entity goes through during its life-cycle in the system.

4E.3. IMPLICATIONS IN RELATION TO MODEL GENERATION

The ability in a (future) simulation engine to set up a system state and generate the system's behaviour from a 'world view less' articulation would have important implications for the model generation system described in chapter 5. In this case the simulation engine would generate its own code implicitly and it would be entirely transparent to the user. It should further permit the user to concentrate on the problem in hand and allow the model generation system to concentrate entirely on the application domain knowledge to generate the suitable model(s).

A further implication would be that it should be possible to generate an executable simulation model in a given programming language, from the logic specification of the model in a 'world view free' form by writing a program generator for that language.

END OF ANNEXE 4E

CHAPTER 5: A PROTOTYPE KNOWLEDGE-BASED DISCRETE SIMULATION MODEL GENERATION FACILITY

INTRODUCTION

The previous chapter demonstrated the feasibility of implementing a simulation engine using the logic programming paradigm which would accept a model expressed as a set of Prolog clauses and generate the system's simulated behaviour directly. The feasibility of supporting the process form for the articulation of a simulation model when the simulation engine was working in a three phase cycle was also demonstrated. Since Prolog clauses have a database interpretation (chapter 3), the clauses used for the articulation of simulation models in a given application domain could constitute a knowledge-base from which new models could be retrieved and/or assembled. This approach although viewed as feasible was seen as lacking generality. Research into more general forms for the knowledge representation was indicated, which is described in this chapter.

In this chapter an initial section covers a brief review of the previous approaches to providing computer support for the construction of simulation programs and looks at the various forms that such support has taken in the past. The motivation to approach this problem afresh from the knowledge-based systems point of view has been described. The rest of the chapter covers the design and implementation of a prototype knowledge-based discrete simulation model-builder which has been implemented in Prolog. A subset of the simulation world has been defined in which the various processes in a simulation model can only interact over the use of resources. The knowledge representations devised and the method developed for the knowledge-based construction of simulation models within this simplified simulation world have been discussed. Examples of knowledge-based model building have been presented by building a sequence of partial versions of the 'lorry' model (Annexe 4A). This has been done by first building a very simple partial version of this model where only one entity (merchant) 'flows' through the system. This is followed by building successive partial versions by adding other entities one at a time (ncb and train) and identifying other possible variations and finally by building the full version of the model as described in chapter 4. An example of constructing a simplified 'harbour' model from [POOLE & S, 77] has also been included.

In addition to the interaction of the processes over the use of resources another form of process interaction where the processes can also interact through messages has been attempted by augmenting the knowledge-base. This has been demonstrated with the help of an example, by constructing a more complex model which includes both the 'harbour' and the 'lorry' as sub-models, such that a process from the 'harbour' sub-model can interact with a process from the 'lorry' sub-model through messages (i.e. symbolic tokens) left on a conceptual 'blackboard'. Another version of the harbour model has been used to demonstrate that it is possible to preplan the editing of the code generated by the model builder. The chapter ends with conclusions and some ideas about further research.

5.1. COMPUTER SUPPORT FOR CONSTRUCTING SIMULATION PROGRAMS

It has been noted in the earlier chapters that the difficulties related to faithfully transcribing a simulation model into an executable simulation program, and those related to quickly modifying it when required during experimentation are among the major obstacles in the way of widespread use of simulation technology for problem solving.

> "A dynamic (changing) logical model needs to be turned into a computer model with relative ease. Otherwise, if this part of the process takes a long time, contact with the real world problem starts to diminish.
> p 3;[PAUL, 88]

Computer assistance in the form of simulation program generators (SPGs) has been seen as useful for speeding up the initial programming phase and for making it less error prone. An SPG is, however, a tool for the analyst and the need for communication between the decision maker and the analyst remains. [SHANNON, 86] has voiced the need for a greater amount of computer assistance so that the decision maker can build his/her own simulation models. This in turn implies a need to capture the domain specific knowledge and the knowledge of simulation methodology in the form of a computer software system so that the decision maker can interact with it to conduct his/her own simulation modelling or even a complete simulation study.

In the past, simulation program generator software systems have concentrated on reducing some of the burden of transcribing the semantics of the simulation model expressed in one of the diagrammatic formalisms (e.g. activity cycle diagrams, network models, Petri-nets, system theoretic representations and the like) into an

executable simulation program. These items of software have been known by the name of simulation program generators e.g. CAPS [CLEMENTSON, 80], DRAFT [MATHEWSON, 84].

A major objection to the use of simulation program generators has been that a simulation program produced by such generators needs to be edited to incorporate complex conditions. Generally, it is not possible to capture these conditions by the particular diagrammatic formalism employed to represent the model and therefore the program generator which has been built around such formalism can not produce the executable code for these complex conditions. This has been regarded as an antithesis. People involved with computer programming know it very well that modifying and debugging computer programs, particularly the ones not written by themselves, can take more time and effort than would be required to write the program originally from the start (e.g. [AHMAD, 78]. Further, in order to modify a program produced by a program generator, one still requires a comprehensive knowledge of the diagrammatic formalism employed to define the model and the particular behaviour generation approach built into the program generator and, of course, the syntax of the language in which the program has been generated. The objective of decision makers being able to build and test simulation models with the assistance of computer alone fails, because of the need for implementing changes in the code produced by the simulation program generators.

The following brief review identifies the various forms in which computer support has been provided in the past for the purposes of simulation programming.

5.1.1. THE INTERACTIVE ENTRY OF MODEL COMPONENTS EXPRESSED IN A DIAGRAMMATIC FORMALISM

Once the model has been defined using one of the diagrammatic formalisms and various names have been assigned to its different components, an elementary form of computer assistance provided by simulation program generators has been to permit the interactive entry of these names and other parameters, like the activity durations and the data recording requirements, into the computer. Along with the interactive entry of the model components some form of checking is also provided to keep a check on the consistency and to trap some of the logical errors at the time of the entry. The user interface generally consists of a sequence of computer initiated dialogues. As such, these systems can be described as offering the simulation methodology related assistance, but none related to the application

domain. The simulation methodology is built into the program generator software and the user has little freedom to influence it. An Example is CAPS/ECSL system [CLEMENTSON, 80].

5.1.2. SIMULATION PROGRAM GENERATION IN ALTERNATE LANGUAGES

A simulation program generator, being a distinct item of software, needs to be 'geared' to a programming language or a simulation package which can be described as the target language or package. Simulation languages and packages being of a specialised nature are much less subject to standardisation as compared with the general-purpose computer programming languages for which international standards exist. In recent years, therefore, the trend has been to generate simulation programs in a general purpose language (e.g. CASM Project [PAUL, 88]).

The applicability of a simulation program generator is therefore conditional to the availability of the target simulation language or package. To overcome this problem intermediate representations for the simulation models have been devised. From these representations an executable simulation model, in a number of different simulation languages or packages, can be generated by employing a software module related to that language. Using this approach an existing simulation program generator can be extended to produce a program in a new simulation language by writing a module related to that language without affecting the user interface part. Examples are DRAFT/GASP, DRAFT/SIMSCRIPT systems [MATHEWSON, 74], [MATHEWSON, 84] and [MATHEWSON, 85].

Such intermediate representations are also important in terms of generalising the articulation of simulation models and writing new simulation languages or simulation engines which could accept a simulation program expressed directly using the intermediate representation, thus moving towards what can be described as a unified 'world view'.

5.1.3. MODEL ENTRY AND OUTPUT USING ALTERNATE WORLD VIEWS

The ideas related to having intermediate representations for generating simulation programs in different simulation languages have been further extended to allow for the entry of the model using alternative 'world views' and formalisms. This tends to further modularise the simulation program generator software by

employing a module to capture the model expressed in one of the prevalent formalisms and another to output the simulation model in one of the languages.

In this line of research [DAVIES, 76] and [DAVIES, 79] has proposed a basic unity for expressing the information content of simulation models and has also formulated a grammar for it. The essential information content of a simulation model can be expressed in terms of this basic unity, as a set of 'descriptive units'. Using these ideas it has been reported that it is feasible to design a modular discrete simulation modelling environment which could accept the simulation model in alternative formalisms and also produce the generated program in a number of languages. Using this approach it is possible to provide a simulation program generator which is capable of providing the widest possible applicability, as flexibility can be provided at both input and output ends.

[SUBRAHMANIAN & C, 81] has also reported a descriptor language based on the system theoretic approaches that have been advanced by Oren and Zeigler (e.g. [ZEIGLER, 84]).

The increase in the number of input and output formalism related options offered by these systems does not provide the solution to the problem that the generated program could still require editing before it truly represents the system under study. Further, the nature of support these systems provide essentially remains unaltered, i.e. the simulation methodology is coded into the program generator with little option for the user to influence it.

5.1.4. ASSISTANCE IN MODEL FORMULATION

[DOUKIDIS & P, 85] has reported research into expert systems to aid the simulation model formulation using a natural language understanding approach. This approach attempts to mimic the natural language communication between the decision maker and the analyst, while the software system takes up the role of analyst.

5.1.5. KNOWLEDGE-BASED SIMULATION MODELLING

[KETTENIS, 86] has reviewed the problems and possibilities related to knowledge-based model storage and retrieval, while taking into account the level of detail which would be adequate in a given modelling situation.

[STANDRIDGE, 86] has reviewed the progress related to the development of models from modules. From this paper it appears that different theoretical approaches have concentrated on building models from software modules (e.g. subroutines), whereas practical implementations are only beginning to appear.

> "The automation of model development from modules is beginning to appear in simulation systems such as TESS and MAGEST. The future will see more such systems and more sophisticated ways of linking modules into models."
> p 117; [STANDRIDGE, 86].

[REDDY & FNM, 86] has reported an implementation of a knowledge-based model building system KBS (similar to ROSS [McARTHUR & KN, 84]) using the database approach to knowledge retrieval and assembly, while producing a simulation model in an object oriented programming language.

> "A KBS model is a collection of Schema Representation Language schemata that represent physical and abstract system entities. The schema is the basic unit that represents objects, processes, ideas and so forth."
> p 27;[REDDY & FNM, 86]

5.1.6. KNOWLEDGE-BASED SOFTWARE SPECIFICATION AND PROGRAM SYNTHESIS

In the area of software specification and maintenance [LEUNG & C, 85] has listed a number of advantages related to the use of a Prolog knowledge-base:

> "Using a Prolog knowledge-base to hold software information offers a number of advantages. These include the representation of implicit module relationships, deductive capability, and a user-friendly interface. Through derivation rules, it is only necessary to 'hard-store' a minimal amount of factual data, thus easing maintenance and reducing the risk of inconsistency. Such a software knowledge-base also supports an evolutionary and incremental construction, and is able to respond to change and expansion in a flexible manner."
> p 139;[LEUNG & C, 85].

The following Abstract also points to the use of the deductive approach to program synthesis.

> "Program synthesis is the systematic derivation of a program from a given specification. A deductive approach to program synthesis is presented for the construction of recursive programs. This approach regards program synthesis as a theorem-proving task and relies on a theorem-proving method that combines the features of transformation rules,

unification and mathematical induction within a single framework."
[MANNA & W. 81].

5.2. MOTIVATION

While using a simulation program generator, all the application domain related information needs to be supplied by the user, whereas the role of the program generator is to receive, organize, validate and store it for the final program generation. Knowledge based systems were seen to provide a possibility where the application domain knowledge in the knowledge-base could be offered to the user, thus reducing the burden from the user and at the same time the computer providing 'intelligent' support in the model formulation and the program generation. The 'intelligent' part, of course, would come from the knowledge-base available and the method employed for generating the model. It was hoped that by using the knowledge-based systems paradigm a greater and more pertinent form of computer support could be made available to the user and the interaction between the computer and the user could be made at a higher level than the interactive entry of the names of model components, which can be easily stored in the knowledge base.

It was hoped that by providing a knowledge-base which caters for both the application domain knowledge as well as the simulation methodology knowledge, the user would have the option to influence and extend both -- such flexibility is not possible with simulation program generators. Only the method employed for the model generation would be provided by the software in the form of an inference engine. Further, it was hoped to lead to an extended amount of computer support which would include the domain specific knowledge, in addition to the simulation methodology knowledge provided by the simulation program generators.

Computer assistance in the model formulation and the program generation simultaneously would shift a larger amount of the burden on to the machine, as compared with what has been possible through simulation program generators which provide assistance in the program generation only. It was hoped that it might be possible to do away with the initial requirement of constructing a model using one of the diagrammatic formalisms.

In view of the major objection to the use of simulation program generators, that the program produced by these needs further editing, to explore if by employing a knowledge based framework it is possible to improve upon this situation, and if it

is possible to define a model completely entirely interactively with reference to the knowledge available in the knowledge base. If this could be achieved then a way could be opened for the decision makers to formulate and run their simulation models with knowledge-based computer assistance.

The knowledge based framework was seen as providing an opportunity of devising a model generation facility which could generate all the possible models from a very high level generic articulation. Such a facility was seen relevant within the context of an overall problem solving system working on a 'generate and test' basis, in which a request could be passed on to the model generation facility which could systematically generate all the possible models which are specialisations of the generic articulation. Such models could be tested by other modules of the envisaged problem solver by experimenting with them to ultimately arrive at a suitable configuration of the system which would adequately satisfy the performance criterion (see chapter 7 for a further elaboration of this framework). The ability to systematically alter the logical structure of a simulation model was seen as a novel feature in terms of simulation model generation.

5.3. OBJECTIVE

The objective was set to write a generalized knowledge-based discrete simulation model building system for a simplified simulation domain using Prolog in order to demonstrate the feasibility of such a system. Such a system was envisaged to be capable of accepting a very high level generic specification of the simulation models within the specified domain, and by suitable reference to the knowledge in the knowledge-base, and by 'intelligent' interaction with the user be able to complete the specification and generate executable simulation programs which the simulation engine (chapter 4) could run.

Using the expert systems terminology this involved devising suitable knowledge representations and writing a special purpose inferencing engine for the knowledge-based generation of simple simulation models.

5.4. A SUB SET OF SIMULATION MODELS

It was decided to begin work on a sub set of simulation models. Any experimental results could be applied to larger, more complex simulation models. The following sub sections describe the restrictions to define this sub set.

5.4.1. CLASSES OF ENTITIES

The classes of entities are generated in the beginning and loaded into their respective 'infinity-sets' from where they 'arrive' (as in DRAFT [MATHEWSON, 74]).

At any time an entity is either waiting in a queue (a waiting-set) to take part in a 'service' activity or is taking part in the 'service' activity in what can be called an 'activity-set'.

5.4.2. QUEUES

A queue is a set where entities wait to take part in a specific activity. The discipline of all queues in a model has been limited to 'first in first out' (FIFO) and therefore entities join the queue at the tail-end and leave the queue from the head-end. Entities from more than one class may join the queue on a FIFO basis.

5.4.3. ACTIVITY-SETS

An activity-set is a set associated with a particular 'service' activity in the model. The entity taking part in an activity is 'moved' into an activity-set for the duration of the activity. An activity can take place with an entity on its own (a delay) or an entity together with a resource.

5.4.4. RESOURCES

A resource can have only two states: 'busy' and 'idle'. A resource would be in the 'busy' state while taking part in an activity along with an entity and would be 'idle' when available for its related activity.

5.5. DESIGN ASPECTS

5.5.1. DESIGN PHILOSOPHY

As has been noted earlier, the main objection to the earlier forms of computer support, for simulation program generation, has been that the code produced almost always needs to be edited. In order to improve upon this situation, the domain specific complex conditions can be coded in a fragmentary form in a generic manner and maintained in a knowledge base by the analyst. Using the

knowledge-based systems approach this knowledge can then be brought into play at the time of model building to build a complete model which should not require editing. If this could be shown to be feasible, then a way could be open for the decision makers to build their own domain specific models entirely with the support from a knowledge based model builder.

5.5.2. A FORM FOR THE GENERIC SPECIFICATION OF SIMULATION MODELS

Having decided to use Prolog for the implementation it was decided to use a set of Prolog clauses to articulate the model. The model was viewed as a set of the names of the entities which 'flow' through the model and the names of the activities these engage in during their life cycle in the system. The function symbol 'actor' was used to identify an entity whereas the function symbol 'subgoal' was used to identify an activity. The choice of these function symbols is arbitrary and this particular choice represents a world view for the generic specification of simulation models that entities arriving into the system have a number of subgoals (at least one) to achieve. A sequence of subgoals for an entity represents its life cycle in the system. The names of the entities are captured by a Prolog clause with the predicate name 'model', and there is a 'goal' clause for each entity. This form of articulation presents the model at a generic level of specification. As an example, the following four Prolog clauses articulate the 'lorry' model (chapter 4):

```
model(lorry) :-
        actor(merchant),
        actor(ncb),
        actor(train).

goal(merchant) :-
        subgoal(weigh_in),
        subgoal(m_load),
        subgoal(weigh_out).

goal(ncb) :-
        subgoal(weigh_in),
        subgoal(other_work),
        subgoal(weigh_out).

goal(train) :-
        subgoal(t_unload).
```

An articulation in this form assumes that the knowledge base contains the necessary knowledge about the three 'actors' and the five 'subgoals' specified. A further restriction has been imposed that a 'subgoal' can not be repeated for the same 'actor'.

5.5.3. THE FORM OF THE EXECUTABLE MODEL

On the output side, a design decision was made to generate the executable model in the process form (e.g. Annexe 4C). The interaction between the processes would be limited to the use of the resources.

5.6. A PROTOTYPE KNOWLEDGE-BASED DISCRETE SIMULATION MODEL GENERATION FACILITY

A prototype knowledge-based model generation system was implemented using Prolog. An initial and brief exposition of the system, [AHMAD & H, 88], has been included in Appendix I. A more detailed description follows.

5.6.1. AN OVERVIEW

Figure 5.1 presents an overview of the model generation system as implemented. The generic model at a very high level is presented to the prototype model builder in the form of a computer file which proceeds to complete the specification and realises the specific model the user has in mind. This is done by making reference to the application-domain knowledge in the knowledge-base and through user interaction, if necessary. When the model is completely specified, an executable simulation program is generated with reference to the simulation methodology knowledge also available in the knowledge-base. The executable model consists of two files, one defining the 'static' part of the model and the start-up events, whereas the other file contains the 'dynamic' part of the model which consists of a process description for each 'actor'. These two files can be presented to the simulation engine (Chapter 4) which can 'drive' the model.

5.6.2. THE REPRESENTATION OF THE KNOWLEDGE

Two types of knowledge need to be represented: the application domain knowledge and the simulation methodology knowledge. The representations attempted are discussed in the following.

(A) THE SIMULATION METHODOLOGY KNOWLEDGE

Having decided upon the process form for the executable model, the representation of the simulation methodology knowledge was based on expressing frequently occurring situations in simulation models as process code-segments in a

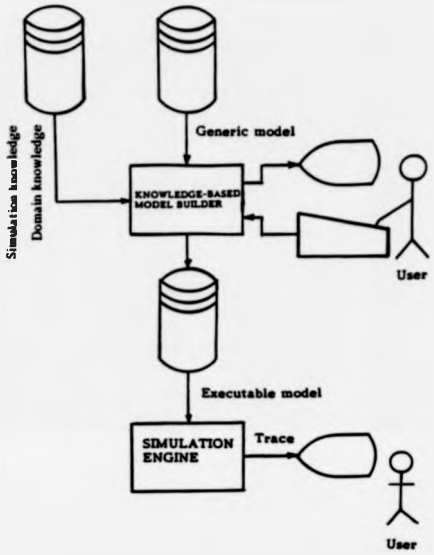


Figure 5.1. An overview of the prototype knowledge-based simulation model building environment.

somewhat generalised form. This approach was seen to provide a possibility for the user to extend the knowledge base by supplying the process fragments of the situations specific to the domain.

To begin with, the process code-segments for four frequently occurring situations in the simulation models were incorporated in the knowledge base. These are stored in the knowledge base by using a predicate 'script'. Each process code segment is identified by a unique name which is the first argument of the 'script' predicate. This name will be used to link a particular script to the knowledge in the application domain. The use of the word script for the predicate name has been motivated by its use by [SCHANK & A, 77] signifying a similar meaning.

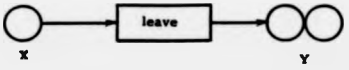
The four 'script' clauses together with their respective diagrammatic equivalents are presented in the following:

Script 'arrive'


```
script(arrive(Process, I_arrival), X, Y) :-
    move(X, Y),
    gen_next(Process, I_arrival, X).
```

As a part of the simplification each entity must 'arrive' before it can start taking part in its first 'subgoal'. A 'subgoal(arrive)' therefore is not explicitly stated in the model articulation.

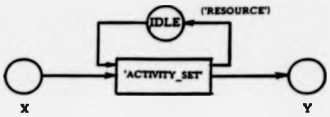
Script 'leave'




```
script(leave, X, Y) :-
    exit_system(X, Y).
```

This script is a dummy activity of zero duration. The 'exit_system' process step 'disengages' the entity from its process when it completes its life cycle in the system and finally returns back to its infinity queue. This is assumed to be the last 'subgoal' of each entity and is therefore not explicitly expressed in the model articulation.

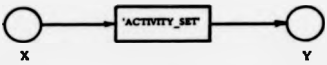
Script 'a'




```
script(a, X, Y) :-
     wait_until(head_of(X)),
     wait_until(idle['RESOURCE']),
     move(X, 'ACTIVITY_SET'),
     seize('RESOURCE'),
     hold('DURATION'),
     release('RESOURCE'),
     move('ACTIVITY_SET', Y).
```

This is one of the most common situations encountered in the simulation models in the service domain. An entity waits in a queue until it is at the head of the queue and the resource is available then the activity is started. After the duration of the activity the resource is released and the entity is moved to another queue

Script 'b'




```
script(b, X, Y) :-
     move( X, 'ACTIVITY_SET'),
     hold('DURATION'),
     move('ACTIVITY_SET', Y).
```

This is also a common situation in simulation models which represents a delay.

(B) THE APPLICATION DOMAIN KNOWLEDGE

The representation of the application domain knowledge has been approached from two directions. The knowledge related to the entities and the knowledge related to the activities. The knowledge related to the entities has been captured by the use of the predicate 'actor_frame'. The following three Prolog clauses represent the knowledge about the entity 'merchant' in the 'lorry' model of chapter 4.

```
actor_frame(merchant, number_in_model(20)).
actor_frame(merchant, arrival_pattern(neg_exp(7.27))).
actor_frame(merchant, first_arrival(7.11)).
```

The first argument signifies the name of the entity, whereas the second argument is a term whose functor signifies a particular aspect of the entity and its argument represents the default value. Any number of aspects related to the entity can be expressed with as many clauses. The representation is therefore flexible and extendible. This representation can be seen as a frame [MINSKY, 75] where the first argument of the predicate 'actor_frame' signifies the name of the frame whereas the second argument represents the 'slot' name and the default value. Although the inheritance aspects related to frames have not been implemented as a part of this particular model building system, these were initially successfully attempted separately with a similar knowledge representation scheme.

The knowledge related to the activities has been captured by a similar set of Prolog clauses using the predicate 'subgoal_frame'. The following three Prolog clauses represent the knowledge about the 'weigh_in' activity of the 'lorry' model (chapter 4).

```
subgoal_frame(weigh_in, resource(weigh_bridge)).
subgoal_frame(weigh_in, duration(default(2.42))).
subgoal_frame(weigh_in, script(a)).
```

The representation is very similar to the one used to capture the knowledge about the entities. The generic nature of the representation should be noted. In the 'lorry' model both the 'merchant' and 'ncb' entities go through the weigh_in but these are not represented specifically for any entity. Another point to note is that there is a reference to one of the 'script' clauses described earlier in the knowledge base. This reference provides a link between the two types of knowledge and is used for code generation purposes.

5.6.3. THE METHOD EMPLOYED FOR MODEL BUILDING

The method employed for the generation of simple models from this knowledge representation consists of first analysing the model as articulated, for the identification of possible process interactions. The various instances of activity-sets and resources are identified and if multiple possibilities exist, these are referred to the user for resolving. Once all the instances of activity-sets and resources have been completely identified a bottom-up synthesis phase is entered into by generating the various queue names to 'couple' the various 'subgoals' for each entity present in the model. All this specific information is then substituted into the respective code-segments represented as 'script' clauses to assemble the process for each entity. At the same time the code for defining the static part of the model is generated from the substitutions, along with the code for scheduling the start-up events.

In the following section the ideas related to the knowledge representation and the method of model building have been further elaborated by following the construction of a number of models of increasing complexity by using the model builder as implemented.

5.7. EXAMPLES OF BUILDING SIMPLE MODELS

Five examples consisting of four partial versions of the 'lorry' model (chapter 4) and one complete version were built to illustrate the working of the model builder. These have been included in Annexes 5A to 5E. The working as presented has been taken directly from the computer files used during the construction of these models and these are un-edited. The headings under which the material has been organised are: the model as articulated at a very high level, the knowledge available in the knowledge base, the executable model as output and the contents of the working memory. For the first illustration (Annexe 5A) the contents of the working memory have been shown at different stages of the model development. In the subsequent examples one snap shot of the working memory at the point just before the executable model is output, this has been included to avoid unnecessary repetition. The contents of the knowledge-base have been repeated when more knowledge is added to it, in order to maintain clarity and to keep the Annexes self contained.

The contents of the working memory consist of a number of Prolog terms stored under various keys. Some of the symbols used for the functors of these terms

have had to be abbreviated and these are expanded to facilitate understanding and have been listed in alphabetic order in the following:

```
res_inst = resource_instance
res_inst_sg_inst = resource_instance_subgoal_instance.
res_inst_sg_instances = resource_instance_subgoal_instances.
resource_sg_instances = resource_subgoal_instances.
sg_inst = subgoal_instance
sg_inst_actr = subgoal_instance_actor.
sg_instance_actors = subgoal_instance_actors.
```

5.7.1. THE PARTIAL 'lorry' MODELS ('merchant' ONLY)

To begin the exposition, a very simple partial version of the 'lorry' model will be considered. In this model only one entity ('merchant') is included and the rest ('mch' and 'train' have been excluded). The model is articulated at a generic level as follows:

```
model(my_model) :-
     actor(merchant).

goal(merchant) :-
     subgoal(weigh_in),
     subgoal(m_load),
     subgoal(weigh_out).
```

From this very high level articulation of the model and the knowledge in the knowledge base (section 5A.1.2.) two models are possible. The first would have one instance of the weigh_bridge serving both the 'weigh_in' and the 'weigh_out' whereas the second would have two instances of the weigh_bridge one serving the 'weigh_in' and the other serving the 'weigh_out'. These possibilities have been shown in Figs. 5.2 and 5.3. As there is nothing in the articulation of the model or in the knowledge base to determine the number of instances of the 'weigh_bridge' in the model the user is asked a question in this regard. Annexe 5A shows the building of the model when the user responds that there is one instance of the 'weigh_bridge' whereas Annexe 5B presents the case when there are two instances of the 'weigh_bridge'.

5.7.2. THE PARTIAL 'lorry' MODELS ('merchant' AND 'mch' ONLY)

The addition of another entity adds to the complexity of the model and therefore to the process of model building. If the two entities are allowed to mix in the queues for weighing in and weighing out then there would be only two possible models depending on the instances of the 'weigh_bridge'. If, however, the two entities queue separately for weighing in and for weighing out then there can be one, two, three or four instances of the 'weigh_bridge' serving the 'weigh_in' and the 'weigh_out' in the various configurations of the model.

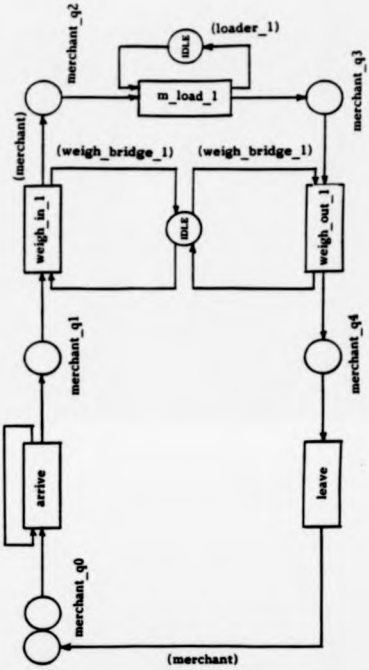


Figure 5.2. The entity cycle diagram for the partial 'lorry' model (merchant only, one weigh-bridge) (Annexe 5A).

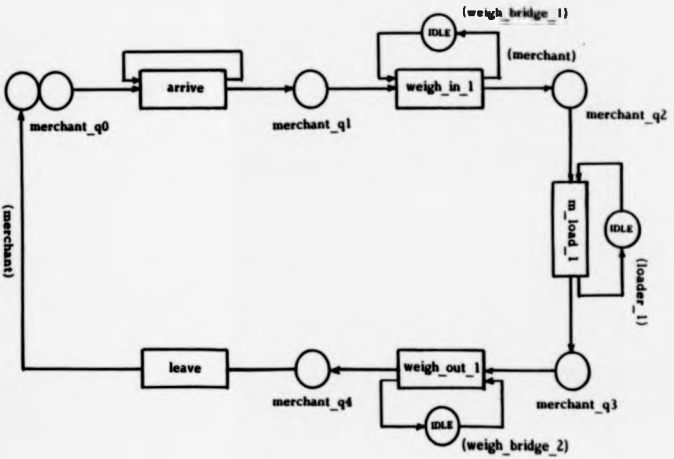


Figure 5.3. The entity cycle diagram for the partial 'lorry' model (merchant only, two weigh-bridges) (Annexe 5B).

Annexe 5C presents the simpler of the two cases (i.e. allowing mixed queueing, which is default) and fig. 5.4 depicts this case using an entity cycle diagram. This model has been articulated as follows:

```
model(my_model) :-
     actor(merchant),
     actor(ncb).

goal(merchant) :-
     subgoal(weigh_in),
     subgoal(m_load),
     subgoal(weigh_out).

goal(ncb) :-
     subgoal(weigh_in),
     subgoal(other_work),
     subgoal(weigh_out).
```

By the addition of two further clauses with the predicate 'own_activity_set' (Annexe 5D) the other case has been specified as shown below:

```
model(my_model) :-
     actor(merchant),
     actor(ncb).

goal(merchant) :-
     subgoal(weigh_in),
     subgoal(m_load),
     subgoal(weigh_out).

goal(ncb) :-
     subgoal(weigh_in),
     subgoal(other_work),
     subgoal(weigh_out).

own_activity_set([merchant], weigh_in).
own_activity_set([merchant], weigh_out).
```

These clauses signify that the two entities queue separately for the weigh in and for the weigh out (fig. 5.5). In both partial models one instance of the weigh_bridge was resolved during user interaction.

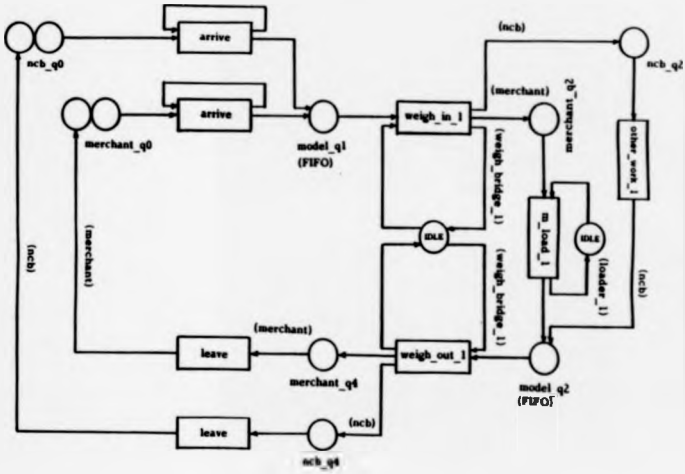


Figure 5.4. The entity cycle diagram for the partial 'lorry' model (merchant and ncb only, one weigh-bridge, mixed queuing) (Annexe 5C).

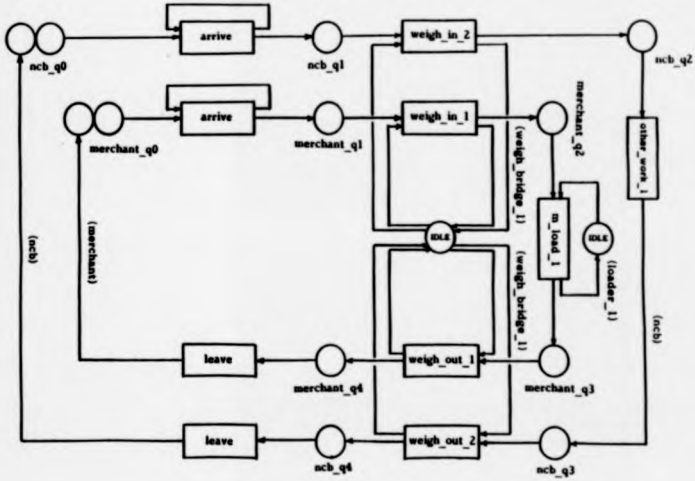


Figure 5.5. The entity cycle diagram for the partial 'lorry' model (merchant and ncb only, one weigh-bridge, separate queuing) (Annexe 5D).

5.7.3. A COMPLETE VERSION OF THE 'lorry' MODEL

Annexe 5E presents the same version of the 'lorry' model as has been included in Annexe 4C and uses the following high level articulation of the model.

```
model(my_model) :-
    actor(merchant),
    actor(nch),
    actor(train).

goal(merchant) :-
    subgoal(weigh_in),
    subgoal(m_load),
    subgoal(weigh_out).

goal(nch) :-
    subgoal(weigh_in),
    subgoal(other_work),
    subgoal(weigh_out).

goal(train) :-
    subgoal(t_unload).

own_activity_set([merchant], weigh_in).
own_activity_set([merchant], weigh_out).
```

By the addition of the 'train' in the model there is a further possible interaction of the merchant with the 'train' over the use of the resource 'loader' and the number of possible models is now twice the number that was previously possible with the 'merchant' and the 'nch' only. It should be noted that there are slight differences between the model in Annexe 4C and the model under discussion (Annexe 5E). The durations of the 'weigh_in' and the 'weigh_out' for the 'nch' have assumed the same default values, as have those for the 'merchant', whereas these are different in the model in the previous chapter. This is so because at present only the logical structure of the model is resolved by the model builder and the default values are used for the durations and the number of instances of each resource. Another difference is that of the presence of a dummy queue alongwith the dummy subgoal 'leave' which is mandatory.

5.7.4. A HARBOUR MODEL

(A) DESCRIPTION OF A 'HARBOUR' MODEL

In the following a simplified description of a 'harbour' model is presented. This has been derived from an original description in [POOLE & S, 77].

A harbour is approached by a narrow entry channel along which only one ship at a time can pass. The following type of ships use the harbour:

1. passenger
2. tanker, and
3. cargo.

The size of the harbour is limited and therefore a ship must not start crossing the channel into the harbour unless its respective unloading berth is vacant.

Of the three types of ship, the passenger boats arrive regularly every 2 hours. They take 12 minutes to go through the channel and their unloading time is given by a uniform distribution between 20 and 40 minutes.

The tankers arrive randomly; their average inter-arrival time is 13 hours. They take 1 hour 40 minutes to pass through the channel and their unloading time is given by an Erlang distribution (mean = 36 hours, std. dev. = 12 hrs).

The cargo ships also arrive randomly; their average inter-arrival time is 6 hours 15 minutes. Their channel passage time is 48 minutes and their unloading time is uniformly distributed between 15 hours and 35 hours.

5.7.5. 'harbour-1' MODEL

For illustration purposes in this section a simplified version of the harbour model as described above will be built using the knowledge-based model builder. In this model the restriction related to the size of the harbour will not be catered for and any number of ships will be permitted in the harbour. In the section 5.8.2 the knowledge-based construction of a 'harbour-2' model will be demonstrated, which will cater for this condition.

Annexe 5F includes the model as articulated, the relevant knowledge in the knowledge base and the executable model as generated. Fig. 5.6 shows this model with the help of an entity cycle diagram.

