

Polka : A Parlog Object Oriented Language.

Andrew Davison

A thesis submitted to the University of London
for the degree of Doctor of Philosophy

Department of Computing
Imperial College of Science, Technology and Medicine

September 1989

Abstract

Programming language design is an evolutionary process. For instance, the concurrent Logic Programming (LP) paradigm and its associated languages could not have been developed without the work on LP and on models of concurrency during the last decade.

This evolution is clearly visible in Polka which is, first and foremost, a language derived from the fusion of the concurrent LP and Object Oriented Programming (OOP) paradigms. The result is more than a sum of its parts since many of the inadequacies of one programming model are compensated for by features of the other.

One of the other design goals behind Polka is its utilisation as a 'higher level' concurrent LP language. This is possible because Polka contains abstractions which are well suited to programming typical kinds of concurrent LP problems.

The last design goal is the incorporation of meta level programming concepts from LP into Polka, which greatly increase the functionality of the class mechanism.

This thesis presents the background to, and motivation for, the design of Polka, and describes the language. The utility of Polka is demonstrated through numerous small examples, and three larger applications concerned with graphics, blackboard problem solving, and simulation. Also, an operational semantics is developed for a subset of the language, called kernel Polka, and a simple and efficient implementation is derived from it.

Acknowledgments

I am, of course, greatly indebted to my supervisor Keith Clark, who has encouraged my work, and dispensed invaluable advice. In addition, his comments on this thesis have been consistently useful and enlightening. Most of all, I thank him for his friendship.

I must also thank a number of people who read earlier drafts of the thesis, and supplied extremely informative comments. They are (in no particular order) : Alastair Burt, Melissa Lam, Colin Atkinson, Ian Foster, Gul Agha, and Ken Kahn.

The working environment has been excellent, and this must be attributed to my fellow members in the Parlog Group, some of whom have now left for pastures new. Apart from those already mentioned, I must thank (again in no particular order) : Steve Gregory, Graem Ringwood, Jim Crammond, Matthew Huntbach, Reem Bahgat, David Gilbert, Bob Kemp, Fukumi Kozato, Priscila Lima, and numerous Parlog Group secretaries.

The Parlog Group is only a small part of the larger Logic Programming section, and friends and colleagues in the section have made my time at Imperial very happy. There being too many people to mention, I simply thank Bob Kowalski and Cheryl Anderson.

My research has been supported by the Science and Engineering Research Council. Other recent work on Polka at Imperial College has been funded by various research grants.

Contents

Part 1. Introductory Material.	16
1. Introduction.	17
1.1 The Context of the Research : Language Design.	17
1.2 Contributions of the Thesis.	18
1.3 Assumptions about the Readership.	19
1.4 Preview of the Contents.	19
2. Logic Programming Overview.	21
2.1 Logic Programming.	21
2.2 Concurrent Logic Programming.	22
2.3 Parlog.	23
2.3.1 The Process Interpretation in Parlog.	25
2.3.2 The Control Metacall.	29
2.3.3 Semantics of Parlog.	29
3. Design Issues for LP-based Object Oriented Languages.	31
3.1 Motivation for combining OOP and LP.	32
3.1.1 The reasons for adding OOP to LP.	32
3.1.2 The reasons for adding LP to OOP.	33
3.1.3 Applications.	33
3.2 Object and Class Representation.	34
3.2.1 The ADT View of Objects and Classes.	35
3.2.2 The Theory View of Objects and Classes.	37
3.2.3 The Process View of Objects and Classes.	37
3.2.3.1 The Committed Choice Process View.	37
3.2.3.2 The Backtracking Process View.	40
3.3 Message Passing.	41
3.3.1 Message Passing in the ADT View.	42
3.3.2 Message Passing in the Theory View.	44
3.3.3 Message Passing in the Process View.	44
3.3.3.1 Message Passing in the CCPV.	44
3.3.3.2 Message Passing in the Backtracking Process View.	47
3.3.3.2.1 Wait Declarations.	47
3.3.3.2.2 Events.	48

3.3.3.2.3 D-clauses.	49
3.4 State.	51
3.4.1 State in the ADT View.	51
3.4.2 State in the Theory View.	52
3.4.3 State in the Process View.	52
3.4.3.1 State in the Committed Choice Process View.	53
3.4.3.2 State in the Backtracking Process View.	53
3.5 Inheritance.	54
3.5.1 Inheritance in the ADT View.	55
3.5.2 Inheritance in the Theory View.	59
3.5.3 Inheritance in the Process View.	59
3.5.3.1 Inheritance in the Committed Choice Process View.	59
3.5.3.2 Inheritance in the Backtracking Process View.	61
3.6 Other Paradigms.	62
3.6.1 Other Paradigms in the ADT View.	62
3.6.2 Other Paradigms in the Theory View.	63
3.6.3 Other Paradigms in the Process View.	63
3.7 Summary of the Views.	63
3.7.1 Summary of the ADT View.	63
3.7.2 Summary of the Theory View.	64
3.7.3 Summary of the Process View.	64
3.7.3.1 Summary of the Committed Choice Process View.	64
3.7.3.2 Summary of the Backtracking Process View.	65
 4. An Overview of Polka.	 66
4.1 Polka Features.	66
4.2 The Advantages of Polka.	68
4.3 The Disadvantages of Polka.	68
 Part 2. Language and Applications.	 71
 5. The Language.	 72
5.1 Class Structure.	72
5.1.1 A 'receiver' Class.	73
5.1.2 A 'filestore' Class.	74
5.2 Default Polka Clauses.	77

5.3 Variables.	78
5.4 The initial Section.	79
5.5 Clauses.	80
5.6 Input Messages.	81
5.7 Becomes.	83
5.8 Output Messages.	84
5.9 I_am.	85
5.10 The code Section.	87
5.11 Inheritance.	88
5.12 Self Communication.	99
5.13 'Higher level' Parlog.	101
 6. Meta Level Programming.	 104
6.1 What is Meta Level Programming?	104
6.2 The Uses of Meta Level Programming.	107
6.3 Program Transformation.	108
6.4 Load Balancing.	112
6.5 Parametrised Classes.	114
6.6 Extensible Classes.	117
6.6.1 Computer.	119
6.6.2 Square.	119
6.6.3 Circle.	120
6.6.4 Window.	121
6.6.5 Support Classes.	122
 7. Three Examples.	 127
7.1 Graphics.	127
7.1.1 Overview of the Classes.	127
7.1.2 Screen.	128
7.1.3 Shape.	131
7.1.4 Turtle.	134
7.1.5 The Parlog Approach.	137
7.1.6 Possible Changes to the Programs.	140
7.2 Blackboard Systems.	142
7.2.1 Background to Blackboard Systems.	142
7.2.1.1 The Blackboard.	143
7.2.1.2 Knowledge Sources (KSs).	143

7.2.1.3	Control.	143
7.2.1.4	Parallelism Issues.	144
7.2.1.5	Manipulation Issues.	145
7.2.2	Polka Blackboard Systems.	146
7.2.2.1	Polka Blackboards.	146
7.2.2.2	Polka KSs.	150
7.2.3	A Version of Hearsay II.	155
7.2.4	Summary of the Blackboard Example.	157
7.3	Simulation.	158
7.3.1	Overview of Simulation.	158
7.3.1.1	Simulation in Logic Programming Languages.	158
7.3.2	Distributed Time.	160
7.3.2.1	Distributed Time in Polka.	161
7.3.3	Polka Simulation Classes.	164
7.3.3.1	Random Number Generators.	165
7.3.3.1.1	Uniform.	166
7.3.3.1.2	Randint.	166
7.3.3.2	Resources.	167
7.3.3.3	Report Generation.	170
7.3.4	Two Simulation Examples.	171
7.3.4.1	A Car Wash.	171
7.3.4.2	An Information Retrieval System.	173
7.3.5	Summary of the Simulation Example.	175
8.	Concurrency.	176
8.1	What is Concurrency?	176
8.2	Synchronisation.	177
8.3	Rendezvous-like Synchronisation.	180
8.4	Liveness.	189
8.5	Arrival, Selection and Completion.	191
8.6	Separation of Synchronisation from Functionality.	192
8.6.1	The Readers-Writers Problem.	193
8.6.2	Readers Priority.	195
8.7	The Actor Model.	197

Part 3. Semantics and Implementation. 203

9. Kernel Polka. 204

9.1 Default Clauses.	204
9.2 Variables.	205
9.3 Input Messages.	205
9.4 Clauses.	205
9.5 Becomes.	205
9.6 Output Messages.	206
9.7 Inheritance.	207
9.8 The Parlog Component.	207

10. Semantics. 208

10.1 Transition Systems.	209
10.2 The Configuration.	210
10.2.1 Tuple Goals.	210
10.2.2 Tuple State.	210
10.2.3 Tuple Flags.	211
10.2.4 Tuple Program.	211
10.3 A Labelled Transition System for Kernel Polka.	212
10.4 The Creation and Linking of the Configuration.	213
10.4.1 The Creation of the 'i' Tuple Set.	213
10.4.2 Linking the 'i' Tuples internally.	214
10.4.3 Linking the 'i' Tuples externally.	215
10.4.4 Executing the 'i' Tuple Set.	218
10.5 Executing a single 'i' Tuple.	219
10.5.1 Goals Transitions.	219
10.5.2 Goal Transitions.	222
10.5.3 Processing Input Messages.	223
10.5.4 Processing Kernel Polka Operations.	226
10.5.4.1 The Becomes Operation.	226
10.5.4.2 The Send Operation.	226
10.5.4.3 The Self Operation.	228
10.5.4.4 The Super Operation.	228
10.5.4.5 The I _{am} Operation.	229
10.5.5 The Code Section.	230
10.6 Term Classes.	231

11. Implementation.	232
11.1 The 'top' Predicate.	232
11.2 The 'o' Predicate.	234
11.2.1 Bottom / no Inheritance.	235
11.2.2 Inside / no Inheritance.	236
11.2.3 Bottom / Inheritance.	236
11.2.4 Inside / Inheritance.	237
11.3 The Initial Section.	238
11.4 The 'i' Predicate.	238
11.4.1 Variables.	240
11.4.2 Input Messages.	240
11.4.3 Separators.	241
11.4.4 Becomes.	241
11.4.5 Sends.	242
11.4.6 Inheritance.	243
11.4.7 Self Communication.	244
11.4.8 I_am.	244
11.4.9 The Code Section.	244
11.5 Higher Level Parlog.	246
11.6 Term Classes.	248
 Part 4. Conclusions.	 249
 12. Conclusions.	 250
12.1 Design Aims.	250
12.2 A Measure of Success.	250
12.3 Futher Research.	253
 Appendix. BNF Description of Polka.	 255
 References.	 260

List of Figures

2.1	Two 'number' processes sending to 'receiver'.	28
3.1	A file store.	39
3.2	Inheritance graph for 'edit' window.	54
3.3	Inheritance graph with two occurrences of 'cursor'.	56
3.4	Inheritance graph with one occurrence of 'cursor'.	57
3.5	'database' inherits 'filestore'.	60
5.1	Informal BNF for a Polka class.	72
5.2	Inheritance graph for 'a'.	89
5.3	Inheritance graph for an instance of 'b'.	89
5.4	Inheritance graph which inherits 't' twice.	90
5.5	Instance inheritance tree for 'q'.	90
5.6	Instance inheritance tree for 'nasty_clock'.	97
5.7	Invocation structure for 'nasty_clock'.	98
5.8	Instance inheritance tree for 'a'.	99
6.1	Initial inheritance tree for 'window'.	118
6.2	New inheritance tree for 'window'.	118
6.3	Data flow in the 'tobj_plus' clause.	123
7.1	Inheritance graph for the graphics example.	127
7.2	Device dependent graph.	128
7.3	WAX on the graph.	131
7.4	Shapes on the graph.	134
7.5	Compass points.	135
7.6	A turtle trail.	136
7.7	Further turtle movements.	137
7.8	A KS with control.	144
7.9	A KS with a trigger.	145
7.10	A Polka KS.	150
7.11	The four components of 'syllable'.	153
7.12	'syllable' as one entity.	154
7.13	Screen dump of execution of the Hearsay implementation.	156

7.14	A typical Polka object.	161
7.15	A distributed time object.	161
7.16	A distributed time object with named streams.	163
7.17	An invocation of 'entity1'.	164
7.18	Screen dump of execution of the car wash.	172
7.19	Screen dump of execution of the information system.	174
8.1	Three stage communication protocol.	188
10.1	Instance inheritance tree for 'a'.	208

List of Tables

10.1	Definition for 'smatch'.	255
------	--------------------------	-----

List of Programs

2.1	Concatenate two lists.	21
2.2	Merge two lists.	24
2.3	Output a list of numbers.	25
2.4	Output a stream of messages.	26
2.5	Receive messages.	27
3.1	A file store.	38
3.2	Broadcast a message.	43
5.1	A 'receiver' class.	73
5.2	A 'filestore' class.	75
5.3	A 'tax' class.	79
5.4	A 'clock' class.	82
5.5	An 'unemployed' class.	85
5.6	An 'employed' class.	86
5.7	A 'pulse' class.	87
5.8	A 'clock' class with an alarm.	91
5.9	A random number generator.	92
5.10	A 'nasty_clock' class.	92
5.11	A 'response' predicate.	95
5.12	A 'responseL' predicate.	97
5.13	A database class.	99
5.14	A database class with suspendible messages.	100
5.15	A 'merge' class.	102
6.1	An evolving factorial	109
6.2	A factorial distributor.	113
6.3	A type store (only 'complex' shown).	115
6.4	A parametrised stack class.	116
6.5	A 'computer' term class.	119
6.6	A 'square' term class.	119
6.7	A 'circle' term class.	120
6.8	A 'window' term class.	121
6.9	A 'tobj_plus' class.	123
6.10	A 'to_inherits' class.	124

7.1	A 'screen' class.	129
7.2	A 'shape' class.	132
7.3	A 'turtle' class.	135
7.4	Predicates for 'screen'.	137
7.5	Predicates for 'shape'.	139
7.6	A 'bd' class.	146
7.7	An 'oboard' class.	147
7.8	A 'joint' predicate.	149
7.9	A 'ks' class.	151
7.10	A 'syl_control' class.	152
7.11	A 'syllable' predicate.	153
7.12	An 'entity1' predicate.	163
7.13	A 'uniform' class.	166
7.14	A 'randint' class.	166
7.15	A 'resource' class.	168
8.1	A 'database' class which uses message peeking.	177
8.2	A 'screen_display' class.	178
8.3	A 'taxi' class.	178
8.4	A 'window' class.	179
8.5	The producer/consumer problem in (pidgin) Ada.	180
8.6	A 'count' predicate.	182
8.7	A 'wait_accept' predicate.	182
8.8	The producer/consumer problem in Polka.	183
8.9	An or- parallel buffer controller.	184
8.10	A 'fail_accept' predicate.	185
8.11	A buffer controller which can sleep.	185
8.12	A producer of 'msg' messages.	185
8.13	Predicates for 'time_out'.	186
8.14	A 'wait_accept' predicate which deals with 'msg' messages.	186
8.15	A 'check_time' predicate which signals confirmation.	187
8.16	A 'confirmed' predicate.	187
8.17	A 'wait_test' predicate.	188
8.18	An 'access_type' predicate.	188
8.19	A 'delete' predicate.	189
8.20	A 'resource_syn' class.	193

8.21	An 'any_reads' predicate.	195
8.22	An actor-style 'factorial' class.	199
8.23	A 'factorial' actor.	200
8.24	A 'factorial' predicate.	200
10.1	A 'add_inherits' predicate.	214
10.2	Predicates for 'iconnect'.	215
10.3	A 'sconnect' predicate.	215
10.4	An 'io_interface' predicate.	216
10.5	An 'input' predicate.	217
10.6	An 'output' predicate.	217
10.7	A 'checkend' predicate.	220
10.8	Predicates for 'do_becomes'.	221
10.9	A 'match' predicate.	224
10.10	A 'matchL' predicate.	225
10.11	A 'messageL' predicate.	225
11.1	The two top 'filestore' predicates.	233
11.2	The two top 'foo' predicates.	233
11.3	A 'bottom/no inheritance' 'o_p' predicate.	235
11.4	An 'inside/no inheritance' 'o_p' predicate.	236
11.5	A 'bottom/inheritance' 'o_p' predicate.	236
11.6	An 'inside/inheritance' 'o_p' predicate.	237
11.7	The resulting 'i_init_filestore' predicate.	238
11.8	A 'filestore' class.	239
11.9	The resulting 'i_filestore' predicate.	239
11.10	Augmented predicates in a 'foo' class.	245
11.11	Rewritten 'dub' and 'single' predicates.	245
11.12	And- parallel 'dub' predicate.	246
11.13	Rewritten and- parallel 'dub' predicate.	246
11.14	A 'merge' class.	247
11.15	The resulting 'i_merge' predicate.	247
11.16	The top 'merge' predicate.	247

Part 1. Introductory Material.

Chapter 1. Introduction.

1.1. The Context of the Research : Language Design.

A programming language should offer a set of constructs which taken together naturally aid programming in a particular paradigm. This definition depends crucially on the meaning of ‘naturally’, which covers a number of important properties [Horowitz 1984], including :

- a well-defined syntactic and semantic description
- orthogonality
- machine independence
- generality
- uniformity
- extensibility
- ability to use language subsets
- consistency with commonly used notation

A few of these ideas require a brief explanation. Orthogonality means that the features should be separately understandable and free from interaction when combined. Generality is the idea that all features should be composed of different aspects of a few basic concepts. Uniformity means that similar things should have similar meanings, and a language is extensible if additional data abstractions and operations can be defined.

Also important in language design, but not directly related to the underlying programming paradigm are :

- reliability
- fast translation
- efficient object code

The meaning of ‘paradigm’ is also difficult to define, but is related to the abstract model of a problem upon which the user bases his solution. Examples of successful programming paradigms include functional programming, logic programming (LP), and object oriented programming (OOP). The success of a paradigm is probably rooted in its restricted view of the world, which for instance permits only relations in LP, or objects in OOP. It may also be successful because it

has a strong mathematical basis, as with LP, or more practical benefits in the case of OOP, which emphasises decomposition, encapsulation, and reuse.

The restrictions imposed by a paradigm mean that it can never be suitable for all problem domains, and it is this insufficiency which drives the generation of new programming models, and new computer languages. In fact, two evolutionary paths can be discerned : a paradigm can be replaced by a more expressive model, or a language based on the paradigm can be extended. A good example of this is LP, which was the starting point for the development of the concurrent LP paradigm, and also for constraint LP. In addition, pure LP languages have been superseded by more expressive languages which contain extra control primitives and constructs for such things as asserting and retracting facts.

1.2. Contributions of the Thesis.

The main original contributions are the following :

1. A new language called Polka which can naturally express a larger class of problems than most languages, simply because it combines two extremely powerful paradigms : concurrent LP and OOP.
2. The ability to utilise Polka as a 'higher level' Parlog by virtue of the way that its features support common Parlog programming styles in a less verbose and more intuitive manner.
3. The development of object oriented meta level constructs for the language, based on ideas from LP.
4. The investigation of application areas which benefit from the concurrent LP and OOP features found only in Polka.
5. The development of an operational semantics for the language which maps onto an efficient and easily understandable implementation.

1.3. Assumptions about the Readership.

This thesis assumes that the reader is familiar with the basic ideas of LP, and also knows something of Prolog. However, detailed knowledge of these subjects is not required, and the necessary extra material is summarised along with the topics of concurrent LP and Parlog. In addition, the essential concepts of OOP will be explained.

1.4. Preview of the Contents.

This thesis is subdivided into 4 parts :

Part 1 (chapters 1 to 4) cover introductory material related to LP and OOP.

Chapter 2 introduces LP, concurrent LP, and Parlog.

Chapter 3 examines the language design issues related to the combination of LP and OOP, and includes a discussion of OOP principles.

Chapter 4 gives a brief summary of Polka and outlines its advantages and disadvantages.

Part 2 (chapters 5 to 8) looks at the language and some of its applications.

Chapter 5 details the features of Polka which include all of the typical OOP and concurrent LP constructs, as well as new ones which have resulted from the fusion of the two paradigms.

The meta level programming parts of Polka are examined in chapter 6.

Three large Polka programming examples are presented in chapter 7. The first application shows how complex classes for dealing with graphics can be built by reusing simpler classes. The second investigates how blackboard systems can be represented, while the last example describes a set of general purpose simulation classes.

Chapter 8 looks at the various forms of concurrency and synchronisation in Polka, culminating in an examination of how rendezvous-like constructs, like those in Ada, can be programmed. In addition, the separation of synchronisation and functionality of a class is considered and some techniques for achieving this are presented. Finally, the actor model is discussed, because of its close relationship with the concurrency model in Polka.

Part 3 (chapters 9 to 11) deal with the semantics and implementation of the language.

As a preliminary task, chapter 9 presents kernel Polka which is a simplified version of the language. This simplification is achieved by isolating a core set of Polka features from which all other constructs can be defined.

Chapter 10 describes an operational semantics for the kernel language, based on Plotkin's state transition system formalism.

Chapter 11 explains the implementation of the kernel language, and shows how it is based on the semantics of the preceding chapter.

The final part of the thesis consists of chapter 12, which presents the conclusions and the proposals for further research.

A BNF description of Polka is given in the appendix.

Chapter 2. Logic Programming Overview.

This chapter contains an overview of Logic Programming (LP), concurrent LP, and the concurrent LP language Parlog. However, it is not intended to constitute an introduction to any of these topics, and suitable reading material will be referred to during the discussion.

2.1. Logic Programming.

For the purposes of this review logic programs are sets of Horn clause implications of the form :

$$P \text{ if } Q_1, Q_2, \dots, Q_n, \quad n \geq 0$$

where $P, Q_1, \dots, Q_n$ are all atomic formulae. All variables appearing in the implication are implicitly universally quantified. As a statement it says that for all values of the variables, P is true if each $Q_1, \dots, Q_n$ is true. Hence, the *declarative semantics* of a logic program for some predicate is that it is a set of statements about the predicate. For instance, the following clauses of program 2.1 define the 'concatenate' predicate :

```
concatenate( [], L, L ).
concatenate( [ X | L1 ], L2, [ X | L3 ] ) if
    concatenate( L1, L2, L3 ).
```

Program 2.1 : Concatenate two lists.

The clauses may be read as :

The empty list ($[]$) concatenated with any list ' L ' is simply ' L '. A non-empty list consisting of ' X ' followed by remaining elements ' $L1$ ' concatenated with list ' $L2$ ' is the list consisting of ' X ' followed by remaining elements ' $L3$ ' where ' $L1$ ' concatenated with ' $L2$ ' is ' $L3$ '.

Under Kowalski's procedural interpretation [Kowalski 1979] of Horn clause logic each Horn clause implication is also a procedure for computing instances of the relation. The head of a clause is a procedure entry point, a goal is a procedure call, and a procedure is a set of clauses with the same head predicate. For example, the clauses for 'concatenate' can be considered to be a procedure for concatenating the elements of two given lists (amongst other uses). The procedure has two entry points corresponding to whether or not the first of the two input lists is empty. One of the clauses makes a recursive call to the same procedure.

A *procedural semantics* is given by a top-down resolution inference system. This defines a non-deterministic machine programmed by Horn clauses. Control

information, such as the ordering of the procedure calls may be used to guide the inference machine. However, the control information does not affect the declarative semantics. Every predicate instance computed by the machine is a logical consequence of the statements of the program.

The control provided by Prolog [Sterling and Shapiro 1987] means that a goal unifies with the first possible clause appearing in the program text. The matching clause instance is then activated by executing in turn, from left to right, each of the goals in its body (if any). If at any time the system fails to match for a goal, it backtracks. This causes the rejection of the most recently activated clause, undoing any substitutions. Then it reconsiders the original goal which activated the rejected clause, and tries to find a subsequent clause which also matches the goal.

2.2. Concurrent Logic Programming.

A program in a concurrent LP language is a collection of Horn clauses, but a computation involves the parallel evaluation of a goal with respect to these clauses. It is this parallel evaluation, and in particular the and- and or- parallelism exhibited, that differentiates a concurrent logic program from one written in a sequential LP language such as Prolog [Clark and Gregory 1986].

Shapiro distinguishes between concurrent and sequential LP from another point of view : that of transformational and reactive systems [Shapiro 1989]. A transformational system, which typifies a sequential logic program, receives an input at the beginning of its operation and yields an output at its end. A concurrent logic program is more like a reactive system whose purpose is not necessarily to obtain a final result but rather to maintain some interaction with its environment. Common transformational systems include programs that encode numerical methods, while typical reactive systems are operating systems and database management systems.

Another way of distinguishing concurrent and sequential LP languages is based on their underlying interpretation. Both paradigms allow programs to have a declarative and procedural interpretation. In concurrent LP, a third interpretation is possible where a conjunction of goals can be regarded as a system of concurrent processes. Each process is an executing recursive predicate, and the processes communicate by partially instantiating shared variables.

Several concurrent LP languages have been defined, and a survey and comparison of some of them is given in [Shapiro 1989].

2.3. Parlog.

Parlog will be discussed because it was chosen as concurrent LP element of Polka. The reasons for this include its use of deep guards, the simplicity of its unification mechanism, and the control metacall, all of which will be described below.

Parlog differs from Prolog in three important respects : concurrent evaluation, don't-care non-determinism, and its (optional) use of mode declarations to specify communication constraints on shared variables.

A Parlog clause is a Horn clause optionally augmented with a commit operator, ':', which is used to separate the right hand side of the clause into a conjunction of guard conditions and a conjunction of body conditions :

$$r(t_1, \dots, t_k) \leftarrow \langle \text{guard conditions} \rangle : \langle \text{body conditions} \rangle$$

where $t_1, \dots, t_k$ are argument terms. Both the $\langle \text{guard conditions} \rangle$ and the $\langle \text{body conditions} \rangle$ are conjunctions of predicate calls. There are two types of conjunction : the parallel ',' ($C1, C2$) in which the conjuncts $C1$ and $C2$ are evaluated concurrently and the sequential '&' ($C1 \& C2$) where $C2$ will only be evaluated when $C1$ has successfully terminated. The use of the parallel conjunction allows Parlog to display and- parallelism in the execution of its goals.

During the evaluation of a call $r(t_1', \dots, t_k')$, all of the clauses for the predicate 'r' can be searched in parallel for a candidate clause. The above clause is a candidate clause if the head $r(t_1, \dots, t_k)$ matches the call $r(t_1', \dots, t_k')$ and the guard succeeds. It is a non-candidate if the match or the guard fails. If all clauses are non-candidates the call fails, otherwise one of the candidate clauses is selected and the call is reduced to the substitution instance of the body of that clause. There is no backtracking on the choice of candidate clause since we 'don't care' which candidate clause is selected. In practice, the language implementation dictates that the first one (chronologically) to be found is chosen.

The absence of backtracking in Parlog is compensated for by guard tests which can be used to guarantee that the correct clause is chosen. This idea has been more formally stated in Gregory's *sufficient guards law* [Gregory 1987], which identifies the situations under which a Parlog computation returns a solution to a query, if a solution exists. It states that Parlog predicates should be written so that guards in each clause guarantee that if a clause is selected to reduce a goal then either :

- a solution to the call can be computed using this clause, or
- no solution can be found using any other clause

The search for a candidate clause can be controlled by using either the parallel clause search operator '.' or the sequential clause search operator ';' between clauses. For instance, if a predicate is defined by the clauses :

```
Clause1 .
Clause2 ;
Clause3.
```

Then Clause3 will not be tried for candidacy until both Clause1 and Clause2 have been found to be non-candidate clauses. The presence of the parallel search operator allows Parlog to display or- parallelism during the search for a candidate clause.

A Parlog predicate definition may have a mode declaration associated with it, which states whether each argument is input (?) or output (^). For example, the predicate 'merge' has the mode (?, ?, ^) to merge the input lists 'X' and 'Y' to create the output list 'Z' :

```
mode merge(?, ?, ^).
merge( [ El | X ], Y, [ El | Z ] ) <-      % output 'El' element from 'X'
    merge( X, Y, Z ).
merge( X, [ El | Y ], [ El | Z ] ) <-      % output 'El' element from 'Y'
    merge( X, Y, Z ).
merge( [], Y, Y ).                        % 'X' is []
merge( X, [], X ).                        % 'Y' is []
```

Program 2.2 : Merge two lists.

Syntax similar to that of Edinburgh Prolog [Clocksin and Mellish 1981] is used for Parlog program examples throughout.

The first clause places an element from the 'X' list onto the 'Z' list, while the second does a similar task for an element from the 'Y' list. The parallel search operator between these two clauses means that a non-deterministic choice will be made between them if there is a 'merge' call with elements on both 'X' and 'Y'. The third clause allows 'Y' to become the remaining part of the 'Z' list by unification since the end of the 'X' list has been reached (it is []). Similarly, 'X' becomes the rest of the 'Z' list when the 'Y' list is exhausted.

A query for 'merge' might be :

```
<- merge( [ 1, 2, 3 | X ], [ a, b, c | Y ], Z ).
```

which may produce the binding :

```
Z = [ 1, a, b, 2, 3, c | _ ]
```

The '_' at the end of the list is an uninstantiated variable which indicates that the list is only partially instantiated because both the 'X' and 'Y' lists are not fully bound. This will mean that the evaluation of 'merge' will deadlock and then terminate after producing the 'Z' binding shown.

Non-variable terms that appear in input argument positions in the head of a clause can only be used for input matching. If an argument of the call is not sufficiently instantiated for an input match to succeed, the attempt to use the clause suspends until some other process further instantiates the input argument of the call. For example, the first clause for 'merge' has [El | X] in its first input argument position. Until the call has a list or partial list structure of the form [El | X] in the first argument position the first clause is suspended. If all clauses for a call are suspended, the call suspends but a candidate clause can be selected even if there are other suspended clauses.

Parlog utilises unification for output bindings which means that bindings to output variables will be unified with any values in those argument positions in the goal. This is of practical use for manipulating the component parts of the output, as can be seen in the following query :

```
<- merge( [ 1, 2, 3 ], [ a, b, c ], [ Head | Tail ] ) .
```

The [Head | Tail] structure will unify with the output list and bind 'Head' to the first value in the list, while 'Tail' will become equal to its tail.

More details on Parlog can be found in [Clark and Gregory 1986] and [Conlon 1989].

2.3.1. The Process Interpretation in Parlog.

As mentioned above, a distinguishing feature of concurrent LP, and so of Parlog, is that programs can be interpreted as processes. To illustrate this, we shall examine a program which incrementally binds an output list using a starting value supplied by the user. 'number' is defined in program 2.3.

```
mode number( ?, ^ ) .
number( No, [ No | Rest ] ) <-                               % output a number
    No1 is No + 1 ,
    number( No1, Rest ) .
```

Program 2.3 : Output a list of numbers

The 'No' number is used as the value for the head of the output list. Also, it is incremented using the Parlog expression evaluator 'is', but because of the single assignment rule of logic programming, the value must be placed in a new variable 'No1'. This new variable then becomes the first argument of the recursive call to 'number' which means that it will be treated as 'No' when the new 'number' goal is executed. A query :

```
<- number( 0, L ) .
```

will produce a non-terminating list :

```
L = [ 0, 1, 2, ...
```

This is due to the fact that there is no way of terminating the 'number' query.

A process view of 'number' would interpret 'No' as a state variable, the output list as an output stream, and each number of the output list as a message. Also, since 'number' does not wait for any kind of reply to its messages, it can be thought of as carrying out asynchronous message passing. In addition, 'number' does not know where its messages are going, since 'L' is acting only as a output stream, and not as the address of another process.

'number' can be brought under control by including a variable in each output message which is used as a reply mechanism by the receiver of the message. This technique is known as 'back communication'.

'number' becomes :

```
mode number( ?, ?, ^ ) .
number( No, stop, [] ) .                               % terminate
number( No, continue, [ message( No, Reply ) | Rest ] ) <- % output a message
    No1 is No + 1 ,
    number( No1, Reply, Rest ) .
```

Program 2.4 : Output a stream of messages.

'number' now has an extra input argument which is used to hold the 'Reply' value from the last output message. Also, the messages are more complicated, having the form :

```
message( < number >, < reply > )
```

The binding for 'Reply' is stored in the recursive call to 'number', and its value is checked by the two clauses. If it is 'stop' then the output stream is closed (set to []) and 'number' terminates by simply not recursing. If 'Reply' is 'continue' then the process generates another message.

The use of 'Reply' illustrates two interesting points : the first is that the two tests of 'Reply' are done in parallel because of the parallel search operator '.' between the clauses. The other point is that the process will suspend until the 'Reply' variable is bound, which means that it is in synchronous communication with the receiver of its messages.

A query is now :

```
<- number( 0, continue, L ) .
```

which will produce :

```
L = [ message( 0, Reply ) | _ ]
```

If 'Reply' is not bound, then 'number' will deadlock at this point.

A receiver process for 'number' would be useful, since it could encode the mechanism for receiving messages, printing out their 'No' contents, and binding their 'Reply' variables. It could also contain some code for terminating 'number' when it had produced enough numbers. A possible definition is given in program 2.5.

```
mode receiver( ? ) .
receiver( [] ) .                                % terminate
receiver( [ message( 200, Reply ) | Rest ] ) <-
    Reply = stop ,                               % send 'stop' to sender
    receiver( Rest ) ;
receiver( [ message( No, Reply ) | Rest ] ) <-
    Reply = continue ,                          % send 'continue' to sender
    write( No ) ,                               % print number
    receiver( Rest ) .
```

Program 2.5 : Receive messages.

'receiver' has one input argument which can be thought of as its input stream, The first clause handles the termination of its input when the stream is []. The second clause deals with the case when the number in the message is 200 and 'receiver' binds 'Reply' to 'stop'. This will be communicated back to the 'number' process which sent the message and will cause it to terminate. The final clause, which can be thought of as the default case, binds the 'Reply' variable to 'continue' and writes the number on the screen using the Parlog 'write' primitive. The final clause is made the default by being preceded by a ';' which means that it is not tried for candidacy until the first two clauses have been rejected.

An important feature of 'receiver' is that it does not know who is communicating with it, but only that it can receive a certain kind of message. Also, it is unaware of how its 'Reply' binding will be used.

A query using 'number' and 'receiver' could be :

```
<- number( 0, continue, L ) , receiver( L ) .
```

This creates two concurrent processes which communicate by partially instantiating the shared logical variable 'L'. In process terms, 'L' is a message stream from 'number' to 'receiver' which also allows replies to be sent back along it.

The result of the query will be the printing of a sequence of numbers on the screen (0 to 199) and then the termination of both processes.

Since streams are actually logical variables, this enables them to be manipulated by predicates, or even be treated as arguments of messages. The full consequences of this will be described in subsequent chapters, but the following simple example shows how 'merge' can be used to link the output streams from two 'number' processes into a single 'receiver' :

```
<- number( 0, continue, L1 ) ,           % first process
   number( 150, continue, L2 ) ,        % second process
   merge( L1, L2, Input ) ,             % merge the two streams
   receiver( Input ) .
```

Graphically, the following process network is created :

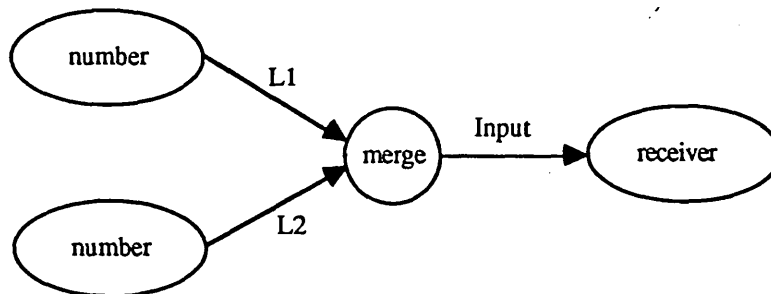


Figure 2.1 : Two 'number' processes sending to 'receiver'.

The ovals (and circle) are processes while the arrows are message streams. Note that 'merge' can be treated as a process which has two input streams and a single output stream.

The result of this query will be the printing of two sequences of numbers (0 to 199 and 150 to 199) which will be arbitrarily merged. It is likely that the second 'number' process will terminate first because it has less numbers to output, but this depends on how fairly 'merge' combines the two message streams. An interesting property of this network is that the termination of one of the 'number' processes will

not affect the communication between the other 'number' and 'receiver'. It is only when the last 'number' process terminates that the input to 'receiver' is closed and it terminates.

2.3.2. The Control Metacall.

The control metacall enables additional monitoring and control of the evaluation of a goal. An outline definition of it is :

```
mode call( ?, ^, ? ) .
call( Goal, Status, Control ) .
```

The first argument is a Parlog goal, the second is an output stream used for status messages, and the third is an input stream for control messages. The successful evaluation of a goal will bind the status stream to *succeeded*, while the failure of the goal will not cause the metacall to fail, but instead bind the stream to *failed*. This allows the metacall to protect its enclosing program from the possible failure of the goal. Other possible status stream messages include *suspend*, *continue*, and *stopped* which are issued in direct response to *suspend*, *continue*, and *stop* messages being sent into the metacall along the 'Control' stream. The *suspend* and *continue* messages are useful for delaying the execution of a goal, while *stop* can be used to terminate the evaluation.

The control metacall can represent such things as negation by failure, or-parallel execution in terms of and- parallel processes, and the sequential conjunction. An important class of programs which are easy to write are UNIX-like shells [Clark and Gregory 1984].

2.3.3. Semantics of Parlog.

Until recently, only operational semantics have been proposed for Parlog [Gregory 1987] but a number of approaches for representing its denotational semantics have now appeared [Richard and Rizk 1988; Murakami 1988; Boer 1989]. Both sorts of semantics are more complex than those for sequential LP languages such as Prolog, because of the problems of data flow dependencies between processes, and also the increased significance of non-termination. In particular, non-termination means that it is not adequate to use a proof of a goal statement from a program as the definition of a successful computation. An alternative method [Shapiro 1989] is to use

the partial computation of a program as the proof of a (conditional) statement which corresponds to the unproved computation. However, the essential properties of Parlog's proof procedure are still like those of Prolog : it is correct but incomplete, and suffers from problems of divergence.

Chapter 3. Design Issues for LP-based Object Oriented Languages.

A large number of languages have been proposed which combine OOP and LP, and the intention of this chapter is look at the design issues behind these languages. In the course of this discussion, only the basic principles of OOP will be examined, but there are many detailed introductions to OOP [Cox 1986; Stefik and Bobrow 1986; Thomas 1989; Wegner 1989].

In order to include some interesting languages, the usual meaning of OOP is relaxed so that any language which offers some form of encapsulated entity, and communicates with others via message passing is considered to be an OOP language. However, most emphasis will be placed on languages which contain other OOP features, such as inheritance, and self communication.

The six design issues discussed are :

- Motivation for combining OOP and LP.
- Object and Class Representation.
- Message Passing.
- State.
- Inheritance.
- Other Paradigms.

Motivation was chosen as a design issue because of its affect on the type of language produced, while object and class representation, message passing, state, and inheritance are examined because of their central roles in OOP. All LP-based object oriented languages combine the LP and OOP paradigms, but a few also include other paradigms which affect the overall language design. For this reason, other paradigms are included as the sixth design issue

As a further structuring device, each design issue will be classified into four implementation approaches :

- the abstract data type (ADT) view
- the theory view
- the committed choice process view
- the backtracking process view

The ADT view is based on extensions to conventional Prolog, while the theory view uses meta level programming techniques. The committed choice process view (which Polka uses) exploits concurrent LP languages as its LP part, and the backtracking

process view extends and- parallel versions of Prolog. These four categories were motivated by similar ones proposed in various papers [Kowalski 1986; Hogger and Kowalski 1987; Kwok 1988].

After the design issues have been examined, the various advantages and disadvantages of the four views will be summarised.

When references are cited in the text to support a statement, not all the relevant languages will necessarily be included, but only those which highlight the point in a particularly clear manner.

3.1. Motivation for combining OOP and LP.

The reasons can be separated into two categories : the reasons for adding OOP to LP and, conversely, the reasons for adding LP to OOP.

3.1.1. The reasons for adding OOP to LP.

The main advantage of OOP for LP is the introduction of the OOP paradigm as a guiding principle for writing programs. This offers a simple but powerful model for representing programs as computational entities which communicate with each other via message passing. In addition, the use of inheritance increases the reusability of programs and makes rapid prototyping of classes easier. Also, message passing allows each class to have a well defined interface, and polymorphism means that the interface is flexible.

The OOP features in a language make programs shorter, more understandable, and therefore reduces the number of errors. In particular, languages based on concurrent LP, such as Polka, benefit a great deal from the OOP notation for encapsulation, state changing, stream manipulation, creation, initialisation, and termination.

The OOP constructs will remove one of the weaknesses of LP : its lack of structure, especially for 'programming in the large', because of its reliance on the predicate as its main structuring mechanism. A module mechanism could be added instead but OOP offers far more than just encapsulation. At the very least, a module system must specify how communication is to take place between modules, and how modules can be reused in different applications.

The deciding factor for using OOP techniques is that they form part of a single paradigm which views a problem as a system of objects communicating by message

passing. This abstraction, used on the right sort of problem, allows it to be decomposed very easily. Also important is the recognition that a class can be specialised from another, by using inheritance. However, perhaps the main difference between conventional modules and OOP is that a class is a collection of operations and *state*, and so allows a programmer to deal with state changes within the object oriented formalism.

3.1.2. The reasons for adding LP to OOP.

Most OOP languages do not have formal semantics, although semantics have been developed for simpler languages, such as those based on actors, and these can be viewed as object oriented [Agha 1986]. Therefore, valid semantic insights can be gained by representing an OOP language as a form of LP language [Goguen and Meseguer 1986; McCabe 1987a].

Another reason for using LP is that it is especially suited for defining relationships between entities [Newton and Watkins 1988]. This is useful because conventional OOP is particularly poor for specifying relationships between classes, except those based on inheritance.

The don't know non-determinism of LP is useful for dealing with search problems, and the logical variable can be used for such things as assignment, testing, data access, data construction, and various other forms of parameter passing.

The key advantage of concurrent LP for OOP is the availability of simple ways to specify concurrency, synchronisation, and state change. Concurrent LP also offers committed choice non-determinism which may, in certain situations, be more suited to programming simulations and computer systems than don't know non-determinism. This is because in such systems it should not be possible to undo an action once taken, except by explicitly doing so, which is not the case with the backtracking mechanism. Guards are also useful, since they allow messages to be tested in arbitrarily complex ways before they are accepted.

3.1.3. Applications.

A measure of the utility of a language, and indeed of a programming paradigm, is whether worthwhile programs can be written using it. Using such a measure, LP-based object oriented languages have proved very successful, as shown below.

Anjewierden has described an OOP virtual machine for window environments, which is linked to the language via a small set of predefined commands [Anjewierden 1986]. High level user interfaces can be built by using these commands, and by utilising objects to represent the various entities on the screen in a machine independent form. This approach has been used, in an expanded form, in the ProWindows product [Quintus 1988].