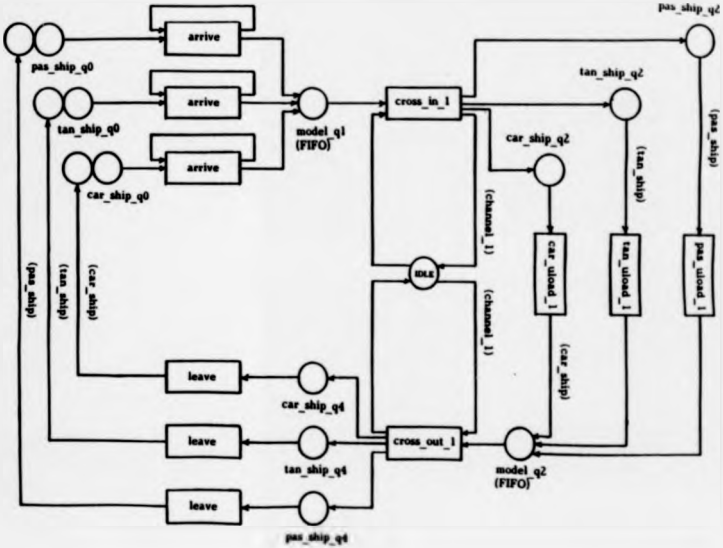


Figure 5.6. The entity cycle diagram for the 'harbour_1' model (Annexe 5F).

For ease of reference the articulation of the model has been repeated in the following:

```
model(my_model) :-
    actor(pas_ship),
    actor(tan_ship),
    actor(car_ship).

goal(pas_ship) :-
    subgoal(cross_in),
    subgoal(pas_uload),
    subgoal(cross_out).

goal(tan_ship) :-
    subgoal(cross_in),
    subgoal(tan_uload),
    subgoal(cross_out).

goal(car_ship) :-
    subgoal(cross_in),
    subgoal(car_uload),
    subgoal(cross_out).
```

The knowledge base now includes the knowledge related to three 'actors' and five 'subgoals' and there is no need for further 'script' clauses. The model as articulated implies that ships mix in the queues for 'cross_in' and 'cross_out', as there is no 'own_activity_set' clause. During the user interaction it was resolved that there is one instance of the channel. An option existed to provide for three separate 'cross_in' subgoals, one for each of the 'actors' as is the case with the unloading activity, because the three ships have different crossing in times. The same could apply for the 'cross_out' subgoal. A second option would be to have only one frame for 'cross_in' and one for 'cross_out' (i.e. as it is now) together with a more elaborate user interaction where the default values of the durations are verified or supplied at the time of the model generation. A further possibility could be that a problem-specific knowledge-base is interactively derived from the generic knowledge-base which would incorporate non-default values and this knowledge base is made available to the model builder at the time of the model building. This approach would support the economy of expression and the generality of knowledge in the knowledge base which is made specific at the time of its use.

5.8. EXTENSIONS TO ALLOW MORE COMPLEX MODELS

5.8.1. A LARGER MODEL

An attempt was made to build a larger model which covers both the 'harbour-1' and the 'lorry' models as sub-models, with slight alterations. For illustration purposes, it will be assumed that the cargo ship in the harbour model brings the

coal and that both the 'merchant' and the 'train' come to load the coal. Also one ship-load of coal is equivalent to two train-loads and enough additional coal to meet all the merchants' requirements adequately. This model specification at a high level is as follows:

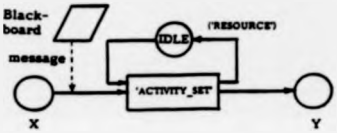
```
model(my_model) :-
    actor(merchant),
    actor(ncb),
    actor(train),
    actor(pas_ship),
    actor(tan_ship),
    actor(car_ship).

goal(merchant) :-
    subgoal(weigh_in),
    subgoal(m_load),
    subgoal(weigh_out).

goal(ncb) :-
    subgoal(weigh_in),
    subgoal(other_work),
    subgoal(weigh_out).

goal(train) :-
    subgoal(t_load).

goal(pas_ship) :-
    subgoal(cross_in),
    subgoal(pas_uload),
    subgoal(cross_out).

goal(tan_ship) :-
    subgoal(cross_in),
    subgoal(tan_uload),
    subgoal(cross_out).

goal(car_ship) :-
    subgoal(cross_in),
    subgoal(car_uload),
    subgoal(cross_out).
```

Annexe 5G depicts the construction of this model. Two more 'script' clauses have been added to the knowledge base to allow the interaction between the cargo ships and the trains through the passing of the message 'coal' and waiting for this message. These scripts are depicted in fig 5.7 diagrammatically. It would be possible for two trains to be loaded after a cargo ship has finished unloading. A message passing facility (a blackboard) and related primitives were added to the simulation engine (chapter 4) to enable the running of the code generated. One instance of each of the channel, the weigh bridge and the loader were resolved during user interaction.

5.8.2. THE 'harbour-2' MODEL

This example has been included to demonstrate that conditions like the one related to the size of the harbour in the harbour model, can be incorporated even when the model builder has the limited capacity for recognising and resolving only

Script 'c'



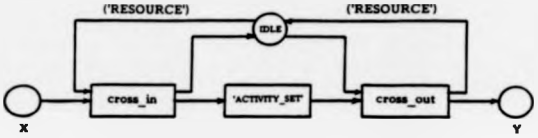
Script 'd'

Figure 5.7. The scripts to provide an interaction between the processes through message passing (Section 5G.1.2.).

one form of process interaction and of substituting for only three keywords (i.e. atoms: 'RESOURCE', 'ACTIVITY_SET' and 'DURATION') in a 'script' clause for the purpose of the code generation. In this example an approach to editing the code generated by the model builder has also been suggested. The articulation of the model is as follows:

```
model(my_model) :-
    actor(pas_ship),
    actor(tan_ship),
    actor(car_ship).

goal(pas_ship) :-
    subgoal(pas_unload).

goal(tan_ship) :-
    subgoal(tan_unload).

goal(car_ship) :-
    subgoal(car_unload).
```

Annexe 5H provides the construction of this model which has been also shown in fig. 5.8. It is easy to recognise the similarity in the processes for the three ships and keeping this in view a 'script' clause corresponding to the following diagram has been devised and included in the knowledge base.



There are three activities in this diagram whereas at present we can substitute only for one 'ACTIVITY_SET' and one 'DURATION' in the script clause. This indicates that the code generated will need editing. This has been incorporated by including the word 'edit' for the durations of cross_in and cross_out in the body of the 'script' clause. In this way it has been possible to preplan the editing which would be needed. It is hoped that this approach to editing the generated code would reduce the extent of the major objection to the use of simulation program generators.

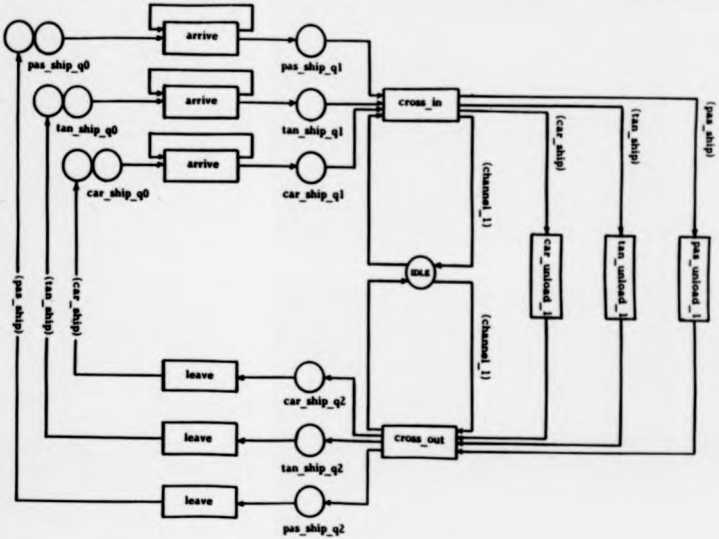


Figure 5.8. The entity cycle diagram for the 'harbour_2' model (Annexe 5H).

5.9. CONCLUSIONS AND FUTURE DEVELOPMENT

5.9.1. CONCLUSIONS

This work has demonstrated that it is feasible to build discrete simulation models by using the knowledge-based systems paradigm while using the logic programming paradigm for implementation. It has been shown that it is possible to separate the model building method from both the application domain as well as the simulation methodology by identifying and reasoning about the possible process interactions and generating a simulation program when all the possible interactions have been fully resolved.

This form of computer assistance in simulation model building shifts a larger amount of the burden onto the computer and the involvement of the user is at an 'intelligent' level as compared with the previous forms of computer assistance related to simulation program generation. The availability of both the simulation methodology knowledge and the application domain knowledge provide for these advantages. Also, the simulation model does not need to be captured with the help of one of the diagrammatic formalisms as is required by most simulation program generators.

The knowledge representations employed provide the knowledge engineer (analyst) with the flexibility of influencing both the simulation methodology as well as the application domain knowledge in the knowledge-base and therefore it is possible to provide for the generation of simulation programs containing complex conditions as their generation is not 'hard-wired' in the program generator software. Also, it is possible to preplan the editing of the generated code when the model complexity exceeds the limits of the system.

5.9.2. FUTURE DEVELOPMENT

The future development of the work described in this chapter would naturally aim at building more complex models than have been described. This would involve adding more complex types of scripts and building facilities for resolving further types of interactions. The articulation of the initial generic model can also be enhanced by further constructs to incorporate complex conditional routes for the entities in the model.

For the model builder to appear more 'intelligent' the knowledge base may be enhanced by a set of rules which would permit or prohibit certain interactions between the processes under varying circumstances. These sets of rules can also form part of an experimental frame and can be partitioned to represent the options to be explored, possibly in a hierarchical fashion. In principle such a rule base can be made comprehensive so that user interaction is minimal, if at all.

In addition to the rules related to the process interactions, another set of rules may concern itself with the preliminary calculations to predict an unstable model (e.g. based on the average interarrival time and the total of the average service times). A warning system may be set up based on these computations to alert the user of a potentially unstable model before experimentation.

Further additions to the knowledge base could include the knowledge of a simulation language intended to be the target language. Such knowledge may be incorporated in the form of the grammar for the language.

A further direction of development could be to provide facilities to output the model in a graphical form for human communication, together with the executable code for experimentation with the model.

END OF CHAPTER 5

ANNEXE 5A

5A.1. THE PARTIAL 'lorry' MODEL (MERCHANT ONLY, ONE INSTANCE OF WHICH BUILDS)

5A.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL

```
model(my_model) :-
    actor(merchant).

goal(merchant) :-
    subgoal(weigh_in),
    subgoal(m_load),
    subgoal(weigh_out).
```

5A.1.2. THE KNOWLEDGE-BASE

The knowledge base consists of three sets of Prolog clauses as follows:

(A) 'actor_frame' CLAUSES

```
actor_frame(merchant, number_in_model(20)).
actor_frame(merchant, arrival_pattern(neg_exp(7.27))).
actor_frame(merchant, first_arrival(7.11)).
```

(B) 'subgoal_frame' CLAUSES

```
subgoal_frame(weigh_in, resource(weigh_bridge)).
subgoal_frame(weigh_in, duration(default(2.42))).
subgoal_frame(weigh_in, script(a)).

subgoal_frame(m_load, resource(loader)).
subgoal_frame(m_load, duration(default(5.84))).
subgoal_frame(m_load, script(a)).

subgoal_frame(weigh_out, resource(weigh_bridge)).
subgoal_frame(weigh_out, duration(default(1.17))).
subgoal_frame(weigh_out, script(a)).
```

(C) 'script' CLAUSES

```
script(arrive(Process, I_arrival), X, Y) :-
    move(X, Y),
    gen_next(Process, I_arrival, X).

script(leave, X, Y) :-
    exit_system(X, Y).

script(a, X, Y) :-
    wait_until(head_of(X)),
    wait_until(idle('RESOURCE')),
    move(X, 'ACTIVITY_SET'),
    seize('RESOURCE'),
    hold('DURATION'),
    release('RESOURCE'),
    move('ACTIVITY_SET', Y).
```

5A.1.3. THE EXECUTABLE MODEL AS GENERATED

The following model, as generated by the model builder, is acceptable to the simulation engine (chapter 4) which can 'drive' the model to provide a trace of the simulated behaviour. The model consists of two files. One file contains the model's statics (i.e. the items which are established before the first time advance) and in the other file the model dynamics (i.e. a process for each 'actor') are delivered.

(A) MODEL STATICS AND START-UP EVENTS

```
:- nl, nl, write('$Defining sets$').
:- vset(m_load_1[1,2,3,4,5,6,7]).
:- vset(merchant_q0[1,2,3,4,5,6,7]).
:- vset(merchant_q1[1,2,3,4,5,6,7]).
:- vset(merchant_q2[1,2,3,4,5,6,7]).
:- vset(merchant_q3[1,2,3,4,5,6,7]).
:- vset(merchant_q4[1,2,3,4,5,6,7]).
:- vset(weigh_in_1[1,2,3,4,5,6,7]).
:- vset(weigh_out_1[1,2,3,4,5,6,7]).

:- nl, nl, write('$Defining resources$').
:- vresource(loader_1,1).
:- vresource(weigh_bridge_1,1).

:- nl, nl, write('$Defining classes$').
:- vclass(merchant(1,2),1,20).

:- nl, nl, write('$Loading classes in their pools$').
:- vload(merchant,1,20,merchant_q0).

:- nl, nl, write('$Scheduling initial events$').
:- introduce(merchant(1),merchant_process,7.11).
```

(B) MODEL DYNAMICS

```
process(merchant_process) :-
    move(merchant_q0,merchant_q1),
    gen_next(merchant_process,7.27,merchant_q0),
    wait_until(head_of(merchant_q1)),
    wait_until(idle(weigh_bridge_1)),
    move(merchant_q1,weigh_in_1),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
    move(weigh_in_1,merchant_q2),
    wait_until(head_of(merchant_q2)),
    wait_until(idle(loader_1)),
    move(merchant_q2,m_load_1),
    seize(loader_1),
    hold(5.84),
    release(loader_1),
    move(m_load_1,merchant_q3),
    wait_until(head_of(merchant_q3)),
    wait_until(idle(weigh_bridge_1)),
    move(merchant_q3,weigh_out_1),
    seize(weigh_bridge_1),
    hold(1.17),
    release(weigh_bridge_1),
    move(weigh_out_1,merchant_q4),
    exit_system(merchant_q4,merchant_q0).
```

5A.1.4. SNAPSHOTS OF THE WORKING MEMORY AT VARIOUS STAGES OF THE MODEL DEVELOPMENT

The model is 'grown' in the working memory from the initial state in which it had been articulated to the final state in which all the instances of queues, activity sets and resources are completely specific and are correctly related to each other. The model is developed in the working memory in the form of Prolog terms stored under various keys some of which are model-specific (e.g. merchant) whereas others are general (e.g. 'RESOURCES'). This is carried out by calling the following clause in the modal building system as a goal:

```
build_model(A) :-
        prepare,
        get_model(A),
        analyse_model,
        synthesise_model,
        get_processes,
        save_model.
```

The subgoal 'prepare' sets the scene in terms of creating a partition of the Arity/Prolog database. The 'get_modal' subgoal makes reference to the model which has been previously 'reconsulted' into the database and sets up the initial keys (explained in chapter 4) for recording the Prolog terms to start the process of model building. The 'analyse_model' subgoal retrieves the relevant information from the knowledge base which has also been previously 'reconsulted' and generates the appropriate instances for the activity sets and the resources. If a resource is shared by two subgoals then the user is prompted to resolve the number and the usage of the resource instances. When all the instances of the activity sets and the resources have been resolved then the subgoal 'synthesise_model' is called. This subgoal generates the configuration of the model by generating the instances of the queues which link the activity sets together and records these under the names of the entities which use these queues. The names of the queues are identified from the name of the entity which uses the respective queues (e.g. merchant_q1, merchant_q2) unless it is used by more than one entity in which case these are identified as 'model_q1', 'model_q2' and so on. At this stage the configuration of the model is completely specified as a set of Prolog terms recorded under the appropriate database keys. The 'get_processes' subgoal generates an executable model from this configuration, and with reference to the simulation methodology knowledge-base (i.e. the 'script' clauses) by supplying appropriate substitutions for the keywords used in the body of the 'script' clauses and assembling these as a process for each 'actor'. The 'save_model' subgoal saves the model as a disk file.

The contents of the working memory at the various stages of the model development for the particular model in this Annexe are presented under the following headings: (A) to begin with, (B) after top down analysis of the model with reference to the knowledge in the knowledge-base along with resolving the instances of the resources by interaction with the user, if this was required (C) after bottom-up synthesis of model, and (D) just before an executable model is output when the behaviour generation knowledge has also been incorporated in the model. Some of the information in the following is of necessity repeated. This has been done to maintain clarity. The contents of the working memory at each stage have been included in full even when there is no change in the terms stored under a particular key from the previous stage. In the following annexes, however, only one snapshot of the working memory has been included.

(A) TO BEGIN WITH

Terms recorded under the key 'ACTORS'

```
actor(merchant)
```

Terms recorded under the key merchant

```
subgoal(weigh_in)
subgoal(m_load)
subgoal(weigh_out)
```

(B) AFTER ANALYSIS AND RESOLVING RESOURCES

Terms recorded under the key 'ACTORS'

```
actor(merchant)
```

Terms recorded under the key merchant

```
subgoal(weigh_in)
subgoal(m_load)
subgoal(weigh_out)
sg_inst(weigh_in,1)
sg_inst(m_load,1)
sg_inst(weigh_out,1)
res_inst(weigh_in,weigh_bridge,1)
res_inst(m_load,loader,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(merchant,7.27)
```

Terms recorded under the key 'SUBGOALS'

```
subgoal(m_load)
subgoal(weigh_in)
subgoal(weigh_out)
```

```
subgoal_resource(m_load,loader)
subgoal_resource(weigh_in,weigh_bridge)
subgoal_resource(weigh_out,weigh_bridge)
sg_instance_actors(m_load(1),[merchant])
sg_instance_actors(weigh_in(1),[merchant])
sg_instance_actors(weigh_out(1),[merchant])
sg_inst_actr(m_load(1),merchant)
sg_inst_actr(weigh_in(1),merchant)
sg_inst_actr(weigh_out(1),merchant)
```

Terms recorded under the key 'RESOURCES'

```
resource(loader)
resource(weigh_bridge)
resource_subgoals(loader,[m_load])
resource_subgoals(weigh_bridge,[weigh_in,weigh_out])
resource_sg_instances(loader,[m_load(1)])
resource_sg_instances(weigh_bridge,[weigh_in(1),weigh_out(1)])
res_inst_sg_instances(loader(1),[m_load(1)])
res_inst_sg_instances(weigh_bridge(1),[weigh_in(1),weigh_out(1)])
res_inst_sg_inst(loader(1),m_load(1))
res_inst_sg_inst(weigh_bridge(1),weigh_in(1))
res_inst_sg_inst(weigh_bridge(1),weigh_out(1))
```

(C) AFTER SYNTHESIS

Terms recorded under the key 'ACTORS'

```
actor(merchant)
```

Terms recorded under the key merchant

```
subgoal(weigh_in)
subgoal(m_load)
subgoal(weigh_out)
sg_inst(weigh_in,1)
sg_inst(m_load,1)
sg_inst(weigh_out,1)
res_inst(weigh_in,weigh_bridge,1)
res_inst(m_load,loader,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(merchant,7.27)
coupling(arrive(merchant,7.27),q1,weigh_in(1))
coupling(weigh_in(1),q2,m_load(1))
coupling(m_load(1),q3,weigh_out(1))
coupling(weigh_out(1),q4,leave)
```

Terms recorded under the key 'SUBGOALS'

```
subgoal(m_load)
subgoal(weigh_in)
subgoal(weigh_out)
subgoal_resource(m_load,loader)
subgoal_resource(weigh_in,weigh_bridge)
subgoal_resource(weigh_out,weigh_bridge)
sg_instance_actors(m_load(1),[merchant])
sg_instance_actors(weigh_in(1),[merchant])
sg_instance_actors(weigh_out(1),[merchant])
sg_inst_actr(m_load(1),merchant)
sg_inst_actr(weigh_in(1),merchant)
sg_inst_actr(weigh_out(1),merchant)
```

Terms recorded under the key 'RESOURCES'

```
resource(loader)
resource(weigh_bridge)
resource_subgoals(loader,[m_load])
resource_subgoals(weigh_bridge,[weigh_in,weigh_out])
resource_eq_instances(loader,[m_load(1)])
resource_eq_instances(weigh_bridge,[weigh_in(1),weigh_out(1)])
res_inst_eq_instances(loader(1),[m_load(1)])
res_inst_eq_instances(weigh_bridge(1),[weigh_in(1),weigh_out(1)])
res_inst_eq_inst(loader(1),m_load(1))
res_inst_eq_inst(weigh_bridge(1),weigh_in(1))
res_inst_eq_inst(weigh_bridge(1),weigh_out(1))
```

(D) BEFORE THE OUTPUT OF EXECUTABLE MODEL

Terms recorded under the key 'ACTORS'

```
actor(merchant)
```

Terms recorded under the key merchant

```
subgoal(weigh_in)
subgoal(m_load)
subgoal(weigh_out)
eq_inst(weigh_in,1)
eq_inst(m_load,1)
eq_inst(weigh_out,1)
res_inst(weigh_in,weigh_bridge,1)
res_inst(m_load,loader,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(merchant,7.27)
coupling(arrive(merchant,7.27),q1,weigh_in(1))
coupling(weigh_in(1),q2,m_load(1))
coupling(m_load(1),q3,weigh_out(1))
coupling(weigh_out(1),q4,leave)
```

Terms recorded under the key 'SUBGOALS'

```
subgoal(m_load)
subgoal(weigh_in)
subgoal(weigh_out)
subgoal_resource(m_load,loader)
subgoal_resource(weigh_in,weigh_bridge)
subgoal_resource(weigh_out,weigh_bridge)
eq_instance_actors(m_load(1),[merchant])
eq_instance_actors(weigh_in(1),[merchant])
eq_instance_actors(weigh_out(1),[merchant])
eq_inst_actr(m_load(1),merchant)
eq_inst_actr(weigh_in(1),merchant)
eq_inst_actr(weigh_out(1),merchant)
```

Terms recorded under the key 'RESOURCES'

```
resource(loader)
resource(weigh_bridge)
resource_subgoals(loader,[m_load])
resource_subgoals(weigh_bridge,[weigh_in,weigh_out])
resource_eq_instances(loader,[m_load(1)])
resource_eq_instances(weigh_bridge,[weigh_in(1),weigh_out(1)])
res_inst_eq_instances(loader(1),[m_load(1)])
res_inst_eq_instances(weigh_bridge(1),[weigh_in(1),weigh_out(1)])
res_inst_eq_inst(loader(1),m_load(1))
res_inst_eq_inst(weigh_bridge(1),weigh_in(1))
res_inst_eq_inst(weigh_bridge(1),weigh_out(1))
```

Terms recorded under the key 'STATICS'

```
vset(merchant_q0(1,2,3,4,5,6,7))
vset(weigh_in_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(merchant_q1(1,2,3,4,5,6,7))
vset(m_load_1(1,2,3,4,5,6,7))
vresource(loader_1,1)
vset(merchant_q2(1,2,3,4,5,6,7))
vset(weigh_out_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(merchant_q3(1,2,3,4,5,6,7))
vset(merchant_q4(1,2,3,4,5,6,7))
vclass(merchant(1,2),1,20)
vload(merchant,1,20,merchant_q0)
introduce(merchant(1),merchant_process,7.11)
```

Term recorded under the key 'PROCESSES'

```
merchant / {{{{{move(merchant_q0,merchant_q1) ,
gen_next(merchant_process,7.27,merchant_q0)} ,
wait_until(head_of(merchant_q1)) ,
wait_until(idle(weigh_bridge_1)) , move(merchant_q1,weigh_in_1) ,
seize(weigh_bridge_1) , hold(2.42) , release(weigh_bridge_1) ,
move(weigh_in_1,merchant_q2)} , wait_until(head_of(merchant_q2)) ,
wait_until(idle(loader_1)) , move(merchant_q2,m_load_1) ,
seize(loader_1) , hold(5.84) , release(loader_1) ,
move(m_load_1,merchant_q3)} , wait_until(head_of(merchant_q3)) ,
wait_until(idle(weigh_bridge_1)) , move(merchant_q3,weigh_out_1) ,
seize(weigh_bridge_1) , hold(3.17) , release(weigh_bridge_1) ,
move(weigh_out_1,merchant_q4)} ,
exit_system(merchant_q4,merchant_q0)}
```

END OF ANNEXE 5A

ANNEXE 5B

5B.1. THE PARTIAL 'lorry' MODEL (MERCHANT ONLY, TWO INSTANCES OF WEIGH-BRIDGE)

5B.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL

As the resource instances are resolved during user interaction the articulation of the model remains the same as in section A5.1.1.

5B.1.2. THE KNOWLEDGE-BASE

The knowledge base also remains the same as in section A5.1.2.

5B.1.3. THE EXECUTABLE MODEL AS GENERATED

(A) MODEL STATICS AND START-UP EVENTS

```
:- nl, nl, write($Defining sets$).
:- vset(m_load_1(1,2,3,4,5,6,7)).
:- vset(merchant_q0(1,2,3,4,5,6,7)).
:- vset(merchant_q1(1,2,3,4,5,6,7)).
:- vset(merchant_q2(1,2,3,4,5,6,7)).
:- vset(merchant_q3(1,2,3,4,5,6,7)).
:- vset(merchant_q4(1,2,3,4,5,6,7)).
:- vset(weigh_in_1(1,2,3,4,5,6,7)).
:- vset(weigh_out_1(1,2,3,4,5,6,7)).

:- nl, nl, write($Defining resources$).
:- vresource(loader_1,1).
:- vresource(weigh_bridge_1,1).
:- vresource(weigh_bridge_2,1).

:- nl, nl, write($Defining classes$).
:- vclass(merchant(1,2),1,20).

:- nl, nl, write($Loading classes in their pools$).
:- vload(merchant,1,20,merchant_q0).

:- nl, nl, write($Scheduling initial events$).
:- introduce(merchant(1),merchant_process,7.11).
```

(B) MODEL DYNAMICS

```
process(merchant_process) :-
    move(merchant_q0,merchant_q1),
    gen_next(merchant_process,7.2),merchant_q0),
    wait_until(head_of(merchant_q1)),
    wait_until(idle(weigh_bridge_1)),
    move(merchant_q1,weigh_in_1),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
```

```
move(weigh_in_1,merchant_q2),
wait_until(head_of(merchant_q2)),
wait_until(idle(loader_1)),
move(merchant_q2,m_load_1),
seize(loader_1),
hold(5.84),
release(loader_1),
move(m_load_1,merchant_q3),
wait_until(head_of(merchant_q3)),
wait_until(idle(weigh_bridge_2)),
move(merchant_q3,weigh_out_1),
seize(weigh_bridge_2),
hold(3.17),
release(weigh_bridge_2),
move(weigh_out_1,merchant_q4),
exit_system(merchant_q4,merchant_q0).
```

5B.1.4. THE CONTENTS OF THE WORKING MEMORY BEFORE THE OUTPUT OF THE EXECUTABLE MODEL

At this stage the development of model is complete and the executable model as presented in the previous section (A5.2.3.) can be readily output.

Terms recorded under the key 'ACTORS'

```
actor(merchant)
```

Terms recorded under the key merchant

```
subgoal(weigh_in)
subgoal(m_load)
subgoal(weigh_out)
sg_inst(weigh_in,1)
sg_inst(m_load,1)
sg_inst(weigh_out,1)
res_inst(weigh_in,weigh_bridge,1)
res_inst(m_load,loader,1)
res_inst(weigh_out,weigh_bridge,2)
arrive(merchant,7.27)
coupling(arrive(merchant,7.27),q1,weigh_in(1))
coupling(weigh_in(1),q2,m_load(1))
coupling(m_load(1),q3,weigh_out(1))
coupling(weigh_out(1),q4,leave)
```

Terms recorded under the key 'SUBGOALS'

```
subgoal(m_load)
subgoal(weigh_in)
subgoal(weigh_out)
subgoal_resource(m_load,loader)
subgoal_resource(weigh_in,weigh_bridge)
subgoal_resource(weigh_out,weigh_bridge)
sg_instance_actors(m_load(1),[merchant])
sg_instance_actors(weigh_in(1),[merchant])
sg_instance_actors(weigh_out(1),[merchant])
sg_inst_actr(m_load(1),merchant)
sg_inst_actr(weigh_in(1),merchant)
sg_inst_actr(weigh_out(1),merchant)
```

Terms recorded under the key 'RESOURCES'

```
resource(loader)
resource(weigh_bridge)
resource_subgoals(loader,[m_load])
resource_subgoals(weigh_bridge,[weigh_in,weigh_out])
resource_sg_instances(loader,[m_load(1)])
resource_sg_instances(weigh_bridge,[weigh_in(1),weigh_out(1)])
res_inst_sg_instances(loader(1),[m_load(1)])
res_inst_sg_instances(weigh_bridge(1),[weigh_in(1)])
res_inst_sg_instances(weigh_bridge(2),[weigh_out(1)])
res_inst_sg_inst(loader(1),m_load(1))
res_inst_sg_inst(weigh_bridge(1),weigh_in(1))
res_inst_sg_inst(weigh_bridge(2),weigh_out(1))
```

Terms recorded under the key 'STATICS'

```
vset(merchant_q0(1,2,3,4,5,6,7))
vset(weigh_in_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(merchant_q1(1,2,3,4,5,6,7))
vset(m_load_1(1,2,3,4,5,6,7))
vresource(loader_1,1)
vset(merchant_q2(1,2,3,4,5,6,7))
vset(weigh_out_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_2,1)
vset(merchant_q3(1,2,3,4,5,6,7))
vset(merchant_q4(1,2,3,4,5,6,7))
vclass(merchant(1,2),1,20)
vload(merchant,1,20,merchant_q0)
introduce(merchant(1),merchant_process,7.11)
```

Term recorded under the key 'PROCESSES'

```
merchant / (((((move(merchant_q0,merchant_q1) ,
gen_next(merchant_process,7.27),merchant_q0)) ,
wait_until(head_of(merchant_q1)) ,
wait_until(idle(weigh_bridge_1)) , move(merchant_q1,weigh_in_1) ,
seize(weigh_bridge_1) , hold(2.42) , release(weigh_bridge_1) ,
move(weigh_in_1,merchant_q2)) , wait_until(head_of(merchant_q2)) ,
wait_until(idle(loader_1)) , move(merchant_q2,m_load_1) ,
seize(loader_1) , hold(5.84) , release(loader_1) ,
move(m_load_1,merchant_q3)) , wait_until(head_of(merchant_q3)) ,
wait_until(idle(weigh_bridge_2)) , move(merchant_q3,weigh_out_1) ,
seize(weigh_bridge_2) , hold(3.17) , release(weigh_bridge_2) ,
move(weigh_out_1,merchant_q4)) ,
exit_system(merchant_q4,merchant_q0))
```

END OF ANNEXE 5B

ANNEXE 5C

5C.1. THE PARTIAL 'lorry' MODEL (MERCHANT AND NCB ONLY, ONE INSTANCE OF WEIGH-BRIDGE, MIXED QUEUEING)

The presence of two 'actors' having some of their 'subgoals' the same adds to the complexity and to the number of possible models that can be generated from the same initial articulation and the same knowledge in the knowledge-base. In the following we shall follow the generation of one model in which there is one instance of the weigh bridge and only mixed queueing is allowed (which is default). The next Annexe takes up another version of the same model where the two 'actors' queue separately.

5C.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL

```
model(my_model) :-
    actor(merchant),
    actor(ncb).

goal(merchant) :-
    subgoal(weigh_in),
    subgoal(m_load),
    subgoal(weigh_out).

goal(ncb) :-
    subgoal(weigh_in),
    subgoal(other_work),
    subgoal(weigh_out).
```

5C.1.2. THE KNOWLEDGE-BASE

(A) 'actor_frame' CLAUSES

```
actor_frame(merchant, number_in_model(20)).
actor_frame(merchant, arrival_pattern(neg_exp(7.27))).
actor_frame(merchant, first_arrival(7.11)).

actor_frame(ncb, number_in_model(20)).
actor_frame(ncb, arrival_pattern(erlang(12.14, 4.3))).
actor_frame(ncb, first_arrival(12.22)).
```

(B) 'subgoal_frame' CLAUSES

```
subgoal_frame(weigh_in, resource(weigh_bridge)).
subgoal_frame(weigh_in, duration(default(2.42))).
subgoal_frame(weigh_in, script(a)).

subgoal_frame(m_load, resource(loader)).
subgoal_frame(m_load, duration(default(5.84))).
subgoal_frame(m_load, script(a)).

subgoal_frame(weigh_out, resource(weigh_bridge)).
subgoal_frame(weigh_out, duration(default(3.17))).
```

```
subgoal_frame(weigh_out, script(a)).

subgoal_frame(other_work, duration(default(22.0))).
subgoal_frame(other_work, script(b)).
```

(C) 'script' CLAUSES

```
script(a, X, Y) :-
    wait_until(head_of(X)),
    wait_until(idle('RESOURCE')),
    move(X, 'ACTIVITY_SET'),
    seize('RESOURCE'),
    hold('DURATION'),
    release('RESOURCE'),
    move('ACTIVITY_SET', Y).

script(b, X, Y) :-
    move( X, 'ACTIVITY_SET'),
    hold('DURATION'),
    move('ACTIVITY_SET', Y).
```

SC.1.3. THE EXECUTABLE MODEL AS GENERATED

(A) MODEL STATICS AND START-UP EVENTS

```
:- nl, nl, write('$Defining sets$).
:- vset(m_load_1(1,2,3,4,5,6,7)).
:- vset(merchant_q0(1,2,3,4,5,6,7)).
:- vset(merchant_q2(1,2,3,4,5,6,7)).
:- vset(merchant_q4(1,2,3,4,5,6,7)).
:- vset(model_q1(1,2,3,4,5,6,7)).
:- vset(model_q2(1,2,3,4,5,6,7)).
:- vset(ncb_q0(1,2,3,4,5,6,7)).
:- vset(ncb_q2(1,2,3,4,5,6,7)).
:- vset(ncb_q4(1,2,3,4,5,6,7)).
:- vset(other_work_1(1,2,3,4,5,6,7)).
:- vset(weigh_in_1(1,2,3,4,5,6,7)).
:- vset(weigh_out_1(1,2,3,4,5,6,7)).

:- nl, nl, write($Defining resources$).
:- vresource(loader_1,1).
:- vresource(weigh_bridge_1,1).

:- nl, nl, write($Defining classes$).
:- vclass(merchant(1,2),1,20).
:- vclass(ncb(1,2),1,20).

:- nl, nl, write($Loading classes in their pools$).
:- vload(merchant,1,20,merchant_q0).
:- vload(ncb,1,20,ncb_q0).

:- nl, nl, write($Scheduling initial events$).
:- introduce(merchant(1),merchant_process,7.11).
:- introduce(ncb(1),ncb_process,12.22).
```

(B) MODEL DYNAMICS

```
process(merchant_process) :-
    move(merchant_q0,model_q1),
    gen_next(merchant_process,7.27,merchant_q0),
    wait_until(head_of(model_q1)),
    wait_until(idle(weigh_bridge_1)),
    move(model_q1,weigh_in_1),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
    move(weigh_in_1,merchant_q2),
```

```
        wait_until(head_of(merchant_q2)),
        wait_until(idle(loader_1)),
        move(merchant_q2,m_load_1),
        seize(loader_1),
        hold(5.84),
        release(loader_1),
        move(m_load_1,model_q2),
        wait_until(head_of(model_q2)),
        wait_until(idle(weigh_bridge_1)),
        move(model_q2,weigh_out_1),
        seize(weigh_bridge_1),
        hold(3.17),
        release(weigh_bridge_1),
        move(weigh_out_1,merchant_q4),
        exit_system(merchant_q4,merchant_q0).

    process(ncb_process) :-
        move(ncb_q0,model_q1),
        gen_next(ncb_process,12.14,ncb_q0),
        wait_until(head_of(model_q1)),
        wait_until(idle(weigh_bridge_1)),
        move(model_q1,weigh_in_1),
        seize(weigh_bridge_1),
        hold(2.42),
        release(weigh_bridge_1),
        move(weigh_in_1,ncb_q2),
        move(ncb_q2,other_work_1),
        hold(22.0),
        move(other_work_1,model_q2),
        wait_until(head_of(model_q2)),
        wait_until(idle(weigh_bridge_1)),
        move(model_q2,weigh_out_1),
        seize(weigh_bridge_1),
        hold(3.17),
        release(weigh_bridge_1),
        move(weigh_out_1,ncb_q4),
        exit_system(ncb_q4,ncb_q0).
```

5C.1.4. THE CONTENTS OF THE WORKING MEMORY BEFORE THE OUTPUT OF THE EXECUTABLE MODEL

Terms recorded under the key 'ACTORS'

```
actor(merchant)
actor(ncb)
mix_q(model_q1,[ncb_q1,merchant_q1])
mix_q(model_q2,[ncb_q3,merchant_q3])
model_q(model_q1,ncb_q1)
model_q(model_q1,merchant_q1)
model_q(model_q2,ncb_q3)
model_q(model_q2,merchant_q3)
```

Terms recorded under the key merchant

```
subgoal(weigh_in)
subgoal(m_load)
subgoal(weigh_out)
sg_inst(weigh_in,1)
sg_inst(m_load,1)
sg_inst(weigh_out,1)
res_inst(weigh_in,weigh_bridge,1)
res_inst(m_load,loader,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(merchant,7.27)
coupling(arrive(merchant,7.27),q1,weigh_in(1))
coupling(weigh_in(1),q2,m_load(1))
coupling(m_load(1),q3,weigh_out(1))
```

```
coupling(weigh_out(1),q4,leave)
```

Terms recorded under the key ncb

```
subgoal(weigh_in)
subgoal(other_work)
subgoal(weigh_out)
sg_inst(weigh_in,1)
sg_inst(other_work,1)
sg_inst(weigh_out,1)
res_inst(weigh_in,weigh_bridge,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(ncb,12.14)
coupling(arrive(ncb,12.14),q1,weigh_in(1))
coupling(weigh_in(1),q2,other_work(1))
coupling(other_work(1),q3,weigh_out(1))
coupling(weigh_out(1),q4,leave)
```

Terms recorded under the key 'SUBGOALS'

```
subgoal(m_load)
subgoal(other_work)
subgoal(weigh_in)
subgoal(weigh_out)
subgoal_resource(m_load,loader)
subgoal_resource(weigh_in,weigh_bridge)
subgoal_resource(weigh_out,weigh_bridge)
sg_instance_actors(m_load(1),[merchant])
sg_instance_actors(other_work(1),[ncb])
sg_instance_actors(weigh_in(1),[merchant,ncb])
sg_instance_actors(weigh_out(1),[merchant,ncb])
sg_inst_actr(m_load(1),merchant)
sg_inst_actr(other_work(1),ncb)
sg_inst_actr(weigh_in(1),merchant)
sg_inst_actr(weigh_in(1),ncb)
sg_inst_actr(weigh_out(1),merchant)
sg_inst_actr(weigh_out(1),ncb)
```

Terms recorded under the key 'RESOURCES'

```
resource(loader)
resource(weigh_bridge)
resource_subgoals(loader,[m_load])
resource_subgoals(weigh_bridge,[weigh_in,weigh_out])
resource_sg_instances(loader,[m_load(1)])
resource_sg_instances(weigh_bridge,[weigh_in(1),weigh_out(1)])
res_inst_sg_instances(loader(1),[m_load(1)])
res_inst_sg_instances(weigh_bridge(1),[weigh_in(1),weigh_out(1)])
res_inst_sg_inst(loader(1),m_load(1))
res_inst_sg_inst(weigh_bridge(1),weigh_in(1))
res_inst_sg_inst(weigh_bridge(1),weigh_out(1))
```

Terms recorded under the key 'STATICS'

```
vset(merchant_q0(1,2,3,4,5,6,7))
vset(weigh_in_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(model_q1(1,2,3,4,5,6,7))
vset(m_load_1(1,2,3,4,5,6,7))
vresource(loader_1,1)
vset(merchant_q2(1,2,3,4,5,6,7))
vset(weigh_out_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(model_q2(1,2,3,4,5,6,7))
vset(merchant_q4(1,2,3,4,5,6,7))
vset(ncb_q0(1,2,3,4,5,6,7))
vset(weigh_in_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(model_q1(1,2,3,4,5,6,7))
```

```
vset(other_work_1(1,2,3,4,5,6,7))
vset(ncb_q2(1,2,3,4,5,6,7))
vset(weigh_out_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(model_q2(1,2,3,4,5,6,7))
vset(ncb_q4(1,2,3,4,5,6,7))
vclass(merchant(1,2),1,20)
vclass(ncb(1,2),1,20)
vload(merchant,1,20,merchant_q0)
vload(ncb,1,20,ncb_q0)
introduce(merchant(1),merchant_process,7.11)
introduce(ncb(1),ncb_process,12.22)
```

Terms recorded under the key 'PROCESSES'

```
merchant / (((((move(merchant_q0,model_q1) ,
gen_next(merchant_process,7.27,merchant_q0)) ,
wait_until(head_of(model_q1)) , wait_until((idle(weigh_bridge_1)) ,
move(model_q1,weigh_in_1) , seize(weigh_bridge_1) , hold(2.42) ,
release(weigh_bridge_1) , move(weigh_in_1,merchant_q2)) ,
wait_until(head_of(merchant_q2)) , wait_until((idle(loader_1)) ,
move(merchant_q2,m_load_1) , seize(loader_1) , hold(5.84) ,
release(loader_1) , move(m_load_1,model_q2)) ,
wait_until(head_of(model_q2)) , wait_until((idle(weigh_bridge_1)) ,
move(model_q2,weigh_out_1) , seize(weigh_bridge_1) , hold(3.17) ,
release(weigh_bridge_1) , move(weigh_out_1,merchant_q4)) ,
exit_system(merchant_q4,merchant_q0))

ncb / (((((move(ncb_q0,model_q1) ,
gen_next(ncb_process,12.14,ncb_q0)) ,
wait_until(head_of(model_q1)) , wait_until((idle(weigh_bridge_1)) ,
move(model_q1,weigh_in_1) , seize(weigh_bridge_1) , hold(2.42) ,
release(weigh_bridge_1) , move(weigh_in_1,ncb_q2)) ,
move(ncb_q2,other_work_1) , hold(22.0) ,
move(other_work_1,model_q2)) , wait_until(head_of(model_q2)) ,
wait_until((idle(weigh_bridge_1)) , move(model_q2,weigh_out_1) ,
seize(weigh_bridge_1) , hold(3.17) , release(weigh_bridge_1) ,
move(weigh_out_1,ncb_q4)) , exit_system(ncb_q4,ncb_q0))
```

END OF ANNEXE 5C

ANNEXE 5D

5D.1. THE PARTIAL 'lorry' MODEL (MERCHANT AND NCB ONLY, ONE INSTANCE OF WEIGH-BRIDGE, SEPARATE QUEUEING)

5D.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL

```
model(my_model) :-
    actor(merchant),
    actor(ncb).

goal(merchant) :-
    subgoal(weigh_in),
    subgoal(m_load),
    subgoal(weigh_out).

goal(ncb) :-
    subgoal(weigh_in),
    subgoal(other_work),
    subgoal(weigh_out).

own_activity_set([merchant], weigh_in).
own_activity_set([merchant], weigh_out).
```

5D.1.2. THE KNOWLEDGE-BASE

The knowledge-base remains the same as in A5.3.2.

5D.1.3. THE EXECUTABLE MODEL

(A) MODEL STATICS AND START-UP EVENTS

```
:- nl, nl, write($Defining sets$).
:- vset(m_load_1(1,2,3,4,5,6,7)).
:- vset(merchant_q0(1,2,3,4,5,6,7)).
:- vset(merchant_q1(1,2,3,4,5,6,7)).
:- vset(merchant_q2(1,2,3,4,5,6,7)).
:- vset(merchant_q3(1,2,3,4,5,6,7)).
:- vset(merchant_q4(1,2,3,4,5,6,7)).
:- vset(ncb_q0(1,2,3,4,5,6,7)).
:- vset(ncb_q1(1,2,3,4,5,6,7)).
:- vset(ncb_q2(1,2,3,4,5,6,7)).
:- vset(ncb_q3(1,2,3,4,5,6,7)).
:- vset(ncb_q4(1,2,3,4,5,6,7)).
:- vset(other_work_1(1,2,3,4,5,6,7)).
:- vset(weigh_in_1(1,2,3,4,5,6,7)).
:- vset(weigh_in_2(1,2,3,4,5,6,7)).
:- vset(weigh_out_1(1,2,3,4,5,6,7)).
:- vset(weigh_out_2(1,2,3,4,5,6,7)).

:- nl, nl, write($Defining resources$).
:- vresource(loader_1,1).
:- vresource(weigh_bridge_1,1).

:- nl, nl, write($Defining classes$).
:- vclass(merchant(1,2),1,20).
:- vclass(ncb(1,2),1,20).
```

```
:- nl, nl, write($Loading classes in their pools$).
:- vload(merchant,1,20,merchant_q0).
:- vload(ncb,1,20,ncb_q0).

:- nl, nl, write($Scheduling initial events$).
:- introduce(merchant(1),merchant_process,7.11).
:- introduce(ncb(1),ncb_process,12.22).
```

(B) MODEL DYNAMICS

```
process(merchant_process) :-
    move(merchant_q0,merchant_q1),
    gen_next(merchant_process,7.27,merchant_q0),
    wait_until(head_of(merchant_q1)),
    wait_until(idle(weigh_bridge_1)),
    move(merchant_q1,weigh_in_1),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
    move(weigh_in_1,merchant_q2),
    wait_until(head_of(merchant_q2)),
    wait_until(idle(loader_1)),
    move(merchant_q2,m_load_1),
    seize(loader_1),
    hold(5.84),
    release(loader_1),
    move(m_load_1,merchant_q3),
    wait_until(head_of(merchant_q3)),
    wait_until(idle(weigh_bridge_1)),
    move(merchant_q3,weigh_out_1),
    seize(weigh_bridge_1),
    hold(3.17),
    release(weigh_bridge_1),
    move(weigh_out_1,merchant_q4),
    exit_system(merchant_q4,merchant_q0).

process(ncb_process) :-
    move(ncb_q0,ncb_q1),
    gen_next(ncb_process,12.14,ncb_q0),
    wait_until(head_of(ncb_q1)),
    wait_until(idle(weigh_bridge_1)),
    move(ncb_q1,weigh_in_2),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
    move(weigh_in_2,ncb_q2),
    move(ncb_q2,other_work_1),
    hold(21.8),
    move(other_work_1,ncb_q3),
    wait_until(head_of(ncb_q3)),
    wait_until(idle(weigh_bridge_1)),
    move(ncb_q3,weigh_out_2),
    seize(weigh_bridge_1),
    hold(3.17),
    release(weigh_bridge_1),
    move(weigh_out_2,ncb_q4),
    exit_system(ncb_q4,ncb_q0).
```

5D.1.4. THE CONTENTS OF THE WORKING MEMORY BEFORE THE OUTPUT OF THE EXECUTABLE MODEL

Terms recorded under the key 'ACTORS'

```
actor(merchant)
actor(ncb)
```

Terms recorded under the key merchant

```
subgoal(weigh_in)
subgoal(m_load)
subgoal(weigh_out)
sg_inst(weigh_in,1)
sg_inst(m_load,1)
sg_inst(weigh_out,1)
res_inst(weigh_in,weigh_bridge,1)
res_inst(m_load,loader,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(merchant,7.27)
coupling(arrive(merchant,7.27),q1,weigh_in(1))
coupling(weigh_in(1),q2,m_load(1))
coupling(m_load(1),q3,weigh_out(1))
coupling(weigh_out(1),q4,leave)
```

Terms recorded under the key ncb

```
subgoal(weigh_in)
subgoal(other_work)
subgoal(weigh_out)
sg_inst(weigh_in,2)
sg_inst(other_work,1)
sg_inst(weigh_out,2)
res_inst(weigh_in,weigh_bridge,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(ncb,12.14)
coupling(arrive(ncb,12.14),q1,weigh_in(2))
coupling(weigh_in(2),q2,other_work(1))
coupling(other_work(1),q3,weigh_out(2))
coupling(weigh_out(2),q4,leave)
```

Terms recorded under the key 'SUBGOALS'

```
subgoal(m_load)
subgoal(other_work)
subgoal(weigh_in)
subgoal(weigh_out)
subgoal_resource(m_load,loader)
subgoal_resource(weigh_in,weigh_bridge)
subgoal_resource(weigh_out,weigh_bridge)
sg_instance_actors(m_load(1),[merchant])
sg_instance_actors(other_work(1),[ncb])
sg_instance_actors(weigh_in(1),[merchant])
sg_instance_actors(weigh_in(2),[ncb])
sg_instance_actors(weigh_out(1),[merchant])
sg_instance_actors(weigh_out(2),[ncb])
sg_inst_actr(m_load(1),merchant)
sg_inst_actr(other_work(1),ncb)
sg_inst_actr(weigh_in(1),merchant)
sg_inst_actr(weigh_in(2),ncb)
sg_inst_actr(weigh_out(1),merchant)
sg_inst_actr(weigh_out(2),ncb)
```

Terms recorded under the key 'RESOURCES'

```
resource(loader)
resource(weigh_bridge)
```

```
resource_subgoals(loader,[m_load|])
resource_subgoals(weigh_bridge,[weigh_in,weigh_out])
resource_eq_instances(loader,[m_load(1)])
resource_eq_instances(weigh_bridge,[weigh_in(2),weigh_in(1),weigh_
out(2),weigh_out(1)])
res_inst_eq_instances(loader(1),[m_load(1)])
res_inst_eq_instances(weigh_bridge(1),[weigh_in(1),weigh_in(2),wei
gh_out(1),weigh_out(2)])
res_inst_eq_inst(loader(1),m_load(1))
res_inst_eq_inst(weigh_bridge(1),weigh_in(1))
res_inst_eq_inst(weigh_bridge(1),weigh_in(2))
res_inst_eq_inst(weigh_bridge(1),weigh_out(1))
res_inst_eq_inst(weigh_bridge(1),weigh_out(2))
```

Terms recorded under the key 'STATICS'

```
vset(merchant_q0(1,2,3,4,5,6,7))
vset(weigh_in_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(merchant_q1(1,2,3,4,5,6,7))
vset(m_load_1(1,2,3,4,5,6,7))
vresource(loader_1,1)
vset(merchant_q2(1,2,3,4,5,6,7))
vset(weigh_out_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(merchant_q3(1,2,3,4,5,6,7))
vset(merchant_q4(1,2,3,4,5,6,7))
vset(ncb_q0(1,2,3,4,5,6,7))
vset(weigh_in_2(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(ncb_q1(1,2,3,4,5,6,7))
vset(other_work_1(1,2,3,4,5,6,7))
vset(ncb_q2(1,2,3,4,5,6,7))
vset(weigh_out_2(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(ncb_q3(1,2,3,4,5,6,7))
vset(ncb_q4(1,2,3,4,5,6,7))
vclass(merchant(1,2),1,20)
vclass(ncb(1,2),1,20)
vload(merchant,1,20,merchant_q0)
vload(ncb,1,20,ncb_q0)
introduce(merchant(1),merchant_process,7.11)
introduce(ncb(1),ncb_process,13.32)
```

Terms recorded under the key 'PROCESSES'

```
merchant / (((((move(merchant_q0,merchant_q1) ,
gen_next(merchant_process,7.27,merchant_q0)) ,
wait_until(head_of(merchant_q1)) ,
wait_until(idle(weigh_bridge_1)) , move(merchant_q1,weigh_in_1) ,
seize(weigh_bridge_1) , hold(2.42) , release(weigh_bridge_1) ,
move(weigh_in_1,merchant_q2)) , wait_until(head_of(merchant_q2)) ,
wait_until(idle(loader_1)) , move(merchant_q2,m_load_1) ,
seize(loader_1) , hold(5.84) , release(loader_1) ,
move(m_load_1,merchant_q3)) , wait_until(head_of(merchant_q3)) ,
wait_until(idle(weigh_bridge_1)) , move(merchant_q3,weigh_out_1) ,
seize(weigh_bridge_1) , hold(3.17) , release(weigh_bridge_1) ,
move(weigh_out_1,merchant_q4)) ,
exit_system(merchant_q4,merchant_q0))
```

```
ncb / {{{{{move(ncb_q0,ncb_q1) ,
gen_next(ncb_process,12.14,ncb_q0)} , wait_until(head_of(ncb_q1)}
, wait_until(idle(weigh_bridge_1)} , move(ncb_q1,weigh_in_2) ,
seize(weigh_bridge_1) , hold(1.42) , release(weigh_bridge_1) ,
move(weigh_in_2,ncb_q2)} , move(ncb_q2,other_work_1) , hold(22.0)
, move(other_work_1,ncb_q3)} , wait_until(head_of(ncb_q3)} ,
wait_until(idle(weigh_bridge_1)} , move(ncb_q3,weigh_out_2) ,
seize(weigh_bridge_1) , hold(3.17) , release(weigh_bridge_1) ,
move(weigh_out_2,ncb_q4)} , exit_system(ncb_q4,ncb_q0)}
```

END OF ANNEXE 5D

ANNEXE 5E

5E.1. THE COMPLETE 'lorry' MODEL (MERCHANT, NCB AND TRAIN, ONE INSTANCE OF MERCHANT-ACTOR, ONE INSTANCE OF LOADER, SEPARATE SCRIPTS)

5E.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL

```
model(my_model) :-
    actor(merchant),
    actor(ncb),
    actor(train).

goal(merchant) :-
    subgoal(weigh_in),
    subgoal(m_load),
    subgoal(weigh_out).

goal(ncb) :-
    subgoal(weigh_in),
    subgoal(other_work),
    subgoal(weigh_out).

goal(train) :-
    subgoal(t_unload).

own_activity_set([merchant], weigh_in).
own_activity_set([merchant], weigh_out).
```

5E.1.2. THE KNOWLEDGE-BASE

(A) 'actor_frame' CLAUSES

```
actor_frame(merchant, number_in_model(20)).
actor_frame(merchant, arrival_pattern(neg_exp(7.27))).
actor_frame(merchant, first_arrival(7.11)).

actor_frame(ncb, number_in_model(20)).
actor_frame(ncb, arrival_pattern(erlang(12.14, 4.3))).
actor_frame(ncb, first_arrival(12.22)).

actor_frame(train, number_in_model(20)).
actor_frame(train, arrival_pattern(normal(11.40, 3.4))).
actor_frame(train, first_arrival(11.3)).
```

(B) 'subgoal_frame' CLAUSES

```
subgoal_frame(weigh_in, resource(weigh_bridge)).
subgoal_frame(weigh_in, duration(default(2.42))).
subgoal_frame(weigh_in, script(a)).

subgoal_frame(m_load, resource(loader)).
subgoal_frame(m_load, duration(default(5.84))).
subgoal_frame(m_load, script(a)).
```

```
subgoal_frame(weigh_out, resource(weigh_bridge)).
subgoal_frame(weigh_out, duration(default(3.17))).
subgoal_frame(weigh_out, script(a)).

subgoal_frame(other_work, duration(default(22.0))).
subgoal_frame(other_work, script(b)).

subgoal_frame(t_unload, resource(loader)).
subgoal_frame(t_unload, duration(default(17.50))).
subgoal_frame(t_unload, script(a)).
```

(C) 'script' CLAUSES

```
script(a, X, Y) :-
    wait_until(head_of(X)),
    wait_until(idle['RESOURCE']),
    move(X, 'ACTIVITY_SET'),
    seize('RESOURCE'),
    hold('DURATION'),
    release('RESOURCE'),
    move('ACTIVITY_SET', Y).

script(b, X, Y) :-
    move( X, 'ACTIVITY_SET'),
    hold('DURATION'),
    move('ACTIVITY_SET', Y).
```

5E.1.3. THE EXECUTABLE MODEL AS GENERATED

(A) MODEL STATICS AND START-UP EVENTS

```
:- nl, nl, write($Defining sets$).
:- vset(m_load_1(1,2,3,4,5,6,7)).
:- vset(merchant_q0(1,2,3,4,5,6,7)).
:- vset(merchant_q1(1,2,3,4,5,6,7)).
:- vset(merchant_q2(1,2,3,4,5,6,7)).
:- vset(merchant_q3(1,2,3,4,5,6,7)).
:- vset(merchant_q4(1,2,3,4,5,6,7)).
:- vset(nch_q0(1,2,3,4,5,6,7)).
:- vset(nch_q1(1,2,3,4,5,6,7)).
:- vset(nch_q2(1,2,3,4,5,6,7)).
:- vset(nch_q3(1,2,3,4,5,6,7)).
:- vset(nch_q4(1,2,3,4,5,6,7)).
:- vset(other_work_1(1,2,3,4,5,6,7)).
:- vset(t_unload_1(1,2,3,4,5,6,7)).
:- vset(train_q0(1,2,3,4,5,6,7)).
:- vset(train_q1(1,2,3,4,5,6,7)).
:- vset(train_q2(1,2,3,4,5,6,7)).
:- vset(weigh_in_1(1,2,3,4,5,6,7)).
:- vset(weigh_in_2(1,2,3,4,5,6,7)).
:- vset(weigh_out_1(1,2,3,4,5,6,7)).
:- vset(weigh_out_2(1,2,3,4,5,6,7)).

:- nl, nl, write($Defining resources$).
:- vresource(loader,1,1).
:- vresource(weigh_bridge,1,1).

:- nl, nl, write($Defining classes$).
:- vclass(merchant(1,2),1,20).
:- vclass(nch(1,2),1,20).
:- vclass(train(1,2),1,20).

:- nl, nl, write($Loading classes in their pools$).
:- vload(merchant,1,20,merchant_q0).
:- vload(nch,1,20,nch_q0).
:- vload(train,1,20,train_q0).
```

```
:- nl, nl, write('Scheduling initial events'),
:- introduce(merchant(1),merchant_process,7.11).
:- introduce(ncb(1),ncb_process,12.22).
:- introduce(train(1),train_process,13.33).
```

(B) MODEL DYNAMICS

```
process(merchant_process) :-
    move(merchant_q0,merchant_q1),
    gen_next(merchant_process,7.27,merchant_q0),
    wait_until(head_of(merchant_q1)),
    wait_until(idle(weigh_bridge_1)),
    move(merchant_q1,weigh_in_1),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
    move(weigh_in_1,merchant_q2),
    wait_until(head_of(merchant_q2)),
    wait_until(idle(loader_1)),
    move(merchant_q2,m_load_1),
    seize(loader_1),
    hold(5.84),
    release(loader_1),
    move(m_load_1,merchant_q3),
    wait_until(head_of(merchant_q3)),
    wait_until(idle(weigh_bridge_1)),
    move(merchant_q3,weigh_out_1),
    seize(weigh_bridge_1),
    hold(3.17),
    release(weigh_bridge_1),
    move(weigh_out_1,merchant_q4),
    exit_system(merchant_q4,merchant_q0).

process(ncb_process) :-
    move(ncb_q0,ncb_q1),
    gen_next(ncb_process,13.14,ncb_q0),
    wait_until(head_of(ncb_q1)),
    wait_until(idle(weigh_bridge_1)),
    move(ncb_q1,weigh_in_2),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
    move(weigh_in_2,ncb_q2),
    move(ncb_q2,other_work_1),
    hold(22.0),
    move(other_work_1,ncb_q3),
    wait_until(head_of(ncb_q3)),
    wait_until(idle(weigh_bridge_1)),
    move(ncb_q3,weigh_out_2),
    seize(weigh_bridge_1),
    hold(3.17),
    release(weigh_bridge_1),
    move(weigh_out_2,ncb_q4),
    exit_system(ncb_q4,ncb_q0).

process(train_process) :-
    move(train_q0,train_q1),
    gen_next(train_process,13.4,train_q0),
    wait_until(head_of(train_q1)),
    wait_until(idle(loader_1)),
    move(train_q1,t_unload_1),
    seize(loader_1),
    hold(19.5),
    release(loader_1),
    move(t_unload_1,train_q2),
    exit_system(train_q2,train_q0).
```

5E.1.4. THE CONTENTS OF THE WORKING MEMORY BEFORE THE OUTPUT OF THE EXECUTABLE MODEL

Terms recorded under the key 'ACTORS'

```
actor(merchant)
actor(ncb)
actor(train)
```

Terms recorded under the key merchant

```
subgoal(weigh_in)
subgoal(m_load)
subgoal(weigh_out)
sg_inst(weigh_in,1)
sg_inst(m_load,1)
sg_inst(weigh_out,1)
res_inst(weigh_in,weigh_bridge,1)
res_inst(m_load,loader,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(merchant,7.27)
coupling(arrive(merchant,7.27),q1,weigh_in(1))
coupling(weigh_in(1),q2,m_load(1))
coupling(m_load(1),q3,weigh_out(1))
coupling(weigh_out(1),q4,leave)
```

Terms recorded under the key ncb

```
subgoal(weigh_in)
subgoal(other_work)
subgoal(weigh_out)
sg_inst(weigh_in,2)
sg_inst(other_work,1)
sg_inst(weigh_out,2)
res_inst(weigh_in,weigh_bridge,1)
res_inst(weigh_out,weigh_bridge,1)
arrive(ncb,12.14)
coupling(arrive(ncb,12.14),q1,weigh_in(2))
coupling(weigh_in(2),q2,other_work(1))
coupling(other_work(1),q3,weigh_out(2))
coupling(weigh_out(2),q4,leave)
```

Terms recorded under the key train

```
subgoal(t_unload)
sg_inst(t_unload,1)
res_inst(t_unload,loader,1)
arrive(train,13.4)
coupling(arrive(train,13.4),q1,t_unload(1))
coupling(t_unload(1),q2,leave)
```

Terms recorded under the key 'SUBGOALS'

```
subgoal(m_load)
subgoal(other_work)
subgoal(t_unload)
subgoal(weigh_in)
subgoal(weigh_out)
subgoal_resource(m_load,loader)
subgoal_resource(t_unload,loader)
subgoal_resource(weigh_in,weigh_bridge)
subgoal_resource(weigh_out,weigh_bridge)
sg_instance_actors(m_load(1),[merchant])
sg_instance_actors(other_work(1),[ncb])
sg_instance_actors(t_unload(1),[train])
sg_instance_actors(weigh_in(1),[merchant])
sg_instance_actors(weigh_in(2),[ncb])
```

```
sg_instance_actors(weigh_out(1),[merchant])
sg_instance_actors(weigh_out(2),[ncb])
sg_inst_actr(m_load(1),merchant)
sg_inst_actr(other_work(1),ncb)
sg_inst_actr(t_unload(1),train)
sg_inst_actr(weigh_in(1),merchant)
sg_inst_actr(weigh_in(2),ncb)
sg_inst_actr(weigh_out(1),merchant)
sg_inst_actr(weigh_out(2),ncb)
```

Terms recorded under the key '<u>RESOURCES</u>'

```
resource(loader)
resource(weigh_bridge)
resource_subgoals(loader,[m_load,t_unload])
resource_subgoals(weigh_bridge,[weigh_in,weigh_out])
resource_sg_instances(loader,[m_load(1),t_unload(1)])
resource_sg_instances(weigh_bridge,[weigh_in(2),weigh_in(1),weigh_
out(2),weigh_out(1)])
res_inst_sg_instances(loader(1),[m_load(1),t_unload(1)])
res_inst_sg_instances(weigh_bridge(1),[weigh_in(1),weigh_in(2),wei
gh_out(1),weigh_out(2)])
res_inst_sg_inst(loader(1),m_load(1))
res_inst_sg_inst(loader(1),t_unload(1))
res_inst_sg_inst(weigh_bridge(1),weigh_in(1))
res_inst_sg_inst(weigh_bridge(1),weigh_in(2))
res_inst_sg_inst(weigh_bridge(1),weigh_out(1))
res_inst_sg_inst(weigh_bridge(1),weigh_out(2))
```

Terms recorded under the key '<u>STATICS</u>'

```
vset(merchant_q0(1,2,3,4,5,6,7))
vset(weigh_in_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(merchant_q1(1,2,3,4,5,6,7))
vset(m_load_1(1,2,3,4,5,6,7))
vresource(loader_1,1)
vset(merchant_q2(1,2,3,4,5,6,7))
vset(weigh_out_1(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(merchant_q3(1,2,3,4,5,6,7))
vset(merchant_q4(1,2,3,4,5,6,7))
vset(ncb_q0(1,2,3,4,5,6,7))
vset(weigh_in_2(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(ncb_q1(1,2,3,4,5,6,7))
vset(other_work_1(1,2,3,4,5,6,7))
vset(ncb_q2(1,2,3,4,5,6,7))
vset(weigh_out_2(1,2,3,4,5,6,7))
vresource(weigh_bridge_1,1)
vset(ncb_q3(1,2,3,4,5,6,7))
vset(ncb_q4(1,2,3,4,5,6,7))
vset(train_q0(1,2,3,4,5,6,7))
vset(t_unload_1(1,2,3,4,5,6,7))
vresource(loader_1,1)
vset(train_q1(1,2,3,4,5,6,7))
vset(train_q2(1,2,3,4,5,6,7))
vclass(merchant(1,2),1,20)
vclass(ncb(1,2),1,20)
vclass(train(1,2),1,20)
vload(merchant,1,20,merchant_q0)
vload(ncb,1,20,ncb_q0)
vload(train,1,20,train_q0)
introduce(merchant(1),merchant_process,7.11)
introduce(ncb(1),ncb_process,12.22)
introduce(train(1),train_process,13.33)
```

Terms recorded under the key 'PROCESSES'

```
merchant / {{{{{move(merchant_q0,merchant_q1) ,
gen_next(merchant_process,7.27,merchant_q0)} ,
wait_until(head_of(merchant_q1)) ,
wait_until(idle(weigh_bridge_1)) , move(merchant_q1,weigh_in_1) ,
seize(weigh_bridge_1) , hold(2.42) , release(weigh_bridge_1) ,
move(weigh_in_1,merchant_q2)} , wait_until(head_of(merchant_q2)) ,
wait_until(idle(loader_1)) , move(merchant_q2,m_load_1) ,
seize(loader_1) , hold(5.84) , release(loader_1) ,
move(m_load_1,merchant_q3)} , wait_until(head_of(merchant_q3)) ,
wait_until(idle(weigh_bridge_1)) , move(merchant_q3,weigh_out_1) ,
seize(weigh_bridge_1) , hold(1.17) , release(weigh_bridge_1) ,
move(weigh_out_1,merchant_q4)} ,
exit_system(merchant_q4,merchant_q0)}

ncb / {{{{{move(ncb_q0,ncb_q1) ,
gen_next(ncb_process,12.14,ncb_q0)} , wait_until(head_of(ncb_q1))
, wait_until(idle(weigh_bridge_1)) , move(ncb_q1,weigh_in_2) ,
seize(weigh_bridge_1) , hold(2.42) , release(weigh_bridge_1) ,
move(weigh_in_2,ncb_q2)} , move(ncb_q2,other_work_1) , hold(22.0)
, move(other_work_1,ncb_q3)} , wait_until(head_of(ncb_q3)) ,
wait_until(idle(weigh_bridge_1)) , move(ncb_q3,weigh_out_2) ,
seize(weigh_bridge_1) , hold(1.17) , release(weigh_bridge_1) ,
move(weigh_out_2,ncb_q4)} , exit_system(ncb_q4,ncb_q0)}

train / {{{move(train_q0,train_q1) ,
gen_next(train_process,13.4,train_q0)} ,
wait_until(head_of(train_q1)) , wait_until(idle(loader_1)) ,
move(train_q1,t_unload_1) , seize(loader_1) , hold(17.5) ,
release(loader_1) , move(t_unload_1,train_q2)} ,
exit_system(train_q2,train_q0)}
```

END OF ANNEXE 5E

ANNEXE 5F

5F.1. THE 'harbour-1' MODEL

In this model the default mixed queuing has been allowed and one instance of the channel is present.