McCabe has investigated object oriented graphics by representing simple pictures, such as lines and circles, using objects [McCabe 1987b]. These can be grouped together to form more complex, aggregate pictures, and geometric and non-geometric operations can be applied to them via messages. This technique, called data driven programming, has been used for such diverse applications as differentiation and meta interpreters.

ESP has been used to code the SIMPOS operating system for the PSI Machine [Chikayama 1984]. Other interesting applications written in this language, including a task modelling package for office support [Sato and Matsumoto 1986] and a parser [Miyoshi and Furukawa 1987]. In fact, several languages have been used for parsing programs [Zaniolo 1984; Ishikawa and Tokoro 1986].

A number of applications have been built using Polka, including a blackboard problem solver [Davison 1987b], simulation tools [Davison 1988a], a budget management system [Davison 1989a], and several programming environment utilities [Cowan 1988]. The first two of these will be described in chapter 7, along with a graphics example.

Some other interesting OOP applications include a system configuration language, and a query-the-user expert system [Mello and Natali 1988], while Coscia discusses debuggers and programming with uncertainty [Coscia et al. 1988]. Simulation examples are also popular [Futo and Szeridi 1984 ; Bancelhon 1986]. Fukunaga and Hirose describe a colour graphics oriented user interface, and also a expert system for annotating assembler programs [Fukunaga and Hirose 1986].

3.2. Object and Class Representation.

An object is an instance of a class, and a class is an encapsulating entity or template which normally consists of a set of variables for holding state information and a set of operations. These operations are sometimes called methods or clauses.

The encapsulation mechanism means that the internal workings of a class can be ignored when considering what it is supposed to do, and instead it can be

characterised by the set of messages which it will accept. In fact, many OOP languages do not enforce encapsulation, and it is common to allow parametrised classes.

Although each object has its own problem solving capabilities, it usually still communicates with other objects to deal with larger problems. This leads naturally to a view of programming where each object represents a real world entity, and it is probably for this reason that object oriented systems are so easy to visualise.

The decomposition of a problem into objects means the emphasis is placed on structuring the data in the problem rather than specifying the control. It also encourages programming by experimentation, and fast prototyping.

3.2.1. The ADT View of Objects and Classes.

The languages which subscribe to the ADT view combine OOP and LP by equating a class with a set of Horn clauses, which represent its methods and its state. This permits a class to be viewed as an ADT, which in the OOP context is a behaviour specification or template that can be used to generate object instances having that behaviour.

A major benefit of this view is that it allows a large body of work on the semantics of types to be applied to OOP [Cardelli and Wegner 1985; Goguen and Meseguer 1986; Danforth and Tomlinson 1988]. It may also help to guide equivalence preserving program transformations for optimisations, and aid debugging and verification.

Typically, the clauses of a class are grouped together using a label, which is used as its name. A syntax for this might be :

```
class_label( Parameters ) [ Horn clauses ] .
```

An object instance of such a class may use one of its parameters as a unique 'Id' argument so that objects can be differentiated.

The parameters of the label can be used by the Horn clauses, so specialising them for a particular invocation. An interesting example of this is when a parameter is a label, and is used to invoke another class inside the class.

As described, this approach does not allow classes to be sent messages due to the fact that a class is simply a template from which object instances are created. One way round this is to use 'assert' and 'retract' to save information about an object during its evaluation, which is then used by class methods. This idea appears in ESP where a class can have both class and instance methods and slots for class and

instance values [Chikayama 1984]. A new instance is produced by creating a new set of slots and then applying the instance methods to them. It is also possible for the class methods to use the instance slots to answer messages. In this approach, the class / instance division is only syntactic and there is no real distinction between the two at the Prolog level.

The simplest compilation strategy is to convert a label into a predicate, and make its clauses arguments of that predicate [McCabe 1987a]. Thus :

$$l [x \text{ if } y]$$

becomes :

$$l(x) \text{ if } l(y).$$

This will be known as the *label/predicate* translation technique in later sections.

A valid alternative is to map the label of a class onto a new argument for each of its clauses [Zaniolo 1984; Bancilhon 1986]. Thus :

$$l [x \text{ if } y]$$

becomes :

$$x(l) \text{ if } y(l).$$

This will be known as the *label/argument* translation technique in later sections.

Most implementation work has been done at the compilation level, although there are benefits to be gained by extending the underlying Prolog system to directly support classes and objects [McCabe 1987a; Conery 1988]. These benefits include better compilation because local and global variables can be distinguished, and because most computation is local to a class. Also, method indexing can be optimised since local variables can be ignored.

Another way of representing a class is by using the module system of the underlying Prolog language [Elshiewy 1988; Trenouth 1988]. This requires that module names can be passed as arguments of predicates, and it is also useful if a module can be parametrised, although import lists can be used instead.

Most languages do not deal with OOP solely by compilation to the underlying Prolog, but also execute some part of the user level language directly [Zaniolo 1984; Fukunaga and Hirose 1986; Koseki 1987]. A common example of this is the direct manipulation of the inheritance information for an object at run time, which is discussed further in section 3.5.1.

3.2.2. The Theory View of Objects and Classes.

The run time interpretation of some part of the user level language mentioned at the end of the last section can be taken to its logical conclusion, and an interpreter can be used to execute an entire object, or set of objects [Bowen 1985; Honiden et al. 1985; Black and Manley 1987; Mello and Natali 1988]. In this situation, classes can be thought of as theories consisting of sentences expressed in formal logic. These are executed by some form of meta interpreter which is like the 'demo' predicate found in LP [Kowalski 1979; Bowen and Kowalski 1982]. This enables a class to be manipulated as a first order citizen of the language, which allows it to be altered easily and to be included in Prolog relations as an argument. Coscia observed that the inefficiency of interpretation is removed by partially evaluating the class with respect to its interpreter, which can easily produce an order of magnitude increase in speed [Coscia et al. 1988]. The drawback of this is that the flexibility of manipulating theories is lost, and so selective partial evaluation is proposed.

The power of the theory view is only limited by the expressibility of the meta interpreter utilised. For instance, this approach was used to add OOP to a concurrent LP language in Mandala, which combines the flexibility of theories with the concurrency present in the committed choice process view discussed below [Furukawa et al. 1984]. The theory view is also behind the term class mechanism in Polka, described in chapter 6.

3.2.3. The Process View of Objects and Classes.

The process view is predominately used by concurrent LP-based OOP languages (such as Polka), but a few languages based on and- parallel versions of Prolog also exist. These different views will be distinguished by being called the committed choice and backtracking process views respectively. The names highlight the distinction between how the two kinds of process search their solution space.

3.2.3.1. The Committed Choice Process View of Objects and Classes.

The paper by Shapiro and Takeuchi was the first to investigate the connections between concurrent LP and OOP [Shapiro and Takeuchi 1983]. Their model can be summarised by the following points :

1. An object can be represented as a process which calls itself recursively and holds its internal state in unshared arguments.

2. Objects communicate with each other by instantiating shared variables.
3. An object becomes active when it receives a message, otherwise it is suspended.
4. An object instance is created by process reduction.
5. A response to a message can be achieved by binding a shared variable in the message.

They also showed how it was possible to implement inheritance by passing unrecognised messages along a message stream to another process, but this will be examined in more detail in section 3.5.3.1. Also, the full repercussions of dealing with state in this approach will be considered in section 3.4.3.1.

Shapiro and Takeuchi's model allows a typical Parlog program (and indeed any concurrent LP program) to be viewed as a template, or class. This will be illustrated by looking at a small program called 'filestore'. The basic idea behind the program is that the file store will be able to store 'file' information and retrieve it when prompted. It will also be able to ignore incorrect requests, and report errors.

```

mode filestore( ?, ?, ^ ) .
filestore( [], Files, [] ) .                                % termination

filestore( [ add( Key, File ) | Input ], Files, Error ) <-  % add a file
    filestore( Input, [ file( Key, File ) | Files ], Error ) .

filestore( [ extract( Key, File, Ok ) | Input ], Files, Error ) <- % extract a file
    extract_file( Key, F, Files, NewFiles ) :
        Ok = done ,
        File = F ,
        filestore( Input, NewFiles, Error ) ;

filestore( [ extract( Key, File, Ok ) | Input ], Files, Error ) <- % error during extraction
    Ok = no_entry_for( Key ) ,
    filestore( Input, Files, Error ) ;

filestore( [ Msg | Input ], Files, [ Msg | Error ] ) <- % output message on
    filestore( Input, Files, Error ) .                  % the error stream

mode extract_file( ?, ^, ?, ^ ) .
% only mode given for brevity

```

Program 3.1 : A file store.

In OOP terminology, an invocation of 'filestore' is an object which has one input message stream, a 'Files' state variable for its 'file' terms, and an output stream for its error messages. It can understand two types of message :

add(Key, File)

and `extract( Key, File, Ok )`

In addition, the object will terminate when its input stream is closed and will issue error messages in response to all other kinds of messages.

An 'add' message is dealt with by the second clause which changes the 'Files' state, not by destructive assignment, but by argument replacement when 'filestore' recursively calls itself.

The third and fourth clauses deal with an 'extract' message, and use a guard to do a test before the message is accepted or rejected. The call to 'extract_file' attempts to remove a 'file' term with a 'Key' argument from 'Files'. It will either succeed and return a value for 'File' and bind 'NewFiles' to the new file list, or will fail and so reject the 'extract' message. If rejected, the message is handled by the fourth clause which is a simple error handling clause for 'extract' messages. In either case, the 'Ok' variable is bound to a term which indicates the outcome of the computation and illustrates the way that a response can be communicated back to the sender of the message.

The fifth clause is a general purpose error handling clause and outputs any unrecognised messages on its 'Error' stream.

A crucial attribute of such objects is their fine grained parallelism, which means that an object may be able to process several messages simultaneously, and also carry out more than one action at a time.

A call to 'filestore', with 'Files' original empty, would be :

```
<- filestore( Input, [], Error ),
   Input = [ add( 1, a1 ), add( 2, a2 ), extract( 1, File1, Ok1 ),
             extract( 3, File2, Ok2 ), hello | _ ] .
```

The five messages sent along 'Input' would produce the following bindings :

```
File1 = a1,           Ok1 = done
File2 unbound,       Ok2 = no_entry_for( 3 )
Error = [ hello | _ ]
```

Note that 'File2' is unbound because no entry was found for it.

The object can be graphically represented by figure 3.1.



Figure 3.1 : A file store.

The circle is an object, while the arrows are message streams.

Shapiro and Takeuchi's work did not deal with some of the fundamental issues involved in OOP, such as encapsulation, multiple inheritance, self communication, the accessing of inherited state variables and clauses, and the dynamic manipulation of objects and classes. Also, they did not propose any syntactic support for OOP. As a consequence, the concurrent LP-based OOP languages developed since have tended to investigate different solutions to these problems.

The user level syntax for these languages has been quite varied but two distinct approaches seem possible. Either the language can allow a mixture of objects and predicates, as in Polka and Ohki's language [Ohki et al. 1988], or only objects are allowed, as in Vulcan [Kahn et al. 1987] and A'UM [Yoshida and Chikayama 1987]. This is something like the distinction between the conventional OOP languages Simula67 [Birtwistle et al. 1973] and Smalltalk80 [Goldberg et al. 1983]. Simula combines the class concept with Algol60-like declarations and procedures, while Smalltalk treats everything as an object.

The use of objects and predicates by Polka allow predicates like 'extract_file' to be used inside an object, and for state to be manipulated as terms. This is more expressive than the 'objects only' approach but means that two paradigms are visible at the user level. Also, 'objects only' languages have simpler semantics since only objects must be explained, but this does not necessarily mean that program understandability is simplified. This is because programs which would normally be written as functions or relations need to be recoded in an object and message passing style.

3.2.3.2. The Backtracking Process View of Objects and Classes.

Most of the languages of this type use and-parallel models which equate a set of goals with a system of concurrently executing objects [Naish 1985; Honiden et al. 1985; Elshiewy 1988]. However, the internal execution of an object will normally be sequential, although this is not the case in the language developed by Porto which can specify the parallel execution of internal goals [Porto 1983].

Each object has extra primitives to enable it to synchronise and communicate with other objects. The details of these will be discussed in section 3.3.3.2 but are usually additions to recursive Prolog predicates which represent objects in an analogous way to the Shapiro and Takeuchi model.

One important problem with and- parallel objects based on sequential LP is what to do when backtracking occurs. Backtracking in OOP is related to three kinds of actions : state changes, object creation and message sends, and problems arise out of how to undo these things, if at all, when backtracking causes the computation to roll back. The simplest solution is to leave this up to the programmer, who must then undo state changes, kill objects and send 'undo'-type messages [Chusho and Haga 1986]. A more satisfactory approach, taken in T-Prolog, is to automatically backtrack an object which is in communication with a failed object [Futo and Szeridi 1984]. The definition of communication is quite complicated because T-Prolog objects also communicate via shared data in a database. Since the backtracking of one object may cause many others to backtrack, the efficiency of this approach is quite crucial, and a form of intelligent backtracking is used. A similar technique appears in Delta-Prolog, but there is no shared database and the messages between objects must be fully bound [Pereira and Nasr 1984]. Karam also allows backtracking between and- parallel objects and explains it as a form of backtrackable rendezvous [Karam 1988]. On the other hand, backtracking is not allowed between objects in the work of Mello and Natali [Mello and Natali 1988].

3.3. Message Passing.

Message passing is the way in which one object communicates with another. It can be viewed as a form of procedure call but instead of calling a procedure to perform the operation on an object, the object is sent a message. Message passing is usually either synchronous or asynchronous. Synchronous communication means that the sender of a message waits until the receiver has finished processing the message before it continues. Asynchronous communication allows the sender to continue its execution while the transmitted message is being dealt with by the receiver.

An important related concept is polymorphism which, in the OOP context, is the ability for a message to be understood by many different classes. Alternatively, it can be thought of as a way of overloading the meaning of a message. This is possible because classes can be written which accept the same kinds of messages, but respond in different ways.

Message polymorphism can be increased if dynamic binding is used in the system. This means that an object is selected to handle a message as late as possible during the processing of the message. Message polymorphism is higher because there is nothing to stop new objects being added to the system at run time, and so altering

how a message is handled. Systems which implement dynamic binding cleverly, such as Smalltalk, show that there is very little slow down caused by the run time mapping of a message to a method in an object, while the gains in flexibility are very great.

3.3.1. Message Passing in the ADT View.

As explained in section 3.1, this approach represents a class as a predicate, and so it is natural to treat a message as a goal. The sending of a message to a class instance can be represented by the following syntax :

```
class_label :: Message
```

This would be translated into the Prolog goal :

```
class_label( Message )
```

when the label/predicate translation strategy is being used. When the label/argument translation strategy is utilised, the message will become :

```
Message( class_label )
```

This would seem to require the use of a meta call in the cases when 'Message' is unbound, but axiom schema can be used instead [Warren 1982]. A simple example is :

```
$call( message1( c1 ) ) if message1( c1 ).  
$call( message2( c1 ) ) if message2( c1 ).
```

'\$call' will be used instead of a meta call, and can execute either the method for 'message1' or 'message2'. The same technique can be applied in the first translation method when a class label is unbound at execution time. Thus, in either translation it is relatively straight forward to send an unbound message to an object, or an ordinary message to an unspecified object. Combined with Prolog's non-determinism, this enables all methods of an object to be accessed, and for a message to be broadcast to all objects. This level of polymorphism is seen as one of the powerful features of using LP with OOP [Gullichsen 1985; Gallaire 1986; Conery 1988].

The message/goal equivalence also enables message passing to be included inside predicates, which means many forms of message transmission can be encoded, including different kinds of broadcasting [McCabe 1987a]. For instance, the transmission of a message to a list of objects might be represented using the predicate in program 3.2.

```

and_cast( [], Msg ).
and_cast( [ Label | List ], Msg ) if
    Label :: Msg ,
    and_cast( List, Msg ).

```

Program 3.2 : Broadcast a message.

A call might be :

```
<- and_cast( [ box( 1 ), circle( 2 ) ], draw ).
```

A new inference rule, called the *object clause*, has been proposed by Conery to help communication between predicates and objects [Conery 1987]. A typical object clause looks like :

```
g1, g2 if c1, c2 .
```

This can be rephrased as two Horn clauses :

```

g1 if c1, c2 .
and
g2 if c1, c2 .

```

During execution, two goals are selected from the goal list and simultaneously unified with the head of the object clause. This allows predicates and objects to communicate because one of the goals is supplied by a predicate which acts as a message, while the object is represented by the other goal. The benefits of this approach can be seen in an example by Conery, which compares a expression evaluator written in conventional Prolog with one which uses a stack object [Conery 1988]. In the object version, the Prolog part of the program sends messages to the stack by using a unique stack object label. This replaces the passing of parameters in the Prolog version, which are used to represent the object's state. The object version is clearer since fewer parameters are present, and it is safer because the stack data structure is accessible only by message passing.

One of the problems with message passing is how it interacts with backtracking. One solution is not to send a message until the clause the send is in has succeeded [Chusho and Haga 1986]. This is not entirely satisfactory since the message may itself fail, which should cause the clause to fail. Another possibility is to have different forms of message passing which can report errors instead of failing [Fukunaga and Hirose 1986].

In general, message passing is synchronous, since each message is represented as a goal which must be solved before the main computation can continue. However, variables in messages need not be used until required and can be

shared between objects. This use of logical variables can also be seen as a potential bug because it allows any object which has a copy of such a variable to affect another object.

3.3.2. Message Passing in the Theory View.

In languages which use the theory view, an object can pass a message up to the meta level to be processed by using a reflection mechanism [Kowalski 1979]. Then it is possible to redirect it to any of the other theories which may be currently executing, or to process the message at the meta level. The message may be synchronous if only one 'demo' can execute at once, but if and- parallel execution is possible then message passing between theories can be asynchronous.

Another approach to communication is taken by Honiden, who uses temporal logic and petri net information at the meta level to define how theories can communicate [Honiden et al. 1985].

3.3.3. Message Passing in the Process View.

The two process views differ quite significantly in how they permit objects to communicate, due to the different ways that objects in the two approaches execute internally. In particular, the backtracking mechanism in the backtracking process view means that synchronisation and communication is more complex than in the committed choice process view.

3.3.3.1. Message Passing in the Committed Choice Process View.

Message passing in these languages is based on the idea of partially instantiating lists, as illustrated by the 'filestore' program in section 3.2.3.1. At the user level this mechanism may be hidden, although it is useful to be able to directly manipulate a message stream as a list. A common example of this is when messages must be removed from the stream in an unusual order. In Polka, this is easily achieved by allowing a predicate to search through the stream as if it was a list.

Stream communication need not be the only form of message passing because the underlying use of shared logical variables for communication is much more flexible. One possible alternative is to use channels, which enables two independent objects to send messages along the same channel without specifying an order of transmission [Tribble et al. 1987].

Whatever the form of communication, an object does not know where the destination of its message, since the logical variable is acting only as a communication medium. Also, the object is in a similar position when it receives a message, because the sender is unknown. This means that message passing can be highly polymorphic, since the communication medium can be manipulated at run time as a logical variable. For instance, a message can be duplicated and placed onto several streams, or output and input streams can be linked and decoupled dynamically. If necessary all of this can be hidden from the objects involved.

This flexibility relies on streams being visible entities in the language, but the alternative 'stream hiding' philosophy is found in the work by Ohki [Ohki et al. 1988]. The language supports a 'send' primitive which any object can use to send messages to any other object :

`send( < name of the object> , <message> )`

The advantage of such a construct is that there is no need to manipulate streams between objects, but the disadvantage is that there is a requirement to have a global data structure, which maps names of objects to their streams. In addition, there must be a process which accepts all the messages generated by the use of the 'send' primitive and puts those messages onto the right streams. This process may be the cause of a bottleneck in the system.

Languages based on the Concurrent Prolog family can use the read-only (RO) variable which is more powerful than synchronisation primitives tied to predicate definitions, such as modes in Parlog [Shapiro 1983]. A RO variable is an occurrence of a variable marked by a '?' annotation, and if unification attempts to bind a RO occurrence of a variable, then the offending call suspends. It is reactivated when the variable is instantiated via some non-RO occurrence. One important application of a RO variable is as a form of dynamic constraint because it can unify with an ordinary variable at run time and make it read-only. Little work has been done on the use of RO variables in OOP, but they are useful for the protection of input streams, and the generation of unique and unforgeable tokens [Miller et al. 1987]. Such tokens could be used to protect and identify messages sent between objects. More recently, further synchronisation constructs based on variable annotations have been proposed for versions of Concurrent Prolog [Shapiro 1989]. These include atomic test-and-set and atomic output primitives which are likely to be useful for OOP.

A'UM introduces a notation for streams which differentiates between variables being used as input and output streams [Yoshida and Chikayama 1987]. This annotation permits better compilation of the A'UM code since the compiler

knows more about streams. In Polka, this kind of information is present for variables declared in a class but does not extend to variables in messages.

Polka allows an object to execute a method without waiting for a message to arrive. This is useful when encoding clocks, producers, or other types of generator, but means that point no.3 of the Shapiro and Takeuchi model :

An object becomes active when it receives a message, otherwise it is suspended.

is not always true.

Polka also has a special type of message called a *suspendible* message. In essence, this is an ordinary message tagged with a list of logical variables. The semantics of such a message are that it will not be sent to its destination until all the variables on its associated list become bound. This sort of message is extremely useful for explicitly constraining the processing order of messages, which can be difficult in fine grained parallel objects, where several actions can be executing at once and several messages may be processed simultaneously. A similar technique has also been proposed by Cutcher [Cutcher 1986].

An important goal for Vulcan is a uniform approach to dealing with functions, objects and terms [Kahn et al. 1987]. This has lead to the treatment of terms as immutable objects which means that terms can be sent messages. This is implemented by utilising the predicate :

sendToTerm(Term, Message)

'sendToTerm' defines the action that a message has on a term, and will be heavily used because of message passing to commonplace terms. In particular, terms such as true, false, 0, 1, [], and [__] will appear often. This will mean defining numerous 'sendToTerm' clauses, many of which will have the same first argument, and good clause indexing will become important.

Since terms are immutable objects, they are accessed via messages like so :

1 : add(2, Z)

This example shows an 'add' message being sent to the immutable object '1', which will cause 'Z' to be bound to the immutable object '3'. To help with programming in this style, several immutable classes have been defined including vector, associative table, string, and integer.

3.3.3.2. Message Passing in the Backtracking Process View.

In LP languages where objects execute in and- parallel, asynchronous message passing is possible because of the ability to simultaneously execute goals in different objects [Pereira and Nasr 1984; Honiden et al. 1985; Ishikawa and Tokoro 1986]. In this group, three types of message passing primitives are used : wait declarations, events, and d-clauses.

3.3.3.2.1. Wait Declarations.

In the languages based on 'wait' declarations, communication between two objects is via a shared logical variable which is partially instantiated [Naish 1985; Honiden et al 1985; Kahn and Miller 1988b; Elshiewy 1988]. Each partial instantiation can be thought of as a message which causes the suspended receiver to start executing. The following example of this type of communication is based on a generator object passing a stream of elements to a buffer object. To simplify matters, no object oriented syntax is shown, and instead an object is shown as its underlying process. Termination clauses and other important features are also not shown, since they are not central to the problem of communication. A 'wait' declaration is needed for 'buffer', and is :

```
wait buffer( ?, ? ).
```

which indicates that a 'buffer' goal will suspend until its first and second arguments are bound. 'buffer' is :

```
buffer( [ put( El ) | In ], B ) if in( El, B, C ) : buffer( In, C ) .
```

The first argument is the input stream, while the second stores the buffer contents. This clause shows how 'buffer' deals with a 'put' message. 'in' adds the element to the buffer, which is then committed to via the use of a cut. The cut is not necessary for the communication but ensures that backtracking can not undo the message passing. This simplifies the implementation, since backtracking between objects is expensive. The generator object, called 'gen', does not need a 'wait' statement, and is :

```
gen( [ El | Q ], [ put( El ) | Out ] ) if gen( Q, Out ) .
```

Its first argument is its element list, while its second is the output stream. The clause shows how 'gen' transmits the elements of its first argument as 'put' messages from its second argument. 'gen' and 'buffer' must be linked via a shared variable when they are invoked in and- parallel.

```
<- gen( [ 1, 2, 3, 4 ], Stream ) // buffer( Stream, [] ) .
```

Messages will be passed from 'gen' to 'buffer' by partially instantiating 'Stream'.

3.3.3.2.2. Events.

In the languages based on events, a specified goal in the body of an object will only become true when a communication event occurs, so allowing a clause to execute [Pereira and Nasr 1984; Monteiro 1984; Futo and Szeredi 1984]. Shared variables do not need to be set up between objects, but this means that event names are globally known, and that the communication medium is hidden. The 'buffer' example will become :

```
buffer( B ) if - put( El ) | in( El, B, C ) : buffer( C ) .
```

The '-' is the receive annotation for the event which is separated from the rest of the clause by a '|' operator, and is conditional on the 'in' predicate succeeding. The 'gen' object sends elements like so :

```
gen( [ El | Q ] ) if + put( El ) , gen( Q ) .
```

The '+' is the send annotation for the event. Both the 'buffer' and 'gen' objects can be executed in and- parallel like so :

```
<- gen( [ 1, 2, 3, 4 ] ) // buffer( [] ) .
```

In Multilog, an event is represented by a :

```
wait_accept( Message )
```

predicate, which suspends execution until a suitable message arrives [Karam 1988]. A variant of this is :

```
accept( Message )
```

which fails immediately if no suitable message is present. Karam has equated their use to Ada's rendezvous since message patterns inside the predicates help to select a message from the queue at the object. If a message is later found to be unsuitable, backtracking puts the message back on the queue and another message is selected.

3.3.3.2.3. D-clauses.

The DL language [Monteiro 1984] contains another form of communication, called the d-clause, which looks like :

$$g1, g2 \text{ if } c1, c2 .$$

This can be rephrased as two Horn clauses :

$$\begin{array}{l} g1 \text{ if } c1 . \\ \text{and} \quad g2 \text{ if } c2 . \end{array}$$

However, the two clauses share variables and substitutions must be applied to the atoms simultaneously. The 'gen' and 'buffer' example rewritten using a d-clause becomes :

$$\text{gen}([E1 | Q]), \text{buffer}(B) \text{ if } \text{in}(E1, B, C) : \text{gen}(Q), \text{buffer}(C) .$$

When the communication is phrased in such a way, neither of the two goals in the head of a d-clause need be objects, but could be goals acting as messages. This is true in Mello and Natali's language [Mello and Natali 1986] where the buffer object might be :

$$\text{buffer}(B) \text{ if } \text{input}(B, C), \text{buffer}(C) .$$

'gen' might be : $\text{gen}([E1 | Q]) \text{ if } \text{put}(E1), \text{gen}(Q) .$

and the two objects communicate using the d-clause :

$$\text{input}(B, C), \text{put}(E1) \text{ if } \text{in}(E1, B, C) .$$

with 'in' defined as before.

Since d-clauses make no distinction between objects, messages, and goals, they can be used for more unusual transformations. An operation to combine two messages might be expressed as :

$$\text{put}(E1), \text{put}(E2) \text{ if } \text{put2}(E1, E2) .$$

'put2' can be thought of a message packet, which could be processed by 'buffer' like so :

$$\text{buffer}(B), \text{put2}(E1, E2) \text{ if } \text{in}(E1, B, B1), \text{in}(E2, B1, C), \text{buffer}(C) .$$

A d-clause could also be used to combine two objects :

$$\text{free_taxi}(T), \text{customer}(C) \text{ if } \text{busy_taxi}(T, C) .$$

which could be seen as creating a new 'busy_taxi' object from the 'free_taxi' and 'customer' objects.

The d-clause has hidden overheads for the Prolog inference mechanism since two goals must be chosen from the executable goal list, which means that the computationally efficient left-to-right, selection strategy must be sacrificed.

The object clause used by Conery in his HOOPS language bears some syntactic resemblance to a d-clause but has a different meaning in terms of pure Horn clauses. However, the object clause does suffer from the same termination problems as the d-clause and event, which are discussed next.

At the end of the communication between the 'gen' and 'buffer' objects, the final goals will be :

```
<- gen( [] ) // buffer( [ 1, 2, 3, 4 ] ) .
```

In order for these to terminate successfully, the clauses :

```
gen( [] ) and buffer( B )
```

should be added to the respective objects. However, if the new 'buffer' clause is placed incorrectly, then it may be chosen too soon and cause the objects to terminate prematurely. This problem can be dealt with by syntactically differentiating between these types of clauses and others. The inference engine can then delay their use until it can determine that all communication has finished. This is possible if it knows what predicates represent objects and so what communication-related goals still need to be processed in the executable goal list. When all these communication goals have been solved, termination clauses like :

```
buffer( B )
```

can be used to finish the computation.

A further complication is that the user may wish to include clauses for 'gen' and 'buffer' which should be executed in the absence of communication. When these should be selected is another problem.

The use of new inference rules for events and d-clauses alter the semantics of the underlying logic language, although some of the extensions seem more amenable to explanation than others. For example, a temporal logic is being developed to deal with the notion of event [Monteiro 1984]. Conery points out that such extensions must

affect the referential transparency of pure Horn clauses because of their side effects to state [Conery 1988].

A key difference between the wait annotation and the event or d-clause is how synchronisation is handled. With the wait annotation, it is dealt with by using primitives which use the binding status of a variable to suspend a process. In event and d-clause communication, synchronisation is hidden in the control strategy of the inference engine.

3.4. State.

A key feature of an object is that it encapsulates both operations and state. The state component is important because it allows information to be retained inside an object between the processing of messages. Also, the state of an object can only be accessed via its operations, and so data hiding and data abstraction are supported.

The semantics of LP-based object oriented languages seem to hinge on their treatment of state, which can be classified into 5 main groups by virtue of what mechanisms are used :

- there is no permanent state
- 'assert' / 'retract' type primitives
- new inference rules
- meta level programming
- arguments in processes

Generally, the first three mechanisms are used by the ADT group of OOP languages, the meta level techniques by the theory group, and the process arguments by the process group.

3.4.1. State in the ADT View.

State is only generated as a byproduct of the evaluation of a message in some languages and is not retained between messages [Kahn 1982; Bancilhon 1986]. McCabe has discussed a partial solution to this, which relies on the object label being returned at the end of the message processing [McCabe 1987a]. The label will contain new state values as its parameters, and these can be used as the initial state for the object when it is next invoked. However, all other objects must be sent this label in order for them to use the new state. The main advantage of this approach is that the semantics of objects and classes become much simpler. For example, McCabe has

given such a language both a proof theoretic and model theoretic semantics. The proof theoretic approach introduces two new proof rules to deal with classes and the use of inheritance. The model theoretic approach shows how classes and inheritance can be mapped to Horn clauses.

In most of the other languages which treat classes as sets of Horn clauses, the most popular approach to dealing with state information is to store it as extra clauses, which are asserted or retracted when the state needs to be changed. The main problem with this is that the semantics of these operations are not well defined, and also lead to non-monotonic reasoning. An additional feature, or bug depending on the situation, is that backtracking over these operations does not normally undo them. However, this technique does have the benefit of being relatively simple to implement.

In some languages, the assertion and retraction of state clauses are hidden by using a slot mechanism, which supplies predefined operations for storing, retrieving and changing state information [Chikayama 1984; Fukunaga and Hirose 1986; Iline and Kanoui 1987]. Normally, slots are divided into those for classes and those for instances, but this is not always the case [Gullichsen 1985].

To try to deal with the semantic problems with assertion and retraction, Porto has suggested new rules of inference which accommodate them, although the semantic basis of these rules still needs to be fully explored [Porto 1983].

Chen and Warren have added intensional semantics to Horn clause logic, which allows intension variables to be used in programs [Chen and Warren 1988]. Such a variable is actually a function from a state to a set of values, and carries around a sequence of states which represents the partial history of the object. A frame assumption is used to make the history complete.

3.4.2. State in the Theory View.

In the theory view, state change can be viewed as the generation of a new object [Bowen 1985]. This leads to practical problems since other objects will need to be informed that they should now communicate with the new object. Also, the semantics of such languages appear to be quite complex [Kowalski 1979].

3.4.3. State in the Process View.

State is dealt with in almost exactly the same way in both types of process view.

3.4.3.1. State in the Committed Choice Process View.

Point no.1 of the Shapiro and Takeuchi model is :

An object can be represented as a process which calls itself recursively and holds its internal state in unshared arguments.

This can be seen in action in the 'filestore' example of section 3.2.3.1 where 'Files' is the state of the object. The main advantage of this approach is that state changes are implemented without resorting to metalogical primitives or to new inference rules. However, the committed choice nature of these languages means that earlier states can not be retrieved by backtracking.

At the language level, state variables are normally treated either as ordinary logical variables, as in Polka, or as objects which must be sent messages to access or alter their values, as in Vulcan [Kahn et al. 1987].

Trehan proposes three object models for committed choice non-deterministic logic languages [Trehan et al. 1988]. Each is based on the Shapiro and Takeuchi model, but also describes how information is shared between objects.

The *closed* object model defines the traditional object, which only allows its data to be accessed via message passing.

The *open* object model allows an object to share data with other objects, but care must be taken about updating it.

The *filofax* object model describes an object which stores data about other objects. This can be done by making it regularly poll all the objects in order to keep its information up to date. A more satisfactory approach is to make the objects send information to the filofax object when they change.

3.4.3.2. State in the Backtracking Process View.

The objects in these languages are processes and so state has an explanation as the arguments of a recursive predicate. In the subset of these which use 'wait' annotations [Naish 1985; Elshiewy 1988], Kahn and Miller have shown that state change can be efficiently implemented [Kahn and Miller 1988b].

Some languages allow data to be asserted and retracted, which is harder to do safely in a concurrent language because of the problem of mutual exclusion [Futo and Szeridi 1984; Mello and Natali 1986].

3.5. Inheritance.

Inheritance can be defined as a mechanism in which classes are specialised by sharing the behaviour of other classes. This is a somewhat imprecise definition because there is still no consensus on the meaning of inheritance. For example, inheritance may be strict, which forces behavioral compatibility between a class and its inherited class, or may be non-strict, which allows a class to arbitrarily redefine an inherited method. Another design decision is whether to permit multiple or single inheritance. Multiple inheritance increases behaviour sharing because it makes it possible for one class to combine the behaviour of several other classes. This is illustrated by the inheritance graph in figure 3.2.

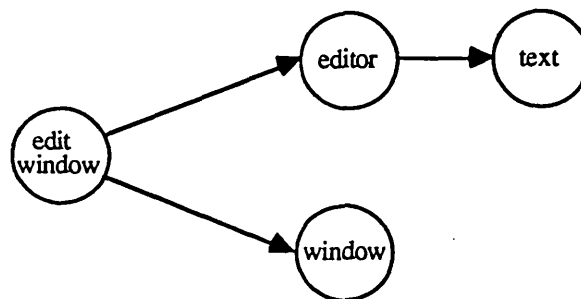


Figure 3.2 : Inheritance graph for 'edit window'.

The circles represent the classes, while the arrows indicate the inheritance links.

This graph shows that the 'edit_window' class uses multiple inheritance in order to use 'editor' and 'window', while 'editor' utilises only single inheritance.

In general, LP-based OOP languages utilise non-strict, multiple inheritance.

Built into most OOP languages are ways for a class to refer directly to its inherited classes (c.f *super* in Smalltalk) and also for inherited classes to refer to the class at the bottom of the graph (c.f *self* in Smalltalk) [Goldberg et al. 1983]. Of these two, self communication is most important since it allows the meaning of the self application of a method in an object to depend on objects below it in the inheritance graph [Stefik and Bobrow 1986]. In practice, this means that a 'self' message is always sent to the object at the bottom of the inheritance graph for a particular invocation, irrespective of the actual object sending it. For example, an invocation of 'editor' causes all the 'self' messages transmitted by its inherited 'text' object to be delivered to 'editor'.

3.5.1. Inheritance in the ADT View.

Inheritance rules can be included inside a class, and so need only mention the inherited classes. An alternative is to specify such rules separately, in which case they will need to include the inheriting class as well, as in :

`square( X ) inherits rectangle( X, X )`

How such a rule is actually represented depends on the class translation method. In the label/predicate approach, the rule :

`l inherits s`

becomes something like :

`l( _ ) if s( _ ) .`

Using the label/argument translation, the single 'inherits' rule will generate one new clause for every clause in the original class. For example, the 'l' class with methods for 'x1' and 'x2' :

`l [x1 if y1 .
 x2 if y2]`

will be translated into two clauses :

`x1(1) if y1(1) .
x2(1) if y2(1) .`

In addition, the 'inherits' rule will generate two extra clauses :

`x1(1) if x1(s) .
x2(1) if x2(s) .`

The meaning of these two new clauses is that if a method in 'l' can not handle a 'x1' or 'x2' message then an alternative method is tried in 's'.

Either implementation strategy means that through the use of backtracking, a message can reach all the clauses of all the inherited objects. However, this can be restricted by specifying clauses which are not to be inherited, or by listing particular clause instantiations which should be ignored [Kauffman and Grumbach 1986; McCabe 1987a]. A possible syntax for this is :

`edit_window inherits editor - cursor_pos / 1`

This states that the 'edit_window' class inherits everything from the 'editor' class, apart from the clauses for 'cursor_pos' of arity 1. This can be implemented at the Prolog level by the use of 'not' [McCabe1987a], but normally restrictions on inheritance are achieved by using cuts [Koseki 1987]. The difference should be

transparent to the user, but the underlying Prolog can not be totally ignored because its search strategy affects the order in which inherited clauses are tried.

When classes have a meaning in terms of types, inheritance can be thought of as a type hierarchy, which places extra constraints on how classes can be specialised [Bancilhon 1986; Goguen and Meseguer 1986]. In Bancilhon's approach for example, multiple inheritance and method overriding are not possible, but more compile time checks can be applied to reduce errors.

An important problem occurs when multiple inheritance is used, and a class is inherited twice, as 'cursor' is in the following inheritance graph in figure 3.3.

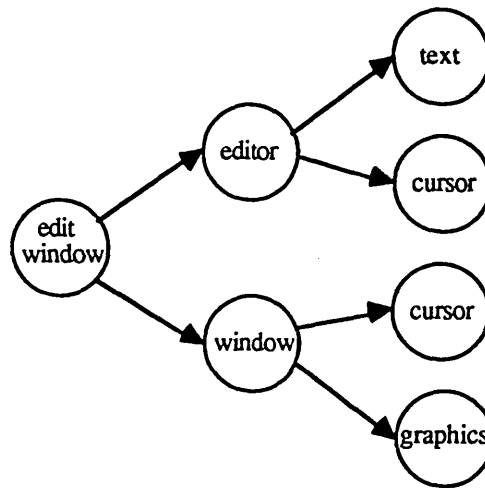


Figure 3.3 : Inheritance graph with two occurrences of 'cursor'.

In the above graph, a message that can be dealt with by 'cursor' may be answered twice. Sometimes such a behaviour is not desirable, but it is not a simple matter to override Prolog's backtracking mechanism. Aside from the use of 'not' and cuts to restrict this, information can be stored during the search to indicate that certain kinds of clauses have already been tried. Another method is to use a metalogical predicate like 'set_of' to return all the answers to a given message and then choose between them [Johnson 1988b]. In the object oriented version of SISCtus Prolog, the problem is dealt with by not allowing multiple inherited instances of the same class [Elshiewy 1988], so making the graph look like figure 3.4.

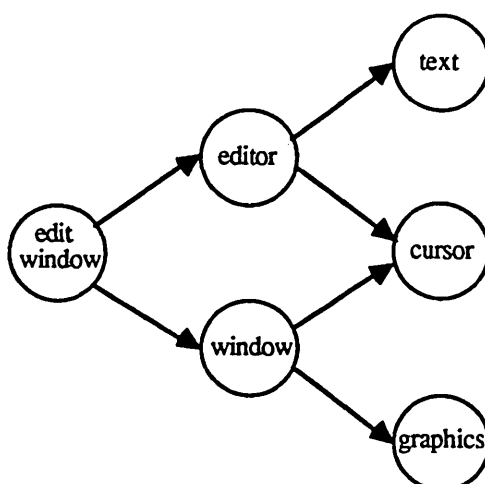


Figure 3.4 : Inheritance graph with one occurrence of 'cursor'.

This may be suitable if the state inside the 'cursor' object is intended to represent the same thing in both 'editor' and 'window'. However, special provision must be made to deal with broadcast messages, which will result in two messages going to 'cursor', and so perhaps incorrectly altering its state.

Although the use of backtracking for implementing multiple inheritance presents some problems, it is in many ways an elegant mechanism. This will become clearer in section 3.5.3.1 when the complicated approaches used in the committed choice process view are examined.

Another technique for searching the inheritance graph is by using an interpreter rather than leaving it to Prolog [Zaniolo 1984; Johnson 1988a]. This enables more flexible types of search to be carried out, such as a breadth-first search.

Just as interpreters can deal with inheritance information in special ways, they can also be used to recognise other types of relationships [Koseki 1987]. One example of this is the 'sub' relation, which returns a class below the current one in the inheritance graph, and through backtracking can return them all. 'sub' can be thought of as the inverse of the inheritance relation [Zaniolo 1984].

Since an inheritance rule is usually represented as a Prolog clause, the passing of a message from an object to its inherited object is done at run time. This means that the choice of inherited object can be dynamically altered, and also allows inherited objects to be changed, although it is difficult to encode such ideas without using metalogical predicates [Chikayama 1984; Gullichsen 1985].

Another implementation approach for inheritance, is to copy all the clauses of the inherited classes over to their inheritor at compile time [Chusho and Haga 1986; Goguen and Meseguer 1986]. This reduces the run time costs but increases the size of

classes and makes them less flexible. A possible advantage is that a change to an inherited clause in a class will not affect the same clause inherited by other classes.

A powerful form of inheritance is based on augmenting unification [Kornfeld 1983; Kahn 1986]. For example, to state that a 'square' class inherits the functionality of 'rectangle', would require the clause :

$$\text{square}(X) = \text{rectangle}(X, X)$$

A message to 'square' might be :

$$\text{square}(10) :: \text{area}(A)$$

This should return the area of a square of sides of length 10, but if such a method is not defined in 'square' then the message can be automatically transferred to 'rectangle', so becoming :

$$\text{rectangle}(10, 10) :: \text{area}(A)$$

The elegance and power of this approach is that it also works in the opposite direction : if 'rectangle' does not understand a message then it can be passed to 'square'. This inheritance mechanism is supported by extending unification but also requires extra control safeguards to stop loops [Kornfeld 1983].

Inheritance is combined with unification in LOGIN, and is claimed to be more economical than resolution [Aït-kaci and Nasr 1986]. This is achieved by replacing resolution with the computation of the greatest lower bound of two symbols relative to an inheritance ordering.