5F.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL

```
model(my_model) :-
     actor(pas_ship),
     actor(tan_ship),
     actor(car_ship).

goal(pas_ship) :-
     subgoal(cross_in),
     subgoal(pas_uload),
     subgoal(cross_out).

goal(tan_ship) :-
     subgoal(cross_in),
     subgoal(tan_uload),
     subgoal(cross_out).

goal(car_ship) :-
     subgoal(cross_in),
     subgoal(car_uload),
     subgoal(cross_out).
```

5F.1.2. THE KNOWLEDGE BASE

(A) 'actor_frame' CLAUSES

```
actor_frame(pas_ship, number_in_model(20)).
actor_frame(pas_ship, arrival_pattern(constant(120.0))).
actor_frame(pas_ship, first_arrival(60.11)).

actor_frame(tan_ship, number_in_model(20)).
actor_frame(tan_ship, arrival_pattern(neg_exp(780.0))).
actor_frame(tan_ship, first_arrival(390.22)).

actor_frame(car_ship, number_in_model(20)).
actor_frame(car_ship, arrival_pattern(neg_exp(375.0))).
actor_frame(car_ship, first_arrival(188.33)).
```

(B) 'subgoal_frame' CLAUSES

```
subgoal_frame(cross_in, resource(channel)).
subgoal_frame(cross_in, duration(default(60))).
subgoal_frame(cross_in, script(a)).
```

```
subgoal_frame(cross_out, resource(channel)).
subgoal_frame(cross_out, duration(default(50))).
subgoal_frame(cross_out, script(a)).

subgoal_frame(pas_uload, duration(default(30))).
subgoal_frame(pas_uload, script(b)).

subgoal_frame(tan_uload, duration(default(2160))).
subgoal_frame(tan_uload, script(b)).

subgoal_frame(car_uload, duration(default(1500))).
subgoal_frame(car_uload, script(b)).
```

(C) 'script' CLAUSES

```
script(a, X, Y) :-
    wait_until(head_of(X)),
    wait_until(idle('RESOURCE')),
    move(X, 'ACTIVITY_SET'),
    seize('RESOURCE'),
    hold('DURATION'),
    release('RESOURCE'),
    move('ACTIVITY_SET', Y).

script(b, X, Y) :-
    move( X, 'ACTIVITY_SET'),
    hold('DURATION'),
    move('ACTIVITY_SET', Y).
```

SF.1.3. THE EXECUTABLE MODEL AS GENERATED

(A) MODEL STATICS AND START-UP EVENTS

```
:- nl, nl, write($Defining sets$).
:- vset(car_ship_q0(1,2,3,4,5,6,7)).
:- vset(car_ship_q2(1,2,3,4,5,6,7)).
:- vset(car_ship_q4(1,2,3,4,5,6,7)).
:- vset(car_uload_1(1,2,3,4,5,6,7)).
:- vset(cross_in_1(1,2,3,4,5,6,7)).
:- vset(cross_out_1(1,2,3,4,5,6,7)).
:- vset(model_q1(1,2,3,4,5,6,7)).
:- vset(model_q2(1,2,3,4,5,6,7)).
:- vset(pas_ship_q0(1,2,3,4,5,6,7)).
:- vset(pas_ship_q2(1,2,3,4,5,6,7)).
:- vset(pas_ship_q4(1,2,3,4,5,6,7)).
:- vset(pas_uload_1(1,2,3,4,5,6,7)).
:- vset(tan_ship_q0(1,2,3,4,5,6,7)).
:- vset(tan_ship_q2(1,2,3,4,5,6,7)).
:- vset(tan_ship_q4(1,2,3,4,5,6,7)).
:- vset(tan_uload_1(1,2,3,4,5,6,7)).

:- nl, nl, write($Defining resources$).
:- vresource(channel_1,1).

:- nl, nl, write($Defining classes$).
:- vclass(car_ship(1,2),1,20).
:- vclass(pas_ship(1,2),1,20).
:- vclass(tan_ship(1,2),1,20).
```

```
:- nl, nl, write('Loading classes in their pools').
:- vload(car_ship,1,20,car_ship_q0).
:- vload(pas_ship,1,20,pas_ship_q0).
:- vload(tan_ship,1,20,tan_ship_q0).

:- nl, nl, write('Scheduling initial events').
:- introduce(car_ship(1),car_ship_process,188.33).
:- introduce(pas_ship(1),pas_ship_process,60.11).
:- introduce(tan_ship(1),tan_ship_process,390.22).
```

(B) MODEL DYNAMICS

```
process(pas_ship_process) :-
    move(pas_ship_q0,model_q1),
    gen_next(pas_ship_process,120.0,pas_ship_q0),
    wait_until(head_of(model_q1)),
    wait_until(idle(channel_1)),
    move(model_q1,cross_in_1),
    seize(channel_1),
    hold(60),
    release(channel_1),
    move(cross_in_1,pas_ship_q2),
    move(pas_ship_q2,pas_uload_1),
    hold(30),
    move(pas_uload_1,model_q2),
    wait_until(head_of(model_q2)),
    wait_until(idle(channel_1)),
    move(model_q2,cross_out_1),
    seize(channel_1),
    hold(50),
    release(channel_1),
    move(cross_out_1,pas_ship_q4),
    exit_system(pas_ship_q4,pas_ship_q0).

process(tan_ship_process) :-
    move(tan_ship_q0,model_q1),
    gen_next(tan_ship_process,780.0,tan_ship_q0),
    wait_until(head_of(model_q1)),
    wait_until(idle(channel_1)),
    move(model_q1,cross_in_1),
    seize(channel_1),
    hold(60),
    release(channel_1),
    move(cross_in_1,tan_ship_q2),
    move(tan_ship_q2,tan_uload_1),
    hold(2160),
    move(tan_uload_1,model_q2),
    wait_until(head_of(model_q2)),
    wait_until(idle(channel_1)),
    move(model_q2,cross_out_1),
    seize(channel_1),
    hold(50),
    release(channel_1),
    move(cross_out_1,tan_ship_q4),
    exit_system(tan_ship_q4,tan_ship_q0).

process(car_ship_process) :-
    move(car_ship_q0,model_q1),
    gen_next(car_ship_process,375.0,car_ship_q0),
    wait_until(head_of(model_q1)),
    wait_until(idle(channel_1)),
    move(model_q1,cross_in_1),
    seize(channel_1),
    hold(60),
    release(channel_1),
    move(cross_in_1,car_ship_q2),
    move(car_ship_q2,car_uload_1),
```

```
hold(1500),
move(car_uload_1,model_q2),
wait_until(head_of(model_q2)),
wait_until(idle(channel_1)),
move(model_q2,cross_out_1),
seize(channel_1),
hold(50),
release(channel_1),
move(cross_out_1,car_ship_q4),
exit_system(car_ship_q4,car_ship_q0).
```

END OF ANNEXE 5F

ANNEXE 5G

5G.1. THE 'harbour+lorry' MODEL

5G.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL

```
model(my_model) :-
    actor(merchant),
    actor(ncb),
    actor(train),
    actor(pas_ship),
    actor(tan_ship),
    actor(car_ship).

goal(merchant) :-
    subgoal(weigh_in),
    subgoal(m_load),
    subgoal(weigh_out).

goal(ncb) :-
    subgoal(weigh_in),
    subgoal(other_work),
    subgoal(weigh_out).

goal(train) :-
    subgoal(t_load).

goal(pas_ship) :-
    subgoal(cross_in),
    subgoal(pas_uload),
    subgoal(cross_out).

goal(tan_ship) :-
    subgoal(cross_in),
    subgoal(tan_uload),
    subgoal(cross_out).

goal(car_ship) :-
    subgoal(cross_in),
    subgoal(car_uload),
    subgoal(cross_out).
```

5G.1.2. THE KNOWLEDGE BASE

(A) 'actor_frame' CLAUSES

```
actor_frame(pas_ship, number_in_model(20)).
actor_frame(pas_ship, arrival_pattern(constant(120.0))).
actor_frame(pas_ship, first_arrival(60.11)).

actor_frame(tan_ship, number_in_model(20)).
actor_frame(tan_ship, arrival_pattern(neg_exp(780.0))).
actor_frame(tan_ship, first_arrival(390.22)).

actor_frame(car_ship, number_in_model(20)).
actor_frame(car_ship, arrival_pattern(neg_exp(375.0))).
actor_frame(car_ship, first_arrival(3.55)).

actor_frame(merchant, number_in_model(20)).
actor_frame(merchant, arrival_pattern(neg_exp(27.27))).
```

```
actor_frame(merchant, first_arrival(7.11)).

actor_frame(ncb, number_in_model(20)).
actor_frame(ncb, arrival_pattern(erlang(12.14, 4.3))).
actor_frame(ncb, first_arrival(12.22)).

actor_frame(train, number_in_model(20)).
actor_frame(train, arrival_pattern(normal(113.40, 3.4))).
actor_frame(train, first_arrival(37.33)).
```

(B) 'subgoal_frame' CLAUSES

```
subgoal_frame(cross_in, resource(channel)).
subgoal_frame(cross_in, duration(default(60))).
subgoal_frame(cross_in, script(a)).

subgoal_frame(cross_out, resource(channel)).
subgoal_frame(cross_out, duration(default(50))).
subgoal_frame(cross_out, script(a)).

subgoal_frame(pas_uload, duration(default(30))).
subgoal_frame(pas_uload, script(b)).

subgoal_frame(tan_uload, duration(default(2160))).
subgoal_frame(tan_uload, script(b)).

subgoal_frame(car_uload, duration(default(50))).
subgoal_frame(car_uload, script(c)).

subgoal_frame(weigh_in, resource(weigh_bridge)).
subgoal_frame(weigh_in, duration(default(2.42))).
subgoal_frame(weigh_in, script(a)).

subgoal_frame(m_load, resource(loader)).
subgoal_frame(m_load, duration(default(5.84))).
subgoal_frame(m_load, script(a)).

subgoal_frame(weigh_out, resource(weigh_bridge)).
subgoal_frame(weigh_out, duration(default(3.17))).
subgoal_frame(weigh_out, script(a)).

subgoal_frame(other_work, duration(default(22.0))).
subgoal_frame(other_work, script(b)).

subgoal_frame(t_load, resource(loader)).
subgoal_frame(t_load, duration(default(17.50))).
subgoal_frame(t_load, script(d)).
```

(C) 'script' CLAUSES

```
script(a, X, Y) :-
    wait_until(head_of(X)),
    wait_until(idle('RESOURCE')),
    move(X, 'ACTIVITY_SET'),
    seize('RESOURCE'),
    hold('DURATION'),
    release('RESOURCE'),
    move('ACTIVITY_SET', Y).

script(b, X, Y) :-
    move( X, 'ACTIVITY_SET'),
    hold('DURATION'),
    move('ACTIVITY_SET', Y).
```

```
script(c, X, Y) :-
     move( X, 'ACTIVITY_SET'),
     hold('DURATION'),
     leave_message(coal),
     leave_message(coal),
     move('ACTIVITY_SET', Y).

script(d, X, Y) :-
     wait_until(head_of(X)),
     wait_until(message_found(coal)),
     wait_until(idle('RESOURCE')),
     remove_message(coal),
     move(X, 'ACTIVITY_SET'),
     seize('RESOURCE'),
     hold('DURATION'),
     release('RESOURCE'),
     move('ACTIVITY_SET', Y).
```

SG.1.3. THE EXECUTABLE MODEL AS GENERATED

(A) MODEL STATICS AND START-UP EVENTS

```
:- nl, nl, write('$Defining sets$').
:- vset(car_ship_q0(1,2,3,4,5,6,7)).
:- vset(car_ship_q1(1,2,3,4,5,6,7)).
:- vset(car_ship_q2(1,2,3,4,5,6,7)).
:- vset(car_ship_q4(1,2,3,4,5,6,7)).
:- vset(car_uload_1(1,2,3,4,5,6,7)).
:- vset(cross_in_1(1,2,3,4,5,6,7)).
:- vset(cross_out_1(1,2,3,4,5,6,7)).
:- vset(m_load_1(1,2,3,4,5,6,7)).
:- vset(marchant_q0(1,2,3,4,5,6,7)).
:- vset(marchant_q2(1,2,3,4,5,6,7)).
:- vset(marchant_q4(1,2,3,4,5,6,7)).
:- vset(model_q1(1,2,3,4,5,6,7)).
:- vset(model_q2(1,2,3,4,5,6,7)).
:- vset(model_q3(1,2,3,4,5,6,7)).
:- vset(model_q4(1,2,3,4,5,6,7)).
:- vset(ncb_q0(1,2,3,4,5,6,7)).
:- vset(ncb_q2(1,2,3,4,5,6,7)).
:- vset(ncb_q4(1,2,3,4,5,6,7)).
:- vset(other_work_1(1,2,3,4,5,6,7)).
:- vset(pas_ship_q0(1,2,3,4,5,6,7)).
:- vset(pas_ship_q2(1,2,3,4,5,6,7)).
:- vset(pas_ship_q4(1,2,3,4,5,6,7)).
:- vset(pas_uload_1(1,2,3,4,5,6,7)).
:- vset(t_load_1(1,2,3,4,5,6,7)).
:- vset(tan_ship_q0(1,2,3,4,5,6,7)).
:- vset(tan_ship_q2(1,2,3,4,5,6,7)).
:- vset(tan_ship_q4(1,2,3,4,5,6,7)).
:- vset(tan_uload_1(1,2,3,4,5,6,7)).
:- vset(train_q0(1,2,3,4,5,6,7)).
:- vset(train_q1(1,2,3,4,5,6,7)).
:- vset(train_q2(1,2,3,4,5,6,7)).
:- vset(weigh_in_1(1,2,3,4,5,6,7)).
:- vset(weigh_out_1(1,2,3,4,5,6,7)).

:- nl, nl, write('$Defining resources$').
:- vresource(channel_1,1).
:- vresource(loader_1,1).
:- vresource(weigh_bridge_1,1).

:- nl, nl, write('$Defining classes$').
:- vclass(car_ship(1,2),1,20).
:- vclass(marchant(1,2),1,20).
:- vclass(ncb(1,2),1,20).
```

```
:- vclass(pas_ship(1,2),1,20).
:- vclass(tan_ship(1,2),1,20).
:- vclass(train(1,2),1,20).

:- nl, nl, write('Loading classes in their pools').
:- vload(car_ship,1,20,car_ship_q0).
:- vload(merchant,1,20,merchant_q0).
:- vload(ncb,1,20,ncb_q0).
:- vload(pas_ship,1,20,pas_ship_q0).
:- vload(tan_ship,1,20,tan_ship_q0).
:- vload(train,1,20,train_q0).

:- nl, nl, write('Scheduling initial events').
:- introduce(car_ship(1),car_ship_process,3.55).
:- introduce(merchant(1),merchant_process,7.11).
:- introduce(ncb(1),ncb_process,12.22).
:- introduce(pas_ship(1),pas_ship_process,60.11).
:- introduce(tan_ship(1),tan_ship_process,390.22).
:- introduce(train(1),train_process,37.33).
```

(B) MODEL DYNAMICS

```
process(merchant_process) :-
    move(merchant_q0,model_q3),
    gen_next(merchant_process,27.27,merchant_q0),
    wait_until(head_of(model_q3)),
    wait_until(idle(weigh_bridge_1)),
    move(model_q3,weigh_in_1),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
    move(weigh_in_1,merchant_q2),
    wait_until(head_of(merchant_q2)),
    wait_until(idle(loader_1)),
    move(merchant_q2,m_load_1),
    seize(loader_1),
    hold(5.84),
    release(loader_1),
    move(m_load_1,model_q4),
    wait_until(head_of(model_q4)),
    wait_until(idle(weigh_bridge_1)),
    move(model_q4,weigh_out_1),
    seize(weigh_bridge_1),
    hold(3.17),
    release(weigh_bridge_1),
    move(weigh_out_1,merchant_q4),
    exit_system(merchant_q4,merchant_q0).

process(ncb_process) :-
    move(ncb_q0,model_q1),
    gen_next(ncb_process,12.14,ncb_q0),
    wait_until(head_of(model_q3)),
    wait_until(idle(weigh_bridge_1)),
    move(model_q3,weigh_in_1),
    seize(weigh_bridge_1),
    hold(2.42),
    release(weigh_bridge_1),
    move(weigh_in_1,ncb_q2),
    move(ncb_q2,other_work_1),
    hold(22.0),
    move(other_work_1,model_q4),
    wait_until(head_of(model_q4)),
    wait_until(idle(weigh_bridge_1)),
    move(model_q4,weigh_out_1),
    seize(weigh_bridge_1),
    hold(3.17),
    release(weigh_bridge_1),
    move(weigh_out_1,ncb_q4),
    exit_system(ncb_q4,ncb_q0).
```

```
process(train_process) :-
     move(train_q0,train_q1),
     gen_next(train_process,113.4,train_q0),
     wait_until(head_of(train_q1)),
     wait_until(message_found(coal)),
     wait_until(idle(loader_1)),
     remove_message(coal),
     move(train_q1,t_load_1),
     seize(loader_1),
     hold(17.5),
     release(loader_1),
     move(t_load_1,train_q2),
     exit_system(train_q2,train_q0).

process(pas_ship_process) :-
     move(pas_ship_q0,model_q1),
     gen_next(pas_ship_process,120.0,pas_ship_q0),
     wait_until(head_of(model_q1)),
     wait_until(idle(channel_1)),
     move(model_q1,cross_in_1),
     seize(channel_1),
     hold(60),
     release(channel_1),
     move(cross_in_1,pas_ship_q2),
     move(pas_ship_q2,pas_uload_1),
     hold(30),
     move(pas_uload_1,model_q2),
     wait_until(head_of(model_q2)),
     wait_until(idle(channel_1)),
     move(model_q2,cross_out_1),
     seize(channel_1),
     hold(50),
     release(channel_1),
     move(cross_out_1,pas_ship_q4),
     exit_system(pas_ship_q4,pas_ship_q0).

process(tan_ship_process) :-
     move(tan_ship_q0,model_q1),
     gen_next(tan_ship_process,780.0,tan_ship_q0),
     wait_until(head_of(model_q1)),
     wait_until(idle(channel_1)),
     move(model_q1,cross_in_1),
     seize(channel_1),
     hold(60),
     release(channel_1),
     move(cross_in_1,tan_ship_q2),
     move(tan_ship_q2,tan_uload_1),
     hold(2160),
     move(tan_uload_1,model_q2),
     wait_until(head_of(model_q2)),
     wait_until(idle(channel_1)),
     move(model_q2,cross_out_1),
     seize(channel_1),
     hold(50),
     release(channel_1),
     move(cross_out_1,tan_ship_q4),
     exit_system(tan_ship_q4,tan_ship_q0).

process(car_ship_process) :-
     move(car_ship_q0,model_q1),
     gen_next(car_ship_process,375.0,car_ship_q0),
     wait_until(head_of(model_q1)),
     wait_until(idle(channel_1)),
     move(model_q1,cross_in_1),
     seize(channel_1),
     hold(60),
     release(channel_1),
     move(cross_in_1,car_ship_q2),
```

```
move(car_ship_q2,car_uload_1),
hold(50),
leave_message(coal),
leave_message(coal),
move(car_uload_1,model_q2),
wait_until(head_of(model_q2)),
wait_until(idle(channel_1)),
move(model_q2,cross_out_1),
seize(channel_1),
hold(50),
release(channel_1),
move(cross_out_1,car_ship_q4),
exit_system(car_ship_q4,car_ship_q0).
```

END OF ANNEXE 5G

ANNEXE 5B

5B.1. THE 'harbour-2' MODEL

5B.1.1. THE MODEL ARTICULATION AT A VERY HIGH LEVEL

```
model(my_model) :-
    actor(pas_ship),
    actor(tan_ship),
    actor(car_ship).

goal(pas_ship) :-
    subgoal(pas_unload).

goal(tan_ship) :-
    subgoal(tan_unload).

goal(car_ship) :-
    subgoal(car_unload).
```

5B.1.2. THE KNOWLEDGE-BASE

(A) 'actor_frame' CLAUSES

```
actor_frame(pas_ship, number_in_model(20)).
actor_frame(pas_ship, arrival_pattern(constant(130.0))).
actor_frame(pas_ship, first_arrival(60.11)).

actor_frame(tan_ship, number_in_model(20)).
actor_frame(tan_ship, arrival_pattern(neg_exp(780.0))).
actor_frame(tan_ship, first_arrival(390.22)).

actor_frame(car_ship, number_in_model(20)).
actor_frame(car_ship, arrival_pattern(neg_exp(375.0))).
actor_frame(car_ship, first_arrival(188.13)).
```

(B) 'subgoal_frame' CLAUSES

```
subgoal_frame(pas_unload, resource(channel)).
subgoal_frame(pas_unload, duration(default(uniform(20, 40)))).
subgoal_frame(pas_unload, script(e)).

subgoal_frame(tan_unload, resource(channel)).
subgoal_frame(tan_unload, duration(default(erlang(2160, 720)))).
subgoal_frame(tan_unload, script(e)).

subgoal_frame(car_unload, resource(channel)).
subgoal_frame(car_unload, duration(default(uniform(900, 2100)))).
subgoal_frame(car_unload, script(e)).
```

(C) 'script' CLAUSES

```
script(e, X, Y) :-
    wait_until(head_of(X)),
    wait_until(is_empty('ACTIVITY_SET')),
    wait_until(idle('RESOURCE')),
    move(X, cross_in),
    seize('RESOURCE'),
    hold(edit),
    release('RESOURCE'),
    move(cross_in, 'ACTIVITY_SET'),
    hold('DURATION'),
    wait_until(idle('RESOURCE')),
    move('ACTIVITY_SET', cross_out),
    seize('RESOURCE'),
    hold(edit),
    release('RESOURCE'),
    move(cross_out, Y).
```

5B.1.3. THE EXECUTABLE MODEL AS GENERATED

(A) MODEL STATICS AND START-UP EVENTS

```
:- nl, nl, write($Defining sets$).
:- vset(car_ship_q0(1,2,3,4,5,6,7)).
:- vset(car_ship_q1(1,2,3,4,5,6,7)).
:- vset(car_ship_q2(1,2,3,4,5,6,7)).
:- vset(car_unload_1(1,2,3,4,5,6,7)).
:- vset(pas_ship_q0(1,2,3,4,5,6,7)).
:- vset(pas_ship_q1(1,2,3,4,5,6,7)).
:- vset(pas_ship_q2(1,2,3,4,5,6,7)).
:- vset(pas_unload_1(1,2,3,4,5,6,7)).
:- vset(tan_ship_q0(1,2,3,4,5,6,7)).
:- vset(tan_ship_q1(1,2,3,4,5,6,7)).
:- vset(tan_ship_q2(1,2,3,4,5,6,7)).
:- vset(tan_unload_1(1,2,3,4,5,6,7)).

:- nl, nl, write($Defining resources$).
:- vresource(channel_1,1).

:- nl, nl, write($Defining classes$).
:- vclass(car_ship(1,2),1,20).
:- vclass(pas_ship(1,2),1,20).
:- vclass(tan_ship(1,2),1,20).

:- nl, nl, write($Loading classes in their pools$).
:- vload(car_ship,1,20,car_ship_q0).
:- vload(pas_ship,1,20,pas_ship_q0).
:- vload(tan_ship,1,20,tan_ship_q0).

:- nl, nl, write($Scheduling initial events$).
:- introduce(car_ship(1),car_ship_process,188.33).
:- introduce(pas_ship(1),pas_ship_process,60.11).
:- introduce(tan_ship(1),tan_ship_process,390.22).
```

(B) MODEL DYNAMICS

```
process(pas_ship_process) :-
    move(pas_ship_q0,pas_ship_q1),
    gen_next(pas_ship_process,120.0,pas_ship_q0),
    wait_until(head_of(pas_ship_q1)),
    wait_until(is_empty(pas_unload_1)),
    wait_until(idle(channel_1)),
    move(pas_ship_q1,cross_in),
    seize(channel_1),
```

```
        hold(edit),
        release(channel_1),
        move(cross_in,pas_unload_1),
        hold(uniform(20,40)),
        wait_until(idle(channel_1)),
        move(pas_unload_1,cross_out),
        seize(channel_1),
        hold(edit),
        release(channel_1),
        move(cross_out,pas_ship_q2),
        exit_system(pas_ship_q2,pas_ship_q0).

    process(tan_ship_process) :-
        move(tan_ship_q0,tan_ship_q1),
        gen_next(tan_ship_process,780.0,tan_ship_q0),
        wait_until(head_of(tan_ship_q1)),
        wait_until(is_empty(tan_unload_1)),
        wait_until(idle(channel_1)),
        move(tan_ship_q1,cross_in),
        seize(channel_1),
        hold(edit),
        release(channel_1),
        move(cross_in,tan_unload_1),
        hold(erlang(2160,720)),
        wait_until(idle(channel_1)),
        move(tan_unload_1,cross_out),
        seize(channel_1),
        hold(edit),
        release(channel_1),
        move(cross_out,tan_ship_q2),
        exit_system(tan_ship_q2,tan_ship_q0).

    process(car_ship_process) :-
        move(car_ship_q0,car_ship_q1),
        gen_next(car_ship_process,375.0,car_ship_q0),
        wait_until(head_of(car_ship_q1)),
        wait_until(is_empty(car_unload_1)),
        wait_until(idle(channel_1)),
        move(car_ship_q1,cross_in),
        seize(channel_1),
        hold(edit),
        release(channel_1),
        move(cross_in,car_unload_1),
        hold(uniform(900,2100)),
        wait_until(idle(channel_1)),
        move(car_unload_1,cross_out),
        seize(channel_1),
        hold(edit),
        release(channel_1),
        move(cross_out,car_ship_q2),
        exit_system(car_ship_q2,car_ship_q0).
```

SE.1.4. THE EXECUTABLE MODEL AFTER EDITING

Editing has been indicated by the underlining of the terms added or edited.

(A) MODEL STATICS AND START-UP EVENTS

```
    :- nl, nl, write('Defining sets'),
    :- vset(car_ship_q0(1,2,3,4,5,6,7)).
    :- vset(car_ship_q1(1,2,3,4,5,6,7)).
    :- vset(car_ship_q2(1,2,3,4,5,6,7)).
    :- vset(car_unload_1(1,2,3,4,5,6,7)).
    :- vset(pas_ship_q0(1,2,3,4,5,6,7)).
    :- vset(pas_ship_q1(1,2,3,4,5,6,7)).
    :- vset(pas_ship_q2(1,2,3,4,5,6,7)).
```

```
:- vset(pas_unload_1[1,2,3,4,5,6,7]).
:- vset(tan_ship_q0[1,2,3,4,5,6,7]).
:- vset(tan_ship_q1[1,2,3,4,5,6,7]).
:- vset(tan_ship_q2[1,2,3,4,5,6,7]).
:- vset(tan_unload_1[1,2,3,4,5,6,7]).
:- vset(cross_in[1,2,3,4,5,6,7]).
:- vset(cross_out[1,2,3,4,5,6,7]).

:- nl, nl, write('Defining resources').
:- vresource(channel_1,1).

:- nl, nl, write('Defining classes').
:- vclass(car_ship[1,2],1,20).
:- vclass(pas_ship[1,2],1,20).
:- vclass(tan_ship[1,2],1,20).

:- nl, nl, write('Loading classes in their pools').
:- vload(car_ship,1,20,car_ship_q0).
:- vload(pas_ship,1,20,pas_ship_q0).
:- vload(tan_ship,1,20,tan_ship_q0).

:- nl, nl, write('Scheduling initial events').
:- introduce(car_ship[1],car_ship_process,188.33).
:- introduce(pas_ship[1],pas_ship_process,60.11).
:- introduce(tan_ship[1],tan_ship_process,390.22).
```

(B) MODEL DYNAMICS

```
process(pas_ship_process) :-
    move(pas_ship_q0,pas_ship_q1),
    gen_next(pas_ship_process,120.0,pas_ship_q0),
    wait_until(head_of(pas_ship_q1)),
    wait_until(is_empty(pas_unload_1)),
    wait_until(idle(channel_1)),
    move(pas_ship_q1,cross_in),
    seize(channel_1),
    hold(12),
    release(channel_1),
    move(cross_in,pas_unload_1),
    hold(uniform(20,40)),
    wait_until(idle(channel_1)),
    move(pas_unload_1,cross_out),
    seize(channel_1),
    hold(12),
    release(channel_1),
    move(cross_out,pas_ship_q2),
    exit_system(pas_ship_q2,pas_ship_q0).

process(tan_ship_process) :-
    move(tan_ship_q0,tan_ship_q1),
    gen_next(tan_ship_process,780.0,tan_ship_q0),
    wait_until(head_of(tan_ship_q1)),
    wait_until(is_empty(tan_unload_1)),
    wait_until(idle(channel_1)),
    move(tan_ship_q1,cross_in),
    seize(channel_1),
    hold(100),
    release(channel_1),
    move(cross_in,tan_unload_1),
    hold(erlang(2160,720)),
    wait_until(idle(channel_1)),
    move(tan_unload_1,cross_out),
    seize(channel_1),
    hold(100),
    release(channel_1),
    move(cross_out,tan_ship_q2),
    exit_system(tan_ship_q2,tan_ship_q0).

process(car_ship_process) :-
```

```
move(car_ship_q0,car_ship_q1),
gen_next(car_ship_process,375.0,car_ship_q0),
wait_until(head_of(car_ship_q1)),
wait_until(is_empty(car_unload_1)),
wait_until(idle(channel_1)),
move(car_ship_q1,cross_in),
seize(channel_1),
hold(48),
release(channel_1),
move(cross_in,car_unload_1),
hold(uniform(900.2100)),
wait_until(idle(channel_1)),
move(car_unload_1,cross_out),
seize(channel_1),
hold(48),
release(channel_1),
move(cross_out,car_ship_q2),
exit_system(car_ship_q2,car_ship_q0).
```

END OF ANNEXE 5H

CHAPTER 6: 'WISE' -- A PROTOTYPE KNOWLEDGE-BASED DISCRETE SIMULATION MODELLING ENVIRONMENT

INTRODUCTION

The coverage of research described in chapter 5 concentrated on the process of knowledge-based model building and the related knowledge representations. This chapter describes further research to build upon the model building method to demonstrate that it can be developed into an expert simulation modelling environment.

In chapter 5 it was assumed that the user would edit the high level generic articulation of his/her intended simulation model into a computer file, before presenting it to the knowledge-based model building system. This in turn implied that he/she would need to have a prior knowledge of the contents of the knowledge-base, which is made available to the model building system during the model construction. In order to provide computer assistance in this regard an interactive knowledge-based model acquisition system was written using Prolog. The initial sections in this chapter cover an exposition of its design and implementation. It proposes to demonstrate that it is feasible to provide adequate knowledge-based computer support to interactively define a simulation model without requiring the user to go through any essential paper and pencil work. This has been seen as a step forward from the other forms of computer assistance for model building, which require the simulation model to be first expressed with the help of a diagrammatical formalism which forms the integral part of the design of such software.

This chapter illustrates the user interface aspects of both the model acquisition system and the model building system of chapter 5 with the help of sequences of screen images. The knowledge-based specification and construction of the 'lorry' model (Annexe 4A) has been used as a vehicle of exposition.

The model acquisition system together with the model building system (chapter 5) and the simulation engine (chapter 4) were compiled separately and were integrated to constitute a prototype knowledge-based discrete simulation modelling environment. This environment shall be referred to as 'WISE' (Warwick Intelligent Simulation Environment).

The final part of this chapter covers the ideas related to the possible ways in which the 'WISE' system approach can be generalised to provide for the construction of more complex simulation models.

6.1. MOTIVATION

To provide an interactive knowledge-based simulation modelling environment for use by the decision makers (managers, engineers, ...), to build and run their own simulation models [SHANNON, 86]. This was seen to be feasible if the knowledge based system could offer its knowledge to the user in an appropriate form to assist her/him in defining the initial high level generic articulation of the intended model, so that the knowledge-based model building could proceed from it (chapter 5).

6.2. THE CONCEPTUAL FRAMEWORK

The high level articulation for a simulation model as described in chapter 5 can be viewed as shown in fig 6.1. where the circle nodes represent 'actors' and the rectangles depict the 'subgoals'. The specification of a simulation model can be regarded as assigning 'actor' names to the circle nodes and appropriate 'subgoal' names to the rectangle nodes. The model is elicited from the user by first offering him/her the choice from the 'actors' on which knowledge is available in the knowledge-base. When the 'actors' have been chosen, then through a similar dialogue a choice of 'subgoals' is offered for each 'actor' previously selected. When the selection is complete the high level generic model is output as a set of Prolog clauses.

Another knowledge-based model specification system KBMC has been reported in [MURRAY & S, 88]. This work is concurrent with the research described in this thesis. The implementation of KBMC using an expert system building tool (the OPS83 system) represents a major advancement, because of the use of the knowledge based systems paradigm having the advantage that the user can extend the capabilities of such a system by adding more rules to the rule-base. This type of extension is not possible in the 'conventional' simulation program generator software. However, in KBMC the application domain knowledge is not represented at all and the names of the elements of the models are acquisitioned through an interactive dialogue much like other 'non-intelligent' simulation program generators (e.g. CAPS [CLEMENTSON, 80]).

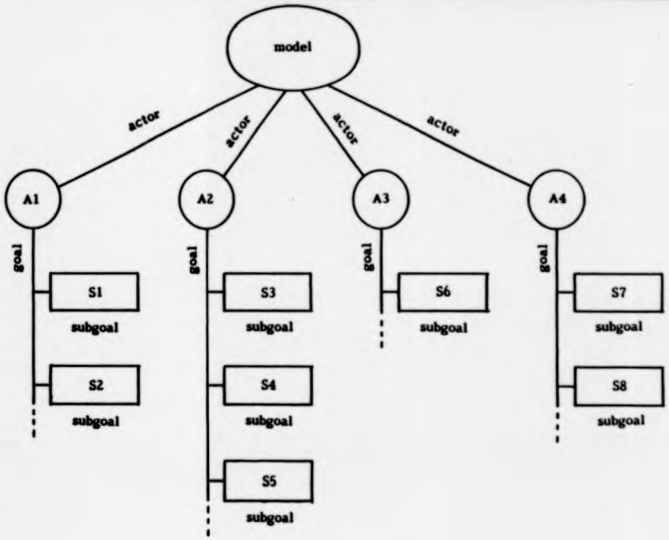


Figure 6.1. A conceptual view of a simulation model at the generic level.

6.3. INITIAL PROBLEMS

During an initial implementation it was realized that it is possible to build quite meaningless models by using the method stated above. For example, suppose in the knowledge-base there is an 'actor' 'train' and also a 'subgoal' 'shopping'. While making selections it is quite possible to select 'shopping' as a 'subgoal' for 'train'. To caution the user of this possibility an extension in the knowledge representation was made. This was done by introducing a 'subgoal_list' slot in the 'actor_frame' clauses for each 'actor' to signify meaningful 'subgoals' for that 'actor'. In the following the 'actor_frame' clauses for 'merchant' (from Chapter 5) have been shown with the new slot added.

```
actor_frame(merchant, number_in_model(20)).
actor_frame(merchant, arrival_pattern(neg_exp(7.27))).
actor_frame(merchant, first_arrival(7.11)).
actor_frame(merchant, subgoal_list([
    weigh_in,
    weigh_out,
    m_load])).
```

Also, it was found useful to keep the 'actor_frame' and 'subgoal_frame' clauses related to a given model in separate files when these clauses are first formulated. These files can be integrated into the knowledge base by using the operating system 'copy' command.

6.4. IMPLEMENTATION

Fig. 6.2 shows an overview of the 'WISE' system. A comparison of this figure with figures 5.1 and 4.1 would assist in visualizing the development of the system.

A display window was implemented by using Prolog for displaying the names of the 'actors' and the 'subgoals'. A chain of Prolog atoms stored under a database key (chapter 4) could be displayed within the window and desired entry could be highlighted with the help of the arrow- and the page-keys on the keypad. An entry could be selected by highlighting it and then pressing the 'return' key. A selected entry could be deleted by highlighting it and then pressing the 'del' key.

A number of dialogues were designed which displayed the respective entries ('actor' or 'subgoal') and invited the user to select among these. A display of helpful hints about the actions the user can take in a given stage of dialogue was implemented. An elementary level of error handling was also implemented e.g. an

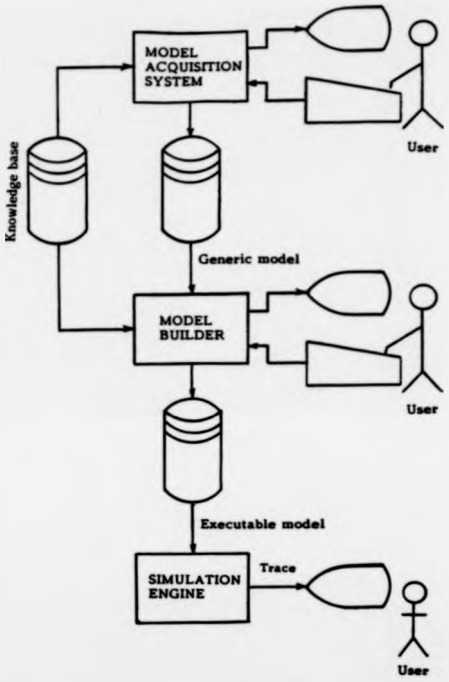


Figure 6.2. An overview of 'WISE' (Warwick Intelligent Simulation Environment).

attempt for duplicate selection of an already selected 'actor' (or 'subgoal') was implemented as an error with suitable reporting.

6.5. AN EXAMPLE

In the following a sample session with the 'WISE' system will be presented with the help of a sequence of screen images. An interactive definition of the 'lorry' model (chapters 4 and 5) will be used as a vehicle for this exposition.

6.5.1. THE KNOWLEDGE-BASE

For the purpose of this example the following knowledge-base was used. The topic of knowledge representation has been covered in Chapter 5.

'actor_frame' CLAUSES

```
actor_frame(merchant, number_in_model(20)).
actor_frame(merchant, arrival_pattern(neg_exp(7.27))).
actor_frame(merchant, first_arrival(7.11)).
actor_frame(merchant, subgoal_list([
    weigh_in,
    weigh_out,
    m_load])).

actor_frame(ncb, number_in_model(20)).
actor_frame(ncb, arrival_pattern(erlang(12.14, 4.3))).
actor_frame(ncb, first_arrival(12.22)).
actor_frame(ncb, subgoal_list([
    weigh_in,
    weigh_out,
    other_work])).

actor_frame(train, number_in_model(20)).
actor_frame(train, arrival_pattern(normal(13.40, 3.4))).
actor_frame(train, first_arrival(13.33)).
actor_frame(train, subgoal_list([
    t_unload])).

actor_frame(pas_ship, number_in_model(20)).
actor_frame(pas_ship, arrival_pattern(constant(120.0))).
actor_frame(pas_ship, first_arrival(60.11)).
actor_frame(pas_ship, subgoal_list([
    cross_in,
    cross_out,
    pas_uload])).

actor_frame(tan_ship, number_in_model(20)).
actor_frame(tan_ship, arrival_pattern(neg_exp(780.0))).
actor_frame(tan_ship, first_arrival(390.22)).
actor_frame(tan_ship, subgoal_list([
    cross_in,
    cross_out,
    tan_uload])).
```

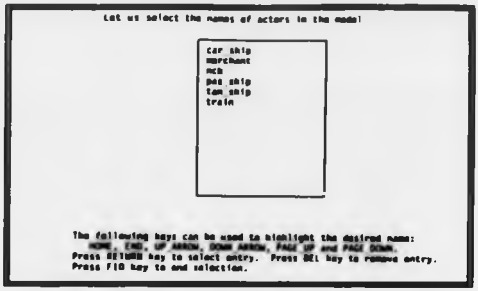
```
    actor_frame(car_ship, number_in_model(20)).
    actor_frame(car_ship, arrival_pattern(neg_exp(375.0))).
    actor_frame(car_ship, first_arrival(188.5)).
    actor_frame(car_ship, subgoal_list([
         cross_in,
         cross_out,
         car_uload])).

'subgoal_frame' CLAUSES

    subgoal_frame(weigh_in, resource(weigh_bridge)).
    subgoal_frame(weigh_in, duration(default(2.42))).
    subgoal_frame(weigh_in, script(a)).

    subgoal_frame(m_load, resource(loader)).
    subgoal_frame(m_load, duration(default(5.84))).
    subgoal_frame(m_load, script(a)).

    subgoal_frame(weigh_out, resource(weigh_bridge)).
    subgoal_frame(weigh_out, duration(default(3.17))).
    subgoal_frame(weigh_out, script(a)).

    subgoal_frame(other_work, duration(default(22.0))).
    subgoal_frame(other_work, script(b)).

    subgoal_frame(t_unload, resource(loader)).
    subgoal_frame(t_unload, duration(default(17.50))).
    subgoal_frame(t_unload, script(a)).

    subgoal_frame(cross_in, resource(channel)).
    subgoal_frame(cross_in, duration(default(60))).
    subgoal_frame(cross_in, script(a)).

    subgoal_frame(cross_out, resource(channel)).
    subgoal_frame(cross_out, duration(default(50))).
    subgoal_frame(cross_out, script(a)).

    subgoal_frame(pas_uload, duration(default(30))).
    subgoal_frame(pas_uload, script(b)).

    subgoal_frame(tan_uload, duration(default(2160))).
    subgoal_frame(tan_uload, script(b)).

    subgoal_frame(car_uload, duration(default(1500))).
    subgoal_frame(car_uload, script(b)).

'script' CLAUSES

    script(arrive(Process, I_arrival), X, Y) :-
         move(X, Y),
         gen_next(Process, I_arrival, X).

    script(leave, X, Y) :-
         exit_system(X, Y).

    script(a, X, Y) :-
         wait_until(head_of(X)),
         wait_until(idle('RESOURCE')),
         move(X, 'ACTIVITY_SET'),
         seize('RESOURCE'),
         hold('DURATION'),
         release('RESOURCE'),
         move('ACTIVITY_SET', Y).

    script(b, X, Y) :-
         move( X, 'ACTIVITY_SET'),
         hold('DURATION'),
         move('ACTIVITY_SET', Y).
```

6.5.2. THE INTERACTIVE DEFINITION OF THE 'lorry' MODEL

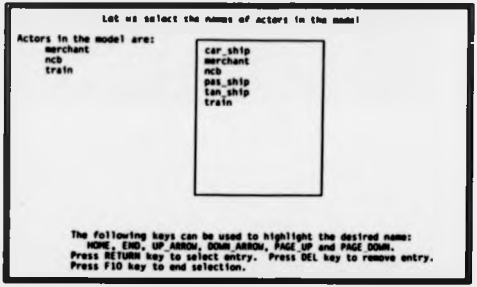
Using the knowledge-base shown previously, an interactive session with the model acquisition system will be described with the help of screen images.

The following screen shows the operating system command level at which the command 'wise' has been entered. Upon invocation, the model acquisition system consults the knowledge base and makes an announcement to that effect, which appears near the lower end of the screen.

After loading the knowledge-base it enters the first phase of the model definition, i.e. selection of the 'actors'. The following screen shows the system presenting the choice of the 'actors' to the user. A window displays in alphabetic order the names of all the 'actors' on which knowledge is available in the knowledge base. At any time one entry is highlighted (the highlight has not been shown). The directions at the bottom of the screen guides the user as to how to select an 'actor'.

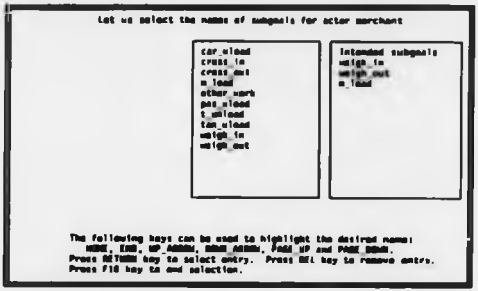


The following screen shows that by following the directions, the user has selected the three 'actors' present in the 'lorry' model (chapter 4). These names are displayed on the left side of the 'actors' window as they are selected (or deleted).

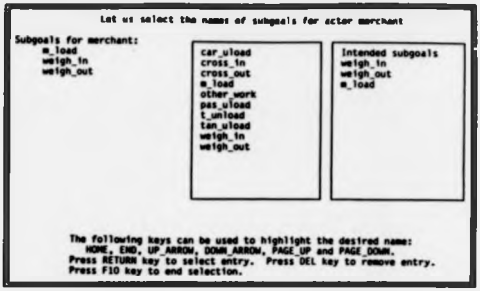


Following the selection of the 'actors' present in the model, the system enters the second phase of the model definition, i.e. the selection of the 'subgoals' for each 'actor'. This phase is carried out in two sub-phases. During the first sub-phase 'subgoals' are selected for each 'actor' and in the second sub-phase the selected 'subgoals' are put in their correct sequence.

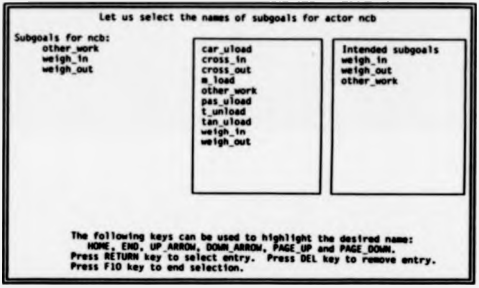
The following screen shows the system offering the choice of the 'subgoals' for the 'actor' 'merchant' which was selected in the previous phase. The presence of another window to the right of the 'subgoal' selection window should be noted. From the domain knowledge included in the additional 'actor_frame' slot (section 6.4) this window informs the user about the 'sensible' 'subgoals' for the 'actor'. It is however possible to select a 'subgoal' not displayed as one of the intended 'subgoals' by 'insisting'. This will be shown in the section on exception handling.

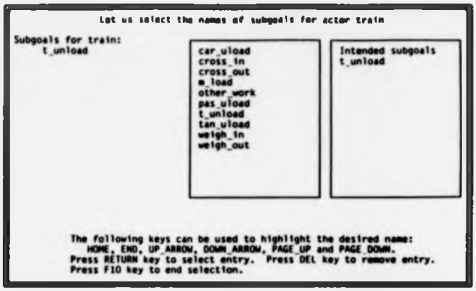


The following screen displays that the user has selected the three 'subgoals' for the 'actor' 'merchant'. As previously, these 'subgoals' are displayed to the left of the 'subgoals' selection window as they are selected.

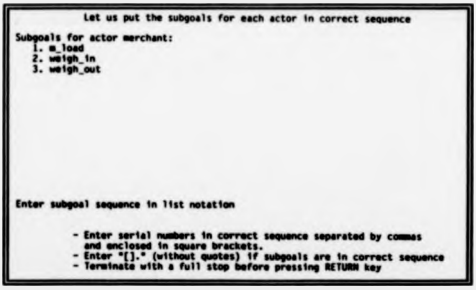


In a similar manner the following two screens show the selection by the user of the 'subgoals' for 'actors' 'ncb' and 'train'.





Having selected the 'subgoals' the system enters the second sub-phase of putting the 'subgoals' selected in the correct sequence. The following screen shows the system displaying the 'subgoals' the user previously selected for the 'actor' 'merchant' and asking the user to enter the serial numbers in the correct sequance.



In the following screen the user has entered the correct sequence.

```
            Let us put the subgoals for each actor in correct sequence

Subgoals for actor merchant:
   1. m_load
   2. weigh_in
   3. weigh_out









Enter subgoal sequence in list notation
     [2,1,3].

          - Enter serial numbers in correct sequence separated by commas
            and enclosed in square brackets.
          - Enter "[]." (without quotes) if subgoals are in correct sequence
          - Terminate with a full stop before pressing RETURN key
```

The sequence entered by the user is confirmed by displaying the correct sequence and asking the user to verify, as has been shown in the following screen.

```
            Let us put the subgoals for each actor in correct sequence

Subgoals for merchant:
     weigh_in
     m_load
     weigh_out









Are subgoals shown in correct sequence?

          Press  y  key to approve
          Press any other key to re-sequence
```

Similarly the following two screens depict the 'subgoal' sequencing sub-phase for the 'actor' 'ncb'. This completes the sub-phase as 'train' only has one 'subgoal' which does not require any sequencing.

```
          Let us put the subgoals for each actor in correct sequence

Subgoals for actor ncb:
   1. other_work
   2. weigh_in
   3. weigh_out




Enter subgoal sequence in list notation
     [2,1,3].

          - Enter serial numbers in correct sequence separated by commas
            and enclosed in square brackets.
          - Enter "[]." (without quotes) if subgoals are in correct sequence
          - Terminate with a full stop before pressing RETURN key
```

```
          Let us put the subgoals for each actor in correct sequence

Subgoals for ncb:
     weigh_in
     other_work
     weigh_out




Are subgoals shown in correct sequence?

          Press  y  key to approve
          Press any other key to re-sequence
```

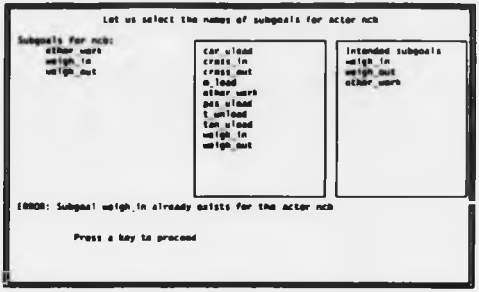
As the definition of the high level generic model is now complete, the system displays the model as captured and announces the name of the file in which it has been saved, as is depicted by the following screen.

```
my_model
model(my_model) :-
    actor(merchant),
    actor(ncb),
    actor(train).

goal(merchant) :-
    subgoal(weigh_in),
    subgoal(m_load),
    subgoal(weigh_out).
goal(ncb) :-
    subgoal(weigh_in),
    subgoal(other_work),
    subgoal(weigh_out).
goal(train) :-
    subgoal(t_unload).


The model is saved in file MY_MODEL.MDL

Happy Simulation!

Press a key to proceed
```

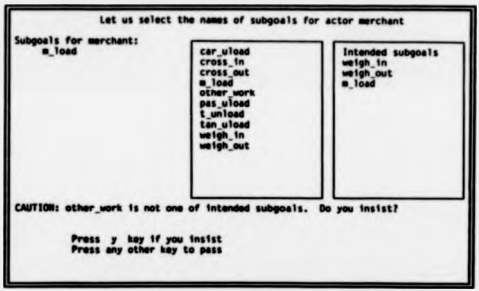
6.5.3. EXCEPTION HANDLING

Two types of exception conditions have been implemented — two error conditions and one caution condition. The following two screens show the two error conditions, when an attempt has been made to select an already selected 'actor' or 'subgoal'. Such conditions are prohibited and therefore are blocked.

```
                Let us select the names of actors in the model

Actors in the model are:
     merchant
                                  car_ship
                                  merchant
                                  ncb
                                  pas_ship
                                  tan_ship
                                  train

ERROR: Actor merchant already exists in the model

          Press a key to proceed
```



The following screen displays a caution condition when an attempt has been made to select a 'subgoal' which is not intended for the particular 'actor' in question. This condition is permitted provided the user 'insists'.



6.5.4. THE USER INTERACTION DURING THE MODEL BUILDING

This section covers the user interface aspects of the user interaction, which is entered into during the model building, for resolving the instances of the resources present in the model (chapter 5). Having interactively defined the high level

generic model by using the knowledge based model acquisition system the user now enters the command 'build' at the operating system prompt, as shown in the screen below. Upon invocation the model building system consults the knowledge base and begins the model building. As it does so it makes announcements to that effect near the lower end of the screen.

Upon the identification that there is a possible interaction between the 'merchant' and the 'train' for the use of the 'loader' it enters into a user interaction by displaying the following screen (see also Chapter 5). As the generic specification of the model does not specify the interactions, the system is asking the user to resolve as to how many instances of the 'loader' there are present in the model. The user is assisted with the syntax of his/her reply with the help of an example.

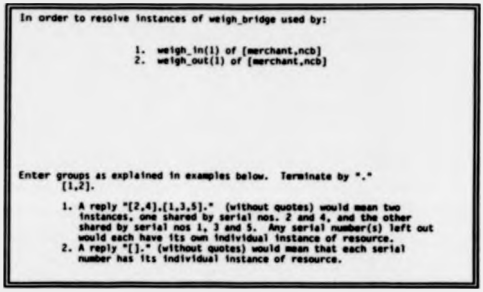
```
In order to resolve instances of loader used by:

                    1.  m_load(1) of [merchant]
                    2.  t_unload(1) of [train]

Enter groups as explained in examples below.  Terminate by "."

      1. A reply "[2,4],[1,3,5]."  (without quotes) would mean two
         instances, one shared by serial nos. 2 and 4, and the other
         shared by serial nos 1, 3 and 5.  Any serial number(s) left out
         would each have its own individual instance of resource.
      2. A reply "[]." (without quotes) would mean that each serial
         number has its individual instance of resource.
```

In the following screen the user has replied that there is one instance of the 'loader' in the model, which is shared by the two instances of the activities shown. If the user wanted to say that there are two instances of the 'loader', one for each (which would in effect mean that there is no interaction between the 'train' and the 'merchant' over the use of the 'loader') the user could answer by typing either "[]." or "[1]." or "[2]." or "[1],[2].".

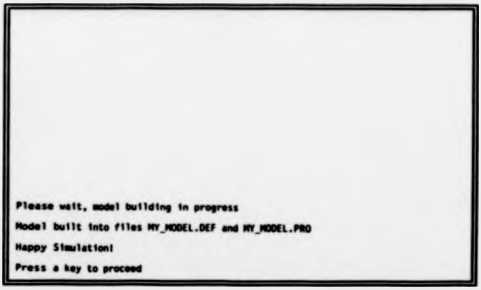
```
In order to resolve instances of loader used by:

                    1.  m_load(1) of [merchant]
                    2.  t_unload(1) of [train]

Enter groups as explained in examples below.  Terminate by "."
      [1,2].

      1. A reply "[2,4],[1,3,5]."  (without quotes) would mean two
         instances, one shared by serial nos. 2 and 4, and the other
         shared by serial nos 1, 3 and 5.  Any serial number(s) left out
         would each have its own individual instance of resource.
      2. A reply "[]." (without quotes) would mean that each serial
         number has its individual instance of resource.
```

Similarly, the following screen shows the resolving of the instances of the weigh bridge in the model. It would be interesting to note that as there is no 'own_activity_set' clause in the model, the 'merchant' and the 'mcb' share a single

instance of the activity set for the 'weigh_in' and the 'weigh_out'. This in turn implies that they mix in the queue before this activity set on a first in first out basis.



Having resolved all the possible interactions present in the model over the use of the resources, the model building system proceeds with its task and ultimately delivers the executable model in two files, as shown in the following screen.

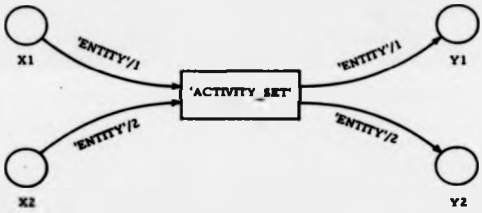


6.6. TOWARDS GENERALIZATIONS

Using the facilities implemented in the prototype system 'WISE', one can build and run simple but non-trivial simulation models. The last section in Chapter 5 pointed to some of the ways in which these facilities can be extended for building more complex simulation models. This section more specifically looks at the possible generalisations that can be made from the experience gained of devising and implementing a prototype knowledge-based discrete simulation modelling environment (the 'WISE' system). These generalisations when implemented should provide for the specification and construction of more complex simulation models, by using a richer knowledge base and more general model building method, than have been implemented.

6.6.1. THE SIMULATION METHODOLOGY KNOWLEDGE

At present a script relates to only one input queue and only one output queue. This can be generalised by having multiple input and output queues and the activity can be viewed from the point of view of entities in each queue. As an example the following diagram shows two entities taking part in an activity.



This generalised situation can be translated into two 'script' clauses as follows:

```
script(a/1, X1, Y1) :-
        wait_until(headof(X1)),
        link_with(['ENTITY'/2]),
        joint_move(X1, 'ACTIVITY_SET'),
        hold('DURATION'),
        move('ACTIVITY_SET', Y1).
```

```
script(a/2, X2, Y2) :-
        wait_until(head_of(X2),
        wait_until(message_found(go)),
        link_with(['ENTITY'/1]),
        joint_move(X2, 'ACTIVITY_SET')
        hold('DURATION'),
        move('ACTIVITY_SET', Y2).
```

The term designating the 'script' (i.e. 'a/1' and 'a/2') has been suitably extended to allow for multiple entities. A further key-term 'ENTITY'/'N' has been used. The value of 'N' (i.e. 1 or 2) corresponds with the value in the script designator. The method of model building would now also be required to identify and allocate the appropriate entity to the appropriate script clause for the purpose of the code generation.

This representation can be employed to capture the simulation knowledge for the activities where more than two entities take part.

6.6.2. CONDITIONAL BRANCHING

In simulation models a situation is frequently encountered where an entity upon completion of an activity determines one of two alternative paths. The condition which determines which path it will take may be based on probability or the system's state. In order to incorporate such conditionals in the specification of the model and in the method of model building some knowledge about these is expected to be available in the knowledge-base. The following two Prolog clauses represent a possible generalized specification of conditionals. The first 'conditional_branch' clause specifies a branch based on the outcome of a random sample, whereas the second clause determines the branch based on the number of entities in one of the two possible destination queues (Y and Z).

```
conditional_branch(cb1, X, Y, Z) :-
        random_sample(A),
        (       [A <= 0.4,
                move(X, Y)]
        ;       move(X, Z)
        ).

conditional_branch(cb2, X, Y, Z) :-
        number_in_queue(Y, N),
        (       [N < 10,
                move(X, Y)]
        ;       move(X, Z)
        ).
```

The above two 'conditional_branch' clauses represent frequently occurring branching conditions. Further domain specific conditions can be incorporated and given a unique name (similar to 'cb1' and 'cb2' in the above example) which can be used in the high level articulation of a model. As an example:

```
goal(entity) :-
        subgoal(service1)
        choose_path(cb1, path(a), path(b)).

path(a) :-
        subgoal(service2),
        subgoal(service4).

path(b) :-
        subgoal(service3)
        path(a).
```

It should be noted that the 'path' and 'choose_path' predicates can be used within the body of the 'path' clauses. This would provide for the specification of complex conditional routes for an entity in the model.

6.6.3. THE FORM OF THE EXECUTABLE MODEL

In the current implementation of 'WISE', the dynamic behaviour of the model is captured by a set of processes, one for each 'actor'. This form does not pose any problems either in the model building or in the execution by the simulation engine. With the introduction of conditionals as described above, 'WISE' could be extended to provide for the appropriate 'flow of control' within a process. The possibilities include borrowing procedural language constructs, e.g. labels and the GOTO statement of FORTRAN or the block structure of PASCAL. These would pose problems both in the model building and in extending the simulation engine to execute the code.

Another interesting alternative however seems to exist and should be explored as it could prove preferable over the use of the procedural language constructs mentioned above. Keeping in view that the simulation engine interprets each process step for each entity at the run time it would probably be easier to keep the process segments separately stored under different Prolog database keys. A control structure representing the conditions can link these keys to provide for a suitable branching at the run time. Such representation would be more in line with the logic programming approach, and would require a simpler model building method, than if it has to provide for the generation of the flow of the control to deliver the model in a procedural form.

6.6.4. THE SUB-MODELS KNOWLEDGE BASE

A high level generic simulation model has been viewed as the specification of the names of the entities which 'flow' through the system (the 'actors') and to each is

associated an ordered list of the names of the activities (the 'subgoals') through which they go during their life cycle in the system being modelled. The possible interactions, when two 'subgoals' have a common resource requirement or two 'actors' have the same 'subgoal' name in their list of 'subgoals', are resolved during a user interaction to completely specify the intended model. Conceptually, this representation of the model can be made use of for specifying larger building blocks in the form of sub-models. These sub-models can refer to the same knowledge base consisting of the 'actor_frame', the 'subgoal_frame' and the 'script' clauses as has been seen previously. In order to achieve this level, additional specification will be required to capture the interactions between the processes within a sub-model. The following set of 'sub_model' clauses provides a possible way of capturing this knowledge. The equivalent sub-model is depicted in a diagrammatic form in fig 6.3

```
sub_model(lorry, actor_list([merchant, ncb, train]).
sub_model(lorry, subgoal_list(merchant, [weigh_in,
        a_load, weigh_out]).
sub_model(lorry, subgoal_list(ncb, [weigh_in,
        other_work, weigh_out]).
sub_model(lorry, subgoal_list(train, [t_unload]).
sub_model(lorry, interaction(resource(weigh_bridge],
        [weigh_in([merchant, ncb]],
        weigh_out([merchant, ncb]]).
sub_model(lorry, interaction(resource(loader],
        a_load([merchant]), t_unload([train]]).
```

Various possibilities exist in relation to the specification of the simulation models and the method of model building, while making use of the sub-models knowledge base. The route of an entity may be described in terms of sub-models (in addition to subgoals), provided that the sub-models specified already include the entity as one of its 'actors'. Such routes (e.g. from one sub-model to another) can be used to 'couple' the two sub-models through linking queues. While specifying simulation models in this way, the need to identify and to resolve any new interactions would arise e.g. a resource shared among two sub-models. Such interactions can be resolved by referring these to the user. Alternatively, it should be possible to specify the sub-models at a higher level, which comprise of two or more sub-models already available in the knowledge base, and to make these also a part of the knowledge base. In this way a hierarchical knowledge base can be set up with reference to the purpose of a simulation study and an experimental frame which can make reference to the sub-models at various level of aggregation defining the scope of the study. These ideas have been depicted diagrammatically in fig. 6.4.

It should be noted that the sub-models knowledge base consists of the knowledge-based specification of the sub-model, making reference to the knowledge already

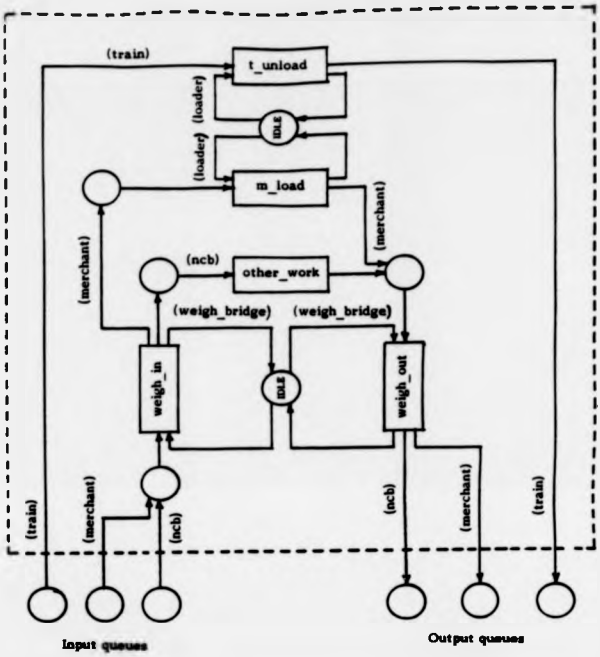


Figure 6.1. The entity cycle diagram for the sub-model 'lorry'.

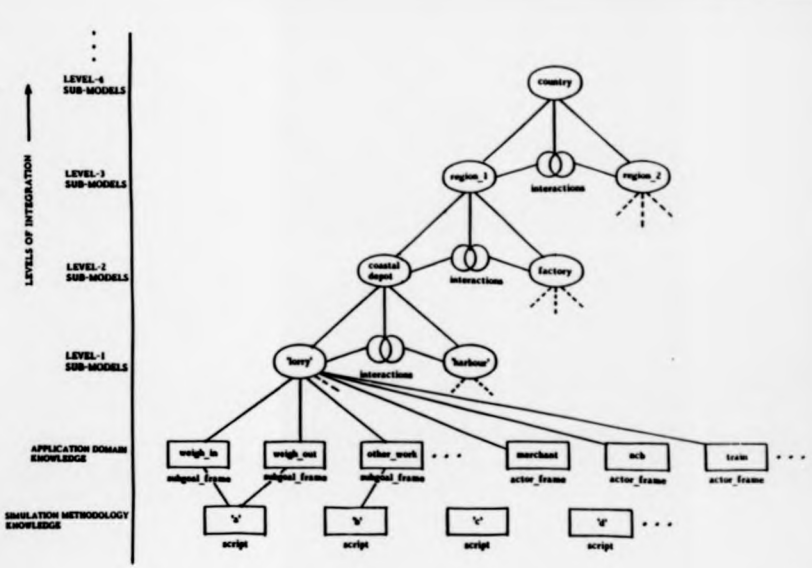


Figure 6.4. The hierarchical structuring of a sub-models knowledge base.

available in the knowledge base ('actors', 'subgoals'). While constructing a specific model, all such specifications would be taken into account and the model viewed in its totality after resolving any newly arising interactions. Using this framework certain types of changes in the underlying knowledge base can be accommodated without affecting the integrity of the knowledge base (e.g. adding further conditions in a 'script' clause). However, some other types of changes would invalidate a particular sub-model specification (e.g. removing a resource from a subgoal which previously had one). The executable model produced would reflect the current knowledge base even if it was different at the time of the specification of the particular sub-models used. The careful management of the knowledge base is therefore indicated and some form of tools developed to ensure its consistency and integrity.

The knowledge-based specification of the sub-models can be facilitated by using a modified combination of the model acquisition system discussed in this chapter and the model building system (chapter 5). The modification would involve stopping the reference to the 'script' clauses for the code generation and providing for the assertion of 'sub-model' clauses instead. Such interactive model acquisition would ensure consistency and should therefore be preferred over any manual method, which would necessarily require additional consistency checking.

6.7. CONCLUSIONS

The work described in this chapter has shown that by using the knowledge-based systems paradigm it is feasible to provide computer assistance for defining a simulation model at a generic level. As a result, keeping in view the work described in chapters 4 and 5, it is feasible to provide an 'intelligent' simulation modelling environment where decision makers can attempt to do their own computer assisted simulation modelling. The knowledge-based systems paradigm is therefore a suitable paradigm for devising such an environment and a logic programming paradigm provides the necessary high level features for representing knowledge and for the implementation of knowledge-based systems.

END OF CHAPTER 6

CHAPTER 7: CONCLUSIONS AND FURTHER RESEARCH

7.1. CONCLUSIONS

7.1.1. CONCLUSIONS RELATED TO THE WORK ON THE SIMULATION ENGINE

It has been possible to demonstrate that it is feasible to implement set representation for the system's state for the purpose of simulated behaviour generation at a symbolic level while using a Logic Programming paradigm. It has also been demonstrated that it is feasible to write a generalised simulation facility using Prolog, which is capable of generating simulated behaviour from an articulation of a simulation model using the three phase or the process 'world views' or a sensible mixture of the two.

The implementation of a simulation engine which works entirely at the symbolic level can be regarded as an improvement over the ones where the symbols represent an underlying programming language data structure (e.g. a FORTRAN integer) which makes it necessary to compile the simulation model and then link it (at binary code level) with the simulation package. These steps (compiling and linking) make debugging cumbersome and time consuming and therefore expensive. The database facilities in Prolog permit the simulation model (e.g. consisting of 'events' and 'activity' Prolog clauses) to be integrated directly with the simulation engine at the symbolic level. Such integration is permissible even when the simulation engine has been compiled for running efficiency.

It has been possible to demonstrate that Prolog provides the adequate features for handling simulated time, which is absolutely necessary for simulated behaviour generation without the need for extending the language interpreter for this purpose, as has been proposed by [FUTO & S, 82].

The use of set representation at the symbolic level can also provide for more intelligible interaction with the model at the run time. The built-in facilities in Prolog allow for the specification of complex searches to be made, to find for example, particular entities or queues which satisfy given conditions, whereas such queries must be provided by user written routines in the case of a procedural

language simulation package, because of the non availability of built in language facilities and also, processing takes place at a lower level.

The entities can be assigned symbolic attributes which can help debugging the model more efficiently than when the attributes are represented, for example, by integers. In a future implementation it should be possible to assign symbolic names to the attributes of an entity (e.g. 'age', 'height' instead of referring to these as first attribute, second attribute, and so on). This can further make writing and debugging simulation models easier, and run-time interaction with the models more intelligible.

Understandably, processing at the symbolic level does not provide for run-time efficiency. This fact calls for the use of faster and/or special purpose symbol processing hardware. The run-time execution speed, however, is not the most important factor in pedagogical environments and the advantages mentioned above can easily outweigh the execution speed related considerations in such situations. A further development could be to build a simulation tutoring system around the prototype simulation engine, as implemented (chapter 4).

The prototype simulation engine therefore represents a consolidation of the simulation technology related to behaviour generation while using the Logic Programming paradigm for implementation. Also noted in chapter 4, a future development of the simulation engine could be to provide for the 'world view free' articulation of simulation models by simply stating the model's components and their interrelationships, thus making the mechanism of behaviour generation entirely transparent.

7.1.2. CONCLUSIONS RELATED TO THE SIMULATION MODELLING ENVIRONMENT

A feasible method for the knowledge-based construction of non-trivial discrete simulation models has been proposed. A knowledge-based model generation system using this method has been implemented in Prolog and its working has been demonstrated by constructing a number of example models. Using this method it has been demonstrated that the model building can start from a very high level of generic specification of the simulation models, and the details are sorted out during user interaction at an 'intelligent' level. It has been demonstrated that such generic specification can itself be developed through user interaction with a

knowledge-based model acquisition system, while using the same knowledge base as used for the executable model construction.

The use of a knowledge-based systems paradigm coupled with the symbolic processing capabilites of Prolog provide a powerful set of tools for implementing simulation modelling environments, which can provide for a greater amount of computer assistance than has been possible using the 'conventional' simulation support tools (e.g. simulation program generators). This is achieved by having the ability to represent both the application domain knowledge as well as the simulation methodology knowledge, which makes it possible to define a simulation model by reasoning with the application domain knowledge while using the simulation methodology knowledge for generating the executable code when the model has been completely defined.

The use of a knowledge based systems paradigm offers the advantage that the knowledge engineer can focus his/her attention on small parts of the system (e.g. one entity, one activity, one script and so on) and such knowledge is retrieved and assembled at the time of the model specification and the model building. Therefore it can be said that the application of a knowledge-based systems paradigm matches the human limitation of being able to concentrate on one thing at a time and therefore can be regarded as an improvement over previous software paradigms for problem solving.

The use of a knowledge-based systems framework provides for an accumulation of the application domain knowledge in a suitable form which can be used for future simulation modelling. Using further enhancements of this knowledge, such a knowledge-base can also be used as a basis for problem solving within the application domain without involving simulation.

There is an interesting implication of the simulation model generation method described in the previous chapters. By a suitable alteration it can be made to generate all the possible configurations of a system from an initial high level generic articulation of the model. Such automatic model generation can be achieved by eliminating the user interaction phase for resolving the resource instances and installing a suitable generator facility instead, which would generate resource-actor-subgoal combinations through backtracking. Such generation of the possible models can be constrained by the addition of a set of rules to the knowledge-base so that only valid and sensible combinations are generated.

A model generator, as described above, has important implications for automating the experimentation with the simulation models, while using simulation for the design of a future system. The ability to influence the logical structure of the model in an automatic way during experimentation can be regarded as a novel feature which has emerged from this research, as previously it was only possible to alter various model parameters (e.g. resource levels) during experimentation once the model had been coded.

7.2. FURTHER RESEARCH

The research described in this thesis has concentrated on the issues of simulated behaviour generation and simulation modelling environments. Concurrent research at Warwick Business School reported in [TAYLOR, 88] has concentrated on an experimental advisory system ('WES') for the experimentation with simulation models. Previously, [FLITMAN, 86] has concentrated on a prototype Prolog simulation engine, separate specification of the logic of the simulation model in Prolog while using MICROSIM for the articulation of the rest of the simulation model, and an experimentation control expert system which exhibits learning capabilities.

Further research is indicated to consolidate this research and to proceed further from there. It is envisaged that the 'intelligent' simulation environment described can be extended to include other phases of the simulation study by enhancing the knowledge representations to include the knowledge about experimentation with models, the knowledge of problems (e.g. congestion), and queueing theory formulae. Using this enhanced knowledge base it should be possible to address system design problems using a 'generate and test' mode of problem solving, which is typical of the Artificial Intelligence approach to problem solving. Starting from the cost/performance requirements and the knowledge available in the knowledge base, the system should be able to automatically generate alternative configurations of the proposed system and test these against the requisite cost/performance criteria. A level of meta-knowledge can be developed to constrain the generation of possible models to the most promising ones (e.g. by making early predictions using queueing theory formulae) therefore enhancing efficiency. In this way a degree of automation coupled with the use of a problem oriented approach can be realised, while transferring a much greater part of the problem solving burden on to the computer.

A number of publications reporting research in the application of artificial intelligence techniques in various phases of a discrete simulation study have been reviewed in chapters 2 and 3. These applications have been researched at various locations and address different isolated aspects of a simulation study. A stage has been reached, when there is a need for further research to integrate and unify the principles that have evolved from these isolated items of research, and implement these as a comprehensive and integrated intelligent simulation environment to make it easier for the decision makers to conduct their own simulation studies.

A great majority of these implementations employ a variety of knowledge representations and programming paradigms. There is therefore a need for further research to integrate the capabilities of these systems in order to provide comprehensive and integrated 'intelligent' computer support for a simulation study. Two approaches seem possible for explorations: (a) to use a multi-expert system architecture (e.g. a blackboard model) where each expert system concentrates on a particular phase of a simulation study, and (b) to evolve more comprehensive knowledge representations and associated inference engines which cover all the phases of a simulation study. Such systems should be able to afford a degree of automation in the conduct of simulation studies. Typically, the problem solving system may start solving a particular problem, posed to it by the user, until it identifies a knowledge gap in its knowledge-base, which can be filled by generating such knowledge from experimentation with an appropriate simulation model. Upon this demand a simulation model can be automatically generated and experimented with, thus delivering the requisite knowledge for the problem solving proper to proceed.

The use of artificial intelligence technology need not be limited to the application in simulation related work, but its use should be explored in relation to other Operational Research techniques. A reasonable starting point seems to be the problem formulation phase for particular OR techniques (e.g. Linear Programming, Decision Theory). The use of the expert systems paradigm in relation to decision support systems has been considered in [RADZIKOWSKI, 84]. At some stage it should be possible to develop an Expert Operational Research Systems which would select and apply the appropriate OR technique when a problem is posed to them.

Prolog with its capability of being used with parallel processing hardware offers some interesting possibilites in relation to simulated behaviour generation. Using

the three phase model for behaviour generation, the c-phase involves searches to determine the possible changes to the state of the model. As such this phase is highly computation intensive and therefore most time consuming. It should in principle be possible to carry out parallel searches to speed up this phase. For example, a processor may be attached to each set (a queue or an activity set) in the model. During the c-phase all such processors may proceed independently to determine the possible next change to the state of the model. Any conflict which may arise as a result of these independent searches can be resolved with reference to a set of priority rules by a master processor which would then change the state of the model.

A further important implication of the parallel processing framework for simulation is that it should be possible to include an inferencing phase within the c-phase, which can allow mimicing of the behaviour of 'intelligent' entities within the model. Such inferencing can be modelled by making available a knowledge base of the application domain. An 'intelligent' entity in the model can make decisions with reference to this knowledge base together with the current state of the model. The multiprocessor approach can also support the multiformalism specification of the parts of the simulation model, the application domain knowledge, the experimental frame, and so on (assuming that these can refer to a common system state representation e.g. set representation).

Another implication of using Prolog can be that its rule base can be dynamically modified by 'intelligent' entities, thus approaching a level of simulating reality more closely than with a set of static rules.

Finally, another possible direction of research can be to explore the integration of visual interactive graphics with the knowledge-based systems framework for simulation modelling, where the user interface is entirely graphical and the system's configurations, as generated, can also be depicted in graphical form.

END OF CHAPTER 7

REFERENCES

[ADELSBERGER, 84]
H. H. Adelsberger. 1984. "Prolog as a simulation language." In [WSC, 84] pp 501-504.