Many languages based on the ADT view do not permit either 'super' messages or self communication, although they are not difficult to implement [Bancilhon 1986; McCabe 1987a; Trenouth 1988]. A 'super' message can be translated into an immediate call to the inherited class, while self communication requires an extra argument in every message. This extra term will contain the name of the initially invoked class, and then if a 'self' message is necessary, the corresponding Prolog goal will use this name to invoke that class. A more complex approach is used in BiggerTalk, which uses a stack to keep account of the self references [Gullichsen 1985]. This allows the destination to be left out of a message, and instead it is popped off a 'self' stack.

3.5.2. Inheritance in the Theory View.

Inheritance in the theory view is seen as just one way in which different theories can be linked at the meta level [Coscia et al. 1988]. Thus, when a goal in one theory can not be answered, it is passed to the meta level and then to the theory representing its inherited object. Usually Horn clause definitions are utilised to specify the search strategy for the theories.

A related method is utilised in the PPS (Parlog Programming System), a multiprocessing programming environment which supports Parlog [Clark and Foster 1987]. Users interact with the PPS by querying and updating collections of logical clauses, termed databases. The PPS understands certain clauses, called meta clauses, as describing meta level information such as system configuration details, the status of user deductions, and relationships between databases. Meta clauses are stored in the meta database for a database. Inheritance is then coded by routing a failed query to an 'inherited' database which is specified in the meta database.

One area in which many theory view languages are weak is in their treatment of forwarded goals. These are goals which could not be answered in one theory and so were forwarded to an 'inherited' theory. If a rule is present in this theory it will be applied to the goal, so creating new subgoals. Usually these are solved with respect to the current theory, but it might be better if they were solved in the theory which originally forwarded their parent goal.

3.5.3. Inheritance in the Process View.

Inheritance in both of the process views is based on passing messages from the object which represents the inheritor to the inherited object. This means that inheritance is implemented in a similar way to message passing.

3.5.3.1. Inheritance in the Committed Choice Process View.

An inheritance link between two classes is usually implemented by connecting them via a shared logical variable which acts as a stream at a run time. In addition, an extra clause is added to the inheritor class, whose purpose is to pass an input message along the stream when it can not be processed by any of the other clauses. Such a clause is added to every class which inherits another class.

If a 'database' class inherits the 'filestore' class, it may be graphically represented as figure 3.5.

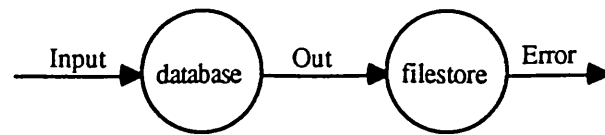


Figure 3.5 : 'database' inherits 'filestore'.

This is encoded in Parlog as :

```
database( Input, Out), filestore( Out, Error )
```

At the user level, the 'Out' stream should be invisible.

The connection between the graphical view of inheritance and its implementation is currently being investigated as a means of defining inheritance relationships more simply [Fellous et al. 1988] .

Multiple inheritance presents a problem since languages based on concurrent LP can not use backtracking to search the inherited objects for the right clause, and once a message has been forwarded to a particular object, it is committed to its choice. One answer to this is to broadcast the same message to all the inherited objects, but this makes it possible for a variable in the message to be bound more than once and so cause one or more of the objects to fail. In Polka, broadcasting primitives exist which automatically make unique copies of a message for each inherited object. Even so, problems still remain about what to do if more than one copy of the message is successfully processed, or if all the copies fail to be dealt with.

Self communication is normally implemented by adding an extra output message stream to every inherited object. All of these streams are merged into a single stream which enters the object at the bottom of the inheritance graph. This mechanism then permits self communication to be converted into a message send along the 'self' output stream of the particular object. Such streams can exist throughout the lifetime of the objects, as in Polka, but an alternative is to only set them up for the duration of the computation of a forwarded message, as is done in Vulcan [Kahn et al. 1987].

Some of the important problems with self communication include making sure that 'self' messages are given priority over external communication, and that state changes caused by self communication do not interfere with calculations using the old state.

Although 'inheritance as message forwarding' is the norm, Vulcan also permits copying, so that the 'database' example could be expanded to include the clauses from 'filestore' [Kahn et al. 1987]. This makes self communication simpler to implement but also leads to duplication of code, potentially very large classes, and makes it more difficult to incrementally recompile a class.

Inheritance is implemented in a totally different way by Ohki : inherited objects do not have a separate existence, but are invoked inside a meta call when required [Ohki et al. 1988]. Since inheritance is implemented as a procedure call, messages can be sent to inherited objects from guards. This is not possible in most other languages because a guard does not allow the manipulation of output streams until it has committed. The main drawback of Ohki's approach is that an inherited object can not retain its state between message processing. This is solved by letting the bottom object in the graph of inherited objects carry all the state variables around itself. These are passed to an inherited object when it is called, which means that there is no encapsulation of data in the inherited object and the bottom object must initialise all of its variables. Also, an inherited object does not seem able to communicate with the bottom object in the inheritance graph, which is essential for self communication.

Trehan points out that many committed choice process view languages do not deal with inheritance of state [Trehan et al. 1988]. This is actually quite a serious problem to deal with because inherited state is normally stored in the inherited objects. This means that it can only be accessed and altered by means of message passing, and in a fine grained parallel language the use of such messages may interact unfavourably with messages from other parts of the object network which are also manipulating the same state. In most languages, including Polka, it is left to the programmer to enforce the correct use of inherited state.

3.5.3.2. Inheritance in the Backtracking Process View.

In these kinds of languages, inheritance is encoded by passing an unrecognised message to some other object which is specified as its inherited object. The actual form of communication mechanism varies but depends on the mechanism used for ordinary message passing. An important factor is the presence of backtracking, which means that the search of multiple inherited objects is much more straight forward than in the committed choice process view. However, self communication is still complicated and some languages [Pereira and Nasr 1984] avoid the problem by disallowing self communication, while others use a weaker form [Ishikawa and Tokoro 1986] where 'self' messages are implemented as internal predicate calls. This avoids the 'self' message priority problem but requires a global name space for predicates.

3.6. Other Paradigms.

All the languages in this survey are based on LP and OOP and so are multi-paradigm languages to a greater or lesser extent. Some designers have taken this approach further and have incorporated other paradigms into their languages.

3.6.1. Other Paradigms in the ADT View.

The most common extra paradigm in languages of this type is functional programming, which has been studied in detail by McCabe [McCabe 1988]. The aim of that work was not only to add first order functional ability to the language, but also to include lambda expressions and function variables for higher-order programming. However, this is constrained by his desire to map these extensions to first order logic, so it is not possible to compare arbitrary relations in the language, which would require higher order unification. Of the functional additions, lambda expressions are particularly useful since they can be used to directly describe a function. The nearest Prolog equivalent is the meta call which is really a mapping from a constant symbol to a relation. The extensions are based on condition equalities of the form :

$$f(\text{Args}) = \text{Answer} \text{ if Conditions .}$$

where 'Answer' is returned if the call unifies with 'f(Args)', and all the conditions evaluate to true. Related extensions have been proposed which extend unification rather than compile equalities down to first order logic [Kornfeld 1983; Kahn 1986]. However, the results are the same since functional, logic and object oriented programming can be carried out, and Kornfeld also notes that constraint logic programming is possible.

Goguen and Meseguer have developed a many sorted first order Horn clause logic with equality [Goguen and Meseguer 1986]. The sorts are not essential, being required only for strong typing, and if removed the language becomes similar to other languages with equality [Kornfeld 1983]. Additional information on the properties of predicates and functions can be included, such as whether they are associative, commutative, or idempotent. This helps when the axioms are being verified.

Along more conventional lines, Smalltalk, Lisp, and Prolog have been combined together [Takeuchi et al. 1986; Ishikawa and Tokoro 1986].

Several languages contain the notion of data-oriented programming, which allows demons to be fired when certain events occur [Chikayama 1984; Koseki 1987; Bartual 1989]. Some also use forward chaining, although this is easily simulated by using the backward chaining functionality of Prolog [Koseki 1987; Bartual 1989].

3.6.2. Other Paradigms in the Theory View.

Since a theory is executed by a meta interpreter, the actual contents of a theory can be almost anything so long as a suitable interpreter can be found to execute it, and other theories can communicate with it through message passing. For instance, theories in Mandala can include Prolog clauses, first order logic, or equations [Furukawa et al. 1984].

3.6.3. Other Paradigms in the Process View.

Vulcan subscribes to the committed choice process view of objects but the language is also important as one part of the Vulcan system which will contain many forms of abstractions and tools [Kahn 1987]. Although the system is still at an early stage, these abstractions and tools may include a functional language [Levy 1986], an or-parallel Prolog [Shapiro 1987], channels [Tribble et al. 1987], types [Yardeni and Shapiro 1987], a keyword notation [Hirsh et al. 1987], a constraints language [Saraswat 1988], lexical FCP [Rauen 1987], public key encryption [Miller et al. 1987] and a hardware description language [Weinbaum and Shapiro 1986]. Consequently, the Vulcan language does not contain all the functionality and the tools necessary for OOP, but instead will use the facilities in the rest of the Vulcan system.

3.7. Summary of the Views.

The six design issues discussed here have indicated the wide range of languages that combine OOP and LP. Which language is best is impossible to say, since different languages tend to emphasis different issues. Some concentrate on semantics, some on functionality, and some on investigating particular extensions, such as augmented unification. However, the preceding sections have highlighted most of the strengths and weaknesses of the four main approaches for combining OOP and LP, and this section will summarise them.

3.7.1. Summary of the ADT View.

The ADT view is probably best for explaining the semantics of OOP, either as a syntactic extension to Horn clause logic, or as types. The conservative extensions to LP required for supporting OOP also mean that other paradigms can be easily added, such as functional programming. The ADT approach also has a straight forward

means of dealing with inheritance (multiple inheritance in particular), and self communication.

On the negative side, ADT-based languages are not particularly good at dealing with state, except at the expense of semantic clarity. Also, backtracking causes some problems with respect to the undoing of object creation, message passing, and state changes. On a more general level, backtracking may not be a suitable mechanism for typical OOP applications such as systems software and simulations.

3.7.2. Summary of the Theory View.

The theory view is the most flexible and powerful approach to combining OOP and LP. In particular, inheritance can be viewed as just one of many possible relationships between theories which can be simply expressed in Horn clause logic at the meta level. Also, the 'class as object' approach, as typified by Smalltalk, follows directly from treating a theory as a first order entity.

The drawbacks of the theory view are that it is hard to implement efficiently, and also has a complex semantics.

3.7.3. Summary of the Process View.

The process view has been subdivided into the committed choice and backtracking approach, but they share a number of qualities. They both deal with the representation of state in a simple manner, and both allow concurrency at the inter-object level. However, their semantics have not yet been fully realised because of the complex interaction of concurrency with Horn clause logic.

3.7.3.1. Summary of the Committed Choice Process View.

This approach contains a powerful message passing mechanism, based on shared logical variables, which permits many unusual communication strategies to be encoded. Also, the fine grained parallelism allows an object to carry out many actions simultaneously and to process many messages at once. However, the internal concurrency is something of a hindrance when dealing with self communication and inherited state variables, and the language level resolution of these problems is an important issue. The don't care non-determinism of objects makes their execution simpler to understand, and perhaps closer to what is required by an OOP language. Unfortunately, it makes the treatment of multiple inheritance more difficult.

3.7.3.2. Summary of the Backtracking Process View.

This methodology has produced some interesting forms of message passing, which offer new ways of enabling objects to communicate. The use of sequential LP concepts inside the objects means that multiple inheritance is as simple as in the ADT view, but the presence of backtracking causes problems, which are compounded by the availability of inter-object concurrency.

Chapter 4. An Overview of Polka.

A summary of the features in Polka are presented in this chapter, followed by two sections which discuss the advantages and disadvantages of the language.

4.1. Polka Features.

The three main design goals behind Polka are :

- the combination of OOP with concurrent LP
- the creation of a 'higher level' Parlog
- support for meta level programming

The concurrent LP part of the language means that Polka utilises the committed choice process view of LP-based OOP languages, and that its implementation follows the Shapiro and Takeuchi model.

Polka is most influenced by the first design goal, which implies that both classes and predicates can be encoded and used together. The result of this is a combination of conventional OOP and concurrent LP constructs along with several new features which are based on both paradigms. The basic Polka structure is the class, which means that the OOP ideas of encapsulation and data abstraction are present. Also, objects communicate by message passing, and can utilise multiple inheritance and self communication.

The Parlog component is apparent in the way that messages are represented by Parlog terms, and in how input pattern matching is used to select a clause to handle a message. In addition, state variables are encoded as Parlog terms, which permits their manipulation by predicates.

Variables in a class are typed and can also be made visible to the user, which allows parametrised classes to be defined. Also, internal variables, which are invisible to the user, can be given initial values before any messages are accepted, and these values can be generated by predicates or Polka operations, if required.

State variables, although represented by Parlog terms, are assigned new values using a Polka operation called *becomes*. This allows state changes to be encoded in a much less verbose way than in Parlog where arguments representing the state must be included in every clause. Although that mechanism is used in the implementation of Polka, it is hidden at the language level.

A clause in a class can use conjunctions of Parlog predicates and Polka operations. The parallel conjunction is available, which permits a message to initiate

several simultaneous actions. Also, it is possible for finer grain parallelism to be exhibited since an object can process more than one message at once.

Clauses are separated by the Parlog sequential search operator, or the parallel operator which allows a message to try several candidate clauses at once. In addition to the clauses specified by the programmer, Polka adds extra clauses to handle the termination of an object, the reporting of errors, and the forwarding of unrecognised messages to inherited objects. An if-then-else clause form is also supported.

Predicates can be defined locally in a class, and these can use some of the Polka operations, which is impossible for ordinary Parlog predicates.

The input and output streams for a class are visible and are represented by logical variables. This allows them to be manipulated by predicates, or even be treated like messages. This is a powerful feature of the language, and many unusual communication and synchronisation mechanisms can be encoded because of it.

Although clauses are normally invoked because of an input message, it is also possible to invoke a clause independently of input by utilising the `null` keyword. This is contrary to the Shapiro and Takeuchi model which rules out such functionality, but the extension is useful for programming such things as clocks and sources.

The closure of input and output streams is specified by the `last` keyword which hides the fact that streams are implemented as partially instantiated lists, and that their closure is equivalent to their tails being bound to `[]`.

The inclusion of the logical variable in messages permits many forms of message protocols to be developed apart from the usual synchronous and asynchronous kinds. Polka supports one particularly useful one at the language level, called *suspendible* message passing. One of its uses is for separating the synchronisation and functionality components of a class.

Class mutation seems an odd idea since it allows an object to change into another object, or to split into several objects. However, it is one of the core concepts of the actor model [Agha 1986], and is included in Polka via the `i_am` construct.

The features described above can be viewed as object oriented, but a subset of them can also be thought of as ‘higher level’ Parlog abstractions. This subset is essentially made up of the standard Polka constructs except for those related to inheritance and self communication. These constructs are ideal for writing programs which represent processes, but in a less verbose, more understandable, and error free manner than the corresponding Parlog.

The final strand of the design is the incorporation of meta level programming ideas from LP, but altered so that they become applicable to OOP. The main meta level construct is the *term class*, which is a (slightly simplified) Polka class. Such a class can be treated like a Parlog term, which means that it is a first order entity, and so can be passed as an argument to a predicate, stored in a state variable, or sent as a message to another object. This flexibility is augmented by a number of built-in predicates, of which the most important is a meta interpreter for creating instances of a term class.

4.2. The Advantages of Polka.

A powerful feature of Polka is the unique way in which OOP and concurrent LP are combined. For instance, the utility of representing state variables as terms, and using predicates to manipulate them is invaluable. The logical variable is also extremely useful for message passing, stream manipulation, and many other tasks.

Since communication in Polka is based around the logical variable, an object is unaware of the name of the object sending it messages, and of the destination of its output messages. When this is combined with the ability to manipulate the communication stream, the polymorphism of message passing is greatly increased.

Committed choice non-determinism is at the heart of the computational model of Polka which means that problems with don't know non-determinism are absent. In particular, state changes, object creation, and message sending are much simpler to understand when it is impossible for them to be 'undone' by backtracking. In fact, committed choice non-determinism is arguably the more natural model for many object oriented application areas, such as simulation and operating systems, where actions should not be silently rescinded.

As discussed in the last section, Polka can be used as both a LP-based OOP language, and as a 'higher level' Parlog, which extends its usefulness outside of the OOP field. Also, the meta level constructs increase the utility of the class concept by allowing term classes to be treated as first order entities.

4.3. The Disadvantages of Polka.

Polka is a combination of two paradigms : concurrent LP and OOP, and so the user must be familiar with them both before the full functionality of the language can be utilised. This problem is overcome to some degree by using subsets of the language. The smallest subset is the 'higher level' Parlog component, which extended

with inheritance, self communication and certain other features becomes an OOP language. The language is complete when the meta level features are included.

The other related problem with a language based on two paradigms is that its semantics become more complex. However, the operational semantics developed in chapter 10 shows that this complexity need not be too great.

While the fine grained parallelism of Polka can be viewed as an advantage in some situations, it can be a major disadvantage in others. The most significant problem concerns message processing order, where the programmer wants to ensure that the processing caused by one message is constrained by the processing of another message. Polka has numerous constructs for restricting the parallelism of an object, but its most innovative feature in this area is the suspendible message.

The absence of backtracking was considered to be an advantage in the previous section, but it also means that the search abilities of Prolog-based OOP languages are missing. This is a problem with all committed choice process view languages, and Polka goes some way to dealing with it by being able to broadcast messages and specify how their replies are handled.

There is no dynamic inheritance in Polka, by which we mean the ability to change the inheritance graph for an instance at run time. This is not a limitation of the committed choice process view and an earlier version of Polka [Davison 1987a] contained operations for this task. However, the complexity of the implementation seemed too high for such a rarely used feature, and it was removed. In the current Polka implementation therefore, the links between the inherited objects that make up an instance are fixed at compile time. This reduces the polymorphism of an instance, but this disadvantage seems to be outweighed by the simplicity of the implementation (and the resulting semantic clarity).

A drawback of the committed choice process view is the difficulty it has in handling inherited state. The implementation approach dictates that inherited state is stored in inherited objects, and so is only accessible by message passing. This means that the programmer must include clauses in his class for accessing and altering inherited state values. The problem does not end there since fine grained parallelism permits many messages to be processed at once. In the context of inherited state, this can cause a value to be changed in an unpredictable manner because of the order in which messages are processed in the inherited objects. Polka has no absolute answers to these problems but suspendible messages can be utilised, and the order in which messages are forwarded to an inherited object can be controlled.

Polka has a weak typing system, but it does more than most LP-based OOP languages, since class variables are differentiated into stream and state types, which

permits the compiler to capture illegal operations. However, Polka does not go as far as a language like A'UM which also types variables inside messages [Yoshida and Chikayama 1987]. However, this kind of functionality can be encoded by using guards, which can be very expressive when user-defined guard predicates are employed.

Part 2. Language and Applications.

Chapter 5. The Language.

The main features of Polka will be described first, followed by a discussion of how a subset of the language can be viewed as a 'higher level' Parlog. However, an examination of the constructs for meta level programming will be delayed until chapter six.

5.1. Class Structure.

All Polka classes have a similar structure which can be summarised by the informal BNF in figure 5.1.

```

< class name >
[ inherits < class names > ]
< variable declarations >

[ initial < actions > ]

clauses
  < clauses >
[
code
  < predicates >
]
end .

```

Figure 5.1 : Informal BNF for a Polka class.

A complete, and more formal, BNF can be found in the appendix.

The words in bold are Polka keywords, the phrases in angled brackets are informal descriptions of the text that should be placed there, while square brackets around text denotes that it is optional.

The BNF above shows how a Polka class starts with its name, may have an **inherits** line, and then declares its variables. These variables define streams and states for the class which may be visible to the user or invisible (local to the class). The optional **initial** section is next, followed by the **clauses** section which contains the clauses for manipulating the variables. Each clause is separated by either the Parlog parallel search operator '.' or the sequential search operator ';'. By using '.', or-parallelism is supported.

A clause has the form :

```
< input messages > => < actions > < separator >
```

The actions to the right of the '=>' will be a sequence of calls to Parlog predicates or Polka operations separated by Parlog's parallel conjunction ',' or

sequential conjunction '&', with the possibility of a guard. By using ',', and-parallelism is supported.

A clause is chosen by input pattern matching its message part against messages arriving on the input stream. If the match is successful and any guard part also succeeds, then the clause body is executed.

The code section is the last part of a Polka class and is optional. It contains Parlog predicates which are visible only to the class.

The class definition finishes with the **end** keyword.

5.1.1. A 'receiver' Class.

As a more concrete example of Polka class structure, a class for 'receiver' will be defined. It is based on the 'receiver' process discussed in section 2.3.1 which receives messages of the form :

message(< number >, < reply >)

The class is :

```

receiver
clauses
  message( 200, Reply )    =>  Reply = stop ;      % bind Reply to stop
  message( No, Reply )    =>  Reply = continue ,    % bind Reply to continue
                           write( No ) .           % write number
end .

```

Program 5.1 : A 'receiver' class.

The name of the class is 'receiver', it inherits nothing, and also does not appear to have any stream or state variables. In fact, this is not the case because, by default, a class always has one input stream called 'Ins' which enables it to receive messages.

The class has two clauses : the first deals with a message of the form

message(200, Reply)

This causes 'Reply' to be bound to 'stop' by using the Parlog '=' operator. The second clause is only tried if the first is rejected because of the ';' separator, and deals with the more general 'message' form. In that case, 'Reply' is bound to 'continue' and 'No' is printed using the 'write' predicate.

In the original process definition, a clause is required for terminating the process when its input is set to []. This situation is handled automatically by the class which has a default termination clause added to it by the compiler.

The compiler also augments the class with a delegation clause. This captures any messages which are not recognised by the user defined clauses and places them onto an output stream called 'Error'. This means that an invocation of 'receiver' will have an extra argument in addition to its input stream, which is used for outputting rejected messages. For example , a call to 'receiver' will be :

```
<- receiver( Ins, Error ),
   Ins = [ message( 199, R1 ), message( 200, R2 ), hello ] .
```

which produces the bindings :

```
R1 = continue      R2 = stop
```

```
and                Error = [ hello ]
```

'Ins' is the input stream, and 'Error' is the error output stream. Note that Polka classes are called in exactly the same way as Parlog predicates which makes it possible to use the two interchangeably.

The 'Error' stream is bound to [hello] because one message was rejected (the third one) and then when 'receiver' terminated, its 'Error' stream was closed.

5.1.2. A 'filestore' Class.

A slightly more complex class definition is required for the Polka version of 'filestore' discussed in section 3.2.3.1. It can understand two types of message :

```
add( Key, File )
```

```
and                extract( Key, File, Ok )
```

which add and extract file information respectively. The process version had a 'files' state variable which the class will also use, along with the 'extract_file' predicate. 'filestore' is :

```

filestore
invisible Files state <= []           % initialise 'Files'
clauses
  add( Key, File ) =>
    Files becomes [ file( Key, File ) | Files ] .           % add a file

  extract( Key, File, Ok ) =>
    extract_file( Key, F, Files, NewFiles ) :               % extract a file
    Ok = done, File = F,
    Files becomes NewFiles ;

  extract( Key, File, Ok ) => Ok = no_entry_for( Key ) .    % return an error

code
  mode extract_file( ?, ^, ?, ^ ) .
  % only mode given for brevity

end.

```

Program 5.2 : A 'filestore' class.

This class defines 'Files' as an invisible state variable whose initial value is []. The **invisible** attribute means that the variable will not be seen by the user when the class is invoked, unlike the 'Ins' and 'Error' streams for example. The **state** attribute is a typing device which is used to check that the variable is used correctly in certain Polka operations. The [] is assigned to 'Files' when an object is first created and before any messages are processed.

The first clause of 'filestore' adds 'File' information to 'Files' using the Polka **becomes** operator. **becomes** does not carry out destructive assignment, and it is more accurate to call it a replacement operator which only takes effect when the next message is processed. This means that the new list :

$$[\text{file}(\text{Key}, \text{File}) \mid \text{Files}]$$

will only become the new value of 'Files' when the processing of the next message begins.

The second clause shows how Polka can use Parlog predicates, and if the predicate succeeds then 'Ok' and 'File' are bound and 'Files' is updated to the 'NewFiles' value. However, if the guard fails then the third clause is tried and the 'Ok' variable is set to an error message. This illustrates how errors can be caught by using deep guards.

As with the last example, no delegation or termination clause needs to be included since these will be added by the compiler.

This class has a **code** section which contains the Parlog definition for 'extract_file'. It is a standard definition but is only visible to the class.

An invocation of 'filestore' might be :

```
<- filestore( Ins, Error ),
   Ins = [ add( 1, a1 ), add( 2, a2 ), extract(1, File1, Ok1 ),
           extract( 3, File2, Ok2 ), hello | _ ] .
```

The bindings produced are :

	File1 = a1	Ok1 = done
	File2 unbound	Ok2 = no_entry_for(3)
and	Error = [hello _]	

This is identical to the earlier process call but does not rely on the user initialising 'Files', since this is done internally by the object.

It is fruitful to compare the Parlog and Polka version of 'filestore'. The Polka version is shorter because its clauses do not contain a head and body call for 'filestore', and arguments do not need to be stated in each clause. The implicit input stream, error stream, delegation clause, and termination clause reduce the actual code size of the Polka program. Also, the use of an invisible variable and the `code` section help to enforce encapsulation and data abstraction since the object hides its internal representation.

The powerful features of Parlog are still present : and- and or- parallelism, unification and the logical variable. Of special interest is the use of back communication as a reply mechanism for messages.

More important than the reduction in code size, is the conceptual aid that Polka gives. The view of 'filestore' as a class is reinforced by the way that a message causes an action, and local variables and predicates are defined and used. The Parlog version tends to obscure this by showing the recursive nature of the 'filestore' predicate. The verbosity of Parlog code for other OOP techniques, such as multiple inheritance and self communication, is even worse, and this is probably why such techniques have not been utilised in concurrent LP.

In the remaining sections, the constructs of the language will be examined in more detail, with the help of code fragments.

5.2 Default Polka Clauses.

Usually, a class will have two Polka clauses added to it at compile time, called the termination clause, and the delegation clause.

The termination clause waits for the default input stream to close which is denoted by the special Polka keyword **last** appearing in the message part of the clause. This causes the object to terminate after closing its output streams and the streams linked to its inherited objects. At the Parlog level, **last** can be signalled by binding the end of the object's input stream to [].

A programmer can define his own termination clause, which stops the default one from being added :

```

filestore
:
clauses
  last => true .      % user-defined termination clause
:
end.
```

A more complicated user defined clause should also specify actions for closing the streams which go to the inherited objects, and for closing the output streams, since these things will no longer be automatically carried out. How these tasks are achieved will be discussed later.

The delegation clause carries out one of two different operations depending on whether the class uses inheritance. In both cases however, the clause will only be executed if all the other clauses fail to handle a message which has arrived on the default input stream. If inheritance is present, the delegation clause forwards the message to the first inherited object named on the **inherits** line (see section 5.1.1). If inheritance is not used then an extra output stream called 'Error' is added to the object and the message is transmitted along it, back to the user level. This puts the onus on the user of an object to deal with illegal messages, since the object will merely ignore them. One common technique used in Polka programs is to connect the 'Error' stream of every object to a 'print' object which prints the messages it receives on the screen. Note that an 'Error' stream is only added to an object which does not use inheritance.

The programmer can define his own delegation clause, which should be the last clause in the class, and be preceded by a ';'. This ensures that it will be tried only after all the other clauses have been rejected as candidates. However, the compiler generated delegation clause will still be added, but after the user's clauses, which means that it will only be executed if the user defined delegation clause is discarded as

a candidate. The manner in which a delegation clause is coded will become clearer after section 5.8 on output messages and section 5.11 on inheritance.

5.3. Variables.

The 'filestore' example showed how an invisible variable can be declared. Visible variables can be defined in a similar way by using a **visible** declaration list, but this keyword can be omitted, as shown below. Another, more important difference, is that only invisible variable declarations can include initial values.

Both visible and invisible variables are typed, and can be either input state variables, output state variables, input stream variables, or output stream variables. The Polka names for these four types are : state, ostate, istream, and ostream. The distinction between state and stream is useful for the compiler because it can use them to check that Polka operations are being called correctly.

The least used type is ostate, which is usually associated with variables that act as flags. For instance :

```
foo Died ostate
clauses
:
  last => Died = true .      % set 'Died' flag to true
end.
```

This code fragment illustrates how a programmer defined termination clause can set the 'Died' flag to 'true'. At run time, 'foo' would be invoked like so :

```
<- foo( Ins, Died, Error ) .
```

'Died' could then be monitored by another object or predicate. This call also shows how the 'Ins' stream is always the first argument of an invocation, while 'Error' is always the last.

Although visible variables must appear as the arguments of a class, it is not necessary to bind them at invocation time. This makes it possible to program with partially instantiated objects, as in program 5.3.

```

tax  Rate state      % tax rate is in 'Rate'
clauses
  add_tax( Value, NewValue ) =>
      NewValue is Value + (Value * Rate / 100) .

  set_tax( NewR )      =>      Rate becomes NewR .

end.

```

Program 5.3 : A 'tax' class.

The 'tax' class adds tax to a value supplied to it in a 'add_tax' message, and also contains a clause which sets the tax to a new rate. The class could be invoked like so :

```

<- tax( Ins, Rate, Error ) ,
Ins = [ add_tax( 100, Val1 ), add_tax( 200, Val2 ),
        set_tax( 20 ), add_tax( 100, Val3 ), ... ] .

```

This will immediately bind 'Val3' to 120, but 'Val1' and 'Val2' will remain unbound until 'Rate' is bound, because 'tax' was using 'Rate' until the 'set_tax' message.

5.4. The initial Section.

When a class uses an **initial** section, it is placed after the variable declarations and before the **clauses** section. Its general format is a conjunction of calls to Parlog predicates and Polka operations which *may* be executed before any messages are processed, depending on the separator at the end of the conjunction. If the symbol is '&.' then the calls will be executed before the first message is accepted. Alternatively, a '.' separator permits messages to be processed immediately, although they may suspend if they depend on bindings generated by calls in the **initial** section.

Essentially, the purpose of an **initial** section is to enable more complex initialisations of invisible variables to be carried out than is possible during their declaration.

For example, 'filestore' might be :

```

filestore
invisible Files state

initial    initialise( NewFiles ) ,
           Files becomes NewFiles .

clauses
:

code
    mode initialise( ^ ) .
    initialise( [ file( 1, a ) ] ) .
end.

```

The **initial** section calls the **code** section predicate 'initialise' to return an initial value for 'Files'. The '.' separator means that messages may start being processed in the interim but any operations on 'Files' will still suspend until it has a value.

5.5. Clauses.

Until now, each clause has been separated by either the '.' or the ';' search operator, but both of these operators can also have a '&' prefix. '&.' and '&:' have the same parallel and sequential search meaning respectively, but also force all the actions of the clause to have terminated before another message is processed. This is sometimes useful if messages should be dealt with sequentially, and it highlights the fact that normally Polka attempts to maximise parallelism. This means that many goals may be executing concurrently in response to a message, and also that many messages may be processed at once.

The ';' search separator allows if-then-else constructs to be encoded in a longish fashion since the message must be included in both clauses, as in :

```

message( No, Reply )    =>    No == 200 : Reply = stop ;
message( No, Reply )    =>    Reply = continue .

```

This can be rephrased as one clause :

```

message( No, Reply )    =>    No == 200 : Reply = stop
                             else Reply = continue .

```

The else construct takes the place of the second clause. Multiple else parts and extra guards are allowed, like so :

```

message( No, Reply )    =>    No < 200      : Reply = less
                             else No > 200 : Reply = more
                             else Reply = stop .

```

That clause is equivalent to the following three clauses :

```
message( No, Reply ) => No < 200 : Reply = less ;
message( No, Reply ) => No > 200 : Reply = more ;
message( No, Reply ) => Reply = stop .
```

There is also a '&else' construct which is analogous to '&:' :

```
message( No, Reply ) => No < 200 : Reply = less
                        &else Reply = stop .
```

This is the same as :

```
message( No, Reply ) => No < 200 : Reply = less &;
message( No, Reply ) => Reply = stop .
```

5.6. Input Messages.

Only the input of messages on the default input stream has been discussed so far, but if an object requires several input streams, then these are declared in the usual way as visible *istream* variables. However, when input streams are explicitly defined, the 'Ins' stream is no longer added, and it is also necessary to label a message with its associated input stream. A labelled message has the form :

```
< input stream > :: < message >
```

The use of multiple input streams and labelled messages can be seen in the example below :

```
foo Strm1, Strm2 istream      % two input streams
clauses
  Strm2 :: mesg => ...        % input message from 'Strm2'
  :
end.
```

The labelling notation also extends to stream termination. Thus, the fact that stream 'S' has closed is denoted by :

```
S :: last => ...
```

This will also cause the object to terminate, after carrying out the actions specified in the clause body.

Several inputs on a stream at one time are possible, by grouping the messages in brackets like so :

```
Strm1 :: ( msg1, msg2 )    => ...
```

This means that this clause will be chosen if the next input message on 'Strm1' is 'msg1' immediately followed by 'msg2'.

A clause can refer to several input streams, as in the following fragment :

```
Strm1 :: msg1, Strm2 :: ( msg2, msg3 )    => ...
```

This means that these three messages must be on the two streams before the clause is chosen. Multiple inputs are especially useful for representing the non-determinism necessary for coding 'merge' type classes.

When a stream is not mentioned in the input part of the clause, that stream is not examined when a candidate clause carries out input pattern matching.

For encoding classes representing clocks, producers or sources, it is sometimes useful to execute a clause regardless of the input to the object. The **null** keyword can be utilised for this, which is the Polka syntactic equivalent of ignoring messages sent to an object. A simple clock class, which uses **null**, is given as program 5.4. It continually prints a sequence of numbers on the screen until its input stream is set to [].

```
clock
invisible Time state <= 0                % initialise the time
clauses
  null => var( Ins ) :
          write( Time ) &                % write out the time
          sleep(1000) &                  % delay for 1000 ms
          NewTime is Time + 1 &          % increment the time
          Time becomes NewTime &.
end.
```

Program 5.4 : A 'clock' class.

A call to this class might be :

```
<- clock( Ins, Error ).
```

The **null** mechanism offers a way to have an active object independent of input messages, but if the object was totally independent of input then there would be no way to stop it. Thus, **null** merely states that input messages are ignored, but does not stop messages arriving. In fact, divergency can be avoided since all classes have at least one input stream and a termination clause. This means that as long as the

assumption that clauses are selected fairly is true, then a closed message stream will eventually cause termination. Unfortunately, in practice the underlying Parlog system is not fair, and so a program may not stop.

In 'clock', the single clause will write a number every 1000 ms, due to the use of the Parlog 'sleep' predicate which delays the execution by a certain number of milliseconds. The other important built-in predicate in the program is 'var' which tests if a variable is unbound. Here it is being used on the stream variable as a guard test which enforces the fairness assumption. When 'Ins' is bound to [] the guard will fail, which will allow the default termination clause to terminate the object. If the 'var' test was not present, then the [] might not terminate the object.

Issues related to fairness and divergence are discussed in more depth in chapter eight.

The ability to treat message streams as logical variables (as is done in the example above) means that they can be manipulated inside predicates or even used as arguments of messages. This enables many unusual communication and synchronisation techniques to be employed, some of which are discussed in the following chapters.

5.7. Becomes.

Apart from the simple uses of **becomes** already encountered, it can also have an expression on its right hand side, like so :

Time becomes Time + 1

This can be viewed as a shorthand form for a call to the built-in 'is' predicate, followed by a standard **becomes** :

NewTime is Time + 1 ,
Time becomes NewTime

A multiple **becomes** is possible :

A, B becomes 1, 2

which can be rephrased as two **becomes** :

A becomes 1 ,
B becomes 2

In all cases, the changes do not take affect until the processing of the next message.

A useful, if sometimes confusing, alternative notation for **becomes** is **#**. Quite often a new value for a variable is returned by a predicate and this is then stored using **becomes**, as in :

```
extract( Key, File, Ok )    =>  extract_file( Key, F, Files, NewFiles ) :
                                Ok = done, File = F,
                                Files becomes NewFiles ;
```

In the clause, 'extract_file' returns 'NewFiles' which is subsequently used by **becomes**. The **#** version of this is :

```
extract( Key, File, Ok )    =>  extract_file( Key, F, Files, #Files ) :
                                Ok = done, File = F ;
```

This means the same thing as the first clause but overloads the output binding coming from 'extract_file' to also include an implicit **becomes** call.

5.8. Output Messages.

Output streams transmit messages using a send operation, which has the general form :

```
< output stream > :: < messages >
```

For instance :

```
foo Out ostream
clauses
  double( M ) => Out :: ( M, M ) .    % output two 'M' messages
  :
end.
```

The 'double' clause sends two 'M' messages out on 'Out', which is declared to be an output stream. As with input streams, a single message need not have brackets around it. The last keyword can be used to close a stream, as in :

```
last => Out :: last ,
      Error :: last .
```

This closes the 'Out' and 'Error' streams.

An **on** extension can be included to encode a *suspendible* message :

```
delay_message( Ok, M ) => Out on Ok :: M .
```

This means that the clause succeeds immediately but that 'M' is held by the sending object until 'Ok' is bound, and then sent along 'Out'. It is not the same as :

```
delay_message( Ok, M ) => data( Ok ) & Out :: M .
```

This clause suspends until 'Ok' is bound before the message is sent. In this example, it does not matter, but there are cases when such clause suspension can cause the deadlock of the object.

Suspendible messages are discussed in more detail in chapter 8, where they are the central construct behind the separation of synchronisation and functionality in a class.

When an object does not use inheritance, and so is given an 'Error' output stream, error messages can be placed on it in the same way as other output streams. For instance, the following clause can be encoded :

```
Msg => Error :: Msg .
```

If this was last in the **clauses** section and was preceded by a ';', then it would be doing the same job as the default delegation clause.

5.9. I_am.

Sometimes it is necessary to have an object change into another object, or make an object split into several others. These things are possible by using the **i_am** operation, which can specify a replacement class.

The 'unemployed' class uses **i_am** to change into an 'employed' class :

```
unemployed  Name state
clauses
  name( N )      =>  N = Name.
  job( Job )     =>  i_am employed( Ins, Name, Job, Error ) .

end.
```

Program 5.5 : An 'unemployed' class.

The first clause retrieves the 'Name', while the second states that when a 'job' message arrives, the **i_am** causes the replacement of the 'unemployed' object by an 'employed' object. In other words, an 'employed' object is created with the default

input stream, 'Name', the 'Job' variable from the message, and the default error output stream, while the 'unemployed' object terminates.

The 'employed' class is defined as :

```

employed  Name, Job  state
clauses
  name( N )      =>   N = Name.
  job( J )       =>   J = Job.
  sacked         =>   i_am unemployed( Ins, Name, Error ) .
end.

```

Program 5.6 : An 'employed' class.

The 'employed' class is very similar to the 'unemployed' one, but a 'sacked' message replaces it by an 'unemployed' object.

A call to 'unemployed' might be :

```

<- unemployed( Ins, andrew, Error ) ,
Ins = [ name( N1 ), job( refuse_operative ) | Ins1 ] ,

```

Giving :

```

N1 = andrew

```

After the 'job' message, the object will become an 'employed' object. If more messages are sent :

```

Ins1 = [ job( J1 ), sacked | Ins2 ]

```

this gives :

```

J1 = refuse_operative

```

After the 'sacked' message, the object will revert to an 'unemployed' object.

i_am can be used for object splitting by including a clause which contains two or more **i_am**'s. For instance :

```

couple  Man, Woman state
clauses
  separate      =>   i_am single( Ins, Man, Error1 ) ,
                    i_am single( Ins, Woman, Error2 ) ,
                    merge( Error1, Error2, Error ) .
  :
end.

```

This clause splits the 'couple' object into two 'single' objects, both of which share the remainder of the input stream to 'couple'. Also, their new error streams are merged back into the error stream for 'couple'.

The argument of **i_am** can actually be a predicate as well as a class, and so it is possible to use **i_am** to terminate an object, as in the following example :

```

foo Val state
clauses
  terminate( V ) => V = Val ,
                    Error :: last ,      % close the error stream
                    i_am true .         % become a 'true' goal
:
end.

```

The 'terminate' message returns the state of the object and closes the 'Error' stream, before the object is replaced by the goal 'true' which immediately succeeds. If this coding technique is used, then the programmer must ensure that output streams are closed.

i_am should not be used when an object utilises inheritance, since there is no way to transfer the inheritance links of the object to the new instance which **i_am** creates. This weakness in the notation may eventually be remedied.

5.10. The code Section.

The code section need not contain only pure Parlog predicates definitions, since its predicates can be augmented with any of the Polka operations for outputting messages. Program 5.7 illustrates this feature.

```

pulse Out ostream
invisible No state <= 1
clauses
  set( N )      => No becomes N .      % set pulse number
  send_pulses   => sendp( No ) .      % transmit 'No' pulses
code
  mode sendp( ? ) .
  sendp( 0 ) ;
  sendp( N ) <-
    Out :: pulse( N ) &                % output a pulse
    N1 is N - 1 &
    sendp( N1 ) .
end.

```

Program 5.7 : A 'pulse' class.

A call to this 'pulse' class might be :

```
<- pulse( Ins, Out, Error ) , Ins = [ set( 2 ), send_pulses ] .
```

which gives :

```
Out = [ pulse( 2 ), pulse( 1 ) ]
```

The first clause allows the user of 'pulse' to set 'No' to a value of his own choice. If 'set' is not used then 'No' will have a value of 1. The second clause calls the predicate 'sendp' and passes it 'No'. 'sendp' is a recursive predicate which transmits 'No' number of 'pulse' messages out on 'Out'. The line :

```
Out :: pulse( N ) &
```

in the second clause of 'sendp' means that the output is sequentialised and so generates the message order :

```
pulse( 2 ), pulse( 1 )
```

If the line was :

```
Out :: pulse( N ) ,
```

then the message order can be :

```
pulse( 2 ), pulse( 1 )
```

or

```
pulse( 1 ), pulse( 2 )
```

This is possible because the ',' conjunction in the predicate causes messages to be added to the 'Out' stream in parallel, so making their order non-deterministic.

The ability to have Polka send operations in code section predicates means that an object is not limited to a finite output for a given input. For instance, if 'No' is set to a negative number, 'sendp' will output an infinite number of pulses.

In addition to being able to send output messages, the transmission of messages to inherited objects and to oneself is also allowed. These features will be discussed in the next two sections.