[ADELSBERGER & PSW, 86]
H. H. Adelsberger, U. W. Pooch, R. E. Shannon and G. N. Williams. 1986. "Rule based object oriented simulation systems." In [Luker & A (eds), 86] pp 107-112.

[AESSP, 85]
The Alvey Expert Systems Starter Pack. 1985. Manchester: The National Computing Centre Ltd.

[AHMAD, 78]
A. Ahmad. 1978. Transfer of an application from MUMPS to BASIC. M.Sc. Operational Research dissertation. Department of Engineering Production. University of Birmingham.

[AHMAD & H, 88]
A. Ahmad and R. D. Hurrion. 1988. "Automatic model generation using a Prolog model-base." In [Henson (ed), 88] pp. 137-142.

[AMAREL, 87]
S. Amarel. 1987. "Problem Solving." In [Shapiro (ed), 87] pp 767-779.

[Aronofsky (ed), 69]
Progress In Operations Research, Volume III: Relationship Between Operations Research and the Computer. 1969. J. S. Aronofsky (editor). New York, etc.: John Wiley. Operations Research Society of America Publications in Operations Research no. 16.

[BHARATH, 86]
R. Bharath. 1986. "Logic Programming: A Tool for MS/OR?" Interfaces. 16:5 (September-October) pp 80-91.

[BIRTWISTLE, 81]
G. M. Birtwistle. 1981. "The design decisions behind DEMOS." In [UKSC, 81] pp 97-107.

[BIRTWISTLE & DMN, 79]
G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug and K. Nygaard. 1979. Simula Begin. 2nd Edition. England: Chartwell-Bratt Ltd (ISBN 0-86238-009-X); West Germany: Bratt Institut fur Neues Lernen (ISBN 3-88598-018-5); Sweden: Studentlitteratur (ISBN 91-44-06212-5)

[BIRTWISTLE & K, 86]
G. Birtwistle and J. Kendall. 1986. "A view of LISP." In [Luker & A (eds), 86] pp 157-162.

[Birtwistle (ed), 85]
AI, Graphics and Simulation. 1985. Proceedings of the SCS Multiconference, January 1985, San Diego, California, USA. G. Birtwistle (editor). San Diego: Society for Computer Simulation. ISBN: 0735-9276.

[BOBROW, 84]
D. G. Bobrow. 1984. "If Prolog is the answer, what is the question." In [FGCS, 84] pp 138-145.

[BONGULIELMI & C, 84]
A. P. Bongulielmi. and F. E. Cellier. 1984. "On the usefulness of deterministic grammars for simulation languages." ACM Simuletter vol. 15 no. 1 (January). 14-36.

[BROOKING, 84]
A. G. Brooking. 1984. "The Fifth Generation game." In [Forsyth (ed), 84] pp 18-35.

[BROWN, 78]
J. C. Brown. 1978. Visual interactive simulation: Further developments towards a generalised system and its use in 3 problem areas associated with a high technology company. MSc Thesis, School of Industrial Business Studies, University of Warwick.

[BUNDY, 83]
A. Bundy. 1983. "What stories should we tell Prolog Students?" DAI Working Paper no. 156. Department of Artificial Intelligence, University of Edinburgh.

[Cellier (ed), 82]
Progress in Modelling and Simulation. 1982. F. E. Cellier (editor). London, etc.: Academic Press. ISBN: 0-12-164780-3.

[CHANDRASEKARAN, 86]
B. Chandrasekaran. 1986. "Generic Tasks in Knowledge-based Reasoning: High-level Building Blocks for Expert System Design." IEEE Expert. Fall 1986 pp 23-30.

[CLEARY & GU, 85]
J. Cleary, K.-S. Goh and B. Unger. 1985. "Discrete event simulation in Prolog." In [Birtwistle (ed), 85] pp 8-13.

[CLEMENTSON, 80]
A. T. Clementson. 1980. "ECSL/CAPS Detailed Reference Manual." University of Birmingham: The Lucas Institute for Engineering Production.

[CLOCKSIN & M, 84]
W. F. Clocksin and C. S. Mellish. 1984. Programming in Prolog. 2nd edition. Berlin, etc.: Springer-Verlag ISBN: 3-540-15011-0 (also 0-387-15011-0).

[COELHO, 83]
H. Coelho. 1983. "PROLOG: A Programming Tool for Logical Domain Modeling." In [Sol (ed), 83] pp 37-45.

[COLMERAUER, 85]
A. Colmerauer. 1985. "Prolog in 10 Figures." Communications of ACM vol. 28 no. 12 pp 1296-1310.

[COOKE & S, 84]
S. Cooke and N. Slack. 1984. Making Management Decisions. London: Prentice-Hall. ISBN 0-13-547837-5.

[CROOKES, 82]
J. G. Crookes. 1982. "Simulation in 1981." European Journal of Operational Research, 9, pp 1-7.

[D'ANGELO, 83]
G. J. D'Angelo. 1983. "Tutorial on Petri Nets." ACM Simuletter, vol. 14, no. 1-4, pp 10-25.

[DAVIES, 76]
N. R. Davies. 1976. "On the information content of a discrete-event simulation model." Simulation, October 1976. pp 123-128.

[DAVIES, 79]
N. R. Davies. 1979. "Interactive Simulation Program Generation." In [Zeigler & EKO (eds), 79] pp 179-200.

[DAVIS, 79]
R. E. Davis. 1979. Generating correct programs from logic specifications. PhD Dissertation. Information Sciences. University of California, Santa Cruz, USA.

[DAVIS, 85]
R. E. Davis. 1985. "Logic Programming and Prolog: A Tutorial." IEEE Software, September:53-62.

[DOUKIDIS, 87]
G. I. Doukidis. 1987. "An Anthology on the Homology of Simulation with Artificial Intelligence." J. Opl Res. Soc. Vol. 38, No. 8, pp. 701-712.

[DOUKIDIS & P, 85]
G. I. Doukidis and R. J. Paul. 1985. "Research into Expert Systems to Aid Simulation Model Formulation." J. Opl Res. Soc. Vol. 36, No. 4, pp. 319-325.

[DOWNES & B, 84]
V. A. downes and R. T. Bosch. 1984. "Discrete event simulation with Ada." In [UKSC, 84] pp 68-78.

[ELZAS, 80]
M. S. Elzas. 1980. "Simulation and the processes of change." In [Oren & ZR (eds), 80] pp 3-18.

[ELZAS, 86]
M. S. Elzas. 1986. "Relations between artificial intelligence environment and modelling & simulation support systems." In [Elzas & OZ (eds), 86] pp 61-77.

[Elzas & OZ (eds), 86]
Modelling and Simulation Methodology in the Artificial Intelligence Era. 1986. M. S. Elzas, T. I. Oren and B. P. Zeigler (editors). Amsterdam, etc: North-Holland. ISBN: 0 444 701303.

[ESAPP, 85]
Expert systems and their applications - 5th International Workshop. Avignon, France: 13-15 May 1985. France: Agence de l'Informatieque.

[ESAPP, 86]
Expert systems and their applications - 6th International Workshop. Avignon, France: 28-30 April 1986. France: Agence de l'Informatique. ISBN: 2-86851-033-X.

[ESTC, 84]
Research and Development in Expert Systems: Proceedings of the Fourth Technical Conference of the British Computer Society Specialist Group on Expert Systems. University of Warwick: 18-20 December 1984. M. A. Bramer (editor). Cambridge, etc.: Cambridge University Press. ISBN: 0 521 30652 3.

[ESTC, 85]
Expert Systems 85. Proceedings of the 5th Technical Conference of the British Computer Society Specialist Group on Expert Systems. University of Warwick: 17-19 December 1985. M. Merry (editor). Cambridge, etc.: Cambridge University Press. The British Computer Society Workshop Series. ISBN: 0-521-32596-X.

[FGCS, 84]
Fifth generation computer systems 1984. Proceedings of the International Conference. Tokyo: 6-9 November 1984. Tokyo: OHMSHA (ISBN: 4-274-07221-5) and Amsterdam: North-Holland (ISBN: 0-444-87673-1).

[FIDDY & BH, 81]
E. Fiddy, J. G. Bright and R. D. Hurrion. 1981. "SEE-WHY: Interactive simulation on the screen." In Proceedings Institute of Mechanical Engineers, C293/81, pp 167-172.

[FISHER, 82]
M. W. J. Fisher. 1982. The application of visual interactive simulation in the management of continuous process chemical plants. PhD Thesis. University of Warwick, School of Industrial and Business Studies.

[FISHMAN, 78]
G. S. Fishman. 1978. Principles of Discrete Event Simulation. New York, etc.: John Wiley & Sons. ISBN: 0-471-04395-8.

[FLITMAN, 86]
A. Flitman. 1986. Towards the application of artificial intelligence techniques for discrete event simulation. PhD thesis. University of Warwick, School of Industrial and Business Studies.

[FLITMAN & H, 87]
A. M. Flitman and R. D. Hurrion. 1987. "Linking Discrete-Event Simulation Models with Expert Systems." J. Opl Res. Soc. Vol. 38, No. 8, pp. 723-733.

[FMS-5, 86]
Proceedings of the 5th International Conference on Flexible Manufacturing Systems. 3-5 November 1986. Stratfor-upon-Avon, UK. K. Rathmill (ed). Bedford IFS (Publications) Ltd. ISBN: 0-948507-17-9 (also 3-540-16332-8 and 0-387-16332-8).

[FORDYCE & NS, 87]
K. Fordyce; P. Norden and G. Sullivan. 1987. "Review of Expert Systems for the Management Science Practitioner." Interfaces. 17:2 (March-April) pp 64-77.

[Forsyth (ed), 84]
Expert Systems: Principles and case studies. 1984. R. Forsyth (editor). London, etc.: Chapman and Hall. ISBN: 0-412-26270-3 (hardback), 0-412-26280-0 (paperback).

[Fox (ed), 84]
Expert systems; State of the Art Report 12:7. Edited by J. Fox. Published by Pergamon Infotech Limited, Maidenhead, Berkshire, England. 1984. ISBN: 0 08 028 5929.

[FRANKOWSKI & F, 80]
E. N. Frankowski and W. R. Franta. 1980. "A Process Oriented Simulation Model Specification and Documentation Language." Software--Practice and Experience, vol. 10, 721-742.

[FRANTA, 77]
W. R. Franta. 1977. The Process View of Simulation. New York, etc.: North-Holland. Operating and Programming Systems Series. ISBN: 0-444-00221-9 and 0-444-00223-5 (pbk.).

[FUTO, 85]
I. Futo. 1985. "Combined discrete/continuous modelling and problem solving." In [Birtwistle (ed), 85] pp 23-28.

[FUTO & G, 87]
I. Futo and T. Gergely. 1987. "Logic Programming in Simulation." Transactions of The Society for Computer Simulation vol. 3 no. 3 pp 195-216.

[FUTO & GD, 86]
I. Futo, T. Gergely and T. Deutsch. 1986. "Logic modelling." in [Kerckhoffs & Vz (eds), 86] pp 117-129.

[FUTO & P, 86]
I. Futo and I. Papp. 1986. "The use of TC-PROLOG for medical simulation." In [Luker & A (eds), 86] pp 29-34.

[FUTO & PS, 86]
I. Futo, I. Papp and J. Szeredi. 1986. "The microcomputer version of TC-PROLOG." In [Luker & A (eds), 86] pp 123-128.

[FUTO & S, 82]
I. Futo and J. Szeredi. 1982. "A very high level discrete simulation system T-PROLOG." Computational Linguistics and Computer Languages vol. XV pp 111-131.

[GOLDBERG & R, 83]
A. Goldberg and D. Robson. 1983. Smalltalk-80: The Language and its Implementation. Reading, Massachusetts: Addison-Wesley.

[HADDOCK, 87]
J. Haddock. 1987. "An expert system framework based on a simulation generator." Simulation 48:2 (Februrary):45-53.

[HAYS-ROTH, 87]
F. Hays-Roth. 1987. "Expert Systems." In [Shapiro (ed), 87] pp 287-298.

[Henson (ed), 88]
Artificial Intelligence and Simulation: The Diversity of Applications. Proceedings of the SCS Multiconference, 3-5 February 1988. San Diego, California, USA. T. Henson (editor). San Diego: Society for Computer Simulation International.

[HILL, 71]
P. R. Hill. 1971. HOCUS (Hand or Computer Simulation). Egham, Surrey: P. E. Group.

[HILL & R, 87]
T. R. Hill and S. D. Roberts. 1987. "A prototype knowledge-based simulation support system." Simulation 48:4 (April):152-161.

[HOLBAEK-HANSSEN & HN, 77]
E. Holbaek-hanssen, P. Handlykken and K. Nygaard. 1977. System Description and the DELTA language. DELTA Project Report no. 4. Secon printing. Oslo: Norwegian Computing Center Publication no. 523.

[Holmes (ed), 85]
AI and Simulation. 1985. Proceedings from the Eastern Simulation Conference, March 1985, Norfolk. W. M. Holmes (editor). San Diego: Society for Computer Simulation. ISBN: 0-911801-05-7.

[HOOPER, 86]
J. W. Hooper. 1986. "Activity scanning and the three-phase approach." Technical Comment. Simulation November 1986. pp 210-211.

[HUNT, 86]
V. D. Hunt. 1986. Artificial Intelligence and Expert Systems Sourcebook. New York and London: Chapman & Hall. ISBN: 0-412-012111.

[Huntsinger & KKV (eds), 88]
Simulation Environments and Symbol and Number Processing on Multi and Array Processors. 1988. Proceedings of the European Simulation Multiconference, June 1-3, 1988, Nice, France. R. C. Huntsinger, W. J. Karplus, E. J. Kerckhoffs and G. C. Vansteenkiste (editors). San Diego, California: Society for Computer Simulation. (also Ghent, Belgium: Society for Computer Simulation Europe). ISBN: 0-911801-39-1.

[HURRION, 76]
R. D. Hurrion. 1976. The design, use and required facilities of an interactive visual computer simulation language to explore production planning problems. PhD Thesis. University of London.

[HURRION, 85]
R. D. Hurrion. 1985. "Interactive Discrete Event Simulation Model Building Using an Integrated Process/Activity/Event Approach." Unpublished paper. Available from author at Warwick Business School.

[Hurrion (ed), 86]
Simulation: Applications in Manufacturing. 1986. R. D. Hurrion (editor). Bedford: IFS (Publications) Ltd. ISBN:0-948507-33-0 (also 3-540-16357-3 and 0-387-16357-3).

[ISTEL, 86]
WITNESS User's Manual. 1986. Redditch, England: ISTEL Ltd.

[JACKSON, 86]
P. Jackson. 1986. Introduction to Expert Systems. Wokingham, etc.: Addison-Wesley Publishing Co. International Computer Science series. ISBN: 0-201-14223-6 (pbk.)

[JOHNSON & K, 85]
L. Johnson and E. T. Keravnou. 1985. Expert Systems Technology, A Guide. Kent: Abacus Press. Information Technology and Systems Series. ISBN: 0-85626-446-6.

[Kerckhoffs & Vz (eds), 86]
AI Applied To Simulation. Proceedings of the European Conference at the University of Ghent. February 25-28, 1985, Belgium. E. J. H. Kerckhoffs, G. C. Vansteenkiste and B. P. Zeigler (editors), San Diego, California: Society for computer simulation. Simulation series vol. 18 no. 1. ISSN: 0735-9276.

[KETTENIS, 86]
D. L. Kettenis. 1986. "Knowledge-based model storage and retrieval: Problems and possibilities." In [Elzas & OZ (eds), 86] pp 101-111.

[KHOSHNEVIS & C, 87]
B. Khoshnevis and A.-P. Chen. 1987. "An automated simulation modelling system based on AI techniques." In [Luker & B (eds), 87] pp 87-91.

[KLAHR, 86]
P. Klahr. 1986. "Expressibility in ROSS, an object-oriented simulation system." In [Kerckhoffs & Vz (eds), 86] pp 136-139.

[KLEIJNEN, 87]
J. P. C. Kleijnen. 1987. "Statistical Tools for Simulation Practitioners." New York and Basel: Marcel Dekker, Inc. ISBN 0-8247-7333-0.

[KLIR, 79]
G. J. Klir. 1979. "General systems problem solving methodology." In [Zeigler & EKO (eds), 79] pp 3-28.

[KOWALIK, 88]
J. S. Kowalik. 1988. "Impact of parallel processing on simulation and artificial intelligence." In [Henson (ed), 88] pp 8-11.

[KOWALSKI, 79]
R. Kowalski. 1979. Logic for Problem Solving. New York, Amsterdam, and Oxford: North-Holland; New York: Elsevier Science Publishing Co., Inc. ISBN 0-444-00365-7 (hbk.) and ISBN 0-444-00368-1 (pbk.).

[KOWALSKI & H, 87]
R. A. Kowalski and C. J. Hogger. 1987. "Logic Programming." In [Shapiro (ed), 87] pp 544-558.

[KREUTZER, 88]
W. Kreutzer. 1988. "A modeller's workbench - Simulation based on the desktop metaphor." In [Henson (ed), 88].

[LAW & K, 82]
A. M. Law and W. D. Kelton. 1982. Simulation Modelling and Analysis. New York, etc.: McGraw-Hill Book Co. McGraw-Hill series in Industrial Engineering and Management Science. ISBN: 0-07-036696-9.

[LEC_NOTES, 85]
R. D. Hurrion. 1985. Lecture notes handed out during Simulation course for M.Sc MS/OR. Coventry: Warwick Business School, University of Warwick.

[LEUNG & C, 85]
C. H. C. Leung and Q. H. Choo. 1985. "A knowledge-base for effective software specification and maintenance." In [SSD, 85] pp 139-142.

[Luker & A (eds), 86]
Intelligent Simulation Environments. 1986. Proceedings of SCS Multiconference 23-25 January, 1986, San Diego, California, USA. P. A. Luker and H. H. Adelsberger (editors). San Diego: Society for Computer Simulation. Simulation Series vol. 17 no. 1. ISSN: 0735-9276.

[Luker & B (eds), 87]
Simulation and AI. Proceedings of SCS Multconference, 14-16 January 1987, San Diego, California, USA. P. A. Luker and G. Birtwistle (editors). San Diego: Society for computer simulation. Simulation Series: vol. 18 no. 3. ISSN: 0735-9276.

[MANNA & W, 81]
Z. Manna and R. Waldinger. 1981. "A Deductive Approach to Program Synthesis." In [Webber & N (eds), 81] pp 141-172.

[MATHEWSON, 74]
S. C. Mathewson. 1974. "Simulation program generators." Simulation. December 1974. pp 181-189.

[MATHEWSON, 84]
S. C. Mathewson. 1984. "The Application of Program Generator Software and Its Extensions to Discrete Event Simulation Modeling." IIE Transactions, vol. 16, no. 1 (March):3-18.

[MATHEWSON, 85]
S. C. Mathewson. 1985. "Simulation Program Generators: Code and Animation on a PC." J. Opl Res. Soc. Vol. 36 No. 7. pp 583-589.

[McARTHUR & KN, 84]
D. McArthur, P. Klahr and S. Narain. 1984. "ROSS: An Object-Oriented Language for Constructing Simulations." R-3160-AF. Rand Corporation.

[McCABE, 86]
F. G. McCabe. 1986. "Logic and objects." Dept. Computing. Imperial College, London. DOC 86/9.

[MIDDLETON & Z, 86]
S. Middleton and R. Zanconato. 1986. "BLOBS: An object-oriented language for simulation and reasoning." In [Kerckhoffs & VZ (eds), 86] pp 130-135.

[MILLS, 86]
R. I. Mills. 1986. "Simulation for manufacturing systems - a critical review." In [FMS-5, 86] PP 225-234.

[MINGERS, 87]
J. Mingers. 1987. "Expert Systems -- Rule Induction with Statistical Data." J. Opl Res. Soc. Vol. 38, No. 1, pp. 39-47.

[MINSKY, 75]
M. Minsky. 1975. A framework for representing knowledge. In [Winston (ed), 75]

[MOREIRA da SILVA, 82]
C. A. R. Moreira da Silva. 1982. The development of a decision support system generator via action research. PhD thesis. University of Warwick. School of Industrial and Business Studies.

[MUETZELFELDT & RUB, 87]
R. Muetzelfeldt, D. Robertson, M. Uschold and A. Bundy. 1987. Computer-aided construction of ecological simulation programs." DAI Research paper no. 314. Department of Artificial Intelligence, University of Edinburgh.

[MURRAY & S, 88]
K. J. Murray and S. V. Sheppard. 1988. "Knowledge-based simulation model specification." Simulation. March 1988 pp 112-119.

[NANCE, 84]
R. E. Nance. 1984. "Model Development revisited." In [WSC, 84] pp 75-80.

[NEWELL, 69]
A. Newell. 1969. "Heuristic Programming: Ill-Structured Problems." In [Aronofsky (ed), 69] pp 360-414.

[NII, 86a]
H. P. Nii. 1986. "Blackboard systems: The Blackboard model of problem solving and the evolution of Blackboard Architectures." The AI Magazine. Summer 1986.

[NII, 86b]
H. P. Nii. 1986. "Blackboard systems: Blackboard Application systems, Blackboard systems from a knowledge engineering perspective." The AI Magazine. August 1986.

[O'KEEFE, 84]
R. M. Okeefe. 1984. "Developing simulation models: An interpreter for V.I.S." Ph.D. Thesis. Southampton University: Faculty of Mathematical Studies.

[O'KEEFE, 86a]
R. M. O'Keefe. 1986. "Advisory systems in simulation." In [Kerckhoffs & VZ (eds), 86] pp 73-78.

[O'KEEFE, 86b]
R. M. O'Keefe. 1986. "The three-phase approach: A comment on 'strategy-related characteristics of discrete languages and models'." Simulation 47:5 (November): 208-210.

[O'KEEFE & R, 87]
R. M. O'Keefe and J. W. Roach. 1987. "Artificial Intelligence Approaches to Simulation." J. Opl Res. Soc. Vol. 38, No. 8, pp. 713-722.

[OOPW, 86]
Proceedings of Object-oriented Programming Workshop. IBM Yorktown Heights: 9-13 June 1986. Special issue of ACM SIGPLAN Notices vol. 21 no. 10 (October 1986). ISBN: 0-89791-205-5.

[OREN, 82]
T. I. Oren. 1982. "Computer-Aided Modelling Systems." in [Cellier (ed), 82], pp 189-203.

[OREN, 86]
T. I. Oren. 1986. "Artificial intelligence and simulation." In [Kerckhoffs & VZ (eds), 86] pp 3-8.

[Oren & SR (eds), 80]
Simulation with Discrete Models: A State-of-the-Art View. 1980. T. I. Oren (editor-in-chief), C. M. Shub and P. F. Roth (editors). New York: Institute of Electrical and Electronics Engineers (IEEE TH0079-4).

[PAUL, 88]
R. J. Paul. 1988. Simulation Modelling: The CASM Project. London School of Economics and Political Science. Paper presented at (a) Brioni, Yugoslavia: The Annual Operational Research Symposium of Yugoslavia, 11-14 October, 1988 and at (b) Sao Jose dos Campos, Sao Paulo, Brazil: The 2nd Brazilian Workshop on Simulation, 1-2 September, 1988.

[PEGDEN & P, 79]
C. D. Pegden and A. A. B. Pritsker. 1979. "SLAM: Simulation language for alternative modelling." Simulation November 1979 pp 145-157.

[PIDD, 84]
M. Pidd. 1984. Computer Simulation in Management Science. Chichester, etc.: John Wiley and Sons. ISBN: 0-471-90281-0.

[POE & NPS, 84]
M. D. Poe; R. Nasr; J. Potter and J. Slinn. 1984. "A KWIC (Key Word in Context) Bibliography on PROLOG and Logic Programming." J. Logic Programming 1984:1:81:142.

[POOLE & S, 77]
T. G. Poole and J. Z. Szymankiewicz. 1977. Using simulation to solve problems. London, etc.: McGraw-Hill. ISBN:

[PRITSKER, 79]
A. A. B. Pritsker. 1979. "Compilation of definitions of simulation." Simulation. August 1979. pp 61-63.

[RADIYA & S, 87]
A. Radiya and R. G. Sargent. 1987. "Logic programming and discrete event simulation." In [Luker & B (eds), 87] pp 64-71.

[RADZIKOWSKI, 84]
P. Radzikowski. 1984. "Framework of the decision support expert systems." In [WSC, 84] pp 507-515.

[REDDY & FHM, 86]
Y. V. R. Reddy; M. S. Fox; N. Husain and M. McRoberts. 1986. "The Knowledge-Based Simulation System." IEEE Software, March:26-37.

[REICHGELT & V, 86]
H. Reichgelt and F. van Harmelen. 1986. "Criteria for choosing representation languages and control regimes for expert systems." DAI Research Paper no. 287. Department of Artificial Intelligence, University of Edinburgh.

[RICH, 83]
E. Rich. 1983. Artificial Intelligence. McGraw-Hill Book Company. ISBN 0-07-052261-8 and ISBN 0-07-Y66508-7 (International Student Edition).

[ROBERTSON, 86]
P. Robertson. 1986. "A rule based expert simulation environment." In [Luker & A (eds), 86] pp 9-15.

[RUBENS, 79]
G. T. Rubens. 1979. A study of the use of V.I.S. for decision making in a complex production system. MSc Thesis. School of Industrial and Business Studies. University of Warwick.

[RUIZ-MIER & T, 87]
S. Ruiz-Mier and J. Talavage. 1987. "A hybrid paradigm for modeling of complex systems." Simulation 48:4 (April): 135-141.

[SARGENT & R, 88]
R. G. Sargent and M. J. Rao. 1988. "An experimental advisory system for operational validity." In [Henson (ed), 88] pp 245-250.

[SCHANK & A, 77]
R. C. Schank and R. P. Abelson. 1977. Scripts, Plans, Goals, and Understanding. Hilladale, N.J.:Erlbaum.

[SCHMIDT, 88]
B. Schmidt. 1988. "Systems analysis, model construction, simulation: Methodological basis of the simulation system SIMPLEX-II." In [Huntsinger & MKV (eds), 88] pp 39-46.

[SECKER, 77]
R. J. R. Secker. 1977. That V.I.S. offers a viable technique for examining production planning and scheduling problems. MSc Thesis. School of Industrial and Business Studies. University of Warwick.

[SHANNON, 86]
R. E. Shannon. 1986. "Intelligent simulation environment." In [Luker & A (eds), 86] pp 150-156.

[SHANNON & MA, 85]
R. E. Shannon; R. Mayer and H. H. Adelsberger. 1985. "Expert systems and simulation." Simulation 44:6 (June). 275-284.

[Shapiro (ed), 87]
Encyclopedia of Artificial Intelligence. S. C. Shapiro (editor-in-chief). 1987. John Wiley. ISBN: 0-471-62974-x (vol. 1), 0-471-62973-1 (vol. 2), 0-471-80748-6 (set).

[SHEPPARD, 83]
S. Sheppard. 1983. "Applying software engineering to simulation." Simulation January: 13-19.

[SHUB, 80]
C. M. Shub. 1980. "Discrete event simulation languages." In [Oren & SR (eds), 80] pp 107-124.

[SIMON, 81]
H. A. Simon. 1981. "Information-Processing Models of Cognition." Journal of the American Society for Information Science. September 1981. pp 364-377

[SIMON, 87]
H. A. Simon. 1987. "Two Heads Are Better then One: The Collaboration between AI and OR." Interfaces. 17:4 (July-August) pp 8-15.

[SIMON & DHPRSSTTW, 87]
H. A. Simon; G. B. Dantzig; R. Hogarth; C. R. Plott; H. Raiffa; T. C. Schelling; K. A. Shepsle; R. Thaler; A. Tversky and S. Winter. 1987. "Decision Making and Problem Solving." Interfaces 17:5 (September-October) pp 11-31.

[SISSON, 69]
R. L. Sisson. 1969. "Simulation: Uses." In [Aronofsky (ed), 69] pp 17-69.

[SOL, 86]
H. G. Sol. 1986. "Expert systems for modelling of decision support and information systems." In [Elzas & OI (eds), 86] pp 353-363.

[Sol (ed), 83]
Processes and Tools for Decision Support. Proceedings of the Joint IFIP WG8.3/IIASA Working Conference. Schloss Laxenburg, Austria: 19-21 July 1982. H. G. Sol (editor). Amsterdam, etc.: North-Holland Publishing Company. ISBN: 0 444 86569 1.

[SSD, 85]
Proceedings of Third International Workshop on Software Specification and Design. London: 26-27 August 1985. Washington DC, USA: IEEE Computer Society Press. ISBN: 0-8186-0638-X.

[STANDRIDGE, 86]
C. R. Standridge. 1986. "An approach to model composition from existing modules." In [Elzas & OZ (eds), 86] pp 113-120.

[STANDRIDGE & P, 82]
C. R. Standridge and A. A. B. Pritsker. 1982. "Using Data Base Capabilities in Simulation." In [Cellier (ed), 82] pp 347-365.

[STEFIK & B, 86]
M. Stefik and D. G. Bobrow. 1986. "Object-oriented Programming: Themes and variations." The AI Magazine, Vol. 6, No. 4. pp 40-62.

[STERLING & S, 86]
L. Sterling and E. Shapiro. 1986. The Art of Prolog: Advanced Programming Techniques. Cambridge, MA, etc.: The MIT Press. (MIT Press series in Logic Programming) ISBN: 0-262-19250-0 (hard) and 0-262-69105-1 (paper).

[STROUSTUP, 86]
B. Stroustrup. 1986. The C++ Programming Language. Reading, Massachusetts: Addison-Wesley.

[SUBRAHMANIAN & C, 81]
E. Subrahmanian and R. L. Cannon. 1981. "A generator program for models of discrete-event systems." Simulation March:93-101.

[TAYLOR & H, 88]
R. P. Taylor and R. D. Hurrion. 1988. "An expert advisor for simulation experimental design and analysis." In [Henson (ed), 88] pp. 238-244.

[TAYLOR, 88]
R. P. Taylor. 1988. An Artificial Intelligence Framework for Experimental Design and Analysis in Discrete Event Simulation. Ph.D. Thesis. Warwick Business School. University of Warwick.

[TOCHER, 69]
K. D. Tocher. 1969. "Simulation: Languages." In [Aronofsky (ed), 69] pp 71-113.

[TOMLINSON & D, 83]
R. Tomlinson and R. Dyson. 1983. "Some Systems Aspects of Strategic Planning." J. Opl Res. Soc. vol. 34, no. 8, pp 765-778.

[Tomlinson & K (eds), 84]
Rethinking the Process of Operational Research and Systems Analysis. 1984. R. Tomlinson and I. Kiss. Oxford, etc: Pergamon Press. ISBN 0-08-030829-5 (Hardcover). ISBN 0-08-030830-9 (Flexicover).

[UKSC, 81]
Proceedings of the 1981 UKSC Conference on Computer Simulation. 13-15 May 1981, Harrogate, England. Surrey: Westbury House (the books division of IPC Science and Technology Press). ISBN: 0 86103 051 6.

[UKSC, 84]
Proceedings of the 1984 UKSC Conference on Computer Simulation. 12-14 September, 1984. University of Bath, England. D. J. Murray-Smith (ed). London, etc.: Butterworths. ISBN 0-408-01504-7.

[UNGER & DCB, 86]
B. Unger, A. Dewar, J. Cleary and G. Birtwistle. 1986. "The Jade approach to distributed software development." In [Kerckhoffs & VZ (eds), 86] pp 178-188.

[VAUCHER, 85]
J. G. Vaucher. 1985. "Views of modelling: Comparing the simulation and AI approaches." In [Birtwistle (ed), 85] pp 3-7.

[VAUCHER & L, 87]
J. G. Vaucher and G. Lapalme. 1987. "Process-oriented simulation in Prolog." In [Luker & B (eds), 87] pp 41-46.

[WALES & L, 86]
"An Environemnt for Discrete Event simulation." In [Luker & A (eds), 86] pp 58-62.

[WALKER & M, 86]
T. C. Walker and R. K. Miller. 1986. Expert Systems 1987: An Assessment of Technology and Applications. Madison, GA: SEAI Technical Publications. ISBN: 0-89671-082-3.

[WARREN, 80]
D. H. D. Warren. 1980. "Logic Programming and Compiler Writing." Software--Practice and Experience, vol. 10, 97-125.

[Webber & N (eds), 81]
Readings in Artificial Intelligence. 1981. B. L. Webber and N. J. Nilsson (eds). Tioga Publishing Company.

[WHITE, 85]
D. J. White. 1985. Operational Research. Chichester, NewYork, Brisbane, Toronto, Singapore: John Wiley & Sons. ISBN 0-471-90717-0 (cloth). ISBN 0-471-90718-9 (paper).

[WICKELGREN, 74]
W. A. Wickelgren. 1974. How to Solve Problems: Elements of a Theory of Problems and Problem Solving. San Francisco, USA: W. H. Freeman and Company. ISBN 0-7167-0846-9 and ISBN 0-7167-0845-0 (pbk.)

[Winston (ed), 75]
The Psychology of Computer Vision. 1975. P. H. Winston (ed). New York: McGraw-Hill.

[WITHERS, 81]
S. J. Withers. 1981. Towards the on-line development of visual interactive simulation models. PhD Thesis, School of Industrial and Business Studies. University of Warwick.

[WSC, 84]
Proceedings of 1984 Winter Simulation Conference. 28-30 November, 1984, Dallas, Texas. S. Sheppard, U. W. Pooch and C. D. Pegden (editors). San Diego, California: Society for Computer Simulation.

[ZEIGLER, 80]
B. P. Zeigler. 1980. "Concepts and software for advanced simulation methodologies." In [Oren & SR (eds), 80] pp 25-44.

[ZEIGLER, 84]
B. P. Zeigler. 1984. "System-Theoretic Representation of Simulation Models." IIE Transactions vol. 16, no. 1 (March):19-34.

[Zeigler & EKO (eds), 79]
Methodology in Systems Modelling and Simulation. 1979. B. P. Zeigler (editor-in-chief), M. S. Elzas, G. J. Klir and T. I. Oren (editors). Amsterdam, etc.: North-Holland. ISBN: 0 444 853405.

REFERENCES CITED WITHIN QUOTES

(Beauchamp and Field, 1979)
J. N. Beauchamp and R. C. Field. 1979. "Simulation Modelling by Stepwise Refinement." In: Proceedings of the Winter Simulation Conference, San Diego, CA.

(Bell and O'Keefe, 1987)
P. C. Bell and R. M. O'Keefe. 1987. Visual interactive simulation - history, recent developments and major issues. Simulation. In press.

(Beltrami and Bodin, 1974)
E. Beltrami and L. Bodin. 1974. "Networks and vehicle routing for municipal waste collection." Networks, Vol. 4, No. 1, pp 65-94.

(Brown et al., 1986)
J. Brown; J. Cook; L. Groner and E. Eusebi. 1986. "Algorithms for Artificial Intelligence in APL2." IBM Santa Teresa Report TR 03.81, IBM Santa Teresa Lab M75/E42, 555 Bally Ave., P. O. Box 50020, San Jose, California 95150.

(Chandrasekaran, 1986)
B. Chandrasekaran. 1986. "Generic tasks in knowledge-based reasoning: High level building blocks for expert system design." IEEE Expert, Vol. 1, No. 3, pp 23-30.

(Clocksin and Mellish, 1984)
W. Clocksin and C. Mellish. 1984. Programming in Prolog. second edition, Springer-Verlag, New York

(Cromarty, 1985)
S. Cromarty. 1985. "What are current expert system tools missing." Proceedings of the IEEE 1985 Computer Conference (COMPCON-85), IEEE Computer Society Press, Los Alamitos, California, pp.411-418.

(Cutler, 1980)
M. M. Cutler. 1980. A formal program model for discrete event simulation and its use in the verification and validation of system models and implementations. Doct. Dissertation, UCLA, Los Angeles, CA.

(Dahl, 1983)
V. Dahl. 1983. "Logic programming as a representation of knowledge." Computer, Vol. 16, No. 10, pp. 106-113.

(Dykstra, 1976)
E. W. Dykstra. 1976. A Discipline of Programming. Prentice Hall, N. J.

(Estrin, 1978)
G. Estrin. 1978. "A method for design of digital systems supported by SARA at the age of one." AFIPS Conference Proceedings, NCC.

(Evans, 1984)
J. B. Evans. 1984. Simulation and intelligence. Technical Report TR-A5-84, Centre of Computer Studies and Applications, University of Hong Kong.

(Expert Systems Development Environment/VM Reference Manual, 1985)
Expert Systems Development Environment/VM Reference Manual. 1985. manual No. SH20-9609-1, International Business Machines Corporation, Menlo Park, California.

(Hong, 1986)
S. Hong. 1986. "Guest editor's introduction for issue devoted to expert systems in engineering." IEEE Computer, Vol. 19, No. 7, pp. 12-15.

(Karplus, 1976)
Karplus, W. J. (1976), "The spectrum of mathematical modelling and systems simulation", in Dekker, L. (ed), Simulation of Systems, North-Holland, Amsterdam, pp. 5-13.

(Liskove and Zilles, 1974)
B. Liskove and S. Zilles. 1974. "Programming with abstract data types." ACM SIGPLAN Notices, 9 1974.

(Minsky, 65)
M. L. Minsky. 1965. "Matter, Mind, and Models." Proceedings of IFIP Congress, Vol. 1. Spartan Books, pp 45-49.

(Nii, 1986)
P. Nii. 1986. "The blackboard model of problem solving." AI Magazine, Vol. 7, No. 2, pp. 38-53.

(Nijssen, 1977)
Architecture and Models in Data Base Management Systems. 1977. G. M. Nijssen (ed). North Holland Pub. Co., Amsterdam.

(Oren, 75)
T. I. Oren. 1975. Syntactic Errors of the Original Formal Definition of CSSL 1967. Technical Report No. 75-01, Computer Science Department, University of Ottawa, Ottawa, Ontario, 1975, 57 p. (Also available from IEEE Computer Society Repository, No. R75-78.).

(Reitman, 1964)
W. Reitman. 1964. "Heuristic decision procedures, open constraints, and structure of ill defined problems." in Human Judgment and Optimality. M. Shelley and G. Bryans, eds., John Wiley and Sons, New York.

(Robinson, 1983)
J. A. Robinson. 1983. "Logic Programming -- Past Present and Future." New Generation Computing, Springer Verlag. Vol. 1, No. 2.

(Ryan, 1979)
M. T. Ryan. 1979. "Software Engineering and Simulation." In: Proceedings of the Winter Simulation Conference, San Diego, CA.

(Rychener, 1985)
M. Rychener. 1985. "Expert systems for engineering design." Expert Systems, Vol. 2, No. 1, pp. 30-44.

(Rzevski, 1980)
G. Rzevski. 1980. "Systematic design of simulation software." In: Proceedings of Simulation '80, Interlaken, Switzerland.

(Sanguinetti, 1979)
J. Sanguinetti. 1979. "A technique for integrating simulation and systems design." In: Conference on Simulation, Measurement and Modeling of Computer Systems. Sigmetrics/Simulatter, Fall.

(Schor, 1986)
M. Schor. 1986. "Declarative knowledge programming: Better than procedural?" IEEE Expert, Vol. 1, No. 1, pp. 36-46.

(SIGART70, 80)
ACM SIGART Newsletter, Special Issue on Knowledge Representation. No. 70, February 1980.

(Simon and Newell, 1958)
H. Simon and A. Newell. 1958. "Heuristic problem solving: The next advance in operations research." Operations Research, Vol. 6, No. 1, pp. 1-10.

{Strauss et al., 67}
J. C. Strauss. 1967. "The SCi Continuous System Simulation Language (CSSL)." Simulation, Vol. 9, No. 6, December 1967. pp 281-303.

{Teichrow and Hershey, 1978}
D. Teichrow and E. Hershey. 1978. "PSL/PSA: A Computer-aided Technique for structure documentation and Analysis of Information Processing Systems." IEEE Trans. Soft. Eng. 3/1.

{Weizenbaum, 1976}
Weizenbaum, J. (1976), Computer Power and Human Reason, W. H. Freeman & Co., San Francisco.

{Yeh, 1977}
Current trends in programming methodology, volume 1: Software specification and design. 1977. R. Yeh (ed). Prentice Hall, N. J.

{Zeigler et. al.(eds), 1979}
B. P. Zeigler, M. S. Elzas, G. J. Klir and T. I. Oren. 1979. Methodology in systems modelling and simulation. North Holland, Amsterdam.

Published in
ARTIFICIAL INTELLIGENCE AND SIMULATION: The Diversity of Applications. Proceedings of SCS Multiconference on Artificial Intelligence and Simulation, 3-5 February 1988, San Diego, California, USA. A Society for Computer Simulation publication. ISBN 0-911801-30-4. pp 137-142.

APPENDIX I

AUTOMATIC MODEL GENERATION USING A PROLOG MODEL-BASE

Ali Ahmad and Robert D. Hurrion
School of Industrial and Business Studies
University of Warwick
Coventry, CV4 7AL, UK

ABSTRACT

This paper describes part of the research carried out to achieve the capability of generating discrete simulation models by retrieving and assembling relevant components from a PROLOG data base (i.e. Model-base). The paper begins with a short description of the need for such a capability and then moves on to discuss various problems related to the representation of two types of knowledge required to be coded. These are simulation modelling knowledge and domain knowledge in a generalized form suitable for retrieval, instantiation and assembly. The method by which these representation problems have been tackled are described. Implications, advantages, and limitations of using PROLOG as the language for implementation are also discussed. In order to run an automatically assembled model, a prototype PROLOG simulation engine was written. Design features of the engine are described and include the ability to run simulation models consisting of events, activities and processes simultaneously. The need for this feature is ascertained and its relevance within the context of the project is described.

In order to illustrate the use of the model generator, a prototype model-base in the service domain is described. Using the PROLOG model base it is shown that a model can be specified at a very high level by naming various intelligent components from the model base. It is then assembled and run immediately by the simulation engine.

The PROLOG model base contains components that represent either an event or an activity in the form of a segment of a process.

1. INTRODUCTION

There has been considerable practical and research interest shown in the application of the emerging Artificial Intelligence (AI) and Expert Systems technology to Operational Research (O'Keefe 1986). Proponents of AI have stated that it will revolutionise management's use of computing and with particular reference to the practice of simulation, Shannon (1985) suggests that if these claims are even only half true then AI is bound to have a profound effect on the art and science of simulation.

The use of Discrete Event Simulation as a problem solving technique involves the building of an adequate computational model of the system under study and then experimenting with it to gain sufficient understanding of the system's behaviour so that the requisite decisions about the system can be made with an acceptable level of confidence. This paper focuses on the model building part of a simulation exercise.

Various formalisms have been used to describe systems (e.g. activity cycle diagrams, flow diagrams, petri-nets). These are useful from the point of view of documenting the system for human communication and provide the basis for constructing computer executable code and its verification. Each such formalism have its advantages and limitations. Another class of software tools known as simulation model generators are also available which accept the model in a given formalism and generate code in one of the compilable (or interpretable) languages. These software tools, although saving considerable time also have their own limitations. Most of the times the code generated by a model generator must be edited to include those aspects of system which are either not supported by the underlying formalism used or are beyond the capability of the model generator to generate the code for these.

It would be desirable if it were possible to have a system description formalism which is directly understood by humans and at the same time directly executable by computer. This paper supports the view that logic programming paradigm provides necessary features which can be exploited towards the achievement of this ideal (Kowalski 1979).

The developments within Artificial Intelligence (AI) research has led to the application of Knowledge Engineering (KE) techniques in an Expert System framwork for solving problems within specific domains (Jackson 1986). The building of a computer simulation model essentially involves integration of two types of knowledge, namely the simulation methodology and application domain knowledge, in the form of a computer executable program. Successful integration of these two types of knowledge require considerable amount of skill and therefore it is worthwhile to investigate the possibility of incorporating these two types of knowledge in a knowledge-base together with the capability to 'intelligently' retrieve such knowledge for the purpose of automatic generation of a computer executable simulation model.

This paper describes a prototype implementation of an 'intelligent' modelling environment using PROLOG. The reader is assumed to have a working knowledge of PROLOG.

2. AN OVERVIEW

Fig. 1 presents an overview of various components of the modelling environment and the way they relate to each other. In order to facilitate

exposition, a simple shop model will be used as an example. In this model '... Customers enter a general food store. Customers move sequentially through, and are served at, vegetable, meat and fruit stalls before queueing for their final packing and payment, after which they leave the shop ...'

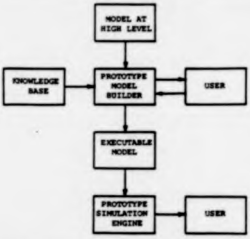


Fig. 1. An overview of the prototype 'intelligent' modelling environment

The model description is at a high level and is presented to the model-builder as a text file. The following two PROLOG clauses represent the shop model:

```
model(shop) :-
        actor(shopper).

goal(shopper) :-
        subgoal(buy_veg),
        subgoal(buy_meat),
        subgoal(buy_fruit),
        subgoal(pay_n_leave).
```

The 'model' clause specifies that there is one entity type in the 'shop' model and it is known by the name 'shopper'. The 'goal' clause specifies that the 'shopper' goes through four service stations in succession which have been given names 'buy_veg', 'buy_meat', 'buy_fruit' and 'pay_n_leave'. The knowledge with regard to the 'shopper' and the four 'subgoals' is contained in the knowledge-base.

As the model description is not complete further information is 'intelligently' sought from the user. After a brief interaction with the user, the model-builder delivers the model as two text files. The first file known as a declaration-file provides clauses for the definition of sets, classes of entities, resources, and initial conditions. The declaration-file for the shop model is shown in Fig. 2. The contents of the declaration file will be described in detail after the section in this paper of

```
:- create_world(simulation).
:- data_world(_, simulation).
:- initialize.

:- nl, nl, write($Defining sets$).
:- vset(buy_fruit_1).
:- vset(buy_meat_1).
:- vset(buy_veg_1).
:- vset(pay_n_leave_1).
:- vset(shopper_q0).
:- vset(shopper_q1).
:- vset(shopper_q2).
:- vset(shopper_q3).
:- vset(shopper_q4).
:- vset(shopper_q5).

:- nl, nl, write($Defining resources$).
:- vresource(balance_1,1).
:- vresource(balance_2,1).
:- vresource(butcher_1,1).
:- vresource(check_out_1,1).

:- nl, nl, write($Defining classes$).
:- vclass(shopper,1,20).

:- nl, nl, write($Loading classes in their
pools$).
:- vload(shopper,1,20,shopper_q0).

:- nl, nl, write($Scheduling initial events$).
:- introduce(shopper(1),shopper_process,7.8).


:- main.
:- nl, nl, nl, nl,
      write($Press a key to start simulation $),
keyb(_,_),
:- simulate.
```

Fig. 2. Listing of declaration-file for the shop model

knowledge representation, however for the moment, 'vset' clauses define sets which are either activity-sets or queues, 'vresource' clauses define resources in the model, 'vclass' clause define a class of entities and 'introduce' clause specifies initial arrivals (e.g. arrival time of first 'shopper').

The second file produced by the model-builder contains the processes for various 'actors' present in the model and therefore is referred to as the process-file. Fig. 3 shows the process file of the shop model which lists the process of the 'shopper'.

The declaration file and the process-file can be edited if required. It is then possible to run the model directly by presenting the two files to the prototype simulation engine.

3. REPRESENTATION OF KNOWLEDGE

Two types of knowledge need to be represented, namely:

- Simulation knowledge, and
- Application-domain knowledge

3.1. Simulation Knowledge-Base

The simulation knowledge consists of a number of situations which are frequently present in simulation models. Four such situations have been incorporated in our prototype knowledge-base. These are shown with the help of diagrams in Fig. 4. Diagram (a) represents the common situation when an entity waits in queue X until it is at the head of the queue and until a resource is available when it can receive service for certain duration. At the end of the service it releases the resource and joins queue Y. The other diagrams are similar. In (b) a resource is not required and represents a delay, in (c) the entity leaves a message on a blackboard before joining queue Y and in (d) the entity can only receive service when there is an idle resource and also when a particular message is present on the blackboard (usually left by another entity). The entity erases the message before the start of the service.

```
process(shopper_process) :-
        move(shopper_q0,shopper_q1),
        gen_next(shopper_process,14.5,shopper_q0),
        wait_until(head_of(shopper_q1)),
        wait_until(idle(balance_2)),
        move(shopper_q1,buy_veg_1),
        seize(balance_2),
        hold(10.1),
        release(balance_2),
        move(buy_veg_1,shopper_q2),
        wait_until(head_of(shopper_q2)),
        wait_until(idle(butcher_1)),
        move(shopper_q2,buy_meat_1),
        seize(butcher_1),
        hold(12.3),
        release(butcher_1),
        move(buy_meat_1,shopper_q3),
        wait_until(head_of(shopper_q3)),
        wait_until(idle(balance_1)),
        move(shopper_q3,buy_fruit_1),
        seize(balance_1),
        hold(15.6),
        release(balance_1),
        move(buy_fruit_1,shopper_q4),
        wait_until(head_of(shopper_q4)),
        wait_until(idle(check_out_1)),
        move(shopper_q4,pay_n_leave_1),
        seize(check_out_1),
        hold(3.5),
        release(check_out_1),
        move(pay_n_leave_1,shopper_q5),
        exit_system(shopper_q5,shopper_q0).
```

Fig. 3. Listing of process-file for the shop model

This simulation knowledge has been represented in the form of PROLOG clauses and is shown in Fig. 5. The head of each clause consists of predicate 'script' with three arguments. The first argument refers to the respective diagram label in Fig. 4 whereas the second and third arguments, PROLOG variables X and Y represent queues X and Y respectively as shown in Fig. 4. The body of each 'script' clause represents a generalised segment of code using the process view of simulation (Franta 1977). The generalisation comes from the use of keywords such as 'ACTIVITY_SET', 'RESOURCE', and 'DURATION'. During the course of model building, individual process segments are assembled to compose a process for each 'actor' present in the model. At this stage these keywords are replaced by appropriate instances of items which each keyword represents.

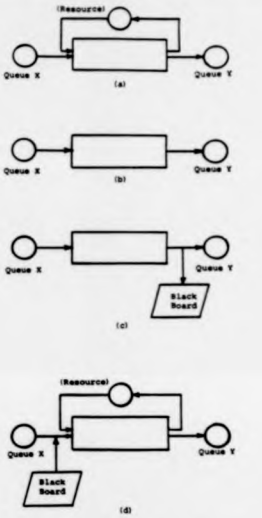


Fig. 4. Simulation-knowlege in diagrametic form.

All 'actors' must 'arrive' before they can start 'achieving' their 'subgoals' and must 'leave' when they have completed the last 'subgoal'. For this purpose two special 'script' clauses have been included in the knowledge-base, one for 'arrive' and the other for 'leave'.

```
script(active(Process, I_arrival), X, Y) :-
    move(X, Y),
    gen_next(Process, I_arrival, Y).

script(leave, X, Y) :-
    exit_system(X, Y).

script(a, X, Y) :-
    wait_until(need_of(X)),
    wait_until(idle('RESOURCE')),
    move(X, 'ACTIVITY_SET'),
    seize('RESOURCE'),
    hold('DURATION'),
    release('RESOURCE'),
    move('ACTIVITY_SET', Y).

script(b, X, Y) :-
    move( X, 'ACTIVITY_SET'),
    hold('DURATION'),
    move('ACTIVITY_SET', Y).

script(c, X, Y) :-
    move(X, 'ACTIVITY_SET'),
    hold('DURATION'),
    leave_message('MESSAGE'),
    move('ACTIVITY_SET', Y).

script(d, X, Y) :-
    wait_until((
        need_of(X),
        message_found('MESSAGE'),
        idle('RESOURCE')
        )),
    remove_message('MESSAGE'),
    move(X, 'ACTIVITY_SET'),
    seize('RESOURCE'),
    hold('DURATION'),
    release('RESOURCE'),
    move('ACTIVITY_SET', Y).
```

Fig. 5. Simulation knowledge base in the form of PROLOG clauses

3.2. [illegible]

The [illegible]-domain knowledge-base is subdivided into two parts. One part represents knowledge of 'subgoals'. It consists of PROLOG clauses with predicate 'subgoal_frame'. These are shown in Fig. 6. The other part represents the knowledge of 'actors'. It consists of PROLOG clauses with predicate 'actor_frame'. These are shown in Fig. 7. As the [illegible] needs support, both use a frame-like formation. The name of individual frame is given by the first argument whereas the second argument gives the name of the slot and its value.

The slots in the 'subgoal_frame' provide necessary information which relates keywords in 'script' clauses (Fig. 5) with respective slot names. For example in 'buy_veg' subgoal frame, the 'resource' slot indicates that keyword 'RESOURCE' in appropriate 'script' clause is to be replaced by an instance of 'balance'. Instances are created by adding an underscore and an instance number. For example first instance of resource 'balance' in our shop model will read as 'balance_1', the second as 'balance_2', and so on. Keyword 'ACTIVITY_SET' is replaced by an instance of the name of the subgoal_frame (e.g. 'buy_veg_1') and therefore there is no separate slot for it. Each subgoal frame must have a 'script' slot and must also

```
subgoal_frame(cross_in, resource(Channel)).
subgoal_frame(cross_in,
                duration(default(12.4))).
subgoal_frame(cross_in, script(a)).

subgoal_frame(cross_out, resource(Channel)).
subgoal_frame(cross_out,
                duration(default(17.2))).
subgoal_frame(cross_out, script(a)).

subgoal_frame(p_unload, duration(default(5.6))).
subgoal_frame(p_unload, script(b)).

subgoal_frame(c_unload, resource(con)).
subgoal_frame(c_unload,
                duration(default(25.3))).
subgoal_frame(c_unload, script(c)).

subgoal_frame(t_unload,
                duration(default(13.7))).
subgoal_frame(t_unload, script(c)).

subgoal_frame(buy_veg, resource(balance)).
subgoal_frame(buy_veg, duration(default(10.2))).
subgoal_frame(buy_veg, script(a)).

subgoal_frame(buy_meat, resource(butcher)).
subgoal_frame(buy_meat,
                duration(default(12.2))).
subgoal_frame(buy_meat, script(a)).

subgoal_frame(buy_fruit, resource(balance)).
subgoal_frame(buy_fruit,
                duration(default(3.0))).
subgoal_frame(buy_fruit, script(a)).

subgoal_frame(pay_n_leave, resource(check_out)).
subgoal_frame(pay_n_leave,
                duration(default(3.3))).
subgoal_frame(pay_n_leave, script(a)).
```

Fig. 6. Knowledge of 'subgoals'

```
actor_frame(p_ship, number_in_model(20)).
actor_frame(p_ship, arrival_pattern(20.5)).
actor_frame(p_ship, first_arrival(10.3)).

actor_frame(tanker, number_in_model(20)).
actor_frame(tanker, arrival_pattern(147.6)).
actor_frame(tanker, first_arrival(78.2)).

actor_frame(c_ship, number_in_model(20)).
actor_frame(c_ship, arrival_pattern(42.9)).
actor_frame(c_ship, first_arrival(24.5)).

actor_frame(shopper, number_in_model(20)).
actor_frame(shopper,
                arrival_pattern(negexp(14.5))).
actor_frame(shopper, first_arrival(7.8)).
```

Fig. 7. Knowledge of 'actors'

have the slots corresponding to the keywords used within the 'script' clause it refers.

The 'actor_frame' has three slots. The 'number_in_model' slot states the size of the class of entities denoted by a particular 'actor' the 'arrival_pattern' slot states the interarrival time distribution for the 'actor' and the 'first_arrival'

slot specifies the time for the 'arrival' of the first 'actor' in the model.

4. ANOTHER LOOK AT THE SHIP MODEL

Having considered the [illegible] scheme used it would now be more meaningful to study Fig. 2 and Fig. 3 which list the declaration-file and the process-file of the ship model as produced by the prototype model-builder by retrieving the relevant knowledge from the knowledge-base as described in the last section. It should be noticed that there are two instances of resource 'balance' present in the model. This was resolved during the user interaction.

5. THE PROCESS OF MODEL BUILDING

The model building process uses top-down decomposition of each 'subgoal' for each 'actor' to resolve various instances of activity sets, resources and other keywords. This resolving is carried out by identifying the interaction of various 'actors' for activity sets and resources, which is ascertained by retrieving relevant clauses from the knowledge base. The user is then presented with the choice to resolve various instances of each. For example two actors who have same 'subgoal' may either wait for service in one queue on a first come first served basis and enter in one activity set for service or they may have two separate queues with two activity sets which may either share one instance of resource required by the subgoal or have two instances of the resource one for each activity set.

Following the top-down decomposition, and resolving through user interaction, a bottom-up synthesis phase occurs. In this phase, individual subgoals are coupled together by generating queue names and identifying those queues in which more than one actors reside. Individual processes for each actor is then built by retrieving the respective scripts and at the same time instantiating the variables by queue names and replacing the key words by corresponding instances as resolved during the decomposition phase. Also during the synthesis phase various declarations are also completed which must be executed to provide necessary house keeping for the processes to execute.

It is interesting to note that although user-interaction is used to extend and complete the initial high level model specifications, in principle it is possible to generate a series of models without user interaction.

6. PROTOTYPE SIMULATION ENGINE

Although the prototype model builder described generates the model using the process world view, the simulation engine is capable of running models written using either the process view and/or event world views.

7. A FURTHER EXAMPLE USING THE SAME KNOWLEDGE BASE

A second example is given to show that a completely different model can be assembled from the same knowledge base.

A harbour is [illegible] by a narrow entry channel alone which only one ship at a time can pass. The following type of ships use the harbour:

1. passenger
2. tanker
3. cargo

The 'knowledge' about these ships is contained in the 'actors' knowledge base (Fig. 7) under the names 'p_ship', 'tanker', and 'c_ship'. Each type of ship, after arrival enter the harbour through the channel and then unload at its respective berth and leaves through the channel. These 'subgoals' have been given names 'cross_in', 'p_unload' for passenger ships, 't_unload' for tankers 'c_unload' for cargo ships, and 'cross_out'. The knowledge of these subgoals is contained in the knowledge base (Fig. 6). A model for the harbour problem can be specified at a high level as follows:

```
model(harbour) :-
        actor(p_ship),
        actor(tanker),
        actor(c_ship).

goal(p_ship) :-
        subgoal(cross_in),
        subgoal(p_unload),
        subgoal(cross_out).

goal(tanker) :-
        subgoal(cross_in),
        subgoal(t_unload),
        subgoal(cross_out).

goal(c_ship) :-
        subgoal(cross_in),
        subgoal(c_unload),
        subgoal(cross_out).
```

The goals and subgoals for the entities are all that one needed to specify the model at the highest level. The simulation knowledge base contains the information on how each subgoal is satisfied. These high level goals are submitted to the model builder which assembles the model asking the user to resolve resource levels.

8. IMPLEMENTATION

The prototype model builder and the prototype simulation engine have been implemented on an *IBM-PC XT with 640 kb memory using **Arity/Prolog interpreter running under *PC-DOS. It was possible to compile an earlier version of the simulation engine which increases the execution speed by a factor of 2 approximately. The execution trace appears on screen with simulated time together with the specific entity (or entities) and their respective events.

9. CONCLUSIONS

With the facility of a knowledge based model builder available, a discrete event simulation model can be constructed by non-experts by specifying their models at a high level which is interpreted 'intelligently' by computer to resolve various interactions within the model to complete the specification of model in an executable form.

*IBM-PC AT and PC-DOS are registered tradenames of International Business Machines Corporation.
**Arity/Prolog is a registered tradename of Arity Corporation.

This paper has shown a method of automatically assembling simulation models. The paper has used two small examples to show its model building principles but the approach is extendible to larger, more complex simulation models.

10. ACKNOWLEDGMENT

The research work described in this paper was supported by a British Council Fellowship Award.

11. REFERENCES

Franta, W. R. 1977. The Process View of Simulation, North-Holland, New York, NY, USA

Jackson, P. 1986. Introduction to Expert Systems, Addison-Wesley, Wokingham, England, UK

Kowalski, R. 1979. Logic for Problem Solving, North-Holland, New York, NY, USA

O'Keefe, R. M.; V. Belton and T. Ball. 1986. "Experiences with Using Expert Systems in OR." J. Opl. Res. Soc. 37, no. 7:657-668.

Shannon, R. E.; R. Mayer and H. H. Adelsberger. 1985. "Expert systems and simulation." Simulation 44, no. 6 :275-284

APPENDIX II

THE PROGRAMMING FACILITIES PROVIDED BY THE MICROSIM SIMULATION PACKAGE

The following is a summary list of the facilities provided by the MICROSIM visual interactive simulation package by Hurrion. This list has been compiled from the user manual for MICROSIM.

A. SYSTEM STARTUP

CALL SETSYS
To initialize the system

B. ELEMENT DEFINITION ROUTINES

CALL VSET
To define a set

CALL VENTIT
To define an entity

CALL VCLASS
To define a class of entities

CALL VHIST
To define a histogram

C. MODEL MANIPULATION ROUTINES

CALL VLOAD
To load a class of entities (or a section of a class) in a set

CALL VADDLA
To add an entity at the last position in a set

CALL VADDFI
To add an entity at the first position in a set

CALL VBEHEA
To remove the entity at the first position in a set

CALL VDELETE
To remove an entity from a set

CALL VBETAI
To remove the entity at the last position in a set

CALL VEMPTY
To remove all entities in a set

CALL SETATT
To set a specific attribute of an entity to a specific value

CALL MOVEXY
To move an entity from one set to the last position of another set while on screen moving in horizontal direction first

CALL MOVEYX
To move an entity from one set to the last position of another set while on screen moving in vertical direction first

D. MODEL STATUS/INSPECTION ROUTINES

I = ISIZE(ISET)
The number of entities in ISET are returned

I = IHEAD(ISET)
The entity at the head of ISET is returned

I = ITAIL(ISET)
The entity at the tail end of ISET is returned

I = IDENT(IPOS, ISET)
The entity at specified position IPOS among the entities in ISET is returned

I = IPOSIT(IENT, ISET)
The entities in set ISET are searched for a specific entity IENT and its postion is returned

I = IATT(IENT, N)
The current value of Nth attribute of entity IENT is returned.

E. ROUTINES FOR EVENT SCHEDULING AND TIME ADVANCE

```
CALL SCHEDL
        To schedule an event

CALL ADVANCE
        To advance simulation time to the next
        scheduled event
```

F. RANDOM VARIATE GENERATION

```
CALL SRESET(ISTRM)
        To reset the random number stream ISTRM back
        to original

R = RNDS(ISTRM)
        A sample from uniform distribution between
        0.0 and 1.0 is returned

R = RNEGEX(RMEAN, ISTRM)
        A random sample from a negative exponential
        distribution with mean RMEAN is returned
        using stream ISTRM

R = RNORM(RMEAN, VAR, ISTRM)
        A random sample from a normal distribution
        with mean RMEAN and variance VAR is returned
        using stream ISTRM.

I = INORM(RMEAN, VAR, ISTRM)
        A random sample (integer) from a normal
        distribution with mean RMEAN and variance VAR
        is returned using stream ISTRM

I = INEGEX(RMEAN, ISTRM)
        A random sample (integer) from a negative
        exponential distribution with mean RMEAN is
        returned using stream ISTRM

I = IPOISS(RMEAN, ISTRM)
        A random sample (integer) from a Poisson
        distribution with mean RMEAN is returned
        using stream ISTRM

I = IRAND(IA, IB, ISTRM)
        A random sample (integer) from a uniform
        distribution between IA and IB is returned
        using stream ISTRM.
```

G. SIMULATION DISPLAY ROUTINES

CALL TFORM
To display text on screen

CALL IFORM
To display integer on screen

CALL RFORM
To display a real number on screen

CALL SFORM
To display blank spaces on screen

CALL FILL
To fill an area with colour

CALL RECT
To draw a rectangle on the screen

CALL EFORM
To display the text associated with an entity

CALL SETDSP
To alter the screen display attributes related to an entity

CALL CLEAR
To clear the screen

CALL LSNOFF(LSN)
To turn the logical screen LSN off

CALL LSNON(LSN)
To turn the logical screen LSN on

L = LSNST(LSN)
The current status of logical screen LSN is returned

H. DATA RECORDING

CALL ADDHIST
To add a value to a histogram

R = RMEAN(IHIST)
The current mean of a histogram is returned

R = STDEV(IHIST)
The current standard deviation of a histogram is returned

CALL TIMEON
To start a 'time clock' associated with an entity

CALL RECORD
To read the 'time clock' associated with an entity and add the value read to a histogram

CALL TIMOFF
To reset the 'time clock' associated with an entity back to zero.

CALL RECON
To switch on all recording

CALL RECOFF
To switch off all recording

I. COMMANDS AVAILABLE IN INTERACTION MODE

RUN
To 'run' the model

GOTO
To 'run' the model upto a specified value of simulation time

ADVANCE
To 'inch' the model by one time unit at a time

BATCH
To run the model without animated graphic display to a specified value of simulation time.

SPEED
To set the speed of animated graphic display

REFORM
To reform the graphic display

DISPLAY
To display the logical screen numbers which are currently on

MONITOR
To display the current mode of recording

STOP
To stop the simulation run and return to operating system

END
To end the simulation by calling a system event.

ELEMENT
To inspect and change the attributes of entities.

OWN
To call the own interaction subroutine supplied by the user.

J. ROUTINES REQUIRED TO BE SUPPLIED BY USER

CALL FORMSC
To form any static graphic screen displays

CALL FORMTI
To display the simulation time

CALL OWNINT
To provide for the options for altering the model parameters during own interaction

BIBLIOGRAPHY

This bibliography represents additional reading and the entries included in the References therefore have not been repeated. An exception has been made for those entries which have been referred to from within this bibliography. Further, the keys which identify the entries have been used only for sorting the entries in alphabetic order and are not necessarily unique.

[ABDEL-HAMID & S, 88]
T. K. Abdel-Hamid and T. R. Sivasankaran. 1988. "Incorporating expert system technology into simulation modeling: An expert-simulator for project management." In [Henson (ed), 88] pp. 268-274.

[ACMCSC, 86]
Proceedings of the 1986 ACM Fourteenth Annual Computer Science Conference. Cincinnati, OH, USA: 4-6 Feb 1986. New York, USA: Association of Computing Machinery.

[ADELSBERGER, 86]
H. H. Adelsberger. 1986. "Introduction to artificial intelligence." In [Luker & A (eds), 86] pp 141-143.

[ARITY, 86]
The Arity/Prolog Programming Language. Concord, Massachusetts, USA: Arity Corporation.

[BAILES, 85]
P. A. Bailes. 1985. "A Low-Cost Implementation of Coroutines for C." Software--Practice and Experience, vol. 15(4) , 379-395, (April).

[BALCI, 86]
O. Balci. 1986. "Requirements for model development environments." Computers and Operations Research. vol. 13 no. 1 pp 53-67.

[BALCI & N, 86]
O. Balci and R. E. Nance. 1986. "Simulation model development environments: A research prototype." Technical report SRC-86-004 Systems Research Center and Department of Computer Science. Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061, USA.

[BALCI & N, 87]
O. Balci and R. E. Nance. 1987. "Simulation Model Development Environments: A Research Prototype." J. Opl Res. Soc. Vol. 38, No. 8, pp. 753-763.

[BALMER & P, 86]
D. W. Balmer and R. J. Paul. 1986. "CASM -- The Right Environment for Simulation." J. Opl Res. Soc. Vol. 37, No. 5, pp. 443-452.

[BAUMAN & T, 86]
R. Bauman and T. A. Turano. 1986. "Production based language for simulation of Petri nets." Simulation 47:5 (November): 191-198.

[BELL, 85]
M. Z. Bell. 1985. "Why Expert Systems Fail." J. Opl Res. Soc. Vol. 37, No. 6., pp. 603-618.

[BIRTWISTLE, 73]
G. M. Birtwistle. 1973. "SIMULA - Its features and prospects" In High level programming languages - the way ahead: Proceedings of British Computer Society Conference. NCC Publications.

[Birtwistle (ed), 85]
AI, Graphics and Simulation. 1985. Proceedings of the SCS Multiconference, January 1985, San Diego, California, USA. G. Birtwistle (editor). San Diego: Society for Computer Simulation. ISSN: 0735-9276.

[BOBROW & MS, 86]
D. G. Bobrow, S. Mittal and M. J. Stefik. 1986. "Expert systems: Perils and promise." Communications of the ACM. Vol. 29, No. 9 (September) pp 880-894.

[BOND & S, 88]
A. H. Bond and B. Soetarman. 1988. "Multiple abstraction in knowledge-based simulation." In [Henson (ed), 88] pp. 61-66.

[BOWEN, 86]
K. A. Bowen. 1986. "New Directions in Logic Programming." In [ACMCSC, 86] pp 19-27.

[BOWEN & K, 82]
K. A. Bowen and R. A. Kowalski. 1982. "Amalgamating Language and Metalanguage in Logic Programming." In [Clark & T (eds), 82] pp 153-172.

[BRACHMAN, 79]
R. J. Brachman. 1979. "On the Epistemological status of semantic networks." In [Findler (ed), 79] pp 3-49.

[BRACHMAN, 83]
R. J. Brachman. 1983. "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks." IEEE Computer. October:30-36.

[BRADY, 79]
M. Brady. 1979. "Expert Problem Solvers." In [Michie (ed), 79] pp 49.

[BRATKO, 86]
I. Bratko. 1986. Prolog Programming for Artificial Intelligence. Wokingham, etc.: Addison-Wesley Publishing Co. International Computer Science Series. ISBN: 0-201-14224-4.

[BRATLEY & FS, 83]
P. Bratley, B. L. Fox and L. E. Schrage. 1983. A Guide to Simulation. New York, etc.: Springer-Verlag. ISBN: 0-387-90820-X. (also 3-540-90820-X).

[BROWN, 81]
M. G. Brown. 1981. "Simulation languages and the development of microprocessor based products." In [UKSC, 81] 44-48.

[BUNDY, 84]
A. Bundy. 1984. "Intelligent front-ends." In [Fox (ed), 84] pp 15-24.

[CARRIE, 88]
A. Carrie. 1988. Simulation of Manufacturing Systems. Chichester, etc.: John Wiley and Sons. ISBN: 0-471-91574-2.

[CAVOURAS, 83]
J. C. Cavouras. 1983. "Implementing a Simulation Tool in a High-level Language with no Multitasking Facilities." Software--Practice and Experience, vol. 13, 809-815.

[CELLIER, 79]
F. E. Cellier. 1979. "Combined continuous/discrete system simulation languages --- usefulness, experiences and future development." In [Zeigler & EKO (eds), 79] pp 201-220.

[CHANDRASEKARAN, 84]
B. Chandrasekaran. 1984. "Expert systems: Matching techniques to tasks." In [Reitman (ed), 84] pp 41-64.

[CHISHOLM & S, 79]
I. H. Chisholm and D. H. Sleeman. 1979. "An Aide for Theory Formation." In [Michie (ed), 79] pp 202-212.

[CHUBB, 84]
D. W. J. Chubb. 1984. "Knowledge engineering problems during expert system development." ACM Simuletter vol. 15 no. 3 (July). 5-9.

[Clark & T (eds), 82]
Logic Programming. Edited by K. L. Clark and S.-A. Tarnlund. A.P.I.C. Studies in Data Processing no. 16. Academic Press. London, etc.

[CLEMA, 80]
J. K. Clema. 1980. "Managing simulation projects." In [Oren & SR (eds), 80] pp 235-241.

[CLEMENTSON, 78]
A. T. Clementson. 1978. "Extended control and simulation language." In [UKSC, 78] pp 174-179.

[COHEN & G, 84]
P. R. Cohen and T. R. Gruber. 1984. "Reasoning about uncertainty: a knowledge representation perspective." In [Fox (ed), 84] pp 25-34.

[COLMERAUER, 82]
A. Colmerauer. 1982. "PROLOG and Infinite Trees." In [Clark & T (eds), 82] pp 231-251.

[COLMERAUER, 86]
A. Colmerauer. 1986. "Theoretical Model of Prolog II." In [van Caneghem & W (eds), 86] pp 3-31.