5.11. Inheritance.

A class inherits others by including an **inherits** line after its name, like so :

```
a
inherits b, c
clauses
:
end.
```

This states that the 'a' class inherits 'b' and 'c', which means that it can understand all of their messages. As usual, 'a' can have its own clauses and data, which may be totally new, or may act as new definitions for clauses or data in the inherited classes.

If 'b' was defined like so :

```

b
inherits d, e
:
end.

```

Then the inheritance graph for 'a' will be as shown in figure 5.2.

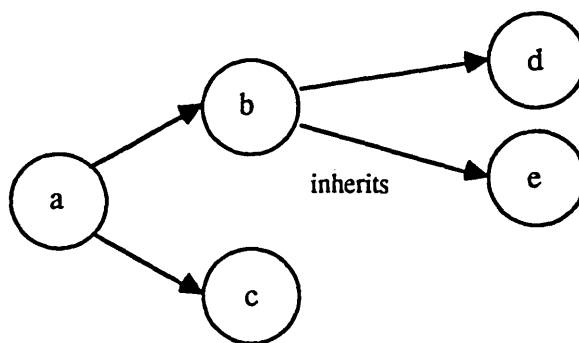


Figure 5.2 : Inheritance graph for 'a'.

The inheritance graph shows the inheritance hierarchy for the classes. This should not be confused with the inheritance hierarchy for an instance of a class, which depends on the instance created. For an 'a' instance, the two hierarchies would correspond because 'a' is at the bottom of them both. However, when a 'b' instance is created, its inheritance hierarchy will be as shown in figure 5.3.

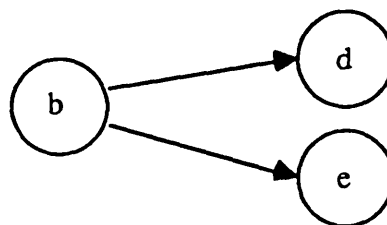


Figure 5.3 : Inheritance graph for an instance of 'b'.

The difference becomes important when we consider the meaning of self communication in section 5.12. In that context, sending a message to self means sending the message to the instance at the bottom of the instance inheritance hierarchy. Which instance this is depends on which class was invoked by the user.

Polka does not permit a class to appear twice on the same **inherits** line, and loops in the inheritors are disallowed. However, the following class inheritance graph is still possible, as illustrated by figure 5.4.

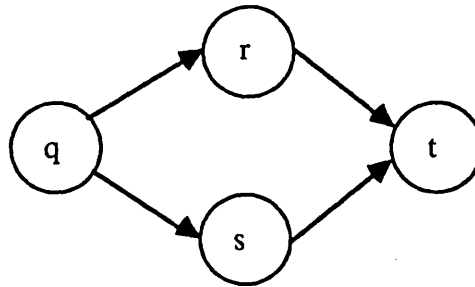


Figure 5.4 : Inheritance graph which inherits 't' twice.

This states that the 'r' and 's' classes both inherit 't', but in Polka this means that they inherit separate instances of the class at run time. In this situation it may be more accurate to show the instance inheritance hierarchy for 'q' as the tree in figure 5.5.

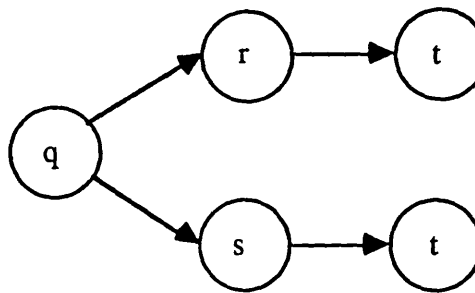


Figure 5.5 : Instance inheritance tree for 'q'.

In future, whenever inheritance is represented graphically, it shall be made clear whether a class inheritance graph or an instance inheritance tree is being shown.

It is not possible for an instance to alter its list of inherited objects at run time, although in earlier versions of Polka [Davison 1987a], this kind of dynamic inheritance was possible. The mechanism was eventually removed because it was found to be rarely needed, while being complex to use and complicated to implement.

To explain inheritance in more depth, two classes will be defined : a clock, and a random number generator. These will both be inherited by a 'nasty_clock' class.

The clock class will be able to accept two types of message :

and

```

display( T )
set_alarm( AlarmTime, Bell )

```

'display' retrieves the current time from the object and binds it to 'T', while 'set_alarm' specifies a time at which 'Bell' will be bound to 'ring'. The class definition is given in program 5.8.

```

clock
invisible Time state <= 0,
          Alarm state <= a( 0, _ )
clauses
  display( T )    =>    T = Time .

  set_alarm( AlarmTime, Bell )    =>
                                Alarm becomes a( AlarmTime, Bell ) .

  null    =>    var( Ins ) :
                chk_alarm( Alarm, Time ) &                % check alarm
                sleep( 1000 ) &
                Time becomes Time + 1 &.                % increment time
code
  mode chk_alarm( ?, ? ) .
  chk_alarm( a( AT, Bell ), T ) <-                % 'ring' bell if time is up
    T ≥ AT : Bell = ring ;
  chk_alarm( Alarm, T ) .
end.

```

Program 5.8 : A 'clock' class with an alarm.

A call to this class might be :

```
<- clock( Ins, Error ), Ins = [ display( T1 ), set_alarm( 6, Bell ) | _ ] .
```

which gives :

```

T1 = 0
Bell = ring          % after a short time

```

The first clause of 'clock' retrieves the current time, while the second sets the 'Alarm' variable. This variable consists of a term containing the time when the alarm is to go off, and a variable which acts as the output for the alarm bell. The third clause is chosen when there is no input, and calls 'chk_alarm' to check the alarm term so that when its time argument is less than 'Time', its 'Bell' variable will be bound. The clause also waits for a second before incrementing the 'Time' state variable.

A random number generator will be able to accept two sorts of message :

```

display( S )
and
rnd( R, N )

```

'display' returns the current seed value in 'S', while 'rnd' takes an integer 'R' and returns a random integer in 'N', which is between 1 and 'R'. The generator is defined in program 5.9.

```

random
invisible Seed state <= 13
clauses
  display( S ) => S = Seed .

  rnd( R, N ) => gen_n( Seed, R, N ) ,      % obtain a value for 'N'
                gen_ns( Seed, #Seed ) .    % generate a new seed

code
  mode gen_n( ?, ?, ^ ) .                  % N is S mod R + 1
  gen_n( S, R, N ) <-
    mod( S, R, Temp ) ,
    N is Temp + 1 .

  mode gen_ns( ?, ^ ) .                    % NS is (125 * S + 1) mod 4096
  gen_ns( S, NS ) <-
    Temp is 125 * S + 1 ,
    mod( Temp, 4096, NS ) .
end.

```

Program 5.9 : A random number generator.

A typical call might be :

```
<- random( Ins, Error ), Ins = [ display( S ), rnd( 10, N1 ), rnd( 9, N2 ) ] .
```

which gives :

```
S = 13, N1 = 7, N2 = 8
```

Now that 'clock' and 'random' have been defined, the 'nasty_clock' class which inherits them both can be specified in program 5.10.

```

nasty_clock
inherits clock, random
clauses
  rnd( R, N ) => random :: rnd( R, N ) .

  set_alarm( AlarmTime, Bell ) => random :: rnd( AlarmTime, LessTime ) ,
                                super :: set_alarm( LessTime, Bell ) .
end.

```

Program 5.10 : A 'nasty_clock' class.

A typical call might be :

```
<- nasty_clock( Ins, Error ),
   Ins = [ display( T1 ), rnd( 17, R1 ), set_alarm( 14, Bell ) | _ ] .
```

which gives :

```
T1 = 0,    R1 = 10
Bell = ring      % after a short while
```

The inclusion of an 'Error' argument in the call to 'nasty_clock' may seem odd since 'nasty_clock' will delegate unrecognised messages to either of its inherited objects. However, the invocation must still include an 'Error' stream because errors will be output from both 'clock' and 'random'. Rather than have a separate error stream for each inherited object, which could lead to a large number of arguments, the error streams for each object are internally merged into a single stream which becomes the 'Error' stream.

In theory, 'nasty_clock' should be able to process 'display' and 'set_alarm' messages because of its inheritance of 'clock', and 'display' and 'rnd' messages because of 'random'. However by default, all messages which are not explicitly processed by clauses in 'nasty_clock' will be delegated to the first object named on its **inherits** line, which is 'clock' in this case. Also, once a message has been forwarded, it can not backtrack and try another inherited object if the object chosen initially fails to understand the message. This restriction is very similar to the committed choice nature of the rest of Polka and Parlog, but can be inconvenient. One way around it is to broadcast the message to all the inherited objects, which is described below.

For 'nasty_clock', this order of delegation means that 'display' and 'rnd' messages intended for the 'random' inherited object will always go to 'clock' unless explicitly redirected. This direction requirement is the motivation behind the first clause of 'nasty_clock' - it captures a 'rnd' message and then routes it to 'random' by using :

```
random :: rnd( R, N )
```

This is an example of how inherited objects are communicated with, and the more general form of the operation is :

```
< inherited object name > :: < messages >
```

The use of '::' as the operator is no coincidence, since it is meant to suggest that sending messages to inherited objects is very like sending messages along output

streams. In fact, all of the various forms available to output messages are also at the disposal of delegated messages. This means that **last** can be utilised :

```
clock :: last
```

This will close the message stream between the object and 'clock', which in turn will cause 'clock' to terminate. Operations like this make up the default termination clause for 'nasty_clock', which looks like :

```
last => clock :: last, random :: last .
```

This states that when the input stream for 'nasty_clock' closes, then 'nasty_clock' terminates after first closing the input to its two inherited objects (and so terminating them).

Suspendible messages are possible, as in :

```
clock on T :: set_alarm( T, Bell )
```

This causes a 'set_alarm' message to be forwarded to 'clock' when 'T' becomes bound.

'set_alarm' messages sent to 'nasty_clock' would normally be delegated to the correct inherited object, but the second clause of 'nasty_clock' overrides this. Instead, a 'rnd' message is transmitted to the 'random' inherited object, which will bind 'LessTime' to a number between 1 and 'AlarmTime'. 'LessTime' is then used as the time argument of the 'set_alarm' message which is sent to the **super** object. This is the default name for the first object on the **inherits** line, which is 'clock'. The following could have been written instead :

```
clock :: set_alarm( LessTime, Bell )
```

It may now be clear why this class is called 'nasty_clock' : the alarm will probably go off early.

Both inherited objects can handle 'display' messages, but at the moment the default action is to delegate such a message to 'clock'. However, the programmer may want to send a 'display' message to 'clock' and 'random', so that both inherited objects display something. One way to program this is :

```
nasty_clock
inherits clock, random
clauses
```

```

display( T )    =>    all :: display( T ) .      % broadcast 'display'
                  :
                  :
end.

```

all causes its associated message to be broadcast to every inherited object, but in doing so it causes any variables in the message to be shared between all the inherited objects. This may be useful if the inherited objects wish to communicate with each other, but it is more likely to be a nuisance, as here. One of the two objects will bind 'T' causing the other object to probably fail because the bound value for 'T' from the first inherited object is unlikely to match the value in the second object. Generally, the simple version of **all** is only useful if fully bound messages are being broadcast.

One way round the shared variable problem is to use the **returning** extension of **all**. With this, the 'display' clause would be :

```
display( T )    =>    all :: display( T ) returning T .
```

This can be thought of as being shorthand for :

```

display( T )    =>    clock :: display( T1 ) ,
                      random :: display( T2 ) ,
                      response( [ T1, T2 ], T ) .

```

Each inherited object now gets a message with an unique variable and each of these unique variables are passed in a list to 'response'. Program 5.11 gives the default definition for 'response'.

```

mode response( ?, ^ ) .
response( [ Val | Rest ], Ans ) <-          % test head
  data( Val ) :                             % value is bound
  Ans = Val .                               % return value
response( [ V | Rest ], Ans ) <-
  response( Rest, Ans ) : true .            % test tail

```

Program 5.11 : A 'response' predicate.

This predicate will return the first variable in the list which becomes bound, and also illustrates the usefulness of deep guards and or- parallelism. The first clause tests the head of the list and succeeds when it is bound, but in or- parallel the tail of the list is examined by 'response' in the guard of the second clause. This enables the 'data' test to be applied to every element in the list in or- parallel, and permits the original 'response' goal to succeed when any of the tests succeed.

If this definition is not suitable, the programmer can redefine 'response' in the **code** section of the class which contains the **all** operation.

One problem with the broadcasting of messages occurs when they are not recognised. An object will always output an unrecognised message on its 'Error' stream if it can not delegate it to an inherited object. Unfortunately, no response is sent back to the sender of the message, since none of the variables in the message are bound. For instance, this means that if a 'print' message was broadcast from 'nasty_clock' like so :

```
print( T ) => all :: print( T ) returning T
```

Then its 'Error' stream would eventually be bound to :

```
[ print( T1 ), print( T2 ) | _ ]
```

The two 'print' messages come from 'clock' and 'random'.

If nothing is done with these messages, then the 'T' variable in the original message will never be bound, and 'response' will remain suspended forever. This would not affect the processing of other messages sent to 'nasty_clock' but it would leave a useless 'response' process in the system.

One simple solution is to define a 'response' predicate which waits for bindings for a fixed time before succeeding without binding its output variable. The obvious problem is how long to wait.

Another solution involves imposing a fixed structure on all messages, with a reserved argument for status information, which could be used to signal failure back to all. For example, a message 'M' would be represented as :

```
message( M, Status )
```

This approach has been investigated, but if the extra structure is to be made transparent to the user and programmer, then the compiler and run time system become very complex.

A solution used in some Polka programs is to pass all error messages to a 'print' object which prints out the error and then deconstructs each term and binds any variables it finds to some dummy value. This is not particularly satisfactory since the binding may be incorrect and cause failure back in the original object.

The **returning** extension can actually return several variables :

```
evaluate( No, X, Y ) => all :: evaluate( No, X, Y ) returning X, Y
```

If the object with this clause inherits 'a' and 'b', then this is equivalent to :

```

evaluate( No, X, Y ) =>  a :: evaluate( No, X11, Y11 ),
                        b :: evaluate( No, X21, Y21 ),
                        responseL( [ [ X11, X21 ], X ], [ [ Y11, Y21 ], Y ] ).

```

The predefined 'responseL' predicate is given in program 5.12.

```

mode responseL( ? ).
responseL( [] ).
responseL( [ [ VList, V ] | Vrs ] ) <-      % a list of replies
    response( VList, V ),                  % deal with each in 'response'
    responseL( Vrs ).

```

Program 5.12 : A 'responseL' predicate.

As with 'response', this definition can be changed by including a different 'responseL' predicate in the code section of the class.

One problem with inherited objects is due to the handling of their visible variables by the inheriting object. For example, if all the variables in 'clock' and 'random' were defined as being visible, then the initial call of 'nasty_clock' would have to supply values for those variables. The problem is in deciding which argument positions of 'nasty_clock' belong to which arguments of the inherited objects. The answer is to consider the tree of inherited objects. 'nasty_clock' will be the bottom object, while the inherited objects will be above it in the tree, in a left to right order dictated by their position in the *inherits* line of 'nasty_clock'. If, for the purposes of this example, 'clock' also inherits 'random', then the instance inheritance tree, with visible variables, looks like figure 5.6.

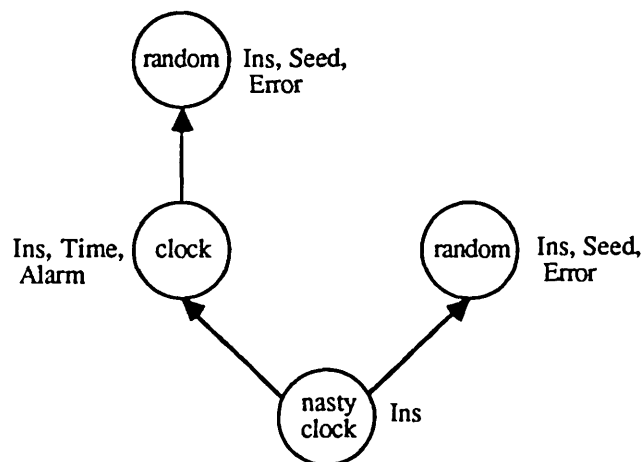


Figure 5.6 : Instance inheritance tree for 'nasty_clock'.

The 'Ins' variables are the visible default input streams for each object but most of these can be ignored since they will be unavailable when an object is inherited. Also, the 'Error' output streams for inherited objects are always merged into a single output stream, and so 'nasty_clock' will have only one 'Error' stream. Of the remaining variables, the call to 'nasty_clock' supplies arguments for its inherited objects in a depth first, left to right order with the exception that the visible variables of the bottom object come first and the 'Error' stream comes last. Thus the invocation structure is looks like figure 5.7.

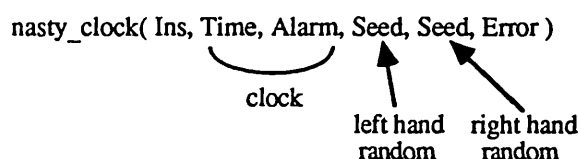


Figure 5.7 : Invocation structure for 'nasty_clock'.

The actual call will be slightly different since unique variable names must be given to each argument.

With larger inheritance trees, it will become harder to decide on the argument ordering, but fortunately Polka contains a programming environment tool called 'invocation', which will print out the correct order. For instance, the call :

```
<- invocation( nasty_clock, full ) .
```

will print :

```
nasty_clock( Ins, Time__clock, Alarm__clock,
             Seed__clock__random, Seed__random, Error )
```

The variable names for the inherited arguments are specially constructed to reflect their original class, and its position in the inheritance tree.

In addition, there is one other way of invoking classes, called the *named* form where arguments are associated with variable names, but more details on this can be found in the Polka user manual [Davison 1988b].

The main lesson of this complexity is that class designers should attempt to keep the number of visible variables to a minimum.

5.12. Self Communication.

As discussed earlier, self communication in an instance does not necessarily refer to the object itself, but to the bottom object in the tree of inherited objects of which the object sending the self communication is a part. This idea can be visualised by looking at the inheritance tree for the 'a' instance in figure 5.8.

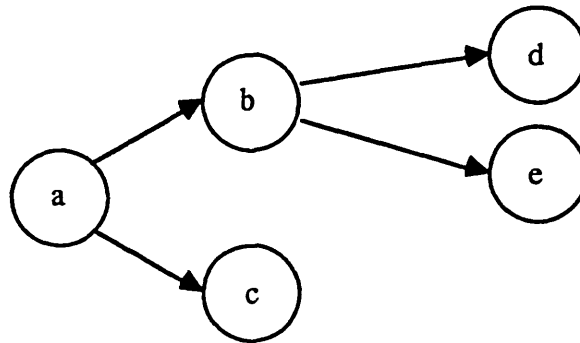


Figure 5.8 : Instance inheritance tree for 'a'.

A self reference in the 'b', 'c', 'd' or 'e' objects will actually mean 'a'. This feature can be used to make clauses redefinable, since they can be written in terms of messages to self, which are processed by different objects depending on the inheritance tree. For example, if the 'a' class is inherited by 'x' and 'x' is invoked, then all self messages will now go to 'x' which may redefine the meaning of any of the messages.

An example of self communication is illustrated by program 5.13, which represents a simple database class.

```

dbase
invisible Db state <= []                                % Db is a list
clauses
  assert( Cl )      =>  Db becomes [ Cl | Db ] .         % add a Cl
  retract( Cl )     =>  retract( Db, Cl, #Db ) .         % remove a Cl
  replace( Old, New ) =>  self :: retract( Old ) &        % 'replace' uses 'retract'
                        self :: assert( New ) .           % and 'assert'
code
  mode retract( ?, ?, ^ ) .    % only mode shown
  :
end.

```

Program 5.13 : A database class.

The new feature is shown in the third clause where the action to deal with the 'replace' message is carried out by sending a 'retract' and an 'assert' message to self. The syntax is like that used for an output message, but the '::' operation has a **self** label. As usual a class which inherits 'dbase' will be able to redefine the meaning of 'assert' and 'retract' messages, but now their redefinition will also alter the meaning of the 'replace' message.

The third clause could also be written as :

```
replace(Old, New)  =>  self :: retract(Old) ,      % and- parallel conjunction
                    self :: assert(New) .
```

so that the two messages are sent in any order. However, the order of 'retract' and 'assert' are likely to be important, and the '&' will enforce that they are sent in a certain sequence. The nature of the implementation means that this ordering will be the same when the messages arrive at the bottom object. However, this does not normally guarantee that they will be processed in the same order, although in this example there is an implicit constraint on their processing because they both manipulate 'Db'.

In fact, the only way to guarantee the processing order of messages is by using the **on** suspendible message extension of **self**. For instance, it might be necessary in 'dbase' because extra cache storage may be added by a class which inherits 'dbase'. This would make it possible for a 'retract' message to be delayed while the cache was being searched, but allows an 'assert' message to immediately add its fact to the database. When 'retract' finally reaches 'Db' it might then remove the newly asserted fact rather than the one that was intended. To avoid this, 'dbase' could be rewritten as program 5.14.

```
dbase
invisible Db state <= []
clauses
  assert( Cl )          =>  Db becomes [ Cl | Db ] .      % as before

  retract( Cl, Done )   =>  retract( Db, Cl, Done, #Db ) . % 4 argument 'retract'

  replace( Old, New )   =>  self :: retract( Old, Done ) ,
                          self on Done :: assert( New ) . % use of self-on form

code
  mode retract( ?, ?, ^, ^ ) .      % only mode given
  :
end.
```

Program 5.14 : A database class with suspendible messages.

A 'retract' message now contains a 'Done' flag, which is set to 'done' by the 'retract' predicate once it has finished. Also, the third clause of 'dbase' uses a suspendible message :

```
self on Done :: assert( New )
```

'Done' is the flag set on completion of the 'retract' message. The clause still sends a 'retract' and 'assert' message to self but the 'assert' message will now be delayed until the 'Done' variable has been bound. This approach guarantees that the 'assert' message will be processed after the 'retract'.

Further discussion of suspendible messages can be found in chapter 8.

There is another way of sending messages to their originating object, but it has none of the flexibility of *self*. To illustrate this technique, the original third clause of 'dbase' can be written as :

```
replace( Old, New )    =>    Ins becomes [ retract( Old ), assert( New ) | Ins ] .
```

By updating 'Ins' in this way, 'dbase' is forced to accept the 'retract' and 'assert' messages before any other input. This approach only works properly if the programmer knows that 'retract' and 'assert' will *not* be redefined by inheritors of 'dbase'. Also, it suffers from a lack of control over the processing order even though the arrival order is enforced.

The *self* operator can also use *last*, but this only closes the self stream for the sending object in the inheritance tree that makes up the instance, and so is not particularly useful. Fortunately, the system ensures that self streams are closed when an object terminates and so the programmer should not worry about closing them using *last* messages.

5.13. 'Higher level' Parlog.

By 'higher level' Parlog, we mean a set of syntactic constructs on top of Parlog which allow more concise and easily understandable representations for common Parlog programming styles. Interestingly, it was found that the types of program which would benefit most from extra syntactic operations were those that used processes. It was a short step from this to the Shapiro and Takeuchi model [Shapiro and Takeuchi 1983], and the relationship between an object and a process described in section 3.2.3.1.

Their model is most useful as a way of viewing concurrent LP programs as classes. For example, the 'merge' and 'filestore' predicates can be seen as simple classes, which do not utilise inheritance, or self communication. The advantage of perceiving programs in this way is that a language which supports the OOP paradigm, will also hide much of the verbosity of process based programs, such as recursion, argument copying, and stream manipulation. Polka falls into this language category, and the 'filestore' class of section 5.1.2. shows that a Parlog predicate can be rephrased in Polka. However, the Polka compiler adds mechanisms to programs for inheritance, self communication, default termination, and other OOP features, but none of these are desirable for a program which is only intended as syntactic sugar for a Parlog predicate. Thus, Polka allows the keyword **parlog** to be included at the start of a program which disables these features. This permits the 'merge' predicate to be represented by program 5.15.

```

parlog merge
  X, Y istream, Z ostream
clauses
  X :: El      =>  Z :: El .
  Y :: El      =>  Z :: El .
  X :: last    =>  Y = Z .
  Y :: last    =>  X = Z .
end.

```

Program 5.15 : A 'merge' class.

The four clauses correspond to the four clauses of the Parlog 'merge', and clearly shows the input streams and output stream, and how they are connected. A call to the Polka 'merge' is carried out in exactly the same way as in the Parlog version :

```
<- merge( [1, 2, 3 | X ], [ a, b | Y ], Z ) .
```

This may give :

```
Z = [ 1, a, 2, 3, b | _ ]
```

The **parlog** keyword prevents the addition of inheritance and self communication mechanisms to a program, which means, for instance, that the program can not use **self** and **super**. Also, default clauses are also not inserted, so forcing the programmer to write his own termination and error clauses. One important benefit of this approach, is that the compiled form is almost as efficient as directly coding the program in Parlog, but the Polka version is also more concise, error free, and understandable. It is more concise because such things as process reduction and argument copying are hidden. There are less errors because Polka carries out more compile time analysis than Parlog, such as type checking of variables. Also, the reduction in text means that the number of errors will be reduced, especially in a

program whose Parlog version recursively calls itself with just one or two arguments changed among many. Polka code is more understandable because the syntactic conventions correspond closely to the Shapiro and Takeuchi model, which is the conceptual representation behind many Parlog programs.

Chapter 6. Meta Level Programming.

This chapter will examine the concepts behind the meta level programming features of Polka, and describe four examples which benefit from these features.

6.1. What is Meta Level Programming?

Meta level programming in Polka is closely related to the work of Bowen and Kowalski [Bowen and Kowalski 1982], except that our meta level and object level languages are both Polka. These two levels are combined via a predicate, written in Parlog, which axiomatises the operational semantics for a subset of Polka. This 'odemo' predicate defines the relationship between a Polka class and an invocation, or instance, of that class. In outline, it is :

```
mode odemo( ?, ?, ^ ).
odemo( TermClass, Invocation, Out )
```

The class argument of 'odemo' is called a term class, because at the meta level it is represented as a term, while the object created when 'odemo' is executed is called the term object. Also, if an object has invoked 'odemo' then it is known as the meta object.

The invocation argument consists of a list of values for creating an instance of the term class. These values, which are terms at the meta level, will be treated by the 'odemo' predicate as Polka streams and states.

'odemo's third argument is 'Out' which can output a list of terms containing information about the execution. 'Out' can also be thought of as a message stream between the object level and the meta level. This stream is utilised at the object level by means of a reflection mechanism [Kowalski 1979; Weyhrauch 1980], which allows subgoals to be passed to the meta level as terms. These terms can also be viewed as messages, which leave 'odemo' on the 'Out' stream. Reflection can be an implicit operation, as when an undefined predicate is called at the object level, but it can also be carried out explicitly, via the meta message operation, which will be described shortly.

It is possible to call 'odemo' from a Parlog predicate, but because the definition of a term class must appear in an ordinary class, it is rare for 'odemo' to be used outside of an object.

An important consideration for the representation of the object level language at the meta level is how to name the various entities of the object level language. A

convenient technique is to let all the constants, literals, terms and functors be named by themselves. The remaining problem is how to represent object level variables, which can not strictly be treated as meta level variables, which are more powerful, since they can range over formulae of the object level. Eshghi [Eshghi 1988] has discussed some of the issues concerning this naming problem, and a variant of his solution is used here. Object level variables will be named by meta level constants, which will be listed next to the clause or goal in which they appear, so allowing them to be distinguished from ordinary constants. Further refinements must be made when manipulating clauses, since they can contain both global and local variables, but these will be explained later. An object level variable will be named by appending 'xx' to its front, so turning the object level variable 'Ins' into the 'xxIns' constant at the meta level, for instance. This type of naming solves the problem of mixing meta level and object level variables, as seen by a call to 'odemo' of the type :

```
<- odemo( TermClass, [ [ message( X ) ] ], Out ) .
```

which is different from :

```
<- odemo( TermClass, [ xxX ] @ [ [ message( xxX ) ] ], Out ) .
```

The first example is an instantiation of a term class with a invocation argument list which contains a single list with a term that uses a meta level variable 'X'. This may be bound to anything, including a constant, a term or a variable. In the second call, the invocation argument is a term of the form :

```
< object level variable list > @ < invocation argument list >
```

The invocation argument list now contains a term with an object level variable 'X'. If the variable list was not present then the 'xxX' would be interpreted as an ordinary constant rather than as the name of an object level variable.

Reflection means 'odemo' need only simulate a subset of the Polka language, while the rest can be handled by passing goals to the meta level. This allows the minimal functionality of term classes to be extended at the meta level in different ways, one of which is described in section 6.6. The current version of 'odemo' allows a term class to have most of the functionality of an ordinary class, including visible and invisible variables, and- and or- parallelism, message operations and an initial section. Unfortunately, a term class does not have a code section, although this will be remedied in the future. At present, a goal will try to use a Parlog predicate defined in the class where the term class was defined. If there is no suitable definition there,

then the goal will try to use a predicate defined at the system level. If that does not contain a definition, then the goal is passed to the meta level.

Inheritance and self communication are not supported, and no default clauses are added. Also, no 'Error' stream is added by the system, although there is a default input stream called 'Ins'.

The major extension to the language for term classes is the **meta** message operation, which looks like :

```
meta :: Message
```

This can appear in the body of a clause, and will send a message :

```
Message
```

out on the 'Out' stream of the 'odemo' call . Multiple messages in a single **meta** operation are also possible.

The **meta** operation can include an **on** extension, such as :

```
meta on Var1 :: Message
```

which will mean that 'Message' is not transmitted until the suspending variable 'Var1' becomes bound.

As would be expected :

```
meta :: last
```

causes the 'Out' stream of the 'odemo' call to be closed.

'Out' is generally used only for **meta** messages, although unrecognised predicate calls are also sent out on it, in the form :

```
goal( Call )
```

An interesting feature is that a term object can call 'odemo', which means that hierarchies of interpreted term objects can be created.

The utility of 'odemo' for controlling term objects is increased by incorporating ideas from the Parlog metacall. As explained in Chapter 2, the metacall uses 'Control' and 'Status' streams to influence and monitor the execution of Parlog goals, and the introduction of these mechanisms into 'odemo' makes it :

```
mode odemo( ?, ?, ^, ? ) .
odemo( TermClass, Invocation, Status, Control )
```

'TermClass' and 'Invocation' are as previously defined, but now 'Out' has been replaced by 'Status', which still acts as an output stream for messages from the object level, but now has a more significant role as a means of passing execution information to the meta level. New 'Status' stream messages include *suspend*, *continue*, and *stopped*, which are responses to *suspend*, *continue*, and *stop* messages being sent into 'odemo' along the 'Control' stream.

At present, the four argument 'odemo' only supports the control stream 'stop' message. This stops the execution of the term object, and puts a 'stopped' message onto the status stream when this has been achieved. However, this restriction will not cause any problems in the following discussion, since the three argument 'odemo' is sufficiently powerful to illustrate the OOP aspects of term classes.

6.2. The Uses of Meta Level Programming.

There are four major benefits of being able to use meta level programming techniques :

1. Term classes can be easily manipulated.
2. Term classes can be treated as messages.
3. Global information can be reduced.
4. Class objects can be encoded.

All of these benefits are due to the fact that term classes are represented as terms, which makes them first order citizens of the Polka language. The consequence of this is that a term class can be assigned to a variable, used as an argument of a predicate, passed as a message between objects, and generally manipulated to a much greater extent than a class which is represented as a Parlog predicate. For instance, clauses and variables can be adding and deleting at run time, which are logical operations, since only a term is being altered. In fact, predicates are included in Polka for these operations, and a number of them are discussed in the following sections. A complete list of meta level predicates can be found in the user guide for Polka [Davison 1988b].

An important property of a term class is its name, which is local to the object that is using it, unlike normal classes, which have globally visible names. Thus, if an ordinary class is replaced by a term class, the problem of its global name is removed, and in a system of classes, much of the global information can be removed in this way. Unfortunately, there will still have to be at least one class with a global name, to

act as the depository for the term classes. A more realistic approach is to have a number of depositories for term classes.

As discussed in chapter 3, a class can usually be thought of as a template or behaviour definition from which instances, or objects, are created. This implies that a class is not itself an object because it is not an instance of another class. However, this need not be the case, as can be seen in Smalltalk80, where a class is an object [Goldberg et al. 1983]. A *class object* still contains the behaviour definition for its instances, but its main attributes are that it has its own state and computation. For instance, this permits a class object to store details about its instances, such as their current status. It is also possible for a class object to manipulate its instance behaviour definition, so altering the structure of the instances which it creates.

In Polka, a class object can be represented by an ordinary class which contains a term class definition. The term class represents the instance behaviour definition for the class object, and at run time the term objects of the class can be monitored and controlled by using 'odemo'.

In the following sections, four applications of term classes in the areas of program transformation, and class functionality will be explained. Some of the other uses of term classes not discussed include meta level reasoning for search and planning problems [Sacerdoti 1974], hypothetical worlds [Brachman and Schmolze 1985], and abstraction mechanisms, such as control abstractions [Bishop 1986]. Also, since new term classes can be created at run time, a network of objects can dynamically evolve, which is an important feature of open systems [Hewitt 1986].

6.3. Program Transformation.

This example will demonstrate how term classes allow greater flexibility in the ways that classes can be modified at run time. An ordinary object can change its state at run time, and may also invoke other objects, but this potential for change must be 'hard wired' in at compile time by including these actions in its clauses. Also, there is no way to arbitrarily alter an ordinary class, via Prolog-like 'assert' and 'retract' operations, for instance. However, such operations are possible on a term class, and can be explained as term manipulation, rather than as meta logical changes. An example of this is seen in the 'e_factorial' class given below, which invokes a term class for the factorial function when it receives the message :

```
f( Num, Val )
```

Using the term class, 'Val' will eventually be bound to the factorial of 'Num'. To calculate this takes a time proportional to the size of the 'Num' number, which is wasteful especially if the same factorial has already been calculated. Thus, after each new factorial calculation, 'e_factorial' modifies its term class to include a new clause for returning the calculated value. A future term object can then carry out an or-parallel search to find previous factorials in constant time before resorting to a more expensive calculation for new factorials. This approach can be viewed as lemma generation [Sterling and Shapiro 1987].

'e_factorial' is defined by program 6.1.

```
e_factorial
invisible Fact state

initial tClass( efact, EFact ),      % retrieve the 'efact' term class
        Fact becomes EFact .

clauses
  f( Num, Val ) =>
    odemo( Fact, [ [ f( Num, Val, Ass ) ] ], Out ),      % produce factorial
    do_assert( Ass, Fact, #Fact, Num, Val ).             % update term class

  print      =>      pprint( Fact ).                     % print term class

termClass
  efact      % definition for 'efact'
  clauses
    f( 0, Val, Ass ) => Val = 1 ,                        % factorial of 0
                        Ass = no_assert ;

    f( N, Val, Ass ) => Ass = yes_assert ,                % factorial of N
                        Val is N * SubVal ,
                        N1 is N - 1 ,
                        Ins becomes [ f( N1, SubVal, _ ) | Ins ] .

  last      => true .

end

code
  mode do_assert( ?, ?, ^, ?, ? ) .
  do_assert( no_assert, F, F, Num, Val ) .                % no change
  do_assert( yes_assert, F, NF, Num, Val ) <-             % add a clause
    cassert( F, NF, 1, [ xxIns ], [ xxV, xxAss ],
      ( xxIns :: f( Num, xxV, xxAss ) =>
        xxV = Val, xxAss = no_assert ), '.' ) .

end.
```

Program 6.1 : An evolving factorial.

The structure of the class is a little different from normal since it contains a new section for term class definitions. The `termClass` section starts just after the clauses of the class but before the `code` section (if one is present). The scope of the variables in a term class are confined to be between its name and the keyword `end`, and none of the variables from the surrounding class or other term classes are visible to it. Aside from this a term class is very like an ordinary class.

'`e_factorial`' has one term class called '`efact`', which utilises a standard recursive algorithm for calculating factorial, but includes an extra '`Ass`' argument in its message part. '`Ass`' is used to indicate if the solution required a recursion and so should be asserted into the term class as a new clause.

As mentioned above, every term class is a first order citizen of the language, and so can be manipulated in a similar way to a term. However, to do this the term class data structure must first be obtained, and this is achieved by calling the '`tClass`' predicate which takes the name of the term class and returns its representation. In '`e_factorial`', this task is carried out by the `initial` section which uses '`tClass`' to retrieve the encoding of '`efact`' and then stores it in '`Fact`'. The purpose of '`Fact`' is to hold the most up to date version of the term class which is gradually changing during the computation of '`e_factorial`'.

In the `clauses` section, the arrival of a

```
f( Num, Val )
```

message causes the object to invoke '`odemo`' using the term class in '`Fact`'. When '`odemo`' finishes, the '`Val`' value is returned to the message sender and '`do_assert`' is called. This does nothing if '`Ass`' is bound to '`no_assert`', but if bound to '`yes_assert`', then '`cassert`' is called. '`cassert`' is the Polka built-in predicate for asserting a clause into a term class, and is defined in outline as :

```
mode cassert( ?, ^, ?, ?, ?, ? ).
ccassert( < term class >, < new term class >,
          < clause position >, < global variable list >,
          < local variable list >, < clause >, < separator > ) .
```

The clause asserted has the form :

```
Ins :: f( Num, V, Ass ) =>  V = Val ,
                          Ass = no_assert .
```

This will become the first clause in the term class and will be tried in or- parallel with the other base cases before the recursive clause. Two arguments are present for variables in '`cassert`' because a clause can contain both global and local variables.

Global variables are ones defined in the variable sections at the top of a term class, while local variables are used only in the clause. A variable list will be of the form :

```
[ xxX, xxIns, ... ]
```

where 'xxX' is the meta level constant representing the object level variable 'X'. In this program, the only global variable is 'Ins', which is the default input stream, while the local variables are 'V' and 'Ass'. In addition, the 'Num' and 'Val' variables are used as values in the clause. After 'do_assert' is finished, the new factorial class is stored in 'Fact' and will be used next time a message arrives.

The second clause of 'e_factorial' calls the Polka built-in predicate 'pprint' which prints out the term class in 'Fact'.

A typical execution of 'e_factorial' is :

```
<- e_factorial( Ins, Error ),
Ins = [ f( 6, V1 ), f( 3, V2 ), f( 4, V3 ), f( 6, V4 ), print | Ins1 ] .
```

which produces the bindings :

```
V1 = 720
```

```
V2 = 2
```

```
V3 = 24
```

```
V4 = 720
```

and prints :

```
efact   Ins istream
clauses
  Ins:: f( 4, _0, _1 )   =>   _0 = 24 ,           % new factorial of 4
                           _1 = no_assert .

  Ins:: f( 2, _0, _1 )   =>   _0 = 2 ,             % new factorial of 2
                           _1 = no_assert .

  Ins:: f( 6, _0, _1 )   =>   _0 = 720 ,           % new factorial of 6
                           _1 = no_assert .

  Ins:: f( 0, _0, _1 )   =>   _0 = 1 ,             % 0 factorial
                           _1 = no_assert ;

  Ins:: f( _0, _1, _2 )  =>   _2 = yes_assert ,     % N factorial
                           _1 is _0 * _3 ,
                           _4 is _0 - _1 ,
                           Ins becomes [ f( _4, _3, _5 ) | Ins ] .

  Ins:: last             =>   true .

end.
```

Local variables in the clauses are represented as '<number>'.

This print out shows the introduction of three new base cases into 'efact' for the factorials of 4, 2 and 6.

During the execution, the first factorial of 6 takes a perceptible longer time to be evaluated than when it is next calculated, which is as expected.

6.4. Load Balancing.

A key advantage of multiprocessor hardware over uniprocessor hardware is that, in theory, programs should run more quickly by being decomposed and executed in parallel on separate processors. The practical drawbacks of this are related to how difficult it is to decompose a program into logically independent parts, or at least into parts which rarely communicate. The obvious solution to this is to force the programmer to specify how his program should be decomposed and which processors the parts should migrate to. A simple example of this idea is presented in this section, in order to illustrate how term classes can help.

When the 'fact_div' program defined below is given a factorial to compute, it is also given a list of nodes which can execute the computation. 'fact_div' will then generate 'fact' term classes which will calculate parts of the factorial, and these are passed to the nodes to be executed. More specifically, 'fact_div' receives messages of the type :

$$f(\text{Num}, \text{Val}, [N1, N2, \dots])$$

The intention is that 'Val' will be assigned the factorial of 'Num' by 'fact_div'. The third argument contains a list of shared variables, which act as message streams to other nodes in the system where subcomputations can be carried out. 'fact_div' uses these to carry out its factorial calculation, by binding each shared variable to :

$$\text{goal}(\text{Fact}, [f(\text{Num}, \text{SubVal}), \text{Lower}])$$

'Fact' contains the term class defined in 'fact_div', and the list contains the arguments required by 'Fact' when it is invoked. The first argument of the list is the message stream to 'Fact' while the second is a value for its visible variable. The answer will be returned in 'SubVal', which is calculated using the equation :

$$\text{Num} * \text{Num} - 1 * \dots * \text{Lower} + 1$$

Conceptually, a node in the system will be waiting for its shared variable called 'N' to be bound. When 'N' is bound to a 'goal' term, the node uses the data in the term in order to invoke the 'odemo' predicate. A code fragment which represents this is :

```
N = goal( Fact, Args ) & odemo( Fact, Args, Out )
```

The answers from the nodes are automatically communicated back to 'fact_div', which multiplies them together to produce the answer to the original query.

'fact_div' is defined in program 6.2.

```
fact_div
clauses
  f( Num, Val, Nodes ) =>
    tClass( fact, Fact ),           % get term class
    length( Nodes, Len ),
    Inc is Num / Len ,
    sub_comp( Fact, Nodes, 0, Inc, Num, Val ) . % generate 'goal's

termClass
fact Lower state
clauses
  f( N, Val ) => N == Lower : Val = 1 ;      % 'Lower' base case

  f( N, Val ) => Val is N * SubVal ,         % recursive case
    N1 is N - 1 ,
    Ins becomes [ f( N1, SubVal ) | Ins ] .

end

code
mode sub_comp( ?, ?, ?, ?, ^ ) .
sub_comp( Fact, Nodes, Low, Inc, Top, Val ) <-
  Top =< Low : Val = 1 ;
sub_comp( Fact, [ Node ], Low, Inc, Top, Val ) <-
  Node = goal( Fact, [ [ f( Top, Val ) ], Low ] ) ; % send term class
sub_comp( Fact, [ Node | Nodes ], Low, Inc, Top, Val ) <-
  NLow is Low + Inc ,
  Node = goal( Fact, [ [ f( NLow, Val1 ) ], Low ] ) , % send term class
  sub_comp( Fact, Nodes, NLow, Inc, Top, Val2 ) ,
  Val is Val1 * Val2 .

end.
```

Program 6.2 : A factorial distributor.

The 'goal' terms are generated by a call to 'sub_comp' which recurses down the 'Nodes' list.

A call to 'fact_div' might be :

```
<- fact_div( Ins, Error ) ,
Ins = [ f( 6, Val, [ N1, N2, N3 ] ), ... ] .
% 3 nodes for the factorial
```

which gives :

```

N1 = goal( fact, [ [ f( 2, Val1 ) ], 0 ] )
N2 = goal( fact, [ [ f( 4, Val2 ) ], 2 ] )
N3 = goal( fact, [ [ f( 6, Val3 ) ], 4 ] )

```

The bindings for 'Val1', 'Val2', and 'Val3' will be :

```

Val1 = 2
Val2 = 12
Val3 = 30

```

Finally, 'Val' will be bound to 720, as expected.

In a more realistic program, it may be necessary to modify the term class before it is passed to the nodes.

6.5. Parametrised Classes.

Since Polka classes can have visible variables, it is possible to write classes which become specialised in different ways, depending on the values given to those variables. The simplest example of this is when a variable contains a value which affects whether a test inside the class will succeed or fail. A more interesting case is when the value supplied is a class, which allows more powerful parametrised classes to be written.

A well known example of a parametrised class is a stack, which has a parameter for the type of its elements. In this case, the type will be a term class which can be used to create type instances and perform various operations on those instances. The type used is for complex numbers and will be defined inside the 'type_store' class, which holds many different types although only the complex number type is shown in program 6.3.

```

type_store      % only complex number type shown
clauses
    get( complex_type, CType ) => tClass( complex, CType ) .
                                % get 'complex' term class
                                :
                                :
                                % other 'get' clauses not shown

termClass
complex
clauses
    type( complex(X,Y), Ans )  =>    Ans = yes .      % check type

```

```

add( complex( XR, Xi ), complex( YR, Yi ), Z ) =>    % add 2 no's
      ZR is XR + YR ,
      Zi is Xi + Yi ,
      Z = complex( ZR, Zi ) .

sub( complex( XR, Xi ), complex( YR, Yi ), Z ) =>    % subtract 2 no's
      ZR is XR - YR ,
      Zi is Xi - Yi ,
      Z = complex( ZR, Zi ) .

create( X, Y, Z )    =>    Z = complex( X, Y ) ;    % create a number

type( C, Ans )    =>    Ans = wrong_type .

add( C1, C2, Z )    =>    Z = add_error .

sub( C1, C2, Z )    =>    Z = subtract_error .

last              =>    true .
end
:                % other types not shown
:
end.

```

Program 6.3 : A type store (only 'complex' shown).

This 'complex' type is retrieved by sending a :

```
get( complex_type, Type )
```

message to 'type_store'. The type must then be invoked in order to create instances.

```

<- type_store( Ins, Error ) ,
Ins = [ get( complex_type, Complex ) | Ins1 ] ,
% get complex type

odemo( Complex, [ CIns ], Out ) ,          % invoke type
CIns = [ create( 1, 2, C1 ), create( 3, 4, C2 ) | CIns1 ] ,

```

'C1' will be bound to `complex( 1, 2 )`, which represents the complex number $1 + i2$, while 'C2' will be bound to `complex( 3, 4 )`, which represents the complex number $3 + i4$

Using the type, these numbers can be added :

```
CIns1 = [ add( C1, C2, C3 ) | CIns2 ] ,
```

which will bind 'C3' to the value `complex( 4, 6 )`, which represents $4 + i6$. If addition is attempted with a number that is not complex, then an error is returned :

```
CIns2 = [ add( C1, 2, C4 ) | CIns3 ]
```

will produce the binding :

```
C4 = add_error
```

As can be seen from its definition, the 'complex' term class can also handle subtraction and type checking.

The main benefit of representing the complex type as a term class is that the type can now be passed as an argument to a parametrised stack class, given in program 6.4.

```
stack  Type state
invisible Stack state <= []
clauses
  push( El, Ans ) => odemo( Type, [ [ type( El, Answer ) ] ], Out ) &
                      Answer = yes :
                      Ans = yes ,
                      Stack becomes [ El | Stack ] .

  empty( Ans )    => Stack == [] : Ans = yes .

  pop( El, Ans )  => Stack = [ El | #Stack ] : Ans = yes ;

  push( El, Ans ) => Ans = wrong_type .

  empty( Ans )    => Ans = not_empty .

  pop( El, Ans )  => Ans = no_elements .

end.
```

Program 6.4 : A parametrised stack class.

'stack' defines the usual functionality of a stack, including push, pop and empty. The interesting clause is the first which uses 'odemo' to send 'Type' a message to check the type of the element which is about to be pushed onto the stack. If the type of the element is satisfactory then the element is added to the stack, otherwise it is ignored and an error is returned.

A complex number stack might be invoked like so :

```
<- stack( Ins, Complex, Error ) ,
Ins = [ push( C1, Ans1 ), push( 2, Ans2 ), pop( El, Ans3 ) ] .
% Complex contains the complex type from 'type_store'
% C1 is a complex number
```

This will produce the following bindings :

```
Ans1 = yes
Ans2 = wrong_type
and El = complex( 1, 2 )
```

The binding of 'Ans2' shows that the attempt to add 2 to the complex number stack failed.

In the above example, the type is only used as a test but it could also be used to manipulate elements on the stack. For instance, the following clause could be added :

```
double => odemo( Type, [ TIns ], Out ),
          double( Stack, #Stack, TIns ).
```

along with the code section predicate :

```
mode double( ?, ^, ^ ).
double( [], [], [] ).
double( [ El | Stack ], [ NewEl | Stack1 ], [ add( El, El, NewEl ) | TIns ] ) <-
    double( Stack, Stack1, TIns ).
```

Now, a 'double' message sent to 'stack' doubles the values of all its elements.

Parametrised classes allow classes to be more general purpose, flexible and reusable. These properties are considerably enhanced by the fact that term classes can be parameters, which is only possible because term classes are first order citizens of the language.

6.6. Extensible classes.

As mentioned earlier, by using the explicit reflection mechanism :

```
meta:: Message
```

it is possible to augment the functionality of a term class by extending it at the meta level. This will be demonstrated by showing how multiple inheritance and self communication can be implemented for term classes. In fact, the inheritance mechanism will allow dynamic alteration of inherited classes, thus giving the term classes more functionality than standard Polka classes.

The example will be a variation of the familiar 'window' scenario [Shapiro and Takeuchi 1983], where the 'window' class inherits a class which defines its shape, and a class which defines the system operations it can carry out. In this example, there are two types of shape class - 'square' and 'circle', and one system class called 'computer'. Initially, the inheritance tree for the objects can be represented by figure 6.1.

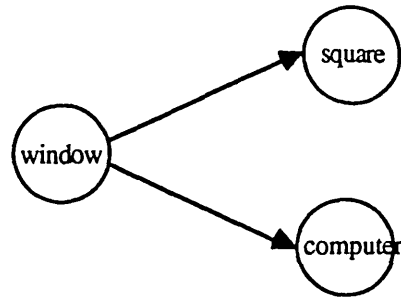


Figure 6.1 : Initial inheritance tree for 'window'.

but during its execution 'window' will change its inherited shape to 'circle', and will make the inheritance tree look like figure 6.2.

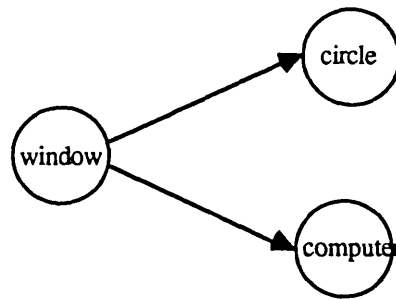


Figure 6.2 : New inheritance tree for 'window'.

To accomplish self communication, the **meta** message :

`self( Message )`

will have the meaning that 'Message' is sent to the bottom object in the inheritance tree.

For inheritance, the **meta** message :

`newI( Name_of_object, Stream )`

will link the output stream 'Stream' to the inherited object called 'Name_of_object'. Any number of these 'newI' messages can be sent to the meta level, so allowing multiple inheritance.

For the dynamic alteration of inheritance links, the **meta** message :

`change( Name_of_object, New_name )`

is defined. This will automatically disconnect the stream output going to 'Name_of_object' and link it to a new object called 'New_name'.

Thus with only three specially defined **meta** messages, the functionality of a term class has been significantly increased.

The term classes used in the example are defined below, without the necessary enclosing ordinary classes.

6.6.1. Computer.

```
termClass
  computer
  invisible Peripherals state <= [],
  clauses
    addP( Device )      => Peripherals becomes [ Device | Peripherals ].
    removeP( Device )   => remove_perhs( Device, Peripherals, #Peripherals ).
    last                => true .
end
```

Program 6.5 : A 'computer' term class.

This class defines operations which record and change the type of peripheral. The 'addP' clause adds a peripheral to the list of ones connected to the machine, while 'removeP' removes a peripheral, by using the system predicate 'remove_perhs'.

6.6.2. Square.

```
termClass
  square
  invisible LeftBot state <= pt( 10, 10 ), Length state <= 5
  clauses
    move_to( X, Y, Ok )  => Ok = okay ,
                          LeftBot becomes pt( X,Y ).

    size( Sz )           => Sz is Length * Length .

    draw( Ok )           => draw_shape( square, LeftBot, Length, Ok ).

    erase( Ok )          => erase_shape( square, LeftBot, Length, Ok ).

    redraw( X, Y, Ok 3 ) =>
      meta:: self( erase( Ok1 ) ),           % uses 3 self messages
      meta on Ok1:: self( move_to( X, Y, Ok2 ) ),
      meta on Ok2:: self( draw( Ok3 ) ).

    last                => true.
end
```

Program 6.6 : A 'square' term class.

'square' defines the position of a window by its bottom left hand coordinate and the length of its sides. Most of the clauses are self explanatory, although 'redraw' is more complicated since it is defined in terms of three 'self' messages :

erase, move_to and draw

This means that the action of 'redraw' is carried out by messages sent to other clauses, and so can change if the definition of any of the clauses associated with those messages is changed. The second and third 'self' messages are suspendible, which enforces the fact that the processing of the messages must be in the order :

erase → move_to → draw

If this wasn't enforced then the concurrent execution of the various objects may allow one action or part of an action to occur in the wrong order. For instance, a new window may be moved, drawn but then erased instead of the old window. The concurrency issues behind this example are discussed in more detail in chapter 8.

6.6.3. Circle.

```
termClass
circle
invisible Centre state <= pt( 10, 10 ), Radius state <= 5,
PI state <= 3.14159
clauses
    move_to( X, Y, Ok )    =>    Ok = okay ,
                                Centre becomes pt( X, Y ) .

    size( Sz )             =>    Sz is PI * Radius * Radius .

    draw( Ok )             =>    draw_shape( circle, Centre, Radius, Ok ) .

    erase( Ok )            =>    erase_shape( circle, Centre, Radius, Ok ) .

    redraw( X, Y, Ok3 )    =>
        meta:: self( erase( Ok1 ) ) ,           % uses 3 self messages
        meta on Ok1:: self( move_to( X, Y, Ok2 ) ) ,
        meta on Ok2:: self( draw( Ok3 ) ) .

    last                  =>    true .
end
```

Program 6.7 : A 'circle' term class.

This has the same interface as 'square' but represents its shape by a centre point and radius, and its drawing and erasing routines are also defined differently. However, the same sequence of 'self' messages are used to define 'redraw'.

6.6.4. Window

This class inherits 'square' or 'circle', and 'computer'. It communicates with its inherited shape via the output stream 'Shape' and with 'computer' via the 'Computer' output stream. In addition, it also defines a header for the window which contains its name and has a X and Y length and coordinate position on the screen.

'window's definition is given as program 6.8.

```

termClass
  window
  invisible   Name, CShape state, Shape, Computer ostream,
              Header state <= pt( 10, 15 ),
              XLength state <= 5, YLength state <= 2

  initial     meta:: newI( square, Shape ),      % set up inheritance links
              meta:: newI( computer, Computer ), % with square and computer
              CShape becomes square .

  clauses
    change_shape( NShape ) =>
      meta:: change( CShape, NShape ),    % new inherited shape
      CShape becomes NShape .

    move_to( X, Y, Ok )  =>   Shape:: move_to( X, Y, Ok ) ,
                              Header becomes pt( X, Y ) .

    size( Sz )           =>   Shape:: size( Sz1 ),
                              Sz2 is XLength * YLength ,
                              Sz is Sz1 + Sz2 .

    draw( Ok )           =>   Shape:: draw( Ok1 ) ,
                              draw_shape( rectangle, Header, Name,
                                          XLength, YLength, Ok2 ) ,
                              joint_ok( Ok1, Ok2, Ok ) .

    erase( Ok )          =>   Shape:: erase( Ok1 ) ,
                              erase_shape( rectangle, Header, Name,
                                          XLength, YLength, Ok2 ) ,
                              joint_ok( Ok1, Ok2, Ok ) .

    redraw( X, Y, Ok )  =>   Shape:: redraw( X, Y, Ok ) .

    last                 =>   Shape:: last , Computer:: last ;

    Msg                  =>   Computer:: Msg .

end

```

Program 6.8 : A 'window' term class.

The **initial** section of 'window' uses two 'newI' **meta** messages to link 'Shape' to a 'square' object, and 'Computer' to a 'computer' object. Then the current shape object is recorded in 'CShape'.

The first clause allows a :

```
change_shape( NShape )
```

to change the inherited shape object by transmitting a 'change' **meta** message, which will automatically link the 'Shape' stream to the new 'NShape' object. The 'move_to', 'size', 'draw' and 'erase' messages all specialise the shape operations inherited from the shape object, so that the header is manipulated along with the shape of the window. The call to 'joint_ok' in the 'draw' and 'erase' clauses ensures that 'Ok' is only bound when both parts of the window have been manipulated. The 'redraw' message is delegated to 'Shape', but because of the use of 'self' messages in the shape object, 'erase', 'move_to' and 'draw' messages will come back to 'window' and affect the header as well as the shape. The final clause passes any other messages to 'Computer'.

6.6.5. Support Classes.

Now that the various term classes have been defined, the ordinary classes which handle the special **meta** messages :

```
self, newI and change
```

must be defined. The main Polka class which does this is 'tobj_plus', which can accept only one type of message :

```
object( <bottom class>, <input stream>, <inherited classes> )
```

This invokes the <bottom class> with the <input stream> as its means of receiving messages. The <inherited classes> are a list of classes which the <bottom class> can inherit. For this example, 'tobj_plus' would be called like so :

```
<- tobj_plus( Ins, Error ),
Ins = [ object( Window, WIns, [ i( square, Square ),
                                i( circle, Circle ),
                                i( computer, Computer ) ] ), ... ] .
```

'Window', 'Square', 'Circle' and 'Computer' are variables containing the term classes just defined, while 'WIns' will be the input stream into the 'Window' instance.

'tobj_plus' is defined in program 6.9.

```
tobj_plus
clauses
  object( Obj, Input, Inherits ) =>
    to_inherits( Meta, Inherits, ISelf ),
    to_self( ISelf, Meta, MySelf ),
    merge( Input, MySelf, In ),
    odemo( Obj, [ In ], Meta ).

code
  mode to_self( ?, ?, ^ ).
  to_self( Self, Meta, MySelf ) <-
    merge( Self, Meta, ToSelf ),
    to_self( ToSelf, MySelf ).

  mode to_self( ?, ^ ).
  to_self( [], [] ).
  to_self( [ self( Msg ) | M ], [ Msg | MySelf ] ) <-
    to_self( M, MySelf );
  to_self( [ Msg | M ], MySelf ) <-
    to_self( M, MySelf ).
end.
```

Program 6.9 : A 'tobj_plus' class.

In the action associated with an 'object' message, a 'to_inherits' object handles the inherited objects and their self communication back to 'Obj', which is the bottom object in the inheritance tree. 'toself' manipulates self messages, and is locally defined.

The easiest way to understand the action of the clause is to look at the data flow between the objects and processes.

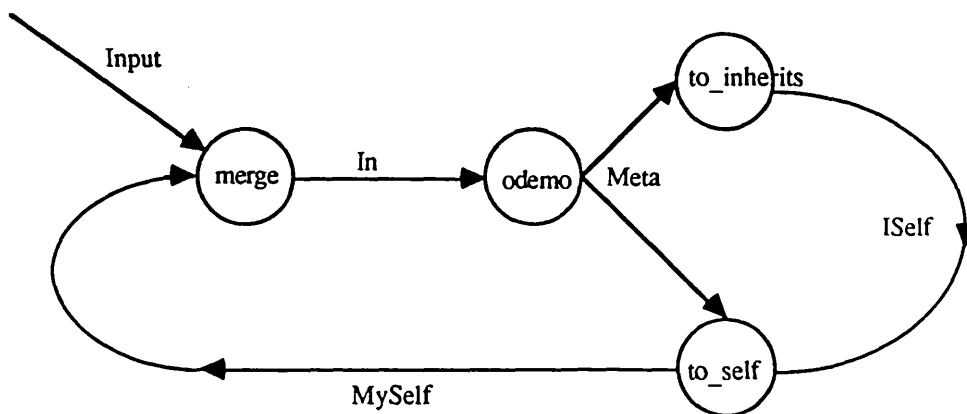


Figure 6.3 : Data flow in the 'tobj_plus' clause.

The streams are represented by arrows, the processes and objects by circles.

The input for 'to_inherits' comes from the 'Meta' output stream of 'odemo', where 'Obj' is executing. 'toself' merges the streams coming from 'to_inherits' and 'odemo' and then extracts any 'self' messages. These are then sent to 'Obj' along 'MySelf', via the 'merge' process.

The 'to_inherits' class is defined in program 6.10.

```

to_inherits MyInhs state, MySelf ostream
invisible Links state <= []
clauses
  newI( Name, Strm )      =>   find( Name, MyInhs, Obj ) :
                                odemo( Obj, [ Strm1 ], Out ),
                                monitor( Strm, Flag, Strm1, Self1 ),
                                mergeL( [ Out, Self1, #MySelf ], MySelf ),
                                Links becomes [ l( Name, Flag ) | Links ].

  change( Name, NewName ) =>   find( NewName, MyInhs, Obj ),
                                lfind( Name, Links, Links1, Flag ) :
                                change_body( Flag, Obj, NFlag ),
                                Links becomes
                                  [ l( NewName, NFlag ) | Links1 ].

  last                    =>   sever_all( Links ).

code
  mode change_body( ^, ?, ? ).
  change_body( Flag, Obj, NFlag ) <-
    sever( Flag, Strm, ThisSelf ),
    odemo( Obj, [ Strm1 ], Out ),
    monitor( Strm, NFlag, Strm1, Self1 ),
    merge( Out, Self1, ThisSelf ).

  mode find( ?, ?, ^ ).          % only modes given for the other predicates

  mode monitor( ?, ?, ^, ^ ).

  mode mergeL( ?, ^ ).

  mode lfind( ?, ?, ^, ? ).

  mode sever( ^, ^, ? ).

  mode sever_all( ? ).
end.

```

Program 6.10 : A 'to_inherits' class.

The 'MyInhs' state contains the inherited term objects, while 'MySelf' is an output stream for self messages. The invisible variable 'Links' is used as a list of executing objects, each stored in a tuple of the form :

l(Name, Flag)

'Name' is the name of the object, while 'Flag' is a variable used to disconnect the input stream from the 'Name' object before it is replaced by another object.

The input of 'to_inherits' are the meta messages coming out of 'Obj', but only the :

newI and change

messages will produce any actions. A 'newI' message will cause the term object called 'Name' to be extracted from 'MyInhs' using the 'find' predicate. The resulting 'Obj' will then be executed in 'odemo' and the 'Strm1' stream will become its input. The 'Strm' stream in the 'newI' message is indirectly connected to 'Strm1' via the 'monitor' process, which passes messages from 'Strm' to 'Strm1' until the 'Flag' variable is set to :

flag(Stream, Self)

This will cause 'Stream' to be unified with the rest of 'Strm', while 'Self', which is an output stream, is unified with 'Self1'. The purpose of 'Flag' is to disconnect 'Strm' from 'Obj' and connect it to a new object, via 'Stream'. The output stream of this new object is 'Self', which by being unified with 'Self1' is connected to 'MySelf'.

The second clause of 'to_inherits' deals with the case when an inherited object is to be changed. Firstly, the new object called 'NewName' is found, along with the 'Flag' for the old 'Name' object. If both these searches are successful then 'change_body' is called and a new object is added to the 'Links' list, and 'change_body' binds 'Flag' to :

flag(Stream, Self)

This will cause the input stream entering 'Name' to be returned in 'Stream', which is then linked to the new object through a 'monitor' process. This is required so that this new object can be altered in the future. Meanwhile, the new 'Out' output stream of the inherited object is merged with a dormant 'Self1' self communication stream and then unified with 'ThisSelf'. 'ThisSelf' is then linked, via the 'Self' stream in 'Flag', to the 'MySelf' stream leaving 'to_inherits'. 'Self1' will not become active unless another object replaces the new object invoked here.

The third clause calls 'sever_all' to terminate all the inherited objects.

As already described, a typical execution of 'tobj_plus' is :

```
<- tobj_plus( Ins, Error ),
Ins = [ object( Window, WIns, [ i( square, Square ),
                                i( circle, Circle ),
                                i( computer, Computer ) ] ), ... ] .
```

Messages sent to the window might be :

```
WIns = [ redraw( 20, 20, IsOk1 ), change_shape( circle ) | WIns1 ]
```

This erases the old window, which is currently a header and square window, and moves it so that its header and window start at (20, 20) and then redraws it. The second message causes the window's shape to be changed to a circle, although this will not be apparent until the window is redrawn.

The next sequence of messages are :

```
WIns1 = [ redraw( 10, 10, IsOk2 ), size( Sz ), addP( line_printer ) | WIns2 ]
```

The 'redraw' causes the window to be erased, moved and the header and circle drawn at (10, 10). The 'size' message binds 'Sz' to 88.5395 which is the header size (2*5) plus the circle size (PI*5*5). The third message adds a lineprinter to the list of peripherals.

The main point of this example is that term class functionality can be easily extended by defining the meaning of certain *meta* messages. This makes term classes potentially much more powerful than ordinary classes.

Chapter 7. Three Examples.

This chapter contains three larger Polka examples, in the areas of :

- graphics
- blackboard systems
- simulation

Each example is self contained, although it is assumed that chapters 5 and 6 have been read, and consequently Polka operations will not be explained in any detail.

7.1. Graphics.

Three classes are described which enable a user to produce graphics on virtually any machine, as long as it can print Ascii characters on a screen. The classes can carry out a wide variety of tasks including drawing, moving and erasing lines, squares, and circles.

More generally, the classes also illustrate how the OOP features of Polka, such as encapsulation, message passing, and inheritance, can be combined with its concurrent LP constructs, such as and- and or- parallelism, the logical variable, and predicates.

In addition, Parlog versions of the classes are outlined and contrasted with the Polka code, and extensions to the functionality of the classes are discussed.

7.1.1. Overview of the Classes.

The classes are called :

	screen
	shape
and	turtle

They are related by the inheritance graph in figure 7.1.

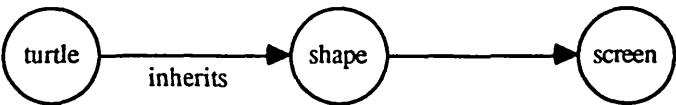


Figure 7.1 : Inheritance graph for the graphics example.

'screen' implements the basic machine dependent operations necessary to print characters. 'shape' inherits this functionality and augments it in a system independent way to allow the printing of circles, lines, strings and squares. 'turtle' specialises 'shape' to enable a character to be moved around the screen by specifying compass headings and distances.

The classes will be described in the order : 'screen', 'shape', and 'turtle'.

7.1.2. Screen.

'screen' manipulates a device dependent graph , which looks like figure 7.2.

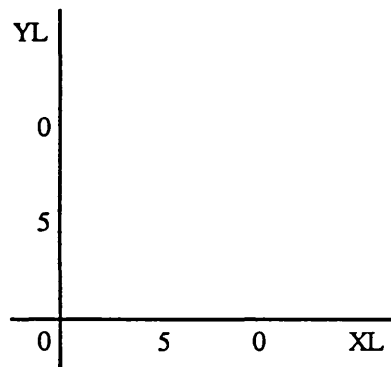


Figure 7.2 : Device dependent graph.

'XL' (the x maximum) and 'YL' (the y maximum) values are supplied as arguments, along with an input stream and error stream, when the class is invoked :

```
<- screen( Ins, XL, YL, Error ) .
```

The class recognises six kinds of message :

```
pt( X, Y )
char( Ch, X, Y )
clear
print( Done )
pts( List )
and
range( X, Y, Reply )
```

In 'pt', 'X' and 'Y' are the graph coordinates which must be positive and less than 'XL' and 'YL'. If they are not, then an error is issued and the graph is not changed, but if 'X' and 'Y' are within the allowable range then an 'X' character is put onto the graph at the (X, Y) coordinate. A 'char' message is more general in that the character placed at (X, Y) is supplied by 'Ch'. 'clear' removes all the points from the graph,

while a 'print' message causes the graph to be printed out on the screen, and binds the 'Done' variable when this is completed. A 'pts' message contains a list which should consist of 'pt' and 'char' messages. A 'range' message can be used to determine if the (X, Y) coordinate lies on the graph. If it does then 'Reply' is bound to 'good', and if not, it is bound to 'bad'.

The Polka program for the 'screen' class is given as program 7.1.

```

screen    XL, YL state
invisible Graph, Axis state

initial   axis( XL, YL, Ax ) &
          Graph, Axis becomes Ax, Ax &.           % initialise the graph

clauses
  clear          =>   Graph becomes Axis .        % clear the graph

  pt( X, Y )     =>   add1( X, Y, 'X', XL, YL, Graph, #Graph ) . % add a 'X'

  char( Ch, X, Y ) =>   add1( X, Y, Ch, XL, YL, Graph, #Graph ) . % add a character

  print( Done )  =>   print_coords( Graph, YL, Done ) .        % print the graph

  pts( List )    =>   add_list( List, XL, YL, Graph, #Graph ) . % add several points

  range( X, Y, Reply ) => range( X, Y, XL, YL, Reply ) . % check X, Y range

code      % Parlog predicates shown here in outline only

  mode add1( ?, ?, ?, ?, ?, ^ ) .
  add1( X, Y, Ch, XL, YL, Graph, NewGraph ) <- ... % add Ch to the graph

  mode axis( ?, ?, ^ ) .

  mode print_coords( ?, ?, ^ ) .

  mode add_list( ?, ?, ?, ?, ^ ) .

  mode range( ?, ?, ?, ?, ^ ) .

end.
```

Program 7.1 : A 'screen' class.

The initial action is a call to the Parlog predicate 'axis' which takes as input the 'XL' and 'YL' values and returns a list of graph coordinates in 'Ax'. These coordinates contain the graph axis and are assigned to 'Graph' and 'Axis' in a **becomes** statement. The idea behind this is that 'Graph' will contain the on-going graph with all the points added by messages, while 'Axis' will be a clean version of the graph. The

coordinates are stored in a sparse list which contains entries only for those points with data. A typical coordinate will be :

```
ch( X, Ch )
```

'Ch' is the character to be printed and 'X' is its X position. This coordinate will be stored as part of a list in a tuple like :

```
row( Y, [ ... ch( X, Ch ), ... ] )
```

where 'Y' is the Y position. These 'row' tuples will be stored as elements of a list which represents the full graph. Such a data structure, although not complicated, can most easily be handled by Parlog predicates rather than classes, and this philosophy has been used throughout for any manipulation of the graph.

The purpose of 'Axis' becomes apparent in the first clause, which clears the graph by resetting 'Graph' to the 'Axis' value.

The second clause adds a point to the graph by calling the 'add1' Parlog predicate. It can also output a message on the 'Error' stream if the 'X' or 'Y' values are out of range.

Whereas the second clause always adds a 'X' to the graph, the third clause calls 'add1' with the 'Ch' character from the message, so that it is added instead.

The fourth clause calls 'print_coords', which is the device dependent predicate that takes the graph coordinates and prints them out on the screen. When this is done it binds 'Done' to 'done', which because it has been unified with the variable in the original message, will communicate its binding back to the sender.

The fifth clause uses 'add_list' to add a list of 'pt' and 'char' coordinates to the graph. This is achieved by 'add_list' calling 'add1' repeatedly until the list is consumed.

The final clause calls a 'range' predicate to determine if 'X' and 'Y' are on the graph

The **code** section predicates are only given in outline since they carry out standard term manipulation operations.

A typical call to 'screen' might be :

```
<- screen( Ins, 15, 15, Error ) .
```

The second and third arguments are values for 'XL' and 'YL' respectively, 'Error' is the error stream, and messages are sent to 'screen' by partially instantiating 'Ins' :

```
Ins = [ char( 'W', 10, 10 ), char( 'A', 11, 10 ), pt( 12, 10 ), print( Done ) | Ins1 ]
```

This will produce the graph in figure 7.3 :

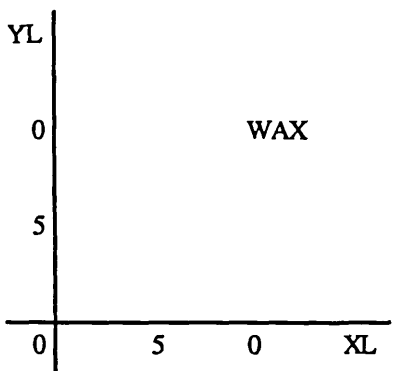


Figure 7.3 : WAX on the graph.

and then bind 'Done' to 'done'.

'screen' demonstrates a number of Polka features but also illustrates some more general points about the combination of OOP and concurrent LP. For example, the encapsulation of data and procedures enforces communication by message passing, which is a good thing since 'screen's implementation can be altered with impunity, as long as its interface remains the same. Another point is that the manipulation of data structures, especially ones involving lists and tuples, is best handled by Parlog. Parlog also allows or- parallel clause selection and and- parallel execution of actions in a clause, and the logical variable allows replies to messages to be coded using back communication.

7.1.3. Shape.

The 'shape' class inherits the functionality of 'screen' and extends it so that circles, lines, strings and squares can be added to the graph directly. 'shape' will recognise 5 sorts of message :

```

circle( Radius, pt( X, Y ) )
line( pt( X1, Y1 ), pt( X2, Y2 ) )
string( String, pt( X, Y ), Direction )
square( pt( X, Y ), Length )
and okline( pt( X1, Y1 ), pt( X2, Y2 ) )

```

A 'circle' message supplies a radius and a coordinate point for the centre of the circle, while a 'line' message contains the two end points of the line to be drawn. A 'string' message supplies a string, the point at which it should start being printed and also a

direction. The direction can either be 'right' or 'down', and describes the string's orientation. A 'square' message contains the lower left hand corner point of the square which is to be drawn, and the length of its sides. An 'okline' message supplies two end points of a line, which are assumed to lie on the graph and so are not checked.

'shape' is defined by program 7.2.

```

shape
inherits screen
clauses
  circle( Radius, pt(X, Y) ) => super:: range( X, Y, Reply ) ,      % add a circle
                                make_circle( Radius, X, Y, Reply, Group ) ,
                                send_group( Group ) .

  line( pt(X1, Y1), pt(X2, Y2) ) =>
                                super:: range( X1, Y1, Reply1 ) ,    % add a line
                                super:: range( X2, Y2, Reply2 ) ,
                                make_line( X1, Y1, X2, Y2, Reply1, Reply2, Group ) ,
                                send_group( Group ) .

  string( String, pt(X1, Y1), Dir ) =>
                                super:: range( X1, Y1, Reply ) ,      % add a string
                                make_string( String, Dir, X1, Y1, Reply, Group ) ,
                                send_group( Group ) .

  square( pt(X1, Y1), Length ) => super:: range( X1, Y1, Reply ) ,    % add a square
                                make_square( X1, Y1, Length, Reply, Group ) ,
                                send_group( Group ) .

  okline( pt(X1, Y1), pt(X2, Y2) ) =>
                                make_line( X1, Y1, X2, Y2, good, good, Group ) ,
                                super:: pts( Group ) .                  % add a 'ok' line

code
  mode send_group( ? ) .                % add several coordinates to the graph
  send_group( [] ) ;
  send_group( G ) <-
    super:: pts( G ) .

  mode make_circle( ?, ?, ?, ?, ^ ) .    % other predicates only given in outline
  mode make_line( ?, ?, ?, ?, ?, ^ ) .
  mode make_string( ?, ?, ?, ?, ^ ) .
  mode make_square( ?, ?, ?, ?, ^ ) .

end.
```

Program 7.2 : A 'shape' class.

Since any unrecognised messages are delegated to 'screen', 'shape' can also accept 'pt', 'char', 'clear', 'print', 'range', and 'pts' messages.

The first clause initially checks the (X, Y) coordinate by sending a 'range' message to 'screen'. Then 'make_circle' takes the radius, the centre, and the reply from 'screen', and generates a list of coordinates for the circle which it returns in 'Group', although if 'Reply' is bound to 'bad' then 'Group' will be []. These coordinates are added to the graph by calling 'send_group'.

The second clause uses 'range' messages to check on the suitability of the two end points of the line. The returned information is used by 'make_line' to either return a list of coordinates between the two end points, or to return [], if one or more of the points is illegal. Again 'send_group' is used to add the generated coordinates to the graph.

In a similar way to the other clauses, the third clause checks the starting point of its string by sending a 'range' message to 'screen'. 'make_string' then converts the string into a list of characters with associated coordinate positions and they are added to the graph.

The clause to handle a 'square' initially checks the arguments of the message, and then the coordinates of the square are generated in 'make_square'.

An 'okline' message causes 'make_line' to be called immediately, with its fifth and sixth arguments both bound to good, which indicates that the four coordinates are legal.

A call to 'shape' might be :

```
<- shape( Ins, 27, 27, Error ) .
```

As usual, 'Ins' is the default input stream, 'Error' is the error stream, while the second and third arguments are the 'XL' and 'YL' arguments, which have been inherited from 'screen'. Messages sent to 'shape', such as :

```
Ins = [ circle( 4, pt( 15, 15 ) ) , string( 'Hi', pt( 14, 15), right ) ,
        square( pt( 5, 5 ), 20 ) , print( Done ) | Ins1 ]
```

would produce figure 7.4 :

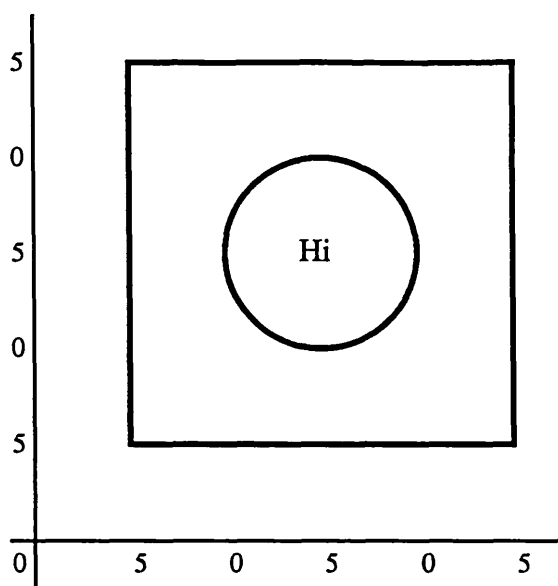


Figure 7.4 : Shapes on the graph.

and set 'Done' to 'done'.

'shape' shows how a class can inherit the functionality of another, and use it to build more high level operations. This is made possible by utilising the **super** operation, and the augmented 'send_group' predicate. 'shape' also relies on Parlog predicates, such as 'make_circle', to generate coordinate data from more abstract shape information.

7.1.4. Turtle.

This class implements simple logo-like graphics [Watt 1984], which enable a 'O' character (supposedly representing a turtle) to be moved about on the graph. Its movements can either be recorded by leaving 'x's on the graph or can be invisible. In addition to the messages which 'turtle' can process because of inheriting 'shape' and 'screen', there are four new messages which it can handle :

```

pen_up
pen_down
move_to( pt( X, Y ), Done )
and
move( Direction, Length, Done )

```

A 'pen_up' message will make the movements of the turtle invisible, and conversely, when a 'pen_down' message arrives, its movements will be shown again. A 'move_to' message makes the turtle move to the stated coordinate and causes the

graph to be printed, after which 'Done' is bound to 'done'. A 'move' message does a similar thing except that the movement is specified by a compass direction which may be one of n, ne, e, se, s, sw, w or nw. These represent the compass points shown in figure 7.5.

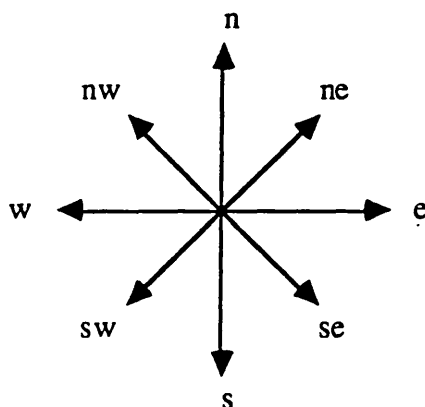


Figure 7.5 : Compass points.

The distance to travel along one of these headings is given by the second argument of 'move'.

The 'turtle' class is defined by program 7.3.

```

turtle
inherits shape
invisible PenState state <= up, Pos state <= pt( 1, 1 )

initial super :: char( 'O', 1, 1 ) &.          % initially place the turtle at (1, 1)

clauses
  pen_up          =>          PenState becomes up .
  pen_down        =>          PenState becomes down .

  move_to( pt(X1, Y1), Done ) =>
    super:: range( X1, Y1, Reply ) ,          % move turtle to (X1, Y1)
    do_line( PenState, Pos, X1, Y1, Reply, Done ) ,
    set_pos( Pos, pt( X1, Y1 ), Reply, #Pos ) &.

  move( Dir, Len, Done )          =>
    new_pos( Pos, Dir, Len, NX, NY ) ,          % move turtle using Dir
    super:: range( NX, NY, Reply ) ,          % and Len
    do_line( PenState, Pos, NX, NY, Reply, Done ) ,
    set_pos( Pos, pt( NX, NY ), Reply, #Pos ) &.

code
  mode do_line( ?, ?, ?, ?, ^ ) .
  do_line( Pst, Pos, X, Y, bad, done ) ;          % draw nothing
  do_line( up, pt( OX, OY ), X, Y, good, Done ) <-
    super :: ( pt( OX, OY ), char( 'O', X, Y ), print( Done ) ) ; % draw the end point
  do_line( down, pt( OX, OY ), X, Y, good, Done ) <-
    super :: ( okline( pt(OX, OY), pt(X, Y) ), char( 'O', X, Y ), print( Done ) ) .

```

```

mode set_pos( ?, ?, ^, ^ ).      % the other predicates given in outline only

mode new_pos( ?, ?, ^, ^ ).

end.

```

Program 7.3 : A 'turtle' class.

This class uses two invisible variables 'PenState' and 'Pos' to store the current state of the pen and the position of the turtle, and these variables are initialised to 'up' and 'pt(1,1)' at invocation. The **initial** section puts the turtle onto the graph at its starting position by sending a 'char' message to 'shape', which will delegate it to 'screen'.

The first two clauses change 'PenState' in the prescribed manner. In the third clause, the new coordinate from the 'move_to' message is checked, and then a line is generated which goes to that point. However, the line may not be printed if 'Reply' is bad or if the pen is up. If the 'Reply' is good but the pen is up, then only the new position of the turtle is generated. If the pen is down then, in addition, a line is drawn to that new position. These decisions are all encoded in 'do_line' by utilising **super** messages. Concurrent with the printing of the graph by 'do_line', a call to 'set_pos' updates the position of the turtle, returning its new value as its fourth argument.

The fourth clause implements the 'move' action, by generating the new coordinate using 'new_pos' and delegating a 'range' message to check that it is legal. It also adds the movement to the graph using 'do_line' and updates the turtle position using 'set_pos'.

A call to 'turtle' might be :

```

<- turtle( Ins, 10, 10, Error ),
Ins = [ pen_down, move( ne, 5, Done1 ) | Ins1 ]

```

which produces figure 7.6 :

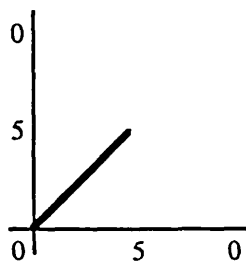


Figure 7.6 : A turtle trail.

and 'Done1' is bound to 'done'. If this is followed by :

```

Ins1 = [ pen_up, move_to( pt( 5, 10 ), Done2 ) | Ins2 ]

```

then the outcome is shown in figure 7.7 :



Figure 7.7 : Further turtle movements.

and 'Done2' equals 'done'.

'turtle' illustrates how new initial actions and new invisible variables can be added to those inherited from other classes. Also, 'do_line' shows how functionality inherited from 'shape' and 'screen' can be used together.

7.1.5. The Parlog Approach.

The Polka classes just described use Parlog predicates to handle the graph manipulation operations, but could everything have been encoded in Parlog? The short answer to this is obviously yes, since Polka programs compile into Parlog. However, coding this problem directly in Parlog means that the advantages of the Polka version are lost. These include encapsulation, message passing, inheritance, augmented predicates, the **becomes** operator, initialisation mechanisms, default termination, and error handling. There are other losses of course, including self communication, term classes, the **i_am** operator, and suspendible messages, but these were not used in these examples, and so will not be considered.

Probably the only feasible way to represent this problem is as a system of communicating processes. Using this technique, the Parlog version of 'screen' might be like program 7.4.

```
mode screen( ?, ?, ?, ^ ).
screen( Ins, XL, YL, Error ) <-
    axis( XL, YL, Ax ),
    screen1( Ins, XL, YL, Ax, Ax, Error ) .
```

% initialise the graph

```

mode screen1( ?, ?, ?, ?, ^ ) .
screen1( [ clear | Ins ], XL, YL, Coords, Axis, Error ) <-      % clear the graph
    screen1( Ins, XL, YL, Axis, Axis, Error ) .

screen1( [ pt( X, Y ) | Ins ], XL, YL, Coords, Axis, Error ) <- % add a 'X'
    add1( X, Y, 'X', XL, YL, Coords, NCoords ) ,
    screen1( Ins, XL, YL, NCoords, Axis, Error ) .

screen1( [ char( Ch, X, Y ) | Ins ], XL, YL, Coords, Axis, Error ) <- % add a character
    add1( X, Y, Ch, XL, YL, Coords, NCoords ) ,
    screen1( Ins, XL, YL, NCoords, Axis, Error ) .

screen1( [ print( Done ) | Ins ], XL, YL, Coords, Axis, Error ) <- % print the graph
    print_coords( Coords, YL, Done ) ,
    screen1( Ins, XL, YL, Coords, Axis, Error ) .

screen1( [ pts( List ) | Ins ], XL, YL, Coords, Axis, Error ) <- % add several points
    add_list( List, XL, YL, Coords, NCoords ) ,
    screen1( Ins, XL, YL, NCoords, Axis, Error ) .

screen1( [ range( X, Y, Reply ) | Ins ], XL, YL, Coords, Axis, Error ) <- % check range
    range( X, Y, XL, YL, Reply ) ,
    screen1( Ins, XL, YL, Coords, Axis, Error ) .

screen1( [], XL, YL, Coords, Axis, [] ) ;                        % termination clause

screen1( [ Msg | Ins ], XL, YL, Coords, Axis, [Msg | Error ] ) <- % error clause
    screen1( Ins, XL, YL, Coords, Axis, Error ) .

% utility predicates as in 'screen' class
mode add1( ?, ?, ?, ?, ?, ^ ) .

mode axis( ?, ?, ^ ) .

mode print_coords( ?, ?, ^ ) .

mode add_list( ?, ?, ?, ^ ) .

mode range( ?, ?, ?, ^ ) .