[CROOKES, 87]
J. G. Crookes. 1987. "Generators, Generic Models and Methodology." J. Opl Res. Soc. Vol. 38, No. 8, pp. 765-768.

[CROOKES & BCP, 86]
J. G. Crookes; D. W. Balmer; S. T. Chew and R. J. Paul. 1986. "A Three-Phase Simulation System Written in Pascal." J. Opl Res. Soc. Vol. 37, No. 6, pp. 603-618.

[D'AGAPEYEFF, 84]
A. d'Agapeyeff. 1984. "Making a start: a view from British industry." In [Fox (ed), 84] pp 3-13.

[DAVIES, 78]
N. R. Davies. 1978. "Program structure and run-time efficiency in discrete event simulation" (abstract only). In [UKSC, 78] pp 172-173.

[DAVIES & D, 87]
H. Davies and R. Davies. 1987. "A Simulation Model for Planning Services for Renal Patients in Europe." J. Opl Res. Soc. vol. 38, no. 8, pp 693-700.

[DAVIS, 82]
R. E. Davis. 1982. "Runnable Specification as a Design Tool." In [Clark & T (eds), 82] pp 141-149.

[DAVIS, 86]
P. K. Davis. 1986. "Applying artificial intelligence tecniques to strategic-level gaming and simulation." In [Elzas & OZ (eds), 86] pp 315-338.

[DEUTCH & FP, 86]
T. Deutsch, I. Futo and I. Papp. 1986. "The use of TC-Prolog for medical simulation." In [Luker & A (eds), 86] pp 29-34.

[DOUKIDIS & P, 86]
G. I. Doukidis and R. J. Paul. 1986. "Experiences in automating the formulation of discrete event simulation models." In [Kerckhoffs & VZ (eds), 86] pp 79-90.

[DUDA & GH, 79]
R. Duda; J. Gaschnig and P. Hart. 1979. "Model Design in the Prospector Consultant System for Mineral Exploration." In [Michie (ed), 79] pp 153-167.

[DUNHAM & K, 81]
N. R. Dunham and A. K. Kochhar. 1981. "Interactive computer simulation for the evaluation of manufacturing planning and control strategies." In [UKSC, 81] pp 82-89.

[ELMAGHRABY & J, 85]
A. S. Elmaghraby and V. Jagannathan. 1985. "An expert system for simulationists." in [Birtwistle (ed), 85] pp 106-109.

[ELZAS, 86]
M. S. Elzas. 1986. "The kinship between artificial intelligence, modelling & simulation: An appraisal." In [Elzas & OZ (eds), 86] pp 3-13.

[ELZAS, 86]
M. S. Elzas. 1986. "The applicability of Artificial intelligence techniques to knowledge representation in modelling and simulation." In [Elzas & OZ (eds), 86] pp 19-40.

[Elzas & OZ (eds), 86]
Modelling and Simulation Methodology in the Artificial Intelligence Era. 1986. M. S. Elzas, T. I. Oren and B. P. Zeigler (editors). Amsterdam, etc: North-Holland. ISBN: 0 444 701303.

[EL SHEIKH & PHB, 87]
A. R. A. El Sheikh; R. J. Paul; A. S. Harding and W. Balmer. 1987. "A Microcomputer-Based Simulation Study of a Port." J. Opl Res. Soc. vol. 38, no. 8, pp 673-681.

[ENNALS, 86]
R. Ennals. 1986. "Teaching Logic as a Computer Language in Schools." In [van Caneghem & W (eds), 86] pp 129-144

[ERNST & N, 69]
G. W. Ernst and A. Newell. 1969. GPS: A Case study in generality and problem solving. New York, London: Academic Press. ACM Monograph Series.

[ESSAR, 84]
Expert systems. State of the Art Report. 1984. "A framework for expert systems." In [Fox (ed), 84] pp 125-133.

[ESSAR, 84]
Expert systems. State of the Art Report. 1984. "Future development: from skill to expertise." In [Fox (ed), 84] pp 135-143.

[ESSAR, 84]
Expert systems. State of the Art Report. 1984. "Claims and achievement." In [Fox (ed), 84] pp 145-155.

[ESSAR, 84]
Expert systems. State of the Art Report. 1984. "International developments." In [Fox (ed), 84] pp 157-164.

[ESSAR, 84]
Expert systems. State of the Art Report. 1984. "Research and development." In [Fox (ed), 84] pp 165-175.

[FARGUES & LDC, 86]
J. Fargues, M.-C. Landau, A. Dugourd and L. Catach. 1986. "Conceptual Graphs for semantics and knowledge processing." IBM J. Res. Develop. vol. 30, no. 1 (January):70-78.

[FARKAS & SS, 86]
Zs. Farkas, P. Szeredi and E. Santane-Toth. 1986. "LDM -- A Program Specification Support System." In [van Caneghem & W (eds), 86] pp 105-116.

[FEIGENBAUM, 79]
E. A. Feigenbaum. 1979. "Themes and Case Studies of Knowledge Engineering." In [Michie (ed), 79] pp 3-25.

[FIKES & K, 1985]
R. Fikes and T. Kehler. 1985. "The Role of Frame-Based Representation in Reasoning." Communications of the ACM. vol. 28, no. 9. 904-920.

[Findler (ed), 79]
Associative networks: Representation and use of knowledge by computers. 1979. N. V. Findler (editor). New York: Academic Press. ISBN:?????

[FISHMAN, 73]
G. S. Fishman. 1973. Concepts and Methods in Discrete Event Digital Simulation. New York, etc.: John Wiley & Sons. ISBN: 0-471-26155-6.

[FISHWICK, 88]
P. A. Fishwick. 1988. "Qualitative simulation: Fundamental concepts and issues." In [Henson (ed), 88] pp. 25-31.

[FORDYCE & NS, 86]
K. Fordyce; P. Norden and G. Sullivan. 1986. "Artificial Intelligence and the Management Science Practitioner: Expert Systems -- Getting a Handle on a Moving Target." Interfaces. 16:6 (November-December) pp 61-63.

[FORDYCE & NS, 87]
K. Fordyce, P. Norden and G. Sullivan. 1987. "Artificial intelligence and the management science practitioner: Links between Operations Research and Expert Systems." Interfaces. 17:4 (July-August) pp 34-40.

[FOX, 84]
J. Fox. 1984. "An annotated bibliography on expert systems." In [Fox_J. (ed), 84] pp 181-197.

[FOX, 84]
M. E. Fox. 1984. "Expert systems for education and training." In [Fox (ed), 84] pp 35-48.

[Fox (ed), 84]
Expert systems: State of the Art Report 12:7. Edited by J. Fox. Published by Pergamon Infotech Limited, Maidenhead, Berkshire, England. 1984. ISBN: 0 08 028 5929.

[FUCHI, 84]
K. Fuchi. 1984. "Revisiting original philosophy of fifth generation computer systems project." In [FGCS, 84] pp 1-2-5

[FUCHI, 86]
K. Fuchi. 1986. "Aiming for Knowledge Information Processing Systems." In [van Caneghem & W (eds), 86] pp 279-305.

[FUTO & G, 86]
I. Futo and T. Gergely. 1986. "Problems and advantages of simulation in Prolog." In [Elzas & OZ (eds), 86] pp 385-397.

[GAINES & S, 85]
B. R. Gaines and M. L. G. Shaw. 1985. "Expert systems and simulation." In [Birtwistle (ed), 85] pp 95-101.

[GENESERETH & G, 85]
M. R. Genesereth and M. L. Ginsberg. 1985. "Logic Programming." Communications of the ACM. vol. 28, no. 9. (September):933-941.

[GEORGE, 80]
F. H. George. 1980. Problem Solving. London: Duckworth. ISBN:

[GIANNESINI & C, 84]
F. Giannesini and J. Cohen. 1984. "Parser Generation and Grammar Manipulation Using PROLOG's Infinite Trees." J. Logic Programming 1984:3:253-265.

[GIESZL, 87]
L. R. Gieszl, 1987. "The expert system applicability question." In [Luker & B (eds), 87] pp 17-20.

[GORDON, 75]
G. Gordon. 1975. The application of GPSS V to Discrete System Simulation. Englewood Cliffs, NJ: Prentice-Hall. ISBN: 0-13-039057-7.

[GUENTHNER, 86]
F. Guenthner; H. Lehmann and W. Schonfeld. 1986. "A theory for the representation of knowledge." IBM J. Res. Develop. vol. 30, no. 1 (January):39-56.

[HANDLYKKEN & N, 81]
P. Handlykken and K. Nygaard. 1981. "The DELTA Description Language: Motivation, main concepts and experience from use." In [Hunke (ed), 81] pp 157-172.

[HANSSON & HT, 82]
A. Hansson; S. Haridi and S.-A. Tarnlund. 1982. "Properties of a Logic Programming Language." In [Clark & T (eds), 82] pp 267-280.

[HANSSON & T, 82]
A. Hansson and S.-A. Tarnlund. 1982. "Program Transformation by Data Structure Mapping." In [Clark & T (eds), 82] pp 117-122.

[HARANDI & Y, 85]
M. T. Harandi and F. H. Young. 1985. "Template based specification and design." In [SSD, 85] pp 94-97.

[HARRIS, 86]
M. R. Harris. 1986. "Methods and Models in Inference Research." DAI Research Paper no. 301. Department of Artificial Intelligence, University of Edinburgh.

[HAWKINS, 85]
R. D. Hawkins. 1985. "Artificial intelligence from the systems engineer's viewpoint." In [Holmes (ed), 85] pp 10-25.

[HAYES-ROTH, 84]
F. Hayes-Roth. 1984. "Knowledge-based expert systems -- the state of the art in the US." In [Fox (ed), 84] pp 49-62.

[HAYS-ROTH, 85]
F. Hayes-Roth. 1985. "Rule-Based Systems." Communications of the ACM vol. 28, no. 9, (September):921-932.

[HENRIKSEN, 83
J. O. Henriksen. 1983. "The integrated simulation environment (Simulation software of the 1990s)." Operations Research vol. 31 no. 6 (Nov-Dec 1983) pp 1053-1073.

[HENRIKSEN, 84]
J. O. Henriksen. 1984. "Discrete event simulation languages: Current Status and future directions." In [WSC, 84] 83-88.

[Henson (ed), 88]
Artificial Intelligence and Simulation: The Diversity of Applications. Proceedings of the SCS Multiconference, 3-5 February 1988. San Diego, California, USA. T. Henson (editor). San Diego: Society for Computer Simulation International.

[HILTON, 88]
M. L. Hilton. 1988. "A multi-level event scheduling mechanism for supporting intelligent objects." In [Hanson (ed), 88] pp. 127-130.

[HIRSCHMAN & P, 86]
L. Hirschman and K. Puder. 1986. "Restriction Grammar: A Prolog Implementation." In [van Canegham & W (eds), 86] pp 244-261.

[HMTREASURY, 85]
H M Treasury. 1985. Expert Systems Some guidelines.

[Holmes (ed), 85]
AI and Simulation. 1985. Proceedings from the Eastern Simulation Conference, March 1985, Norfolk. W. M. Holmes (editor). San Diego: Society for Computer Simulation. ISBN: 0-911801-05-7.

[HONIDEN & UK, 85]
S. Honiden, N. Uchihira and T. Kasuya. 1985. "Software Prototyping with MENDEL." In [LOGICPRO, 85] pp 108-116.

[HOWE, 79]
J. A. M. Howe. 1979. "Learning through Model-building." In [Michie (ed). 79] pp 215-225.

[Hunke (ed), 81]
Software Engineering Environments. 1981. H. Hunke (editor). Proceedings of the symposium held in Lahnstein, Federal Republic of Germany. June 16-20, 1980. Amsterdam, etc: North-Holland Publishing Company. ISBN: 0 444 86133 5.

[Huntsinger & KKV (eds), 88]
Simulation Environments and Symbol and Number Processing on Multi and Array Processors. 1988. Proceedings of the European Simulation Multiconference, June 1-3, 1988, Nice, France. R. C. Huntsinger, W. J. Karplus, E. J. Kerckhoffs and G. C. Vansteenkiste (editors). San Diego, California: Society for Computer Simulation. (also Ghent, Belgium: Society for Computer Simulation Europe). ISBN: 0-911801-39-1.

[ISHIZUKA & M, 84]
M. Ishizuka and T. Moto-oka. 1984. "Overview of expert systems in Japan." In [Fox (ed), 84] pp 63-69.

[JONES, 86]
A. W. Jones. 1986. "Possibilities for expert aids in system simulation." In [Elzas & OZ (eds), 86] pp 145-152.

[JOYCE & BW, 84]
J. Joyce, G. Birtwistle and B. Wyvill. 1984. "ANDES -- an environment for animated discrete event simulation." In [UKSC, 84] 93-101.

[KAUBISCH & PH, 76]
W. H. Kaubisch; R. H. Perrott and C. A. Hoare. 1976. "Quasiparallel Programming." Software--Practice and Experience, vol. 6, 341-356.

[Kerckhoffs & Vr (eds), 86]
AI Applied To Simulation. Proceedings of the European Conference at the University of Ghent. February 25-28, 1985, Belgium. E. J. H. Kerckhoffs, G. C. Vansteenkiste and B. P. Zeigler (editors). San Diego, California: Society for computer simulation. Simulation series vol. 18 no. 1. ISSN: 0735-9276.

[KHOSHNEVIS & C, 86]
B. Koshnevis and A.-P. Chen. 1986. "An expert simulation model builder." In [Luker & A (eds), 86] pp 129-132.

[KIMBLER & W, 88]
D. L. Kimbler and B. A. Watford. 1988. "Simulation program generators: A functional perspective." In [Henson (ed), 88] pp. 133-136.

[KITZMILLER, 88]
C. T. Kitzmiller. 1988. "Simulation and AI: Coupling symbolic and numeric computing." In [Henson (ed), 88] pp. 3-7.

[KLAHR, 84]
P. Klahr. 1984. "Artificial intelligence approaches to simulation." In [UKSC, 84] pp 87-92.

[Klahr & W (eds), 86]
Expert Systems: Techniques, Tools and Applications. 1986. P. Klahr and D. A. Waterman (editors). Reading, MA, etc.: Addison-Wesley Publishing Co.

[KLEIJNEN, 79]
J. P. C. Kleijnen. 1979. "The role of statistical methodology in simulation." In [Zeigler & EKO (eds), 79] pp 425-445.

[KORNFELD, 86]
W. A. Kornfeld. 1986. "The purpose and promise of logic programming." In [ACMCSC, 86] pp 15-17.

[KOWALSKI, 82]
R. A. Kowalski. 1982. "Logic as a Computer Language." In [Clark & T (eds), 82] pp 3-16.

[KOWALSKI, 86]
R. Kowalski. 1986. "The limitations of logic." In [ACMCSC, 86] pp 7-13.

[KRIZ & S, 80]
J. Kriz and H. Sandmayr. 1980. "Extension of Pascal by Coroutines and its Application to Quasi-parallel Programming and Simulation." Software--Practice and Experience vol. 10, 773-789.

[LANGEN, 87]
P. A. Langen. 1987. "Application of artificial intelligence techniques to simulation." In [Luker & B (eds), 87] 49-57.

[LEE, 83]
R. M. Lee. 1983. "Epistemological Aspects of Knowledge-based Decision Support Systems." In [Sol (ed), 83] pp 25-36.

[LEE & S, 85]
S. Lee and S. Sluizer. 1985. "On using executable specifications for high-level prototyping." In [SSD, 85] pp 130-134.

[LEHMANN & RS, 88]
A. Lehmann; G. Roll and H. Szczerbicka. 1988. "Application of expert systems in INT^3: An interactive, intelligent and integrated PC modelling environment." In [Henson (ed), 88] pp. 49-54.

[LEHMANN & KKS, 86]
A. Lehmann, B. Knodler, E. Kwee and H. Szczerbicka. 1986. "Dialog-oriented and knowledge-based modelling in a typical PC environment." In [Luker & A (eds), 86] pp 133-138.

[LEHMANN & KKS, 86]
A. Lehmann, B. Knodler, E. Kwee and H. Szczerbicka. 1986. "Dialog-oriented and knowledge-based modelling in a typical PC environment." In [Kerckhoffs & VZ (eds), 86] pp 91-96.

[LLOYD, 84]
J. W. Lloyd. 1984. Foundations of Logic Programming. Berlin, etc.: Springer-Verlag. ISBN: 3-540-13299-6 (also 0-387-13299-6).

[LLOYD & T, 84]
J. W. Lloyd and R. W. Topor. 1984. "Making PROLOG More Expressive." J. Logic Programming:1984:3:225-240.

[LOGICPRO, 85]
Logic Programming '85. Proceedings of the 4th Conference. E. Wada (editor). Berlin: Springer-Verlag.

[Luker & A (eds), 86]
Intelligent Simulation Environments. 1986. Proceedings of SCS Multiconference 23-25 January, 1986, San Diego, California, USA. P. A. Luker and H. H. Adelsberger (editors). San Diego: Society for Computer Simulation. Simulation Series vol. 17 no. 1. ISSN: 0735-9276.

[Luker & B (eds), 87]
Simulation and AI. Proceedings of SCS Multconference, 14-16 January 1987, San Diego, California, USA. P. A. Luker and G. Birtwistle (editors). San Diego: Society for computer simulation. Simulation Series: vol. 18 no. 3. ISSN: 0735-9276.

[MARKOWITZ, 84]
H. M. Markowitz. 1984. "Proposals for the standardization of status description." ACM Simulatter vol. 15 no. 1 (January). 37-55.

[MATHEWSON & A, 78]
S. C. Mathewson and J. A. Allen. "A commentary on the proposal for a simulation model specification and documentation language." In [UKSC, 78] pp 158-171.

[MAYER & Y, 84]
R. J. Mayer and R. E. Young. 1984. "Automation of simulation model generation from system specification." In [WSC, 84] 571-576.

[McGOWAN & FC, 85]
C. L. McGowan, M. D. Feblowitz and M. Chandrasekharan. 1985. "The metafor approach to executable specifications." In [SSD, 85] pp 163-169.

[McROBERTS & FH, 85]
M. McRoberts, M. Fox and N. Husain. 1985. "Generating model abstraction scenarios in KBS." In [Birtwistle (ed), 85] pp 29-33.

[Michie (ed), 79]
Expert systems in the micro-electronic age. Edited by D. Michie. Edinburgh University Press. Edinburgh.

[MILES, 84]
P. W. Miles. 1984. "A methodology for constructing rule-based data driven discrete event simulations." In [UKSC, 84] pp 57-67.

[MOREIRA da SILVA & B, 86]
C. Moreira da Silva and J. M. Bastos. 1986. "The use of decision mechanisms in visual simulation for flexible manufacturing systems modelling." In [Kerckhoffs & VZ (eds), 86] pp 165-170.

[MOSER, 1986]
J. G. Moser. 1986. "Integration of artificial intelligence and simulation in a comprehensive decision-support system." Simulation 47:6 (December) 223-229.

[NANCE, 84b]
R. E. Nance. 1984. "Simulation Modeling: Two Perspectives." IIE Transactions, vol. 16, no. 1 (March) p 2.

[NARAIN, 86]
S. Narain. 1986. "MYCIN: The Expert System and Its Implementation in LogLisp." In [van Caneghem & W (eds), 86] pp 161-174.

[NAYLOR, 80]
T. H. Naylor. 1980. "Third generation corporate simulation models." In [Oren & SR (eds), 80] pp 131-141.

[NEUMANN, 86]
G. Neumann. 1986. "A Prolog tutorial." In [Luker & A (eds), 86] pp 163-164.

[NEWTON & W, 80]
O. L. Newton and J. E. Weatherbee. 1980. "Guidelines for documenting simulation models: A review and procedures." In [Oren & SR (eds), 80] pp 243-258.

[NYGAARD, 86]
K. Nygaard. 1986. "Basic Concepts in Object Oriented Programming." In [OOPW, 86] pp 128-132.

[O'KEEFE, 83]
R. A. O'Keefe. 1983. "Programming Meta-Logical Operations in Prolog." DAI Working Paper no. 142. Department of Artificial Intelligence, University of Edinburgh.

[O'KEEFE, 83]
R. A. O'Keefe. 1983. "Updatable Arrays in Prolog." DAI Working Paper no. 150. Department of Artificial Intelligence, University of Edinburgh.

[O'KEEFE, 83]
R. A. O'Keefe. 1983. "Classification: a worked exercise in Prolog." DAI Working Paper no. 153. Department of Artificial Intelligence, University of Edinburgh.

[O'KEEFE, 84]
R. A. O'Keefe. 1984. "Reading Sentences in Prolog - a Worked Example." DAI Working Paper no. 159. Department of Artificial Intelligence, University of Edinburgh.

[O'KEEFE, 85]
R. M. O'Keefe. 1985. "Expert Systems and Operational Research -- Mutual Benefits." J. Opl Res. Soc. Vol. 36, No. 2, pp. 125-129.

[O'KEEFE, 86]
R. M. O'Keefe. 1986. "Advisory systems in simulation." In [Kerckhoffs & VZ (eds), 86] pp 73-78.

[O'KEEFE & BB, 86]
R. M. O'Keefe; V. Belton and T. Ball. 1986. "Experiences with Using Expert Systems in O.R.." J. Opl Res. Soc. Vol. 37, No. 7, pp. 657-668.

[O'SHEA, 79]
T. O'Shea. 1979. "Rule-based Computer Tutors." In [Michie (ed), 79] pp 226-232.

[OOPW, 86]
Proceedings of Object-oriented Programming Workshop. IBM Yorktown Heights: 9-13 June 1986. Special issue of ACM SIGPLAN Notices vol. 21 no. 10 (October 1986). ISBN: 0-89791-205-5.

[OPEN UNIV, 75]
D. Morris and L. Jones. 1975. Systems Modelling: A third level course. Milton Keynes: The Open University Press. ISBN: 0 335 060528.

[OREN, 78]
T. I. Oren. 1978. "A personal view on the future of simulation languages." In [UKSC, 78] pp 294-306.

[OREN, 79]
T. I. Oren. 1979. "Concepts for advanced computer assisted modelling." In [Zeigler & EKO (eds), 79] pp 29-55.

[OREN, 86]
T. I. Oren. 1986. "Implications of machine learning in simulation." In [Elzas & OZ (eds), 86] pp 41-57.

[OREN, 86]
T. I. Oren. 1986. "Knowledge bases for an advanced simulation environment." In [Luker & A (eds), 86] pp 16-22.

[OREN, 87]
T. I. Oren. 1987. "Aritificial intelligence and simulation: From congnitive simulation toward cognizant simulation." Simulation 48:4 (April): 129-130.

[OREN, 87]
T. I. Oren. 1987. "Quality assurance paradigms for artificial intelligence in modelling and simulation." Simulation 48:4 (April) :149-151.

[Oren & SR (eds), 80]
Simulation with Discrete Models: A State-of-the-Art View. 1980. T. I. Oren (editor-in-chief), C. M. Shub and P. F. Roth (editors). New York: Institute of Electrical and Electronics Engineers (IEEE TH0079-4).

[OREN & Z, 79]
T. I. Oren and B. P. Zeigler. 1979. "Concepts for advanced simulation methodologies." Simulation. March 1979 pp 69-82.

[OREN & Z, 87]
T. I. Oren and B. P. Zeigler. 1987. "Artificial intelligence in modelling and simulation: Directions to explore." Simulation 48:4 (April): 131-134.

[OVERSTREET & N, 85]
C. M. Overstreet and R. E. Nance. 1985. "A Specification Language to Assist in Analysis of Discrete Event Simulation Models." Communications of the ACM. vol. 28, no. 2. (February):190-201.

[OVERSTREET & N, 86]
C. M. Overstreet and R. E. Nance. 1986. "World view based discrete event model simplification." In [Elzas & OZ (eds), 86] pp 165-179.

[OXBORROW, 87]
E. A. Oxborrow. 1987. "Towards Knowledge Bases - Semantics, Rules and Object-oriented Programming." UKC Computing Laboratory Report no. 45. University of Kent at Canterbury.

[PALME, 76]
J. Palme. 1976. "Experience from the Standardization of the SIMULA Programming Language." Software--Practice and Experience, vol. 6, 405-409.

[PAULI & S, 80]
W. Pauli and M. L. Sofa. 1980. "Coroutine Behaviour and Implementation." Software--Practice and Experience, vol. 10, 189-204.

[PAUL & C, 87]
R. J. Paul and S. T. Chew. 1987. "Simulation Modelling Using an Interactive Simulation Program Generator." J. Opl Res. Soc. Vol. 38, No. 8, pp. 735-752.

[PEREIRA & W, 80]
F. Pereira and D. H. D. Warren. 1980. "Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks." Artificial Intelligence 13(1980) pp 231-278.

[PICHLER, 86]
F. R. Pichler. 1986. "Model components for symbolic processing by knowledge based systems: The STIPS framework." In [Elzas & OZ (eds), 86] pp 133-143.

[PIDD, 87]
M. Pidd. 1987. "Simulating Automated Food Plants." J. Opl Res. Soc. Vol. 38, No. 8, pp. 683-692.

[Pidd (ed), 89]
Computer Modelling for Discrete Simulation. 1989. M. Pidd (ed). Chichester, etc: John Wiley & Sons. ISBN: 0-471-92282-X.

[POLYA, 57]
G. Polya. 1957. How to solve it. Garden City, New York, USA: Doubleday & Company.

[PRERAU, 85]
D. S. Prerau. 1985. "Selection of an Appropriate Domain for an Expert System." The AI Magazine. Summer, 1985. pp 26-30.

[PRITSKER & K, 69]
A. A. B. Pritsker and P. J. Kiviat. 1969. Simulation with GASP II: A FORTRAN based simulation language. Englewood Cliffs, NJ: Prentice-Hall. ISBN:

[QUINLAN, 79]
J. R. Quinlan. 1979. "Discovering Rules by Induction from Large Collections of Examples." In [Michie (ed), 79] pp 168-201.

[RAJAGOPALAN, 86]
R. Rajagopalan. 1986. "Qualitative modelling and simulation: A survey." In [Kerckhoffs & VZ (eds), 86] pp 9-26.

[RAO & S, 88]
M. J. Rao and R. G. Sargent. 1988. "An experimental advisory system for operational validity." In [Henson (ed), 88] pp. 245-250.

[REDDY, 87]
R. Reddy. 1987. "Epistemology of knowledge based simulation." Simulation 48:4 (April):162-166.

[REICHGELT & V, 87]
H. Reichgelt and F. van Harmelen. 1987. "Building expert systems using logic and meta-level interpretation." DAI Research Paper no. 303. Department of Artificial Intelligence, University of Edinburgh.

[REILLY & JD, 85]
K. D. Reilly, M. T. Jones and P. Dey. 1985. "The simulation environment concept artificial intelligence perspectives." In [Holmes (ed), 85] pp 29-34.

[REITMAN, 80]
J. Reitman. 1980. "Interactive graphics and discrete event simulation languages." In [Oren & SR (eds), 80] 125-127.

[Reitman (ed), 84]
Artificial Intelligence Applications for Business. 1984. Proceedings of 1983 NYU Symposium at the New York University Graduate School of Business Administration 18-20 May 1983. Norwood, New Jersey: Ablex Publishing Corporation. ISBN: 0-89391-220-4.

[ROACH & F, 86]
J. W. Roach and T. D. Fuller. 1986. "A Prolog Simulation of Migration Dicision-Making in a Less Developed Country." In [van Caneghem & W (eds), 86] pp 145-151.

[ROBERTSON & BUM, 87]
D. Robertson; Alan Bundy; M. Uschold and R. Muetzelfeldt. 1987. "Synthesis of simulation models from high level specifications." DAI research paper no. 313. Department of Artificial Intelligence, University of Edinburgh.

[ROBERTS & E, 80]
S. D. Roberts. 1980. "Simulation and health care delivery." In [Oren & SR (eds), 80] pp 143-164.

[ROBINSON, 79]
J. A. Robinson. 1979. "The Logical Basis of Programming by Assertion and Query." In [Michie (ed), 79] pp 105-111.

[ROMAN & A, 86]
E. G. Roman and S. V. Ahamed. 1986. "A model-based expert system for decision support in negotiating." In [Elzas & OZ (eds), 86] pp 339-352.

[ROSS, 85]
P. Ross. 1985. Expert Systmes Course. MSc/PhD - 1985/86. DAI Teaching Paper no. 1. Department of Artificial Intelligence, University of Edinburgh.

[ROZENBLIT & Z, 86]
J. W. Rozenblit and B. P. Zeigler. 1986. "Entity-based structures for modelling and experimental frame construction." In [Elzas & OZ (eds), 86] pp 79-100.

[SAGE & T, 80]
A. P. Sage and M. A. H. Thissen. 1980. "Methodologies for systems modeling." In [Oren & SR (eds), 80] pp 45-62.

[SAUER & M, 79]
C. H. Sauer and E. A. Macnair. 1979. "Queueing Network Software for Systems Modelling." Software--Practice and Experience, vol. 9, 369-380.

[SCHMIDT, 84]
J. W. Schmidt, 1984. "Introduction to simulation." In [WSC, 84] pp 65-73.

[SCHRIBER, 74]
T. J. Schriber. 1974. Simulation using GPSS. New York, etc.: John Wiley & Sons. ISBN: 0-471-76310-1.

[SLOMAN, 79]
A. Sloman. 1979. "Epistemology and Artificial Intelligence." In [Michie (ed), 79] pp 235-241.

[SMITH & M, 88]
H. R. Smith and K. McVicar. 1988. "Knowledge-based simulation with frameworks." In [Henson (ed), 88] pp. 72-77.

[SMITH, 88]
P. Smith. 1988. Expert System Development in Prolog and Turbo-Prolog™. Wilmslow, Cheshire: SIGMA Press. ISBN: 1-85058-064-2. Also New York: Halsted Press ISBN: 0-470-20911-9.

[Sol (ed), 83]
Processes and Tools for Decision Support. Proceedings of the Joint IFIP WG8.3/IIASA Working Conference. Schloss Laxenburg, Austria: 19-21 July 1982. H. G. Sol (editor). Amsterdam, etc.: North-Holland Publishing Company. ISBN: 0 444 86569 1.

[SSD, 85]
Proceedings of Third International Workshop on Software Specification and Design. London: 26-27 August 1985. Washington DC, USA: IEEE Computer Society Press. ISBN: 0-8186-0638-X.

[STROM, 86]
R. Strom. 1986. "A comparison of the object-oriented and process paradigms." In [OOPW, 86] pp 88-97.

[SUZUKI, 86]
N. Suzuki. 1986. "Experience with Specification and Verification of a Complex Computer Using Concurrent Prolog." In [van Canaghem & W (eds), 86] pp 188-207.

[SYKES & CY, 88]
D. J. Sykes; J. K. Cochran and H. H. Young. 1988. "Development of diagnostic expert systems using qualitative simulation." In [Hanson (ed), 88] pp. 32-38.

[TAYLOR & H, 88]
R. P. Taylor and R. D. Hurrion. 1988. Support environments for discrete event simulation experimentation. In [Huntsinger & KKV (eds), 88] pp 242-248.

[TERWILLIGER & C, 85]
R. B. Terwilliger and R. H. Campbell. 1985. "PLEASE: Predicate Logic based Executable SpEcifications." Report no. UIUCDCS-R-85-1231. Department of Computer Science, University of Illinois at Urbana-Champaign

[TORN, 85]
A. A. Torn. 1985. "Simulation nets, a simulation modelling and validation tool." Simulation 45:2 (August). 71-75.

[UKSC, 78]
Proceedings of the 1978 UKSC Conference on Computer Simulation. 4-6 April 1978. Chester, England. Surrey: IPC Science and Technology Press. ISBN: 0 902852 92 2.

[UKSC, 81]
Proceedings of the 1981 UKSC Conference on Computer Simulation. 13-15 May 1981, Harrogate, England. Surrey: Westbury House (the books division of IPC Science and Technology Press). ISBN: 0 86103 051 6.

[UKSC, 84]
Proceedings of the 1984 UKSC Conference on Computer Simulation. 12-14 September, 1984. University of Bath, England. D. J. Murray-Smith (ed). London, etc.: Butterworths. ISBN 0-408-01504-7.

[van Caneghem & W (eds), 86]
Logic Programming and its Applications. 1986. M. van Caneghem and D. H. D. Warren (editors). Norwood, New Jersey: Ablex Publishing Corporation. Ablex series in Artificial Intelligence. ISBN: 0-89391-232-8.

[WALKER, 84]
A. Walker. 1984. "Data bases, expert systems, and Prolog." In [Reitman (ed), 84] pp 87-109.

[WALKER, 86]
A. Walker. 1986. "Syllog: An Approach to Prolog for Nonprogrammers." In [van Caneghem & W (eds), 86] pp 32-49.

[WALKER, 86]
A. Walker. 1986. "Knowledge systems: Principles and practice." IBM J. Res. Develop. vol. 30, no. 1 (January):2-13.

[WARREN, 79]
D. Warren. 1979. "PROLOG on the DECsystem-10." In [Michie (ed), 79] pp 112-121.

[WATERMAN, 86]
D. A. Waterman. 1986. D. A. Waterman. 1986. A Guide to Expert Systems. Reading, MA, etc.: Addison-Wesley Publishing Co. The Teknowledge series in Knowledge Engineering. ISBN: 0-201-08313-2

[WINSTON, 84]
P. H. Winston. 1984. Artificial Intelligence. 2nd Edition. Reading, MA, etc.: Addison-Wesley Publishing Company. ISBN: 0-201-08259-4.

[WSC, 84]
Proceedings of 1984 Winter Simulation Conference. 28-30 November, 1984, Dallas, Texas. S. Sheppard, U. W. Pooch and C. D. Pegden (editors). San Diego, California: Society for Computer Simulation.

[YAMAMOTO, 86]
Y. Yamamoto. 1986. "Graphic interfaces for modelling systems." In [Elzas & OZ (eds), 86] pp 399-403.

[YOSHIDA & KS, 85]
H. Yoshida, H. Kato and M. Sugimoto. 1985. "Retrieval of software module functions using first-order predicate logical formulae." In [LOGICPRO, 85] pp 117-127.

[YOUNG, 84]
R. M. Young. 1984. "Human interface aspects of expert systems." In [Fox (ed), 84] pp 101-111.

[ZAHEDI, 87]
F. Zahedi. 1987. "Artificial Intelligence and the Management Science Practitioner: The Economics of Expert Systems and the Contribution of MS/OR." Interfaces. 17:5 (September-October) pp 72-81.

[ZAJICEK, 86]
W. A. Zajicek. 1986. "Transforming a discrete-event system into a logic programming formalism." In [Elzas & Oz, 86] pp 181-192.

[ZEIGLER, 84]
B. P. Zeigler. 1984. "Systems hierarchy as a basis for simulation model description." ACM Simuletter vol. 15 no. 1 (January). 8-13.

[ZEIGLER, 86]
B. P. Zeigler. 1986. "System Knowledge: A definition and its implications." In [Elzas & Oz (eds), 86] pp 15-17.

[ZEIGLER & D.-W, 86]
B. P. Zeigler and L. De Wael. 1986. "Towards a knowledge-based implementation of multifaceted modelling methodology." In [Kerckhoffs & Vz (eds), 86] 42-51.

[Zeigler & EKO (eds), 79]
Methodology in Systems Modelling and Simulation. 1979. B. P. Zeigler (editor-in-chief), M. S. Elzas, G. J. Klir and T. I. Oren (editors). Amsterdam, etc.: North-Holland. ISBN: 0 444 853405.

THE BRITISH LIBRARY DOCUMENT SUPPLY CENTRE

TITLE Towards A Knowledge-Based Discrete Simulation Modelling Environment Using Prolog

AUTHOR Ali Ahmad

INSTITUTION and DATE University of Warwick, June 1989

Attention is drawn to the fact that the copyright of this thesis rests with its author.

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no information derived from it may be published without the author's prior written consent.

THE BRITISH LIBRARY
DOCUMENT SUPPLY CENTRE
Boston Spa, Wetherby
West Yorkshire
United Kingdom

cms 1 2 3 4 5 6

REDUCTION X 20

CIMELIA 3