```

Program 7.4 : Predicates for 'screen'.

This 'screen' would be invoked in exactly the same way as the class :

```

<- screen( Ins, 15, 15, Error ) ,
Ins = [ char( 'W', 10, 10 ), char( 'A', 11, 10 ), pt( 12, 10 ) ] .

```

The recursive calls in the clauses of 'screen1' makes the program longer, and errors are more likely because the recursive calls are easily misspelt, or written with the wrong number of arguments. If new state variables are added to the program, or old ones removed, then all the clauses must be altered. Also, an extra predicate is required to initialise the arguments, and extra clauses must be included to deal with termination and error handling. The predicates 'axis', 'add1', 'print_coords', 'add_list', and

'range' are no longer hidden inside the class, and so can be called directly and may clash with other predicates of the same name. In fact, the program has no overall structure, being simply a flat set of predicates.

'shape' might look like program 7.5.

```

mode shape( ?, ?, ^ ).
shape( Ins, XL, YL, Error ) <-
    shape1( Ins, ToScreen ),
    screen( ToScreen, XL, YL, Error ).

                                % link shape and screen

mode shape1( ?, ^ ).
shape1( [ circle( Radius, pt( X, Y ) | Ins ], TS ) <-
    TS = [ range( X, Y, Reply ) | TS1 ],
    make_circle( Radius, X, Y, Reply, Group ),
    send_group( Group, TS1, TS2 ),
    shape1( Ins, TS2 ).

                                % add a circle

shape1( [ line( pt( X1, Y1 ), pt( X2, Y2 ) ) | Ins ], TS ) <-
    TS = [ range( X1, Y1, Reply1 ), range( X2, Y2, Reply2 ) | TS1 ],
    make_line( X1, Y1, X2, Y2, Reply1, Reply2, Group ),
    send_group( Group, TS1, TS2 ),
    shape1( Ins, TS2 ).

                                % add a line

shape1( [ string( String, pt( X1, Y1 ), Dir ) | Ins ], TS ) <-
    TS = [ range( X1, Y1, Reply ) | TS1 ],
    make_string( String, Dir, X1, Y1, Reply, Group ),
    send_group( Group, TS1, TS2 ),
    shape1( Ins, TS2 ).

                                % add a string

shape1( [ square( pt( X1, Y1 ), Length ) | Ins ], TS ) <-
    TS = [ range( X1, Y1, Reply ) | TS1 ],
    make_square( X1, Y1, Length, Reply, Group ),
    send_group( Group, TS1, TS2 ),
    shape1( Ins, TS2 ).

                                % add a square

shape1( [ okline( X1, Y1, X2, Y2 ) | Ins ], TS ) <-
    make_line( X1, Y1, X2, Y2, good, good, Group ),
    TS = [ pts( Group ) | TS1 ],
    shape1( Ins, TS1 ).

                                % add a 'ok' line

shape1( [], [] );
                                % termination clause

shape1( [ Msg | Ins ], [ Msg | TS ] ) <-
    shape1( Ins, TS ).
                                % delegation clause

mode send_group( ?, ^, ? ).
send_group( [], TS, TS );
send_group( G, [ pts( G ) | TS ], TS ).

% the other predicates are exactly like their 'shape' class counterparts
mode make_circle( ?, ?, ?, ?, ^ ).

mode make_line( ?, ?, ?, ?, ?, ^ ).

```

```
mode make_string( ?, ?, ?, ?, ^ ).
mode make_square( ?, ?, ?, ?, ^ ).
```

Program 7.5 : Predicates for 'shape'.

'shape1' is more complex because its 'TS' output stream allows it to send messages to the 'screen' process. If several processes had to be communicated with, then the code would be more convoluted, since each of them would require its own stream. Also, the 'send_group' predicate must now manipulate an output stream, and predicates can become very difficult to understand if they use a lot of stream communication. A termination clause is also required, as well as a delegation clause to pass unrecognised messages to 'screen'.

The Parlog version of 'turtle' will not be outlined, but has a similar structure to the 'shape' predicate, and similar drawbacks.

Writing communicating processes in Parlog can be very complex, and one of the main reasons for using such an idea - conceptual clarity - is lost beneath the verbosity of the program. This highlights the main reason for using Polka for OOP instead of Parlog : the language features of Polka directly support the OOP view of a problem, while suppressing the implementation details.

7.1.6. Possible Changes to the Programs.

One of the easiest changes to implement is to alter the way that the graph is printed. This would only require the recoding of 'print_coords' in 'screen', and perhaps the addition of invisible state variables to hold the name of a graphics window or the current cursor location. Bearing this in mind, it is straight forward to port 'screen' to different machines, and thus to move 'shape' and 'turtle' as well.

Another useful change is the introduction of new shapes such as arcs, splines, arrows, and polygons, which would be simple to achieve.

The reason why these extensions are straight forward is due in part to encapsulation, message passing and inheritance. All of these contribute to the independence of the classes from the underlying hardware, and thus to their reusability and flexibility.

The assumptions made when the classes were being designed are also important. One of these assumptions was that 'screen' should use a mapping from the

device independent graph to the computer screen based on (X, Y) coordinates and Ascii characters. However, on machines such as the Macintosh, there are system primitives which can draw lines, squares, circles, and other shapes, as single operations [Chernicoff 1987]. To alter the classes to take advantage of these operations would require the rewriting of 'screen' and 'shape', but even these changes would not affect 'turtle'.

7.2. Blackboard Systems.

This example looks at an approach to implementing the blackboard model of problem solving, where each blackboard and knowledge source is an object. One benefit of this is that inheritance and the ease of object composition means that new blackboards and knowledge sources can be created from old ones quickly and simply. The problems of control, and non-monotonic reasoning are also considered, and our solutions to these problems have been demonstrated by the implementation of a simplified version of Hearsay II, the speech recognition system [Erman et al. 1980], which is described briefly.

The following discussion is closely based on a paper called "Blackboard Systems in Polka" [Davison 1987d].

7.2.1. Background to Blackboard Systems.

Blackboard systems are a powerful and flexible means of problem solving and are especially useful for problems with the following properties [Nii 1986a; Nii 1986b] :

- A large solution space.
- A variety of input data and a need to intergrate diverse information.
- The need for many independent or semi-independent pieces of knowledge to cooperate in forming a solution.
- The need to use multiple reasoning methods (for example, backward and forward reasoning).
- The need for multiple lines of reasoning.
- The need for an evolutionary solution.

Blackboard systems have been used to solve a wide variety of problems. These include speech recognition [Erman et al. 1980], plan formation [Hayes-Roth et al. 1979], and the analysis of the structure of proteins [Hayes-Roth 1984].

A blackboard model consists of three major kinds of component : the blackboard data structure, the knowledge source (KS) and a control mechanism. In this section we shall examine each of these, which shall motivate the use of Polka to represent a blackboard architecture as an object oriented system with concurrently executing entities. In addition, the issues of parallelism and the manipulation of blackboard entities will be discussed.

7.2.1.1. The Blackboard.

The blackboard is typically a globally accessible data structure, which can be written to and read, in an incremental fashion. A board may be subdivided into levels [Erman et al. 1980], which represent problem specific logical divisions in the data, and may also be subdivided into subboards [Terry 1983] to represent more important divisions in the meaning or use of the data. A board may actively organize its contents so that new data will be placed in a particular location, but it is usually unaware of how it is read by KSs. To be useful for real problems, which contain misleading or possibly incorrect information, the board should allow data to be deleted or changed, and permit automatic changes to other data whose validity depends on the data just deleted or changed. These non-monotonic aspects of blackboard architectures are difficult to handle because of the complexity of the logical dependencies between data and the execution overheads in maintaining data consistency.

7.2.1.2. Knowledge Sources (KSs).

KSs are the problem solving entities in the blackboard system, and carry out their tasks by using the data on the board. They operate independently of one another but can communicate indirectly by means of data which one KS may generate and add to the board, and another may then read. Typically a KS is a condition-action process which carries out its actions when the blackboard satisfies its conditions. Aside from this, KSs can be implemented in a number of different ways : as procedures [Erman et al. 1980] or rules [Nii et al. 1982] and may also use their own data structures, such as event lists [Nii et al. 1982] to improve efficiency. The independence of KSs means that any number can be set to work on a blackboard, each with its own problem area and reasoning mechanism. It also means that KSs can be easily removed from the blackboard system, if they are not required.

7.2.1.3. Control.

The most important aspect of a blackboard architecture is its control, as explained in [Hayes-Roth 1985]. Control is necessary to organize the problem solving activity, so that a solution is found without excessive generation of data and needless execution of KSs. This requires the formulation of control policies that restrict the execution of some KSs while encouraging others, and these policies should change as partial answers are generated and new sub-problems are discovered. This has led to the idea of including control information on boards, and using control KSs to read, change and delete it, in the normal way.

Common to most blackboard models is the scheduler, which is the mechanism that executes the KSs. A scheduler carries out three jobs : it collects the executable KSs, a KS is chosen from the executable set, and then it is executed. An executable KS is one whose conditions have been satisfied by the data, while the choice of KS will depend on the control information on the board. After the chosen KS is executed and new data has been generated, the scheduler will repeat its three step sequence. However, it will obtain a different set of executable KSs, since the state of the board has changed. The choice of KS may also alter if the board's control information has changed.

7.2.1.4. Parallelism Issues.

The scheduler is a bottleneck in a blackboard system since it must repeatedly stop the problem solving process in order to select an executable KS. Some parallelism may be obtained if a scheduler executes a number of KSs on each cycle, but the basic bottleneck still remains. One answer to this, is to let each KS be an independently executing object, but this is unsatisfactory since the control over which KS to execute has been lost. Control can be reintroduced by considering each KS as a three stage process, as in figure 7.8.

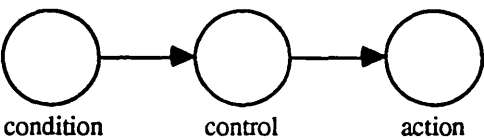


Figure 7.8 : A KS with control.

Now an action will only be executed if a KS's domain conditions and its control conditions are both satisfied. These will be checked on the board by the condition and control parts of the KS, so that a KS now decides for itself whether it can execute its action. The scheduler is no longer needed because every KS will carry out its own tests to see if it can execute its action. We also want to maximise the parallelism of the system and allow all the KSs to test their domain and control conditions in parallel, which may not be possible because of the lack of hardware resources. A partial solution to this is to use a four stage KS, of the form shown in figure 7.9

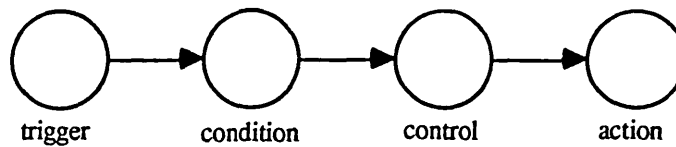


Figure 7.9 : A KS with a trigger.

The trigger will be a computationally inexpensive test which can be applied to the new data entering the board, in order to eliminate a KS. It is now only necessary to allow the trigger parts of all the KSs to execute at once. Even with this refinement there will be a small process running for each KS in the system, and the underlying hardware will need to support the parallel execution of many such processes in order for this approach to be efficient. If the number of concurrently executing processes is not a major concern, then it may be useful if each of the four parts of a KS can execute independently of each other. This would allow a KS to carry out several computations at once.

One further point to consider is how to handle the concurrent updates of a board, which may occur if many KSs are executing at once. In this situation, it must be possible to 'lock' out other KSs while the board is being changed.

7.2.1.5. Manipulation Issues.

By manipulation we mean the dynamic altering, deletion and creation of components of the blackboard model at runtime. This has been called learning [Hayes-Roth and Hewitt 1985], since the manipulation can be governed by changes to the rest of the system or by user intervention. This problem has strong parallels with some of the ideas found in open systems [Hewitt 1985], [Kahn and Miller 1988a], where it is possible for components to evolve in response to changing circumstances, new components be created and old ones killed. However, in open systems these entities are linked in an arbitrary way, whereas in the blackboard model, KSs are grouped around blackboards. Another interesting requirement of an open system is that its components can change their connections with other components during execution. This may have parallels in blackboard architectures, when KSs migrate between subboards at run time.

7.2.2. Polka Blackboard Systems.

In our approach, the various parts of a KS are treated as concurrently executing Polka objects linked via message streams. In addition, instead of a single blackboard data structure, many subboards are used, each controlled by a concurrent object. Control is decentralized, and no scheduler is required. Control KSs generate control information and place it on the blackboards, so allowing the control part of each KS to read it and act upon it. The system also allows a simple form of non-monotonic reasoning and data dependency. All these ideas will be illustrated by Polka code taken from a small scale version of the Hearsay II speech recognition system.

7.2.2.1. Polka Blackboards.

A blackboard will be stored inside a controlling object as an incomplete list, which is a list terminated by a variable. An example is :

```

B = [ 1, 2, 3 | X ]
'X' could be bound to : [ 4, 5 | NewX ]
to make : B = [ 1, 2, 3, 4, 5 | NewX ]

```

Thus the list can have new values added to it while still being referenced by the same variable ('B' in this example). Each element in the incomplete board list will represent a version of the board, with the last one being the most recent version. This is not as inefficient as it may first appear, since a new board will actually consist of pointers to the old version along with the new data. However, this means that pointers must be traversed to find a piece of data in a board. A major benefit is that this technique allows KSs to read blackboard data without sending messages to its controlling object. How this is done will be discussed shortly.

An example of this approach for representing blackboards is the 'bd' class given as program 7.6, which stores its incomplete list in the 'Boards' state variable.

```

bd  Boards state
clauses
  Mesg => Boards = [ B | X ],           % get board B
          self :: do( Mesg, B, NewB ), % process Mesg
          X = [ NewB | X1 ],           % store new board
          Boards becomes X.
end.

```

Program 7.6 : A 'bd' class.

The class accepts one message which can contain anything, since it is a variable. In response, the 'Boards' state is expanded into [B | X], where 'B' is the most recent board and 'X' is a variable. The class also sends itself a message, of the form :

```
do( Mesg, B, NewB )
```

When the message has been answered, 'NewB' will be bound to the new board and this is added to the end of the 'Boards' list, followed by a new variable for any future boards.

The 'bd' class is a basic board class, which allows the latest board on 'Boards' to be accessed and updated. The handling of 'do' messages, which actually alter the board, is separate and is defined by 'oboard', a class that inherits 'bd'. 'oboard' is given as program 7.7.

```
oboard
inherits bd
clauses
    do( change( El, NewEl ), B, B1 ) =>
        change( El, NewEl, B, B1 ) .    % change El for NewEl

    do( del( El ), B, B1 ) =>    del( El, B, B1 ) ;    % delete El

    do( El, B, B1 ) =>    unique( El, B ) : add( El, B, B1 )    % add El if unique
                        else    B = B1 .    % else do nothing

code
    mode unique( ?, ? ) .    % is El unique?
    unique( El, [] ) ;
    unique( El, [ El1 | B ] ) <-
        El =\= El1 &
        unique( El, B ) .

    mode add( ?, ?, ^ ) .    % add El to board B
    add( El, B, [ El | B ] ) .

    mode del( ?, ?, ^ ) .
    del( El, [], [] ) .    % delete El from board B
    del( d( Dt, Valid ), [ d( Dt, Valid ) | B ], B ) <-
        Valid = not_valid ;
    del( El, [ D | B ], [ D | B1 ] ) <-
        del( El, B, B1 ) .

    mode change( ?, ?, ?, ^ ) .
    change( El, NewEl, [], [ NewEl ] ) ;    % change El for NewEl on board B
    change( d( Dt, Valid ), d( NewDt, V ),
        [ d( Dt, Valid ) | B ], [ d( NewDt, V ) | B ] ) <-
        V = Valid ;
    change( El, NewEl, [ D | B ], [ D | B1 ] ) <-
        change( El, NewEl, B, B1 ) .

end.
```

Program 7.7 : An 'oboard' class.

'oboard' inherits 'bd' so that every message from the user will be handled by the inherited clause as 'Mesg'. After extracting the most recent board, 'bd' sends a 'do' message to self with the form :

`do( Mesg, <board>, <new board> )`

This message is dealt with by one of the three 'do' clauses in 'oboard'.

The first clause of 'oboard' handles messages from the user, of the form :

`change( El, NewEl )`

This causes the board to be searched and the element 'El' will be replaced by the new element 'NewEl'. The resulting board will be stored in 'B1' which will eventually be put into 'Boards'.

The second clause handles user messages of the form :

`del( El )`

which causes 'El' to be deleted from the board, with the result being returned in 'B1'. The ';' after the second clause means that the 'do' message must have failed to match against the first two clauses before it tries the third. This implies that the user's message is neither a 'change' or a 'del' message.

The third clause states that the 'El' message will be accepted if it satisfies the guard predicate 'unique', which checks that the message is not stored on the board. If the guard succeeds, the 'add' predicate adds the user's message to the board by creating a new board :

`[ El | B ]`

which consists of the message along with a pointer to the old board. If the guard fails, the else part creates a pointer from the new board to the old one without adding a new element.

In describing the clauses, we have not had to describe the actual structure of an element on the board, since it is hidden inside the Parlog predicates in the code section. A board element consists of a term of the form :

`d( Data , Validity )`

'Data' is the data to be stored, while 'Validity' states whether the data is true or false. When an element is first added to the board, its validity will be true, but if it is deleted, it will become false. Thus, this second argument allows a simple form of non-monotonic reasoning with the blackboard data. 'Validity' is a logical variable, which is initially unbound to mean 'true', and is taken to mean false when it is bound. This approach also enables data dependencies to be set up very simply. For example, if a new element :

`d( c, V3 )`

is created, which should depend on the validity of :

`d( a, V1 ) and d( b, V2 )`

then this can be achieved by using a predicate, such as 'joint' :

```

mode joint( ?, ?, ^ ) .
joint( X, Y, Z ) <-
    data( X ) : Z = not_valid .
joint( X, Y, Z ) <-
    data( Y ) : Z = not_valid .

```

Program 7.8 : A 'joint' predicate.

'data' is a built-in predicate, which suspends until the variable it is given becomes bound. A call to 'joint' may be :

```
<- joint( V1, V2, V3 ) .
```

and might be invoked by the KS creating 'c'. This call creates a small process which will suspend until either 'V1' or 'V2' becomes bound. When one of them does, 'V3' will be set to 'not_valid', and the 'joint' process will terminate. In this way the validity of 'c' has been made to depend on 'a' or 'b'. Other types of dependencies between data can easily be coded using similar techniques and one such method can be seen in the 'change' predicate of 'oboard', which replaces an element in a board by another. More importantly, it also passes on the validity argument of the old element to the new one, which means that the new element is now dependent on all the things that the old one was linked to. The simplicity of the validity argument can be seen in the 'del' predicate, which makes an element false and also explicitly removes it.

'Validity', as described here, can only take on two states, which we have taken to mean true and false, but similar techniques can allow a validity argument to have a multitude of states.

A call to 'oboard' might be :

```
<- oboard( Ins, [ [] | X ], Error ) .
```

'Ins' is the input stream, 'Error' is the error stream, while [[] | X] is the incomplete list of boards, with an empty board, and the future list of boards represented by 'X'. Since 'X' is a logical variable, copies of it can be passed to KSs so that they can read the contents of a board without having to send messages to 'oboard'. Each KS will have to retrieve the latest version of a board from 'X' by searching down it, but this approach means that any number of KSs can read a board without affecting the board's execution. If a KS wants to write on a board, it does so by sending a message along a stream to the board. All these KS message streams will gradually be merged, using a 'merge' predicate. Eventually a single stream will be created which will be used as the input stream into the board ('Ins' in the above example). These merges will enforce an arrival order on the messages sent to the board but their processing

order is still undetermined. 'oboard' processes messages concurrently until a particular message requires an element that has not yet been released by an earlier message. The current message will then suspend until that value becomes free. This suspension will not stop further messages from being processed, although it may constrain their execution.

In summary, the benefits of this approach to blackboard implementation are :

- Very fine grained parallelism is achieved for 'write's to a blackboard, without causing inconsistent updates.
- There is no overhead in the board caused by allowing a KS to read it.
- The object oriented approach means that the process of writing onto a board has been separated from the way the board is kept up to date.
- 'oboard' can be specialized by being inherited by other 'board'-type classes. This was done for some boards in the Hearsay II implementation, discussed later.
- Data dependencies and simple non-monotonic reasoning have been implemented with hardly any overhead.

7.2.2.2. Polka KSs.

As outlined in section 7.2.1.4, each KS will be an independently executing object which will consist of four basic parts, represented by figure 7.10.

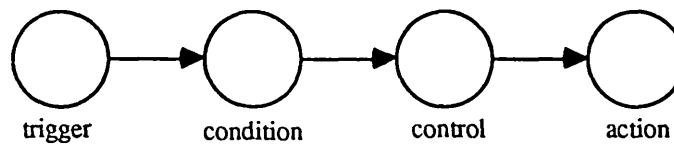


Figure 7.10 : A Polka KS.

In fact, each of these parts, called sub-KSs, will be an object which will be linked to its neighbours via message streams to create a complete KS. In theory, each of the sub-KSs can be executing at once, but it is more likely that many of them will be suspended. This suspension means that either the trigger, condition, control or action sub-KSs could not execute with the new data and the subboards at their disposal.

Each sub-KS will inherit a 'ks' class, which is the basic building block for a sub-KS. It implements the message passing mechanism needed for communication between sub-KSs, and also the mechanism for finding the latest version of a board on a 'Boards' list. 'ks' is defined by program 7.9.

```

ks  Boards state, Out ostream
clauses
  Mesg      =>  find( Boards, #Boards, B ),      % find board B
              self :: test( Mesg, B, Out, #Out ) . % test Mesg
code
  mode find( ?, ^, ^ ) .
  find( [ B | X ], [ B | X ], B ) <-
    var( X ) : true ;
  find( [ OldB | Bs ], NBs, B ) <-
    find( Bs, NBs, B ) .
end.

```

Program 7.9 : A 'ks' class.

It has one state variable called 'Boards', which the KS will use for its board list, and 'Out', which is an output stream to either another sub-KS or perhaps to a subboard. 'Boards' is an incomplete list of boards, so when a message arrives, the list must be searched through using 'find' until the latest board 'B' is found. A new board list is also generated from the tail of the present 'Boards' list, in order that its full length does not need to be searched when another message arrives.

In an analogous way to 'bd', the testing of a message is carried out by sending a 'test' message to self, and clauses which deal with such messages are defined by the objects which inherit 'ks'. In addition to 'Mesg' and 'B', 'test' also includes the 'Out' message stream, so that any results can be output, and the tail of the stream is returned and saved for future use.

The various sub-KSs for 'trigger', 'condition', 'control' and 'action' all inherit 'ks' and define the meaning of 'test' messages. For example, the definition of the 'syllable' KS from the Hearsay II implementation is composed from four sub-KSs : 'syl_trigger', 'syl_condition', 'syl_control' and 'syl_action'. Each of these inherits 'ks' and defines a different form of 'test'. The 'syl_control' sub-KS is defined by program 7.10.

```

syl_control
inherits ks
clauses
    test( Msg, Board, O, O1 ) => focus( Msg, Board ) :    % is Msg important?
                                O = [ Msg | O1 ]           % then transmit it
                                else
                                O = O1 .                  % else ignore it

code
    mode focus( ?, ? ).
    focus( Msg, Board ) <-
        high_pri( Board ),                                % have syllables got a high priority?
        min_syl( Board, Min ) &                          % get the minimum rating
        f_check( Msg, Min ) .                             % check Msg rating

    mode high_pri( ? ).
    high_pri( [ d( high_priority( syllable ), Valid ) | B ] ) <-
        valid( Valid ) : true ;
    high_pri( [ D | B ] ) <-
        high_pri( B ) .

    mode min_syl( ?, ^ ).
    min_syl( [ d( minimum( Rate ), Valid ) | B ], Rate ) <-
        valid( Valid ) : true ;
    min_syl( [ D | B ], R ) <-
        min_syl( B, R ) .

    mode f_check( ?, ? ).
    f_check( d( syl( Syl, BT, ET, CR ), Valid ), R ) <-
        valid( Valid ) &
        R =< CR .

    mode valid( ? ).
    valid( V ) <-
        var( V ).

end.

```

Program 7.10 : A 'syl_control' class.

The 'focus' predicate is used in the guard of the first clause to check the priority of the KS, and whether the syllable in the message has a sufficiently high rating. The priority is checked by calling 'high_pri' which looks through the board for valid 'high_priority' data. If the data is invalid or not present then 'focus' will fail, but if data does exist then the minimum rating is extracted from the board using 'min_syl'. If the 'minimum' data is invalid or not present then 'focus' will also fail. Only if both searches are successful, is 'f_check' called, which checks the rating retrieved from 'min_syl' against the rating in the message. Data in a message will be of the form :

```
syl( <syllable>, <begin time>, <end time>, <rating> )
```

If the data is valid and its rating is higher than the minimum required, then 'focus' will succeed, which will cause the message to be sent out on the output stream in 'ks'. If

'focus' fails then the message is ignored by 'syl_control', which will suspend until another message arrives.

The validity of the 'high_priority' and 'minimum' data could have been tied to that of 'syl', using the 'joint' predicate defined earlier, and this would have caused the 'syl' data to become invalid if the priority or the rating changed.

A call to 'syl_control' could be :

```
<- syl_control( Ins, Boards, Out, Error ) .
```

where 'Ins' is the input stream, 'Boards' is the boards list, 'Out' is the output stream, and 'Error' is the error stream. Similar definitions can be obtained for 'syl_trigger', 'syl_condition' and 'syl_action', which can be combined to produce 'syllable' :

```
mode syllable( ?, ?, ?, ?, ^ ) .
syllable( In, TBoards, CdBoards, CtBoards, ABoards, Out ) <-
    syl_trigger( In, TBoards, Out1, _ ) ,
    syl_condition( Out1, CdBoards, Out2, _ ) ,
    syl_control( Out2, CtBoards, Out3, _ ) ,
    syl_action( Out3, ABoards, Out, _ ) .
```

Program 7.11 : A 'syllable' predicate.

For the purposes of this discussion, the various error streams for the sub-KSs will be ignored.

The use of logical variables to represent message streams means that the composition of objects is very simple. Each sub-KS is executing in parallel internally, and because they have been composed together using ',', each will execute in parallel inside the 'syllable' KS. Thus the parallelism of the KS is maximised.

Diagrammatically, 'syllable' can be viewed as figure 7.11.

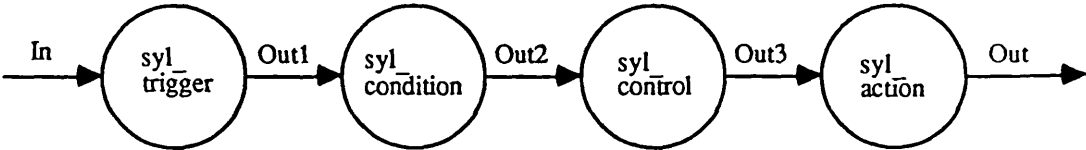


Figure 7.11 : The four components of 'syllable'.

Each line is a message stream, each circle an object. 'syllable' can also be viewed as figure 7.12.

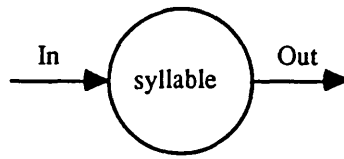


Figure 7.12 : 'syllable' as one entity.

'In' is an input stream, merged together from the streams going into the various subboards in the system, and passed to 'syl_trigger', which can then monitor all the new data being sent to the subboards. 'Out' is an output message stream which will be merged into the inputs of all the relevant subboards, so enabling 'syl_action' to transmit data to them. 'TBoards', 'CdBoards', 'CtBoards' and 'ABoards' are the boards used by the sub-KSs of 'syllable'.

Further refinements of this KS model are possible. For instance, in the Hearsay II implementation, 'syl_trigger' and 'syl_condition' actually inherit a class called 'ks_list', which allows lists of data to be stored in the sub-KS. This is useful if a particular piece of data is rejected but should be stored and perhaps used later when the state of the board has changed. This change causes only a small alteration in the definition of 'syllable', which must connect some extra streams between the sub-KSs.

All the other KSs in the Hearsay II implementation have similar definitions to that of 'syllable', and inherit most of what they need from 'ks' or 'ks_list'.

The benefits of this approach to KSs are :

- A high degree of parallelism is possible, which can be constrained by using control data on subboards to suspend some or all of the sub-KSs of a KS.
- Much of the functionality of a new KS can be inherited from KS classes already present.
- The use of logical variables as message streams makes the composition of sub-KSs very simple.

7.2.3. A Version of Hearsay II.

Using the techniques outlined in the previous sections, a small version of the Hearsay II blackboard system has been programmed. Hearsay II is a speech recognition system which starts by grouping sound input into syllables. Each syllable is given a begin time, end time and a rating to indicate how credibly the syllable matches the sound. These syllables are built up into words, then word sequences. These sequences are partially parsed and predictions are made as to what the full parse might be. These are verified against the actual input data and longer parses may be produced. Eventually a parse covering the entire input is obtained and this is used to query a database. The blackboard solution to this problem required the main blackboard to be partitioned into 7 subboards, and used 13 KSs [Erman et al. 1980].

Our system is considerably simpler and uses 7 subboards and 8 KSs. The subboards are called 'syllable', 'word', 'word_sequence', 'big_sequence', 'parse', 'predict' and 'stop'. The 'stop' subboard is the equivalent of Hearsay's 'data-base interface' subboard, but no database is linked to it. The 8 KSs in our system are called 'kssyl', 'wordformer', 'word-seq', 'parser', 'predictor', 'verifier', 'concat' and 'stopper'. These do much the same things as Hearsay's 'mow', 'wordseq', 'parse', 'predict', 'verify', 'concat' and 'stop'. Hearsay's scheduler was not required in the system because of the use of control subboards in our KSs.

Our system starts with syllables, and so misses out Hearsay's sound transformation stage. The grammar used is fairly elementary and there is a dictionary of only about 30 words. However, the system still has complex problems to solve and each KS is correspondingly complicated. Fortunately, this was fairly easy to deal with since the coding of each KS only required the writing of four classes for the trigger, condition, control and action sub-KSs of each KS. The code produced was very like what has been discussed here, although the 'oboard' class was specialized for the trigger and condition subboards. This was not a difficult task, since the new board class inherited most of its functionality from 'oboard'.

A typical execution starts with syllable data, such as :

```
syl(t,1,2,7), syl(n,15,16,7), syl(ih,2,3,7), syl(aa,14,15,7),
syl(f,5,6,7), syl(l,6,7,7), syl(ow,19,20,7), syl(m,3,4,7),
syl(iy,7,8,7), syl(s,8,9,7), syl(r,18,19,7), syl(aa,17,18,7),
syl(l,10,11,7), syl(iy,11,12,7), syl(k,12,13,7)
```

being input to the syllable data subboard. Each term has four arguments. The first is the syllable, the second its begin time, the third its end time, and the fourth is its credibility rating. During the execution, the KSs will monitor, read data from, and add data to the various subboards until the 'stop' data subboard receives data of the type :

```
stop( [ adj( time ), noun( flies ), verb( like ), det( an ), noun( arrow ) ], 1, 20, 105 )
```

This term also has four arguments, similar to those for syllable. However, the first argument is a simple parse of a sentence which is the system's (correct) interpretation of the syllable input.

In the running system, input to the subboards is made visible by being output to windows on a SUN 3, which incidentally were generated by a window manager written in Polka, using an object for each window. The execution of the above example is shown in figure 7.13. The top left hand window shows the input to the 'syllable' subboard, and the input to the 'stop' subboard is shown in the bottom window on the right of the screen.

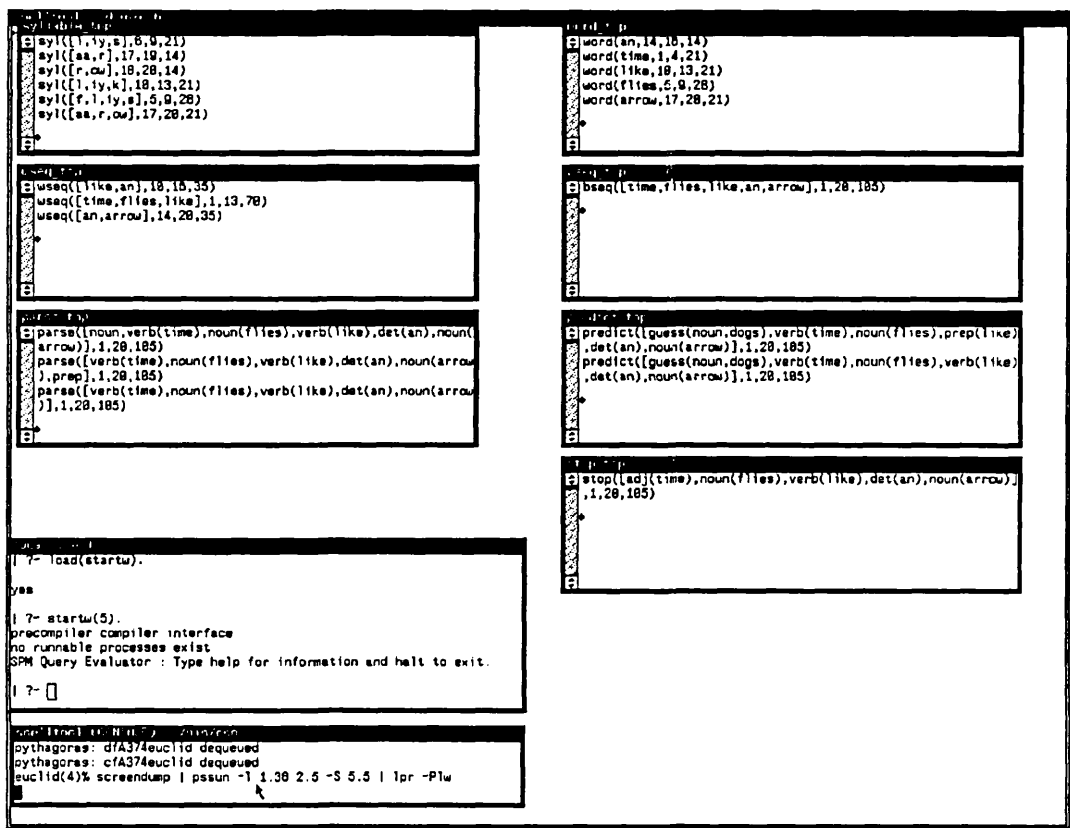


Figure 7.13 : Screen dump of an execution of the Hearsay implementation.

7.2.4. Summary of the Blackboard Example.

We have looked at the use of Polka for the implementation of blackboard architectures and, in general, concurrently executing objects communicating via message passing seem to map easily onto the blackboard formalism.

For blackboards, it was relatively straight forward to implement a class which separated the manipulation of the board from the mechanism to find the data on that board. The implementation also maximised the potential parallelism while avoiding the multiple writers problem. Data could be deleted, changed and data dependencies could be created, allowing simple forms of non-monotonic reasoning. In addition, specialized boards and KSs could easily be created from others, by using inheritance. A KS consists of four sub-KSs, each handling a different KS operation, which can be quickly created by inheriting most of its functionality from predefined KS classes.

The problem of control was considered, and by including trigger and control sub-KSs in each KS, it was found that a central scheduler was not necessary. This removes a major bottleneck from the blackboard model while retaining a high degree of control over the execution of KSs.

These ideas were tested by implementing a small scale version of Hearsay II, and although simpler than the original, it is still complex enough to demonstrate the benefits of using the blackboard model described here.

7.3. Simulation.

The object oriented paradigm has been successfully used in simulation for many years and has resulted in a number of programming languages being developed, including DEMOS [Birtwistle 1979]. However, DEMOS is a sequential language which relies on a global clock to order events. This example illustrates how many of the ideas of DEMOS can be applied to the creation of simulation classes. In addition, by using a distributed discrete-event simulation algorithm, described by Misra [Misra 1986], it is possible to create simulations which do not require a global clock. Two simulations are outlined to show how these classes can be used : one is a car wash model, and the other a simple information retrieval system.

7.3.1. Overview of Simulation.

In its most general sense, simulation is a way of representing certain aspects of the behaviour of a physical or abstract system by the behaviour of another system. In computing, this other system will be a computerised model, and such models have been used extensively, mainly because of the ease of experimenting with them rather than with the real system. Another important consideration is the cost, since programming a simulation will generally be much cheaper than building the physical system.

There are many types of simulation model but a common classification is into continuous and discrete models. In continuous-variable models, equations are used as the basis of the simulation, which means that its variables will change continuously as time changes. Discrete-event models are easier to represent on a digital computer because state changes are abrupt, caused by events such as resource allocation and deallocation. Within discrete-event simulation there are a number of subdivisions, which model events differently, such as event scheduling, activity scanning and process interaction [Fishman 1978]. The one we will investigate here is the process interaction model, which considers an event to have occurred when two processes communicate. What exactly a process is depends on the detail of the simulation, but generally it can be any entity or resource. One important benefit of this model is that a process can be easily mapped onto a Polka object, and the communication between the processes can then be seen as message passing.

7.3.1.1. Simulation in Logic Programming Languages.

The process interaction model was used by Broda and Gregory [Broda and Gregory 1984] to write simulations in Parlog, so allowing the various processes in

the system to execute concurrently. Unfortunately, their approach relied on a global clock to make sure that events occurred in chronological order. This meant that events which could occur simultaneously were unnecessarily sequentialised while they obtained a clock pulse. Also, the clock process became a bottleneck in the system, since all the processes which generated events had to access it. The clock also required a global deadlock detection mechanism, which can be complicated and expensive to implement.

A possible solution was put forward in [Elshiewy 1987], but this required extensions to Parlog, which made its semantics more complex. A more practical drawback was the assumption that every process in a simulation executes at roughly the same rate. On distributed hardware, this is unlikely to be correct, since different processes can be mapped to processors on different machines.

The process interaction model was also used by Cleary for simulation programming in Concurrent Prolog (CP) [Cleary et al. 1984]. Again global time was required, and the language semantics were also extended so a process could include a delay as part of its commit operation. This delay can be seen as an implicit version of Broda and Gregory's approach, so would require a global deadlock detection mechanism in order to work. One of Cleary's example uses a further extension which allows it to backtrack to produce alternative solutions. How this backtracking would interact with CP's committed choice execution strategy was not explained.

Simulation models in Prolog have also been proposed, notably by Futo and Szeredi [Futo and Szeredi 1984], who also considered discrete-event simulation. They introduced a form of coroutining into their version of Prolog, called T-Prolog, which allowed then to represent entities as Prolog processes. However, they also had the problem of reconciling backtracking with concurrent execution, and required a global clock.

The absence of backtracking in Parlog is not as serious as it may first seem. Parlog contains all-solution primitives [Clark and Gregory 1986], which can return all the answers to a query when it is applied to a set of clauses. Also, it will soon be possible to call Prolog from Parlog and vice versa [Clark and Gregory 1987; Bahgat and Gregory 1988]. An alternative approach is to use Parlog meta-interpreters to enhance the execution strategy of Parlog programs to allow them to backtrack, which was done in [Gregory et al. 1985]. The applicability of these techniques to simulation will not be examined, but it seems likely that they will prove useful.

7.3.2. Distributed Time.

The simulation approach used in Polka, which was suggested by work being done by Lam in Parlog [Lam 1987], is to use the distributed discrete-event simulation algorithm described by Misra in [Misra 1986]. Misra represents a simulation as a system of processes, which are connected via directed channels, and communicate by sending messages to each other along these channels. Where his work differs from that of [Broda and Gregory 1984] is that there is no global clock for scheduling events. Instead, each process has its own clock, which is kept in synchronisation with other clocks by timestamping the messages which are sent between the processes. Generally, a process will have several input channels, and to obtain a message it must scan all these input channels and select the message with the least time. This time will become the new local time of the process, and the message will be processed, so causing output messages to be transmitted. The timestamp on these transmitted messages will correspond to the local time of the process. Also, a process must regularly send messages out on all of its output channels, to ensure that its local time is propagated to its neighbours. If a real message can not be sent then the process sends a dummy one, called a null message.

The algorithm assumes that each message takes an arbitrary but finite time to reach its destination, and that messages sent along a channel are delivered in the sequence in which they are sent. In addition, each process is assumed to have an infinite buffer associated with each input channel.

There are a number of ways in which the efficiency of the algorithm can be improved, including the delaying of the transmission of null messages and the manipulation of input buffers. The delaying technique is useful because if a real message is sent just after a null message, then the null need not have been sent, thus reducing the inter-process message traffic. In the buffer method, the buffers for each input channel are analysed and any nulls which occur ahead of real messages are removed. This will speed up the execution of the process, since it will no longer have to handle so many null messages, which do not advance the simulation. Both of these techniques will be used in our simulation classes. Something not possible in Misra's approach is bidirectional communication, which is provided in Polka by using the back communication technique. This must be handled carefully so as not to invalidate the basis communication scheme, but is particularly useful for decreasing the number of separate streams between objects.

Deadlock can occur, but fortunately if the physical system deadlocks then the simulation continues, but only null messages with increasing time values are passed between the objects. Thus, it is not difficult to detect deadlock, and the harder

problem is to recover from it, or to avoid it in the first place. Deadlock recovery and avoidance schemes do exist, but they will not be investigated here.

7.3.2.1. Distributed Time in Polka.

A typical object will have several input and output streams, which are not the same as Misra's channel, since bidirectional communication is possible. The object could be represented in diagram form by figure 7.14.

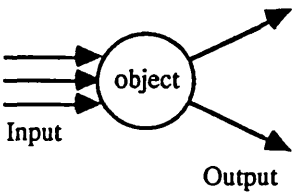


Figure 7.14 : A typical Polka object.

Each of the input streams needs to be scanned to produce a single stream of messages with increasing minimum times. In addition, the scanning process will delete any nulls which are followed by real messages. A monitoring object will also be added to each output stream to delete transmitted null messages if they are closely followed by real messages. The diagram of the object will now look like figure 7.15.

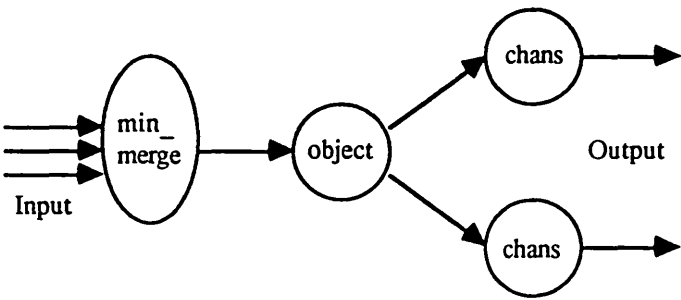


Figure 7.15 : A distributed time object

'min_merge' is a Parlog predicate which does the job of the scanning process, while 'chans' is an object which monitors the output stream it is inserted into. Neither will be discussed here, although their code can be found in a paper entitled "Simulation techniques in Polka" [Davison 1988a].

A typical call to 'min_merge' is given below. Its first argument is a list containing the input streams, while its second is the resulting output stream. In this example, 'min_merge' has two input streams entering it :

```
<- min_merge( [ In1, In2 ], Msgs ),
In1 = [ m( 2, a ), m( 4, null ) | _ ],
In2 = [ m( 3, b ), m( 5, null ) | _ ].
```

which will bind 'Msgs' to :

```
[ m( 2, a ), m( 3, b ), m( 4, null ), m( 5, null ) | _ ]
```

This call illustrates how messages are represented. A message 'M' sent at time 'T' will be a term of the form :

```
m( T, M )
```

A null message sent at time 'T' will be :

```
m( T, null )
```

A typical call to 'chans' is :

```
<- chans( Ins, Out, 10, Error ),
Ins = [ m( 4, a ), m( 5, null ), m( 6, b ) | _ ].
```

The first argument is the input message stream, the last is the error stream, 'Out' is the output stream, while 10 is a value for how long messages are to be delayed inside 'chans' before being sent out on 'Out'. 'Out' will be bound to :

```
[ m( 4, a ), m( 6, b ) | _ ]
```

The three messages will be buffered for a short time and because the third message was real, the null message was deleted. Thus, when the buffer was flushed only the first and third messages were output. The delay is not reflected in the timestamps of the messages, since it is not part of the simulation.

We are now in a position to put 'min_merge' and 'chans' together with an object. The earlier diagram of an object, augmented with a name for each message stream (except error streams), is shown as figure 7.16.

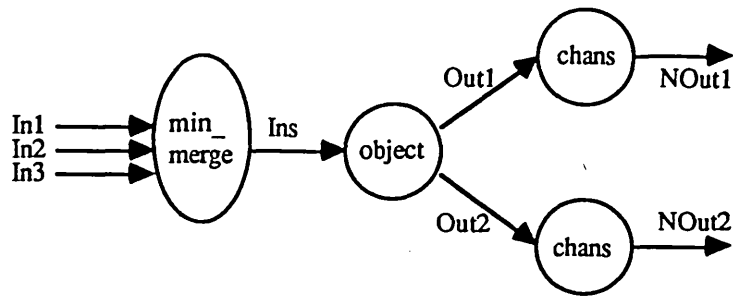


Figure 7.16 : A distributed time object with named streams.

Assuming that the object is called 'entity', then its outline definition is :

```

entity Out1, Out2 ostream
:
end.

```

Then the diagram is coded in Parlog as :

```

<- min_merge( [ In1, In2, In3 ], Ins ),
   entity( Ins, Out1, Out2, _ ),           % error streams are ignored
   chans( Out1, NOut1, 10, _ ),
   chans( Out2, NOut2, 10, _ ).

```

The 'min_merge' predicate and the classes will execute concurrently because of the use of the ',' between them. This also illustrates how Parlog's logical variable can be used to connect predicates and classes together. This linking can be hidden by defining a new predicate called 'entity1' :

```

mode entity1( ?, ?, ?, ^, ^, ? ).
entity1( In1, In2, In3, NOut1, NOut2, Limit ) <-
  min_merge( [ In1, In2, In3 ], Ins ),
  entity( Ins, Out1, Out2, _ ),
  chans( Out1, NOut1, Limit, _ ),
  chans( Out2, NOut2, Limit, _ ).

```

Program 7.12 : An 'entity1' predicate.

which would be called like so :

```

<- entity1( In1, In2, In3, NOut1, NOut2, 10 ).

```

The diagram of 'entity1' would be :

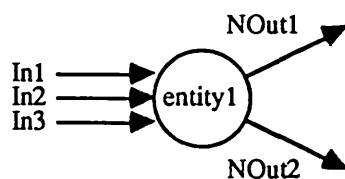


Figure 7.17 : An invocation of 'entity1'.

which shows how the mechanism for obtaining the message with the least time and for manipulating null messages is hidden. This technique will be used when writing all the simulation examples.

In general, a simulation object will input and output messages of the form :

$m(T, \text{null})$

and $m(T, \text{Msg})$ where 'Msg' is a real message

The object must also make sure that when it sends a message out on one output stream that it will eventually output messages on the other output streams as well. This requirement need not be left up to the object, and an alternative is to automatically generate nulls on all the output streams when one of them is given a real message. This can be coded up easily but leads to a proliferation of nulls which will degrade the system too severely.

Apart from these few assumptions, an object does not need to handle the selection of a message with a minimum timestamp, or delete nulls. That is done by putting a 'min_merge' in front of the object's input and passing the object's output through 'chans' objects.

The encoding of Misra's algorithm is a good example of how Polka classes and Parlog predicates can be mixed. The object itself and 'chans' are written in Polka, but 'min_merge' is a Parlog predicate.

7.3.3. Polka Simulation Classes.

Polka is particularly suitable for coding up concurrently executing entities which communicate via message passing. It is also straight forward to specify internal data and procedures for these entities, which means that there is a natural mapping from a discrete-event simulation to a Polka program. In addition, when the distributed time mechanism of the last section is included, a framework is present for a concurrent implementation which does not require a global clock. Polka also offers inheritance, which means that entities from the model can be coded by specialising more general

entities. But what should these entities be? The DEMOS simulation package [Birtwistle 1979] is of some help in this respect.

DEMOS is implemented in SIMULA [Birtwistle et al. 1973], and uses its class mechanism to define a set of discrete-event simulation tools which can be specialised, via inheritance, for particular simulation models. In DEMOS, a simulation is constructed out of entities which represent the various active participants in the model. These entities compete with each other for system resources, cooperate with each other to perform a sequence of task, or even interrupt one another. Among the predefined classes of DEMOS are : an 'entity' class, which can be specialised for particular entities in the application, classes for two types of resources, and three 'queue' classes. In addition, there are various data collection classes for generating histograms, tallies and counts, and also classes for different types of random number generator. Some of these classes will be used as the basis for Polka simulation classes. DEMOS also contains a number of useful procedures, and the analogy in Polka will be Parlog predicates. These predicates will be utilised for manipulating streams between objects, which is easier to do in Parlog because of the way that streams are represented by logical variables. Typical examples are predicates like 'merge' or 'min_merge'.

Currently, Polka simulation classes have been defined for :

- random number generators
- message generators
- resources
- queues
- report generation

Rather than describe all of these classes, which would take a considerable amount of time, only the random number generators, resources, and report generators will be explained. Information on the others can be found in [Davison 1988a].

7.3.3.1. Random Number Generators.

Most simulations require a random element, and so an object which returns a sequence of random numbers is particularly useful. It is known that if a generator exists which returns a sequence of numbers from a uniform distribution, then sequences from other distributions can be generated by using it [Fishman 1978].

7.3.3.1.1. Uniform.

A typical random number generator for a uniform distribution can be found in [Birtwistle 1979], and its Polka version is given as program 7.13.

```
uniform Seed state
initial  mod( Seed, 10, T1 ),
        Seed becomes T1 + 1 .           % modify Seed
clauses
  next( Num ) => T1 is 2 * Seed ,
                mod( T1, 11, T2 ) ,
                Num = T2 / 11 ,         % generate a number
                Seed becomes T2 .      % store a new Seed
end.
```

Program 7.13 : A 'uniform' class.

'uniform' has one visible state variable called 'Seed', which the user will be able to give a value to when the class is invoked, and the **initial** section converts it into a number between 1 and 10. When a 'next' message arrives, 'Seed' is used to generate a random number 'Num' and a new value for 'Seed' is stored ready for the processing of the next message. A typical call to 'uniform' is :

```
<- uniform( Ins, 9, Error ) ,
Ins = [ next( N1 ), next( N2 ) ] .
```

The first argument of the call is the default input stream, the second is an initial value for 'Seed', while the last argument is the error stream. The bindings of the variables in the messages may be :

```
      N1 = 0.273
and    N2 = 0.091
```

7.3.3.1.2. Randint.

'randint' generates random integers in the range [A, B] by specialising the 'uniform' class.

```
randint inherits uniform
  A, B state
clauses
  next( Num ) => super :: next( N ) ,
                Num is A + int( ( B - A + 1 ) * N ) .
end.
```

Program 7.14 : A 'randint' class.

The class inherits 'uniform', and has two visible state variables 'A' and 'B' for the range. 'randint' specialises the action of the 'next' message, by transmitting a **super** message. This obtains a random number 'N' which is then used to calculate another random number 'Num' in the range [A, B]. A call to 'randint' might be :

```
<- randint( Ins, 2, 5, 9, Error )
```

The second and third arguments are for the 'A' and 'B' state variables from 'randint', while the fourth argument is for the inherited 'Seed' state variable from 'uniform'. 'randint' could be sent messages of the form :

```
Ins = [ next( N1 ), next( N2 ) ] .
```

which might produce :

```
N1 = 4
```

```
and N2 = 3
```

Other types of random number generator with other distributions can easily be encoded. Examples found in DEMOS include ones for a negative exponential distribution, a normal distribution, and an Erlang distribution.

7.3.3.2. Resources.

A 'resource' object is one which contains a finite number of resources, which it will allocate to objects that send it messages. However, if there are not enough resources, the requesting object must wait. When an object finishes with some resources, it should return what it has used, which may then allow a suspended object to be granted its request.

The way that resources will be acquired are via :

```
m( Time, acquire( Number, Sender ) )
```

messages. 'Number' stands for the number of resources requested, while 'Sender' is a variable which will be bound to :

```
acquired( Time1, Resources )
```

when the resources are given to the requester. Since the requesting object will be suspended while its request is processed, 'Time1' indicates the time when the resources were acquired, and so 'Sender' is being used as a reply variable. The distributed time algorithm is not invalidated, since this resource allocation scheme only

means that an object will not be able to accept any messages between time 'Time' and 'Time1'. This can be seen as a more realistic version of 'hold', which is used in several simulation languages to simulate the time an object takes to do something, such as obtain resources. The Polka version is more realistic because the delay is calculated by the 'resource' object rather than by the requester.

When an object has finished with a resource, it returns its allocation by sending :

```
m( Time, release( Number, Resources ) )
```

to 'resource'. 'resource' can also accept null messages :

```
m( Time, null )
```

An outline definition of 'resource' is given as program 7.15.

```
resource Name, Number state
invisible ResList state, WaitQ state <= []

initial create_res( 0, Number, #ResList ) .           % create resources
clauses
  m( Tm, acquire( No, Sender ) ) =>
    Number < No :                                     % not enough resources
    append( WaitQ, [ acquire( No, Sender ) ], #WaitQ ) . % so requester waits

  m( Tm, acquire( No, Sender ) ) =>
    No <= Number , WaitQ = [] :                       % enough resources
    rem_res( No, ResList, #ResList, Res ) ,           % and no one waiting
    Sender = acquired( Tm, Res ) ,                   % so allocate resources
    Number becomes Number - No .

  m( Tm, acquire( No, Sender ) ) =>                   % enough resources
    No <= Number , WaitQ <= [] :                     % and somebody waiting
    retry( Tm, WaitQ, [m(Tm, acquire(No, Sender)) | Ins], #Ins ) ,
    WaitQ becomes [] .

  m( Tm, release( No, Res ) ) =>
    append( ResList, Res, #ResList ) ,               % resources returned
    retry( Tm, WaitQ, Ins, #Ins ) ,
    Number, WaitQ becomes Number + No, [] .

  m( Tm, null ) => true .                             % ignore null message

code
  mode create_res( ?, ?, ^ ) .                       % predicates only given in outline
  mode rem_res( ?, ?, ^, ^ ) .
  mode retry( ?, ?, ?, ^ ) .
  mode res_name( ?, ^ ) .

end.
```

Program 7.15 : A 'resource' class.

'resource's two visible variables are for the name and the number of resources that it will look after. The invisible variables include 'ResList', which is a list where resources will be stored in the form :

`r < number >`

'WaitQ' is a queue for storing requests for resources which can not currently be granted. A suspended request looks like :

`acquire( Number, Sender )`

The **initial** section calls the local predicate 'create_res' to insert 'Number' resources into 'ResList'.

Of the five clauses, the first three are to deal with 'acquire' messages. The first clause deals with the case when 'resource' does not have enough resources to meet the request. That results in the 'acquire' message being put onto the 'WaitQ' list.

The second clause deals with the case when there are enough resources and 'WaitQ' is empty. Then the specified number of resources are removed from 'ResList' by using 'rem_res'. These resources and the time of the message are sent back to the sender, and the number of resources is decremented.

The third clause deals with the case when there are enough resources but 'WaitQ' is not empty. Then 'retry' is called which adds all the suspended 'acquires' back onto the input stream but with message times corresponding to 'Tm', the time of the current message. Thus each 'acquire' will be retried but with a later timestamp. The new input stream will contain all the old 'acquire' requests followed by the current 'acquire' and then any future messages to 'resources'. Since 'WaitQ' has been emptied, it must be set to [].

The fourth clause reclaims resources released by an object, and also calls 'retry' to retry all the suspended 'acquire's on 'WaitQ'.

The final clause states that null messages are ignored. The code section will not be discussed, but is about twenty lines long.

A call to 'resource' might be :

`<- resource( Ins, car, 3, Error )`

This 'resource' object will manage three car resources. When the input messages are :

`Ins = [m(5, acquire(2, S1)), m(7, acquire(2, S2)),
m(9, release(2, [r1, r2])) | _] .`

this produces :

`S1 = acquired( 5, [ r1, r2 ] )`

and `S2 = acquired( 9, [ r3, r1 ] )`

The second 'acquire' at time 7 is suspended until time 9 when the resources obtained at time 5 are released. 'S2' gets [r3, r1] because of the way that released resources are added back to 'resource'.

7.3.3.3. Report Generation.

Most of the larger Polka programs written so far use or specialise a set of graphics classes designed to manipulate windows on the SUN 3 Workstation. These classes have also been used in the simulation programs and three new classes have been defined which make the graphics interface more suitable for simulation problems. These classes are called :

- `trace`
- `text_report`
- `icons_report`

The 'trace' class allows trace information to be sent to a text window by using a 'trace' message, and tracing can also be switched on and off.

The 'text_report' class must be used in conjunction with an existing text window, and supports the functionality necessary for textual output to that window. In fact, 'text_report' is inherited by 'trace' to enable trace information to be sent to a window.

The 'icons_report' class requires a window which can display icons, and allows *pos-icons* to be created at an (X,Y) coordinate inside the window. Pos-icons are icons which remain fixed at a certain position, and so can be used as locations where ordinary icons should be created. Also, by specifying the name of a destination pos-icon, icons can be moved from one pos-icon to another. This kind of functionality is useful in simulation programs, because stationary entities can be represented as pos-icons, and messages can be ordinary icons. This enables the sending of a message to be represented by the movement of an ordinary icon from one pos-icon to another. Thus, the simulation can be animated, and so convey a feeling for what is happening, which is harder with textual output.

7.3.4. Two Simulation Examples.

In this section, two examples which use the simulation classes are outlined. The first is a version of one of Broda and Gregory's car wash examples [Broda and Gregory 1984], and the second is a simulation of a simple information retrieval system based on a DEMOS program in [Birtwistle 1979]. At first glance, they do not appear to have much in common, but they actually inherit many of the same simulation classes.

7.3.4.1. A Car Wash.

This simulation has three main entities : the car generator, the queue, and the car wash. The car generator periodically generates cars, which are added to a queue of cars waiting to use the car wash. At the same time, the car wash will try to acquire a worker resource in order to wash a car. The car wash may be delayed at this point since there are only a maximum of three workers, all of which may already be washing cars. When it has obtained a worker, it can then choose a car from the queue based on a priority assigned to each car by the car generator. Once a car and worker have been acquired by the car wash, the car is deemed clean after a short time and it leaves the simulation, and the worker returns to the resource pool. An added feature is that each car is itself an entity, with its own decision making ability, which means that if a car is kept waiting in the queue for too long, then it may decide to leave the simulation without being washed.

Each object has its own clock, uses timestamped messages to communicate, and is specialised to some degree from the simulation classes described in the earlier sections. The car generator, queue and car wash send messages to an 'icons_report' object which puts pos-icons representing them into an icon window. In addition, the car generator invokes a new car object for every car it creates, and these communicate with 'icons_report' to move an ordinary icon representing a car between the three pos-icons. Also, the car wash object manages the worker icons. Apart from the icon window, three text windows are also created for the car generator, queue and car wash.

A screen dump from a typical execution of the program is in figure 7.18.

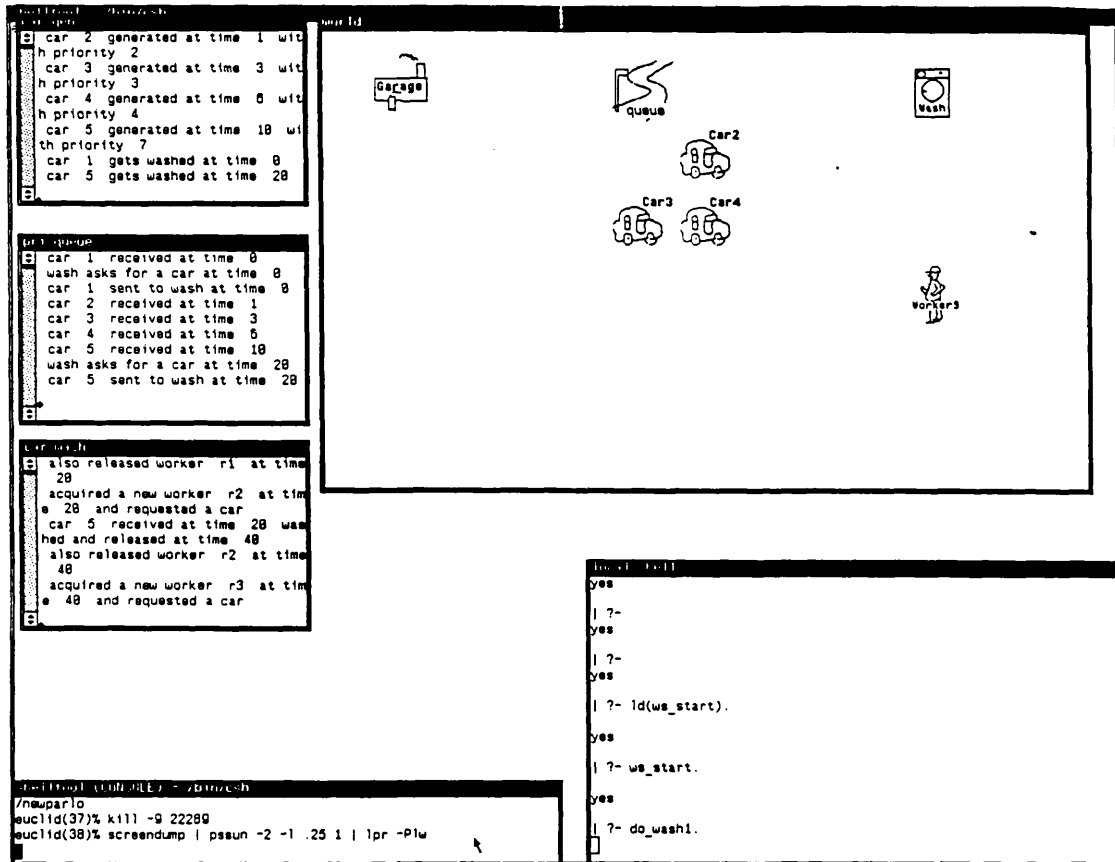


Figure 7.18 : Screen dump of an execution of the car wash.

The icon window at the top right shows the three pos-icons for the car generator (the garage), queue and car wash. In this example, two cars have already been processed by the car wash and have disappeared, along with their workers. A third worker has just started work and a car is about to be taken from the queue. Interestingly, the cars which have been washed are 'car1' and 'car5'. 'car1' was chosen because it was the only car in the queue at time 0, but 'car5' was chosen at time 20 because its priority was higher than cars 2, 3 and 4 which were also in the queue. The current time is 40, and shortly after this both 'car2' and 'car3' leave the queue of their own volition because they were kept waiting too long. 'car4' is then chosen by the car wash because it has the highest priority.

The text windows allow detailed time information to be output, while the icon window gives an overview of how the various entities are interacting.

7.3.4.2. An Information Retrieval System.

This simulation model contains two terminals, both of which may have pending queries for the system. A scanner checks each terminal in turn and if there is one or more queries waiting then it attempts to answer one of them. It does this by allocating a processing buffer resource to the query, but there is only a maximum of three such buffers, so if there are none currently available, then the scanner suspends. Once a query has been allocated a processing buffer, the scanner continues onto the next terminal, and so it is possible for at most three queries from any terminal to be processed simultaneously. This is less restrictive than the DEMOS version, from which this simulation is derived, which only allows one query per terminal to be dealt with at once. After a certain amount of time, the processing of a query is assumed to have been completed and it disappears from the simulation and the buffer returns to the resource.

As in the last example, each object has its own clock and communicates via timestamped messages. The query generator, the two terminals and the scanner all send messages to an 'icons_report' object which will put pos-icons representing them into an icon window. The query generator produces messages which stand for queries, and ordinary icons are used to represent each query as they migrate between the pos-icons. The scanner can also create ordinary icons, which stand for processing buffers. One totally new class was required, called 'router', which routes generated queries to their assigned terminal, and a new stream manipulation predicate also had to be written which allowed the scanner to rotate through the terminals. The query generator, the terminals and the scanner also send information to text windows.

A screen dump of an execution of the program is given in figure 7.19.

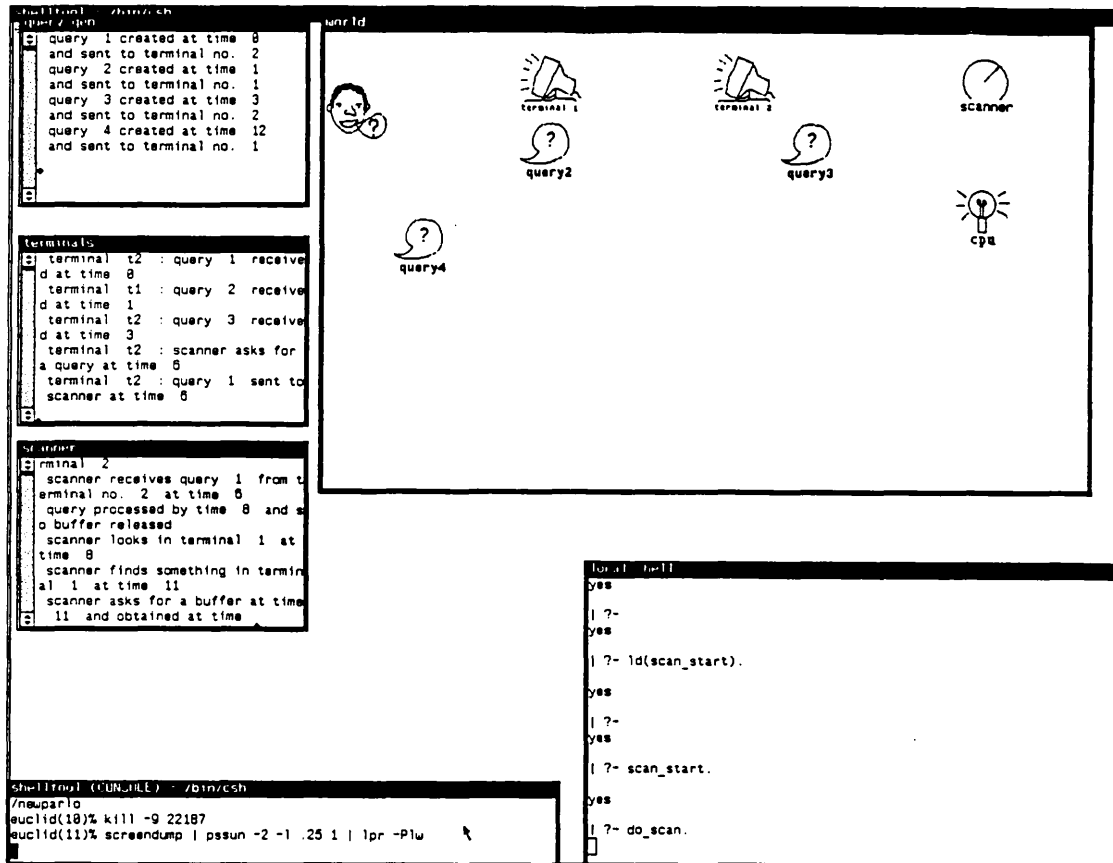


Figure 7.19 : Screen dump of an execution of the information retrieval system.

The icon window shows the situation after 'query1' from terminal no.2 had been processed at time 8. The scanner has now found 'query2' in terminal no.1 at time 11, and has obtained a processing buffer. Shortly after this, the 'query2' icon will move across to the scanner pos-icon and be processed, and 'query4' will be added to terminal no.1.

7.3.5. Summary of the Simulation Example.

A set of Polka classes for building discrete-event simulations were outlined. These allow the various entities and resources in a simulation to execute concurrently and communicate via message passing. In addition, by using distributed simulation techniques, it was not necessary to use a global clock in the simulations. The usefulness of the classes was demonstrated by the ease with which two simulations were programmed. One was a car wash simulation, the other a simple model of an information retrieval system. Both of these examples were much simpler to write and debug by using the predefined simulation classes. This meant there was less new code to produce and the possibility of errors was correspondingly reduced. Some new classes and predicates were required but these were not complex programs. It was also straight forward to create graphical representations of the simulations by specialising predefined graphics classes. This allowed the simulations to be debugged and analysed in a more direct fashion.

The choice of the simulation classes was decided on by looking at DEMOS, a simulation package built on top of SIMULA. It differs from our approach because of its sequential execution, and its use of a global clock.

Chapter 8. Concurrency.

The various forms of concurrency and synchronisation in Polka will be discussed in this chapter, including an explanation of internal and external concurrency, and synchronous, asynchronous, and suspendible messages. The utility of Polka's synchronisation mechanisms will be demonstrated by showing how rendezvous-like constructs, similar to those in Ada, can be coded. There will also be a brief examination of the properties of liveness in Polka, and we will consider how the execution of an object can be decomposed into three parts, namely message arrival, message selection, and completion of message processing. This will motivate a description of a technique which effectively separates the synchronisation issues of a class from its functionality. Finally, there will be a discussion of the actor notation, which utilises a model of concurrency very like that found in Polka.

8.1. What is Concurrency?

Normally, concurrency is defined as the simultaneous execution of sequential programs [Ben-Ari 1982], but in Polka its meaning is in terms of two components. One part is called *external* concurrency because the concurrency is at the inter-object level, there being no assumptions made about the internal execution of the objects. The assumption of sequential execution in many object oriented languages, means that when an object receives a sequence of messages, they will be processed in their arrival order. However, in Polka, and in a few other languages such as actors [Agha 1986], concurrency can be much more fine grained. An object can process many messages at once, and the actions caused by a message may execute simultaneously. This is called *internal* concurrency since it occurs within an object, and has a number of benefits and problems, which will be examined below.

Another property of the concurrency in Polka is that it is explicit, in the sense that parallel operators must be used, if concurrency is required. This allows programs to be written which are closer to the abstract model of the problem, since the parts which are meant to be concurrent can be written in that form. This approach can be contrasted with LP languages which exhibit implicit concurrency, such as Andorra [Yang 1988], where the concurrency is extracted from the program by the compiler.

8.2. Synchronisation.

Polka utilises dataflow synchronisation by means of message passing, but in fact this view of communication is imposed on a much more powerful underlying mechanism - the logical variable. Objects communicate by sharing a logical variable which is treated as a message stream, but there is nothing to stop the variable being treated as another form of communication structure, such as a channel [Tribble et al. 1987], which allows a partial ordering on messages sent between objects, instead of the total ordering found in a stream. Even when the logical variable is treated as a stream it can be manipulated in non-standard ways, as happens in *message peeking*, which enables an object to search along its input stream to find a message rather than just accept the message at the head of the stream. An example of this is given in program 8.1.

```

database
invisible   Store state
clauses
    delete_all =>   undo( Ins, #Ins ) : true      % peek along stream for 'undo's
                    else
                    Store becomes [] .           % else delete store
:
code
    mode undo( ?, ^ ).
    undo( Stream, Stream ) <-                    % current end of stream reached
        var( Stream ) : fail .
    undo( [ undo | Stream ], Stream ) ;           % 'undo' message found
    undo( [ Msg | Stream ], [ Msg | Stream1 ] ) <- % move along stream
        undo( Stream, Stream1 ) .

end.
```

Program 8.1 : A 'database' class which uses message peeking.

The 'undo' call in the guard of the clause means that the 'Ins' stream is searched for an 'undo' message, which if found causes the 'delete_all' message to be ignored. The effect of this search is time dependent since it may be completed before an 'undo' message arrives. Inside the 'undo' predicate, the stream is treated as a partially instantiated list, which is why the first clause causes failure when it encounters a variable, because the current end of the stream has been reached.

There are three main types of message communication in Polka : synchronous, asynchronous and suspendible message passing, although it is possible to use ostream variables in other ways. A synchronous message send is one where the sender waits for a reply from the receiver of the message, as demonstrated in program 8.2.

```

screen_display ToClock ostream
invisible Screen state
clauses
  refresh => ToClock:: get_time( Time ) ,      % get the time
              draw_clock( Time ) ,             % draw the clock
              draw_screen( Screen ) .           % draw the screen
:
end.

```

Program 8.2 : A 'screen_display' class.

The notion of waiting is not as strict as in most languages, because of internal concurrency. In the example, the 'draw_clock' goal waits for 'Time' to become bound but this does not stop 'draw_screen' from drawing the screen or the 'screen_display' object from accepting more messages. One way to control these actions would be to write the clause as :

```

refresh  =>    ToClock:: get_time( Time ) ,
              draw_clock( Time ) &
              draw_screen( Screen ) &.

```

There is no point placing a '&' after the send operation since its sequential or parallel execution is unimportant.

An asynchronous message send is simpler to understand than a synchronous one, since no part of the sender waits for a reply to the message.

A suspendible message is one which is guaranteed not to be sent to the receiver until the message's suspendible variable (or variables) becomes bound, as shown in program 8.3.

```

taxi ToScheduler ostream
invisible Status state
clauses
  pick_up( Customer, Ok ) =>
    ToScheduler:: pick_up( Ok ) ,      % tell scheduler about pick up
    self on Ok:: set_status( busy ) &.  % send self message

  set_status( St )      =>  Status becomes St .
:
end.

```

Program 8.3 : A 'taxi' class.

In this example, 'taxi' sends a 'pick_up' message to the scheduler and also sends a suspendible 'set_status' message to itself. The use of the suspendible form means that the 'set_status' message will not arrive on the input stream of 'taxi' until 'Ok' is bound by the scheduler.

The first clause can be rewritten as :

```

pick_up( Customer, Ok ) => ToScheduler :: pick_up( Ok ) ,
                           data( Ok ) &
                           self :: set_status( busy ) &.

```

The suspendible message send has been replaced by an extra predicate call which tests the 'Ok' variable and suspends until it is bound. Also, because of the use of the '&', the self operation is not carried out until 'Ok' is bound, whereas the suspendible self operation executes immediately but the message is held by the sender until the binding occurs.

Another property of this new clause is that it stops the object from accepting any more input. This may lead to deadlock if the 'scheduler' object can only bind 'Ok' by sending a message to 'taxi'. However, potential deadlock is avoided in the original clause because the delay of the 'set_status' message is independent of the clause which will always succeed.

The main use of suspendible messages is in making data dependencies explicit between messages. Normally, messages are constrained by implicit data constraints within objects, but this means the user knowing far more about how an object processes a message than is necessary or wise. For instance, in systems which use a great deal of inheritance, it will generally be very difficult to know what implicit constraints have changed when objects are changed. Therefore, by making dependencies explicit, many programs become simpler to understand, such as the familiar 'window' example :

```

window    ..
clauses
  redraw( X, Y, Ok3 ) =>
    self :: erase( Ok1 ) ,           % erase a window
    self on Ok1 :: move_by( X, Y, Ok2 ) , % move the window
    self on Ok2 :: redraw( Ok3 ) .   % redraw it
  :
end.

```

Program 8.4 : A 'window' class.

It is quite likely that the operations of 'erase', 'move_by' and 'redraw' are constrained by shared data, but by using suspendible messages the programmer need not concern himself with what the data constraints are, or worry that they may change if any of the 'erase', 'move_by' or 'redraw' actions are redefined.

8.3. Rendezvous-like Synchronisation.

The rendezvous mechanism found in Ada [Young 1983] is one of the most powerful and flexible synchronisation constructs presently available. To show the utility of Polka, we shall discuss how rendezvous-like operations can be encoded, and how they can be generalised to offer more functionality than the Ada equivalent.

The rendezvous has evolved from more primitive synchronisation tools such as spin-locks, semaphores, monitors and message passing [Bishop 1986], and can be thought of as a form of synchronous remote procedure call. The difference lies in the fact that the called part is not a procedure as such, but a sequence of statements within a procedure, introduced by an `ACCEPT` statement. An example of the rendezvous, using a simplified form of syntax from [Bishop 1986], is given in program 8.5. It is a version of the producer/consumer problem, where strings are transferred between the two by using a single buffer. The three entities (the producer, consumer, and buffer) are represented by Ada tasks, which have separate threads of control.

```

TASK buffercontroller ;
var b : string ;
begin
  loop
    ACCEPT deposit( a : string ) do
      b := a ;
    ACCEPT fetch( var a : string ) do
      a := b ;
    end loop ;
  end ; { buffer controller }

  % accept a string deposit
  % store the string in 'b'
  % accept a string fetch
  % serve the request using 'b'

TASK producer ;
var s : string ;
begin
  loop ..
    create a value for s ;
    buffercontroller.deposit( s ) ;
  end loop ;
end ; { producer }

% deposit a string in the buffer

TASK consumer ;
var t : string ;
begin
  loop
    buffercontroller.fetch( t ) ;
    do something with t ;
  end loop ;
end ; { consumer }

% fetch a string from the buffer

```

Program 8.5 : The producer/consumer problem in (pidgin) Ada.

When the buffer controller is first started, it encounters the `ACCEPT` 'deposit' statement. At this point, it must wait until the producer calls 'deposit' to initiate a rendezvous. When this occurs, the actual parameter supplied by the call is copied into the corresponding formal parameter of the `ACCEPT` statement, and then the body of the `ACCEPT` statement is executed. This has the effect of copying the input character into the 'b' variable. On completion of the `ACCEPT` statement, both the producer and the buffer controller proceed again independently. In this case, the latter immediately encounters the `ACCEPT` 'fetch' statement. It therefore waits until the consumer calls 'fetch' to initiate another rendezvous. This time the value of 'b' is copied to the consumer in the body of the `ACCEPT`. Finally, the rendezvous is broken, and both tasks proceed again independently.

It is important to realise that two events are necessary to initiate a rendezvous. Firstly, the caller must call the relevant procedure, and secondly, the server must reach a corresponding `ACCEPT`. It does not, however, matter which event occurs first. If the call occurs first, then the caller is made to wait until the server reaches the `ACCEPT`. If, on the other hand, the server reaches the `ACCEPT` first, then it is made to wait until the server calls the procedure.

As seen in the example, the sequential ordering of `ACCEPTS` means that synchronisation activities can be ordered. Also, since `ACCEPT` statements can be nested, multiple rendezvous are possible. It is also possible to have several `ACCEPT` statements with the same name in a single task, which allows a request to be handled in different ways at different times.

In addition to the `ACCEPT` statement, Ada has a `SELECT` statement, which is a variation of Dijkstra guarded command [Dijkstra 1975] and can be used to choose between several `ACCEPTS` nondeterministically. A `SELECT` statement can have an `ELSE` part, which is executed if none of the `SELECT` choices can be executed. A `TERMINATION` alternative is also possible, and `DELAY` operations can be used for timeouts.

Although most of the flexibility of synchronisation is in the receiver, the sender can use a `SELECT` statement and timeouts to specify how long to attempt a rendezvous, and what to do if time runs out. The sender can also choose which receiver to rendezvous with, unlike the receiver which can only accept calls and does not know the sender.

The underlying model for `ACCEPT`'s in a task is that each has a queue upon which potential rendezvous calls are stored. When the flow of control reaches an `ACCEPT`, the first call is removed from the `ACCEPT`'s queue, or if the queue is empty the `ACCEPT` suspends. There is no direct way to access the queue apart from a

COUNT test which counts the queue elements. This operation must be used with care since new calls may be added or removed during its evaluation, which will alter the queue length and so invalidate any decision based on a COUNT value.

The basic idea behind the Polka 'accept' predicates described below, is that an Ada task with a queue for each of its ACCEPTs can be viewed as a Polka class with a single input stream. The Polka 'accept' predicates selectively search this single stream in order to mimic the action of an Ada ACCEPT. This enables variations of the basic ACCEPT operation to be created, because the input stream is visible and can be manipulated as a first order citizen of the language.

For simplicity, the following examples will only contain classes which use the default input stream 'Ins', but all the techniques discussed are scalable to deal with several inputs.

As previously mentioned, a stream can be viewed as a partially instantiated list, which is normally terminated by a variable. Only upon termination will its end be bound to []. Assuming that message streams have this structure allows predicates to be written which can manipulate them. For instance, a 'count' predicate might be :

```
mode count( ?, ?, ^ ).
count( Stream, No, No ) <-
    var( Stream ) : true .           % current end of stream reached
count( [], No, No ) .               % actual end of stream reached
count( [ Msg | Stream ], No, Total ) <-
    No1 is No + 1 ,                 % increment message count
    count( Stream, No1, Total ) .
```

Program 8.6 : A 'count' predicate.

which can be called like so :

```
<- count( Ins, 0, Total ) .
```

'Ins' is the input stream, and its current length will be returned in 'Total'. The first clause uses the metalogical 'var' predicate to test for the variable at the end of the stream, while the second clause deals with the case when the stream has been terminated.

As mentioned above, each accept-like call in an object selectively searches a single input stream instead of accessing a queue. This leads to the following definition for 'wait_accept' in program 8.7.

```
mode wait_accept( ?, ?, ^ ).
wait_accept( [ Msg | Stream ], Msg, Stream ) ;           % remove 'Msg'
wait_accept( [ M | Stream ], Msg, [ M | Stream1 ] ) <- % keep looking for 'Msg'
    wait_accept( Stream, Msg, Stream1 ) .
```

Program 8.7 : A 'wait_accept' predicate.

A typical call is :

```
<- wait_accept( Ins, deposit( A ), Ins1 ).           % find a 'deposit' message
```

'Ins' is the input stream, the second argument is the message being looked for, and the third argument is the resulting stream after the message has been extracted from it.

The predicate recurses down the input stream and attempts to unify 'Msg' with each message on the stream. The first successful unification causes the predicate to succeed, but it will suspend if no suitable message can be found, or fail if the stream has been terminated. If 'wait_accept' is suspended and a new message arrives then it will wake up and test it.

The producer/consumer program given earlier could now be coded in Polka as program 8.8.

```
buffer_controller
clauses
  null =>
    wait_accept( Ins, deposit( A ), Ins1 ),           % look for a 'deposit' message
    wait_accept( Ins1, fetch( A ), #Ins ).           % then look for a 'fetch'
end.

producer Out ostream
clauses
  null =>  create a value for S ,
           Out:: deposit( S ) .                      % send a 'deposit' message
end.

consumer Out ostream
clauses
  null =>  Out:: fetch( T ) ,                          % send a 'fetch' message
           do something with T .
end.
```

Program 8.8 : The producer/consumer problem in Polka.

The objects would also have to be connected :

```
<-  producer( PIns, POut, _ ) ,                      % error streams are ignored
     consumer( CIns, COut, _ ) ,
     merge( POut, COut, Ins ) ,
     buffer_controller( Ins, _ ).
```

The use of **null** allows the standard message passing input mechanism to be ignored, although input stream arguments still need to appear in the invocation of the classes. In 'buffer_controller' the ordering of the two ACCEPTS is implicit by the sharing of the 'Ins1' stream.

'wait_accept' and other predicates defined later, are not quite the same as the Ada ACCEPT because of the nature of messages in Polka. A message is only synchronous if the sender uses a variable from the message, which the receiver binds. However, in Ada every rendezvous is synchronous and this is related to the length of the body of the ACCEPT statement. Thus, the Ada version of the program causes the producer, consumer, and buffer controller to suspend at various times, whereas the Polka version does not. To achieve the effect of synchronisation for the duration of an accept in Polka, each message should include a variable which the receiver will only bind at the end of its accept. This variable should be used by the sender to suspend its execution until it is given a value.

In Ada, the use of ACCEPTs in a SELECT statement corresponds to the use of or- parallel clauses in Polka classes and calls to 'wait_accept's in the guards. This can be seen in the class in program 8.9.

```

buffer_controller2
invisible String state <= ''
clauses
  null => wait_accept( Ins, deposit( A ), #Ins ) :      % look for a 'deposit'
              String becomes A .                      % store the string

  null => wait_accept( Ins, fetch( A ), #Ins ) :        % look for a 'fetch'
              A = String .                             % assign it the string
end.
```

Program 8.9 : An or- parallel buffer controller.

This allows either a 'deposit' or a 'fetch' at any time, but this functionality can be constrained by further predicates in the guards :

```

  null => wait_accept( Ins, fetch( A ), #Ins ),        % look for a 'fetch'
              upper_case( String ) :                  % and the string is upper case
              A = String .
```

This change to the second clause means that a 'fetch' will only be processed if the string is in upper case characters.

The use of 'wait_accept' is only appropriate if the clause it is in should suspend when no suitable message is on the input stream. A useful variation of this is 'fail_accept', which fails if a message is not present .

```

mode fail_accept( ?, ?, ^ ).
fail_accept( Stream, Msg, Stream1 ) <-
    var( Stream ) : fail .                                % fail at the current end of stream
fail_accept( [ Msg | Stream ], Msg, Stream ) ;            % find 'Msg'
fail_accept( [ M | Stream ], Msg, [ M | Stream1 ] ) <-    % keep looking for 'Msg'
    fail_accept( Stream, Msg, Stream1 ).

```

Program 8.10 : A 'fail_accept' predicate.

'fail_accept' can be used to encode the equivalent of Ada's ELSE statement, as demonstrated by program 8.11.

```

buffer controller3
invisible String state <= ''
clauses
    null => fail_accept( Ins, deposit( A ), #Ins ) :      % find a 'deposit' or fail
        String becomes A .

    null => fail_accept( Ins, fetch( A ), #Ins ) :          % find a 'fetch' or fail
        A = String ;

    null => sleep( 1000 ) &.                                % suspend for 1 sec
end.

```

Program 8.11 : A buffer controller which can sleep.

This object looks for 'deposit' and 'fetch' messages on its input stream and if none are there then the third clause is executed. This calls 'sleep', which will suspend the object for one second (1000 ms) before starting again.

Another powerful feature in Ada is the ability to retract a call if it has not been accepted within a certain time. This can be coded up in Polka by including two extra arguments with each message. The first will be known as the 'Accept' variable, and is bound when the message has been accepted. The other will be called the 'Dead' variable, and is bound by the sender when he wants the message to be removed from the receiver's input stream. A message will now be sent in the form :

```
msg( Message, Accept, Dead )
```

A class might transmit such a message like so :

```

producer Out ostream
clauses
    null => create a value for S ,
        Out: msg( deposit( S ), Accept, Dead ) ,      % send a 'deposit'
        time_out( 1000, Accept, Dead ) .              % timeout for 1 sec
end.

```

Program 8.12 : A producer of 'msg' messages.

The clause sends a 'deposit' message out on 'Out' and then timeouts for one second before killing the message if nothing has happened.

'time_out' is defined in program 8.13.

```

mode time_out( ?, ^ ).
time_out( Time, Accept, Dead ) <-
    sleep( Time ) &                % suspend for 'Time' ms
    check_time( Accept, Dead ) .    % check the message arguments

mode check_time( ?, ^ ).
check_time( Accept, dead ) <-      % kill the message since not accepted
    var( Accept ) : true ;
check_time( accepted, D ) .        % the message has been accepted

```

Program 8.13 : Predicates for 'time_out'.

'time_out' sleeps for the required time and then checks 'Accept'. If it is unbound then the message is killed by binding its 'Dead' variable, otherwise the message has been accepted. This new protocol means that the previously described 'wait_accept' and 'fail_accept' predicates must be altered. For instance, 'wait_accept' becomes :

```

mode wait_accept( ?, ^ ).
wait_accept( [ msg( Msg, Accept, Dead ) | Stream ], Msg, Stream ) <-
    var( Dead ) :                % accept 'Msg' if not dead
    Accept = accepted ;

wait_accept( [ msg( M, Accept, Dead ) | Stream ], Msg,                % keep looking for 'Msg'
    [ msg( M, Accept, Dead ) | Stream1 ] ) <-
    var( Dead ) :
    wait_accept( Stream, Msg, Stream1 ) ;

wait_accept( [ msg( M, Accept, dead ) | Stream ], Msg, Stream1 ) <- % delete a dead message
    wait_accept( Stream, Msg, Stream1 ) .

```

Program 8.14 : A 'wait_accept' predicate which deals with 'msg' messages.

A 'wait_accept' call would be just as before :

```

<- wait_accept( Ins, deposit( A ), Ins1 ) .    % look for a 'deposit'

```

This predicate hides the fact that messages are received inside 'msg' terms, and it also deletes any messages which are 'dead'. The first clause deals with the case when a suitable message has been found and it is not dead. In that case, the message's 'Accept' variable is bound and this value is communicated back to the sender of the message. However, it is possible that between the 'var' test and the arrival of the 'Accept' binding at the sender, the 'Dead' variable has itself been bound. This means that the message just accepted should really have been ignored. To deal with this

requires a more complex protocol since 'wait_accept' must wait for its binding of 'Accept' to be confirmed by the sender. Fortunately, this extra complexity can be hidden by redefining the 'check_time' predicate used by the producer, and also by extending 'wait_accept'.

The new definition for 'check_time' is in program 8.15.

```
mode check_time( ?, ^ ).
check_time( Accept, dead ) <-          % kill the message since not accepted
    var( Accept ) : true ;
check_time( accepted( Ok ), Dead ) <- % the message has been accepted
    Ok = ok .                          % and so signal confirmation
```

Program 8.15 : A 'check_time' predicate which signals acceptance confirmation.

'check_time' now receives an 'accepted' term containing a variable which it must bind in order to send a confirmation back to the receiver.

The first clause of 'wait_accept' becomes :

```
wait_accept( [ msg( M, Accept, Dead ) | Stream ], Msg, Stream1 ) <- % accept 'Msg' if
    copy( Msg, CMsg ) &
    M = CMsg , var( Dead ) :          % 'M' unifies with 'Msg' copy
    Accept = accepted( Ok ) ,         % and 'M' is not dead
    confirmed( Stream, Stream1, Dead, Ok, CMsg, Msg ) ; % and sender confirms acceptance
```

'confirmed' is given as program 8.16.

```
mode confirmed( ?, ^, ?, ?, ?, ? ).
confirmed( Stream, Stream1, dead, Ok, CMsg, Msg ) <- % acceptance rejected
    wait_accept( Stream, Msg, Stream1 ) .
confirmed( Stream, Stream, Dead, ok, CMsg, Msg ) <- % acceptance confirmed
    CMsg = Msg .
```

Program 8.16 : A 'confirmed' predicate.

Notice that 'wait_accept' now unifies a copy of 'Msg' with the message in the 'msg' term. This allows the result to be discarded if the message is subsequently killed.

In essence, 'check_time' and 'wait_accept' define a three stage communication protocol. The sender sends messages to the receiver, which the receiver searches through. If it finds a suitable message it queries the sender to see if that message is still 'live'. The sender replies and the receiver then uses the original message or ignores it, depending on the answer. Diagrammatically, this is shown in figure 8.1.

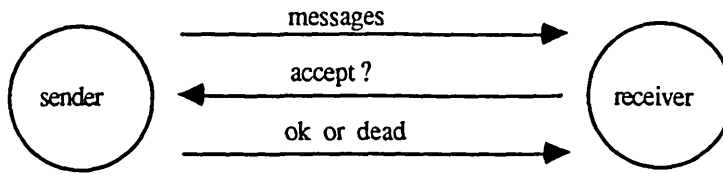


Figure 8.1 : Three stage communication protocol.

The circles represent the objects and the arrows signify the direction of communication, although the diagram is a little inaccurate since a 'dead' message may be sent to the receiver at any time.

The techniques outlined above show how the rendezvous mechanisms of Ada can be mimicked. In fact, many other communication protocols are possible using variations of the methods used here. One example is a predicate which returns the first message from a stream which passes an arbitrary test. It is called 'wait_test' and is defined by program 8.17.

```

mode wait_test( ?, ?, ^, ^ ).
wait_test( [ Msg | Stream ], Test, M, Stream ) <-
  list_to_struct( Test, [ Msg ], Pred ),           % construct and execute test
  call( Pred ) : M = Msg ;
wait_test( [ M | Stream ], Test, Msg, [ M | Stream1 ] ) <- % look for another message
  wait_test( Stream, Test, Msg, Stream1 ) .
  
```

Program 8.17 : A 'wait_test' predicate.

'wait_test' assumes that the test predicate can be called in the form :

Test(Msg)

which is done by the guard in the first clause. A typical 'wait_test' call might be :

```
<- wait_test( Ins, access_type, Msg, Ins1 ) .
```

'Ins' is the input stream, 'access_type' the name of the test predicate, 'Msg' the returned message, and 'Ins1' the resulting input stream. The query looks for a message which passes the 'access_type' test. 'access_type' might be defined as :

```

mode access_type( ? ) .
access_type( fetch( _ ) ) .
access_type( deposit( _ ) ) .
  
```

Program 8.18 : An 'access_type' predicate.

Predicates which manipulate the input stream need not extract messages. For instance, the 'delete' predicate searches the input stream and deletes messages which pass a given test :

```

mode delete( ?, ?, ^ ) .
delete( Stream, Test, Stream ) <-
    var( Stream ) : true .                                % current end of stream reached
delete( [], Test, [] ) .                                  % actual end of stream reached
delete( [ Msg | Stream ], Test, Stream1 ) <-
    list_to_struct( Test, [ Msg ], Pred ) ,                % carry out test on 'Msg'
    call( Pred ) :
    delete( Stream, Test, Stream1 ) ;
delete( [ Msg | Stream ], Test, [ Msg | Stream1 ] ) <-    % look for another message
    delete( Stream, Test, Stream1 ).

```

Program 8.19 : A 'delete' predicate.

A typical call might be :

```
<- delete( Ins, access_type, Ins1 ) .
```

This query will delete all the messages which pass the 'access_type' test.

Many primitives like these have to be built into message passing languages, such as PLITS [Feldman 1979].

This section has shown how high level synchronisation mechanisms can be coded in Polka and in particular how the Ada rendezvous can be simulated. This is mostly due to Parlog and also to the choice of abstract model for the Polka language, which means that logical variables are the same as message streams. Although Ada hides more of the underlying mechanism of the rendezvous, Polka gains by being able to manipulate message streams as first order citizens of the language. This allows many types of protocol to be simulated.

8.4. Liveness.

Liveness deal with the dynamic properties of a concurrent program and in its most general form states that if something is supposed to happen then eventually it will happen [Ben-Ari 1982]. Example of this include the assumption that a message will arrive at an object, and that a test on data will eventually terminate. The key notion in liveness relates to the word 'eventually' which is usually taken to mean 'within an unspecified but finite length of time'. This approach to fairness is sometimes called weak fairness and is related to two other forms called antifairness and strong fairness [Filman and Friedman 1984]. An antifair system is one that makes no commitment about the level of any entity's service. In a strongly fair system,

entities with equivalent priority should be served in the order of their requests. Some well known antifair systems are petri nets, CSP, Argus and Hearsay II. Weakly fair systems are the Act languages, and strongly fair ones include Ada, PLITS and SR [Filman and Friedman 1984].

In concurrent LP, the two types of fairness discussed are called And- and Or- fairness, and correspond to the two forms of nondeterminism present in a concurrent logic program [Shapiro 1989]. And- fairness implies that a given goal will eventually be reduced. Or- fairness implies that a given clause will eventually be chosen if it is repeatedly a candidate clause in the selectable set. Because of the close parallels between Polka and concurrent LP, And- fairness in Polka can be taken to mean the eventual execution of an object within a system of objects, and Or- fairness can be seen as the eventual execution of a candidate clause in the selectable set. In fact, Parlog (and so Polka) is only And- fair, but Or- fairness can be programmed.

Other important fairness issues include deadlock, livelock, starvation and divergence.

Deadlock can occur when objects are attempting to obtain common resources. Each object may obtain some part of the resources but deadlock because the others have the rest and no object is prepared to release its share. The well known example of deadlock is the dining philosophers problem [Dijkstra 1972], which was discussed in Parlog terms in [Ringwood 1988]. Generally, deadlock can be detected by invoking an object, or system of objects, inside a Parlog metacall and observing if a deadlock status message is issued. Unfortunately, most deadlock avoidance algorithms require special knowledge of the problem, as seen in the dining philosophers problem, where a 'waiter' only allows a certain number of philosophers to be sitting at the table at one time.

Livelock occurs when an object becomes locked into a sequence of actions which stops it from doing any useful work. An example of this is the continual polling for a free disk by a CPU when there is none available. Livelock also occurs between objects when they keep repeating the same communication sequence, as may happen during deadlock recovery if two objects keep passing their resources to each other.

Starvation, which is also known as lock out, happens to an object when it is indefinitely delayed from doing something because of the actions of other objects. This is a less serious problem because at least some part of the system may be doing some useful work, but it is also hard to detect and correct.

Divergence is actually a key property of most object oriented systems such as operating systems and simulations. However, even in operating systems, it is

sometimes necessary to interrupt or stop execution, and this is impossible if divergence has occurred. Divergence can be subdivided into two categories : divergence due to a message (such as an interrupt) not being accepted, and the case when an object is looping internally and so can not get round to processing a message.

8.5. Arrival, Selection and Completion.

Liveness issues can best be studied by looking at the stages of the execution of an object. In Polka, three stages can be discerned : the arrival of a message, the selection of a message, and the completion of the processing caused by the message. Each of these stages has its own problems, which shall be looked at in turn.

As with many message passing systems, Polka assumes that a message once sent will arrive, which relies on the Or- fairness of predicates such as 'merge'. Also, inter-object communication assumes that messages arrive in the order they are sent, albeit interleaved with messages sent from other objects. Both of these assumptions can be weakened, by allowing communication mechanisms to be programmed which deal with the loss of messages or with arbitrary arrival order. A sender can decide if a message is lost by using timeouts and 'Accept' flags in each message, as described in section 8.3. Arbitrary message arrival order can be programmed by utilising predicate which manipulate streams.

A message is selected when it is extracted from the receiver's message stream. Normally this is done in a First-Come-First-Served order and so guarantees fairness, but if a predicate like 'wait_accept' is used as a guard then the object may suspend forever waiting for a certain message to arrive.

In sequential languages, the order in which messages complete their processing is always the same as the selection order of messages, since only one message can be processed at a time, However, internal concurrency permits the processing of several messages to be carried out at once. This makes it quite likely that the processing completion order for a set of messages will be different from their selection order.

These ordering are important because of their effect on the meaning of messages sent to an object. For instance, most users of object oriented databases would think that sending a message to add data to a database, followed by a second message to remove that data would be perfectly safe in a single user environment. This is not the case if the ordering of these two messages can change.

The problem is compounded when the processing of a message can be subdivided into messages to other objects, to self, or to inherited objects. All of these may lead to processing outside of the action part of the receiver object.

Many languages avoid such problems by making sure that messages sent to an object always stay in the same order in relation to one another. This has a number of serious consequences, including the removal of all internal concurrency to ensure that the selection order for messages is the same as their processing completion order. This seems too drastic since it means that even harmless parallelism is ruled out.

The answer is not to force messages to follow some implicit ordering, but rather to explicitly state the required orderings between messages. This is achieved by adding an extra variable to each input message and all the messages transmitted during its processing. These act as reply variables to inform the object that they have finished being processed. When processing is completed, the variable in the original message is bound to indicate that its action has been completed. Suspendible messages can use these reply variables to make explicit the intended processing order of messages. This can be seen in the familiar :

erase → move_by → draw

sequence in the 'redraw' action of the 'window' class (program 8.4) in section 8.2.

8.6. Separation of Synchronisation from Functionality.

The main drawback of the approach described above is that it increases the complexity of messages. At the very least, extra arguments must be included inside messages which will signal the completion of some important event. Extra code is also required if messages need to be constrained and it is likely that such constraints will be complex in many cases. There are no absolute answers to these problems but the technique proposed below does help to simplify the user interface to an application.

A user often wants to ensure that one message has been processed to completion before another is sent to the object. This can be thought of as a synchronisation problem, which is separate from the functionality offered by the application. To mirror this, the user's synchronisation constraints will be encoded inside a *synchronisation class* which acts as an interface to the application class. A synchronisation class is like any ordinary class and is only different because of the way it is used. This approach means that the user can send messages without

constraints to a synchronisation object, and also be unaware of the separation of synchronisation and functionality.

8.6.1. The Readers-Writers Problem.

The readers-writers problem will be used to illustrate this coding technique since it is a typical synchronisation problem [Courtois et al. 1971]. In its simplest form it specifies that many readers are allowed to access a resource at once, but only one writer may change it at a time, and while a writer is manipulating the resource, no readers are allowed. In addition to this safety property, a scheduling property can be included which states that all readers and writers must eventually access the resource.

We shall assume the presence of a 'resource' class which can accept two sorts of message :

```

                                read( R, Ok )
and                             write( W, Ok )

```

The 'R' variable in the 'read' message will contain a partially instantiated term which will be used to find a value in the 'resource' object . When this value is unified with 'R', bindings will be returned to the sender and at the same time 'Ok' will be bound. The 'write' message causes the contents of 'W' to be added to the resource in some fashion and when this is done 'Ok' is bound.

The synchronisation class for 'resource' will be called 'resource_syn' and will accept messages of the form :

```

                                read ( R )
and                             write( W )

```

One of 'resource_syn' jobs will be to add 'Ok' arguments to the messages it receives, so relieving the user of that task. However, its main job will be to enforce the synchronisation and scheduling criteria for 'resource'. The definition of 'resource_syn' is in program 8.20.

```

resource_syn      % fair scheduling
invisible ToResource ostream,
               Readers state <= [], Writer state <= ok

initial   resource( ToResource ) .                                % invoke a resource object

clauses
  read( R )  => ToResource on Writer :: read( R, Ok ) ,           % send a 'read'
               Readers becomes [ Ok | Readers ] .                 % store its 'Ok' variable

```

```

write( W ) => all_done( Readers ) &                % wait for readers to finish
               ToResource on Writer :: write( W, Ok ), % send a 'write'
               Writer, Readers becomes Ok, [] .      % store 'Ok' and reset 'Readers'

code
  mode all_done( ? ) .
  all_done( [] ) .
  all_done( [ Rd | Rds ] ) <-
    data( Rd ) &                                % reader has finished
    all_done( Rds ) .
end.

```

Program 8.20 : A 'resource_syn' class.

The actions that 'resource_syn' carries out can more clearly be understood by considering the dependencies for each 'read' and 'write' message. A 'read' must wait until the earlier 'write' has finished, while a 'write' must wait until the earlier 'write' and all the earlier 'read's have finished. In fact, a new 'write' will only depend on one or other of these, but we test for both to simplify matters. These dependencies are encoded as the state variables 'Readers' and 'Writer' : 'Readers' contains a list of 'Ok' variables for every 'read' sent to 'resource', while 'Writer' contains the 'Ok' variable for the last 'write'.

When a 'read' message arrives at 'resource_syn', it is sent to 'resource' with an 'Ok' variable, but as a suspendible message which will wait until 'Writer' becomes bound. In addition, the new 'Ok' variable is added to the 'Readers' list. The 'ToResource' output stream is linked to an invocation of 'resource' which is created when 'resource_syn' is started. This means that the user can not access this resource directly but must go through 'resource_syn'.

The clause for a 'write' message first checks that all the readers have finished by calling 'all_done'. When this is true it sends the 'write' to 'resource', but as a suspendible message tied to the 'Ok' value for the last 'write'. Then 'Writer' is updated so that it contains the 'Ok' value for the current 'write', and 'Readers' is set to [] since no readers are waiting.

A call to 'resource_syn' might be :

```

<- resource_syn ( Ins, Error ),
   Ins = [ write( el( 1, a )), read( el( 1, Val )), write( el( 2, b )) | _ ] .

```

which binds :

```

Val = a

```

This shows how the synchronisation details can be hidden from the user and also separated from the functionality of the problem.

8.6.2. Readers Priority.

A more interesting variation on the basic readers-writers problem assigns a priority to one of the operations. The example we shall consider is the readers priority problem, which gives priority to readers over writers. This will not require any changes to 'resource', but 'resource_syn' will call a new predicate called 'any_reads' in its second clause :

```
write( W ) =>
    any_reads( Ins, #Ins, Writer, Readers, Readers1 ),      % check for readers on 'Ins'
    all_done( Readers1 ) &
    ToResource on Writer :: write( W, Ok ),
    Writer, Readers becomes Ok, [] .
```

The definition of 'any_reads' is given as program 8.21.

```
mode any_reads( ?, ^, ?, ^, ^ ) .
any_reads( Str, Str, Writer, Rs, Rs ) <-                    % current end of stream reached
    var( Str ) : true .
any_reads( [ read( R ) | Str ], Str1, Writer, Rs, NRs ) <-
    ToResource on Writer :: read( R, Ok ),                  % 'read' message sent
    any_reads( Str, Str1, Writer, [ Ok | Rs ], NRs ) ;
any_reads( [ Msg | Str ], [ Msg | Str1 ], Writer, Rs, NRs ) <- % ignore other messages
    any_reads( Str, Str1, Writer, Rs, NRs ) .
```

Program 8.21 : An 'any_reads' predicate.

The 'read' dependencies are as before, which is why the first clause of 'resource_syn' does not need to be changed. However, a 'write' now depends not only on the 'read's and the 'write' ahead of it on the way to 'resource', but also on any other 'read's which are behind it on the input stream into 'resource_syn'. This first criteria is encoded as before, but the second requires the new 'any_reads' predicate. Its action is to recurse down the input stream of 'resource_syn' sending any 'read's to 'resource' before the 'write'. These new 'read's have 'Ok' variables so these are added to the 'Readers' list to create 'Readers1'.

A typical call to 'resource_syn' is :

```
<- resource_syn( Ins, Error )
```

Three messages are sent along 'Ins' :

```
write( el( 1, a ) ), write( el( 1, b ) ), and read( el( 1, Val ) )
```

What 'Val' is bound to is time dependent, and the 'read' may pass neither of the 'write's, only the second or perhaps both. Thus its value may be :

b, a or read_error

The 'read_error' value is generated by 'resource' when a 'read' message arrives when the resource is empty.

It is interesting to note that a writer priority version of the readers-writers problem requires similar sorts of changes to the 'read' message clause instead of the 'write' message clause.

8.7. The Actor Model.

The actor model was designed by Hewitt [Hewitt 1977] and several languages have been proposed which use it and extend it in various ways, including Act-1 [Theriault 1982], Act2 [Theriault 1983], Atolia [Clinger 1981], Act3 [Hewitt et al. 1985], and Acore [Manning 1986]. The model is not object-oriented in nature, although it can be employed in an object-oriented style [Agha 1986]. However, the model deals with many of the same issues as Polka for programming concurrently executing systems which communicate via message passing, and it is for those reasons that the actor model is included here. No one particular actor language will be considered, and only the important language features will be described.

The basic unit in the model is the actor, which is an encapsulated, concurrently executing entity. In a program, there will be potentially many actors communicating with each other, via asynchronous message passing. There are no global data structures, and no global clock.

A message is sent to the mail address of an actor, which is used in a very similar way to message stream variables in Polka. However, the model makes no assumptions about the arrival order of messages at an actor, which is non-deterministic even if the sending order is known. In Polka, the situation is quite different, since message streams are usually encoded as partially instantiated lists, which means the arrival order of messages is the same as their sending order. Fortunately, message streams need not be represented as lists, and it is possible to encode a stream data structure which randomly orders messages. This ability highlights the fact that streams are visible entities in Polka and can be explicitly manipulated, whereas they are not accessible in the actor model.

Non-determinism can occur in Polka if several input streams are allowed into an object - the Polka version of 'merge' is a typical example, but actors avoid this by only having one input entry point.

Common to both Polka and the actor model, is the idea of buffering messages at the entry to an object/actor, which means a message may have to wait before being processed. These buffers are called mail boxes in the actor model.

Messages can contain mail addresses, so allowing an actor to change the list of actors it can communicate with, and thus alter the configuration of the system. This has a direct correspondence in Polka with the passing of message stream variables inside messages.

The internal structure of an actor is very like a Polka class. Typically, it includes a sequence of different possible actions, one of which will be executed when a message arrives and matches the message pattern associated with the action. In more

recent actor languages, message patterns can be tried in parallel, allowing the possibility of non-deterministically choosing an action if several can be carried out. This or-parallelism is exactly like that found in Polka. If no pattern matches, a default action may be carried out, or the message may be delegated to another actor. Message patterns can be quite sophisticated, but the actor model does not use unification, relying instead on simple pattern matching, which rules out back communication. In Act2, an actor utilises a description language based on Omega [Attardi and Simi 1981], a description and deduction system, itself written using actors. The description language can specify pattern matching tests on incoming messages, the data types of state variables, and the structure of output messages.

A message may consist of three parts - the request, the customer and a complaints department. The customer (also called the continuation) is the mail address where the reply should be sent. A complaints department is the mail address of an error handling actor where a rejected request can be routed. Such built in structures are not present in Polka messages.

There may be two sorts of data in an actor : state variables and mailing addresses, but one of the uniform features of the actor model is that everything is an actor. Thus, state variables, mailing addresses, and even messages are actors and must be communicated with via message passing. This can lead to infinite regress when communicating with messages, and this is prevented by having primitive actors in the model, which can be manipulated and accessed like normal data. Apart from primitive actors for messages, there are ones for literals, numbers and mail addresses.

An important feature of the model are the locality laws which define finiteness properties for each actor system [Hewitt and Baker 1977]. One such law is that of finite acquaintances, which states what an actor can know about other actors in the system. An actor can know the names of the actors it has defined, and also the mail addresses of actors which it had initially or was later sent in messages. In addition, it is possible for an actor to forget names and mail addresses. This is very close to how Polka works, since initially an object only knows the names of its inherited objects, its term classes and its output streams, but may receive further term classes and streams inside messages.

The actions of an actor are made up of at most four different types of operation :

1. Simple tests on the received message
2. The sending of messages to one or more of its mail addresses.
3. The creation of new actors along with unique mail addresses for each one.
4. The invocation of a new behaviour for an actor. This typically means the

actor either alters its internal data, or perhaps becomes a new actor.

These four types of operation have direct analogies in Polka.

1. Simple tests in actors correspond to tests in the guard or the body of a Polka clause.
2. Polka sends messages to objects along streams.
3. Creating a new actor is equivalent to invoking a Polka class (or term class).
4. Changing to a new behaviour can be modelled in several ways. If only state variables are being changed then the **becomes** operation can be used. If the actor is mutating into another, then this can be represented in Polka by using the **i_am** operation.

The similarities between the actor model and Polka can be seen by comparing their representation of the factorial function. Polka will be programmed in an 'actor' style to aid the comparison :

```
factorial
clauses
  f( 0, Cust ) =>  Cust = 1 ,                                % factorial of 0
                  i_am factorial( Ins ) .

  f( N, Cust ) =>  N != 0 :                                    % factorial of N
                  Cust = X * N ,
                  N1 is N - 1 ,
                  i_am factorial( [ f( N1, X ) | Ins ] ) . % calculate factorial of N - 1
end.
```

Program 8.22 : An actor-style 'factorial' class.

A typical call might be :

```
<- factorial( Ins ), Ins = [ f( 6, C1 ), f( 4, C2 ) ] .
```

Giving : C1 = 720 and C2 = 24

For comparison, the same function can be coded in Agha's SAL actor language [Agha 1985] as program 8.23.

```

def factorial ( ) O [ N, Cust ]
become factorial ( ) //           % specify replacement behaviour

if N = 0 then                     % factorial of 0
    send [ 1 ] to Cust fi //

if N <> 0 then                     % factorial of N
    def sum ( ) O [ K ]
        send N * K to Cust       % define actor for X * N
    end def
    new X : sum ( ) ;
    send [ N - 1, X ] to factorial % calculate factorial of N - 1
fi
end def

```

Program 8.23 : A 'factorial' actor.

The amount of parallelism in the actor version and the Polka version are the same.

It is unlikely that factorial would be written in this form in either language. An actor language would supply operations to allow the programmer to write it in a functional notation, which would be compiled into the actor version given above. In Polka, a Parlog predicate would be used :

```

mode factorial( ?, ^ ).
factorial( 0, 1 ).           % factorial of 0
factorial( N, Cust ) <-     % factorial of N
    N =\= 0 :
        Cust is N * K, N1 is N - 1,
        factorial( N1, K ).   % calculate factorial of N - 1

```

Program 8.24 : A 'factorial' predicate.

This highlights the point that the actor model is deliberately low level, and is meant to be used as an implementation language. The clearest indication of this lack of high-level expressability is that an actor can not send a message and then suspend, waiting for a reply. The way such a thing is coded is to send a message which includes a mail address for a customer whose job is to accept and act upon the reply to that message. Thus, during the execution of even a simple function, many new actors are created which are used as places to send the intermediate result before the final answer is produced.

An actor's inability to wait for several inputs causes the computation of expressions to be quite complicated. One feature of newer actor languages is the *future* which allows an expression to be evaluated as an actor. When the value of the expression is needed, a message is sent to retrieve it from the future, but all such messages will be buffered until the expression has been evaluated. In Polka, this can

be coded by using a logical variable, which can be seen in the 'factorial' program where 'N1' will eventually be bound but in the meantime the recursive call to 'factorial' is not affected. It is only when 'N1' is used in a calculation that the possible lack of a value may cause suspension.

In every actor system there will be a number of different types of actor. These include primitive actors, which have already been mentioned, and there are also unserialized and serialized actors [Atkinson and Hewitt 1979].

An unserialized actor does not contain state variables and does not alter its behaviour during execution. These restrictions allow it to process many messages in parallel, and a typical unserialized actor example is 'factorial', which can calculate many factorials concurrently.

Serialized actors, on the other hand, can change their behaviour and may have state variables. Because of this, a message must be processed completely before another can be handled. Serialized actors can be built from unserialized ones by using 'lock' and 'unlock' primitives.

In Polka, unserialized actors would be represented as Parlog predicates, or perhaps as Polka classes with no state variables. Serialized actors would normally correspond to Polka classes.

Another difference between Polka and the actor model is object granularity. An actor is a very small computational entity which may be short lived, and do very little work. Consequentially, many actors will be created and die during a systems execution. Objects in Polka are much larger, consisting of at least one Parlog predicate, but normally several. Typically, an object remains in existence during the entire life of the system, and produces large amounts of computation. These two granularities for objects and actors naturally map onto different kinds of hardware. For actors, the architecture should support very many light weight and short lived processes, while for Polka, the number of processes will be fewer but exist for longer and use more resources. A priority for the actor architecture is the efficient construction, delivery and decoding of messages, of which there will be many. In Polka, there may be less communication using messages, since communication inside an object could be carried out via shared memory on the processor the object is mapped to.

There have been three major approaches taken to giving a semantics to the actor model. Hewitt and Baker [Hewitt and Baker 1977] originally defined its behaviour in terms of axioms. Clinger [Clinger 1981] used powerdomains composed from the partial ordering of message arrivals at actors. Recently, Agha [Agha 1986] has used a Plotkin style state transition model, in a similar way to earlier work by

Baker [Baker 1978]. This approach is also utilised to give a semantics for a subset of Polka in chapter 10.

The close correspondence between elements of concurrent LP and the actor model has led to work on using the semantics of actors to provide an adequate model for the concurrent LP language FGHC [Hewitt and Agha 1988].

In summary, it seems clear that the actor model and Polka have a lot in common, and many fruitful analogies can be made between them. However, the actor model is lower level than Polka, and so is not as naturally expressive. Also, Polka has unification and the logical variable, which allows such things as synchronous messages, partial data structures, and the explicit manipulation of message streams. The ability for Polka objects to have more than one input stream and so possibly act in a non-deterministic way is also a major difference. An object's granularity is another important distinction.

Many of the differences can be removed by encoding the necessary functionality in an actor language. A more difficult question is whether such things as unification and the logical variable can be integrated with the actor model without drastically altering its semantics.

Part 3. Semantics and Implementation.

Chapter 9. Kernel Polka.

In the following two chapters on semantics and implementation, the discussion will centre on a subset of the language, called kernel Polka. Kernel Polka has all of the functionality of the full language but does not contain features which can be expressed in other, longer ways. In effect, kernel Polka is the collection of essential constructs necessary for the language. This 'paired down' language permits a simpler exposition of the semantics and implementation, and allows the less important aspects of the language to be ignored. Kernel Polka can be thought of as an intermediate language, generated by the compiler before the remaining constructs are translated into Parlog.

In this chapter, each section will consider a language component, such as output messages, and describe how certain operations of this type are rewritten in simpler terms.

9.1. Default Clauses.

The termination and delegation clauses, which are added by default in full Polka, are visible in kernel Polka.

The default termination clause closes the stream links to its inherited objects and its output streams when its own input stream terminates. These activities are easily expressed :

```
Ins :: last      =>   Out :: last ,
                    random :: last .
```

In this example, the object has one output stream called 'Out' and inherits an object called 'random'.

The delegation clause has two forms depending on if the object uses inheritance. When inheritance is utilised, an object forwards an unrecognised message to the first object named on its **inherits** line. This is coded as :

```
Ins :: Msg      =>   < first name on inherits line > :: Msg .
```

The name is used because **super** is not implemented.

If inheritance is not used then the unrecognised message is put onto the 'Error' output stream, which is represented as :

```
Ins :: Msg      =>   Error :: Msg .
```

In both cases, the clause is tried only after all other clauses have been rejected, and so is preceded by a ';' and is placed at the end of the **clauses** section.

9.2. Variables.

Kernel Polka does not support the automatic inclusion of 'Ins' and 'Error' streams, and so the compiler adds them to the variables section of the class definition.

9.3. Input Messages.

All messages are labelled with their input stream, due in part to the fact that there is no longer a default input stream.

Only input matching takes place between the message pattern and an incoming message, and a variable occurs only once in the pattern. This is not a major restriction since test unification of two variables can be carried out in the guard, and it allows a simple match algorithm to be utilised, based upon one used in the Strand language [Foster and Taylor 1989].

9.4. Clauses.

The use of **else** to connect several clauses is rephrased in the longer form. Thus :

```
Ins :: Message => Guard : Body1 else Body2 .
```

becomes

```
Ins :: Message => Guard : Body1 ;
Ins :: Message => Body2 .
```

Also, no user defined predicates appear in guards and only the parallel conjunction is available. This may seem overly restrictive, but the functionality of deep guards and the sequential conjunction can be encoded with the control metacall [Gregory 1987].

9.5. Becomes.

Multiple **becomes** of the form :

```
X1, X2 becomes Y1, Y2
```

are rewritten as several simpler **becomes** :

$$\begin{array}{l} X1 \text{ becomes } Y1, \\ X2 \text{ becomes } Y2 \end{array}$$

The '#' notation is not allowed, and is replaced by its **becomes** equivalent.

Thus :

$$\text{Ins} :: \text{Msg} \quad \Rightarrow \quad \text{predicate}(F, \#F) .$$

is coded as

$$\text{Ins} :: \text{Msg} \quad \Rightarrow \quad \begin{array}{l} \text{predicate}(F, \text{Result}) , \\ F \text{ becomes } \text{Result} . \end{array}$$

A requirement is that the new variable ('Result' in this example) is unique.

A major advantage of the removal of the '#' notation is that a goal in a clause is now composed of only Parlog, without any extra Polka notation. This greatly simplifies the semantics of goal reduction.

Expressions on the right hand side of **becomes** are not allowed, but can be easily coded by using 'is'. For instance,

$$X \text{ becomes } (2 + 3)$$

is rephrased as :

$$\begin{array}{l} \text{Result is } (2 + 3) , \\ X \text{ becomes } \text{Result} \end{array}$$

Care is taken to ensure that the new variable is unique.

9.6. Output Messages.

Multiple output messages are rewritten as a sequential conjunction of single messages. Therefore :

$$\text{Out} :: (M1, M2)$$

becomes

$$\text{Out} :: M1 \ \& \ \text{Out} :: M2$$

This requires a further transformation to remove the sequential conjunction.

Suspendible messages undergo a transformation to a new term structure whose functor is '\$wait'. For instance, the message :

$$\text{Out on Ok} :: \text{Message}$$

becomes

$$\text{Out} :: '\$wait'([Ok], \text{Message})$$

The first argument of '\$wait' is the list of suspending variables while the second argument is the message.

Although only output to streams have been discussed, these restructuring rules also apply to output messages in code section predicates, messages to inherited objects, and for self communication.

One useful result of these rewrites is that an output message is always a Parlog term or the **last** keyword, which allows a much simpler semantics to be developed for operations which use messages.

9.7. Inheritance.

As mentioned above, **super** is not implemented and is replaced by the name of the first class on the inherits line.

The **all** construct is recoded as a duplicated message to each of the inherited objects. Thus, if an object inherits 'a' and 'b' then

```
all :: print_state
```

becomes

```
a :: print_state ,
b :: print_state
```

The **returning** extension of **all** is not supported, and the compiler adds calls to 'response' or 'responseL' in the way outlined in section 5.11

9.8. The Parlog Component.

The restrictions on input matching mentioned in section 9.3 also apply to head unification in predicate calls, and this is enforced by translating Parlog into kernel Parlog [Gregory 1987]. This also means that there are no mode declarations and that each clause is in 'standard form', where the input matching and output unification is performed by explicit primitive calls in the guard and body. We further restrict our discussion to a flat version of the language which only uses the parallel conjunction.

Chapter 10. Semantics.

The most important benefit of defining the semantics for a language is that it results in an abstract definition which does not contain all the details of an implementation. This permits the language to be readily studied, and for faults in its design to be corrected more easily.

This chapter will concentrate on the operational semantics of kernel Polka, which will be described using a labelled transition system. Most emphasis will be placed on giving a meaning to the creation of an object and to its behaviour during execution, although there will be a brief discussion of term classes. An object will be represented by a set of 'i' tuples which represent the state and computation of the class instances which make up the object. For example, suppose an object 'a' has the inheritance tree in figure 10.1.

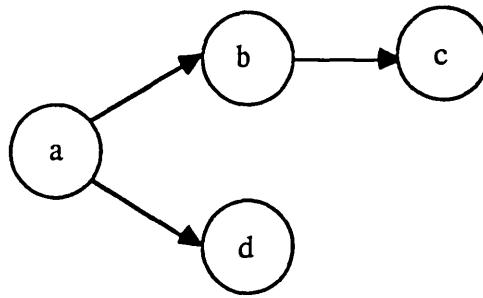


Figure 10.1 : Instance inheritance tree for 'a'.

The semantics of the object 'a' will consist of a set of four 'i' tuples representing the class instances for 'a', 'b', 'c', and 'd'.

Each 'i' tuple consists of five parts : its name, its set of pending goals GS, its state S, a set of flags F, and its program P. Thus, the 'i' tuple set for the 'a' object is :

$$\{ \langle a, GS_a, S_a, F_a, P_a \rangle, \langle b, GS_b, S_b, F_b, P_b \rangle, \\ \langle c, GS_c, S_c, F_c, P_c \rangle, \langle d, GS_d, S_d, F_d, P_d \rangle \}$$

The creation of an object involves four operations :

- creation of the 'i' tuple set
- linking 'i' tuples internally where necessary
- linking 'i' tuples to the outside world
- starting the execution of the tuple set

The creation task involves such things as finding what 'i' tuples are required by the object, and initialising the various arguments of each tuple. Internal linking means that delegation and self communication are possible, since tuples are able to send messages to each other. External linking specifies the interface between the tuple set and the user, which means explaining how messages are sent and received, and how visible state variables are bound. The execution of the tuple set requires a semantics to be given for the evaluation of an 'i' tuple which involves the definition of the various kernel Polka operations.

In the following discussion, each of these four tasks will be examined, but most time will be spent on defining the effect of the kernel Polka operations on the state and computation of an 'i' tuple.

The semantics put forward here can be viewed as a framework for the implementation explored in the next chapter, and many similarities can be seen between the two. A class is implemented using three kinds of Parlog predicates : the 'top', 'o', and 'i' predicates. When an object is invoked, the 'top' predicate is called, which uses the 'o' predicate to create a parallel conjunction of 'i' predicate goals, and to link them together. The 'top' and 'o' predicates can be seen as the implemented version of the semantics for the creation and linking of 'i' tuples, while the 'i' predicate is almost equivalent to an 'i' tuple.

10.1. Transition Systems.

Transition systems were first used by Hennessy and Plotkin [Hennessy and Plotkin 1979; Plotkin 1981; Plotkin 1983], and have already been utilised to give a semantics to a parallel OOP language called POOL [America et al. 1986], and to the actor model [Agha 1986]. In this kind of formalism, the semantics of a program is given in terms of the transitions it can make from one configuration to another. The execution of a program is then modelled by a sequence of configurations with transitions in between, starting from a suitable initial configuration.

A *labelled* transition system may be defined as a quadruple $\langle \Gamma, T, \Lambda, \rightarrow \rangle$.

Γ is a set of configurations, T is the set of terminal configurations such that $\Gamma \supseteq T$, Λ is a set of labels, and $\rightarrow$ is the transition relation such that $\Gamma \times \Lambda \times \Gamma \supseteq \rightarrow$. Relations $\langle \gamma, \lambda, \gamma' \rangle$ in $\rightarrow$ are written $\gamma \rightarrow_\lambda \gamma'$ and have the property that :

$$\forall \gamma \in T, \forall \lambda \in \Lambda, \forall \gamma' \in \Gamma : \text{not } \gamma \rightarrow_\lambda \gamma'$$

Labelled transition systems have been used extensively in the area of concurrent LP [Saraswat 1985; Saraswat 1987; Shapiro 1989; Foster and Taylor 1989].

10.2. The Configuration.

The configuration used here is a set of 'i' tuples, each of which is defined as :

$$iTuple = Name \times Goals \times State \times Flags \times Program$$

Therefore a typical configuration *c* might be :

$$\{ \langle Nm_1, GS_1, S_1, F_1, P_1 \rangle, \langle Nm_2, GS_2, S_2, F_2, P_2 \rangle \}$$

Each of the five parts of an 'i' tuple will be examined in more detail.

10.2.1. Tuple Goals.

A typical goal set *GS* stores a pair of the form :

$$\{ G_1, G_2, \dots G_n ; \theta \}$$

$G_1, \dots G_n$ is a set of goals and θ is the current bindings to the variables. If the binding is empty (i.e ϵ) then the set can be written as :

$$\{ G_1, G_2, \dots G_n \}$$

10.2.2. Tuple State.

The state *S* is a quadruple with the form :

$$\langle OVars, Inherits, Self, Becomes \rangle$$

OVars is used to store name, type and value information about the visible and invisible variables in the 'i' tuple.

Inherits is a list of the names of the inherited classes used by the 'i' tuple.

Self contains the self communication output stream.

Becomes is a list of terms of the form :

$$bvar(OName, Term)$$

which indicates that the variable with the name 'OName' is to take on the 'Term' value. A 'bvar' term is created as a result of a **becomes** operation, and is stored since our semantics dictate that changes of state only occur at the end of a clause.

The type of the state argument is defined as :

$$State = ((OName \times Type) \rightarrow VAR) \times Name^* \times VAR \times (OName \rightarrow TERM)^*$$

OName is the name of a Polka variable which is specified in terms of the function :

name(VAR)

where VAR is a variable.

Type is a set comprised of elements for each kind of Polka variable :

Type = { istream, ostream, state, ostate }

Name is a constant.

10.2.3. Tuple Flags.

Flags is the third argument of an 'i' tuple, and has the form :

< MoreFlag, FFlag >

MoreFlag will have a value taken from the set : { more, nomore }

FFlag will have a value from the set : { finished, notfinished }

'MoreFlag' is used to indicate whether a further message should be processed while the current one is being processed. If the separator of the clause is '&';' or '&.' then 'MoreFlag' will be set to 'nomore', but if the separator is ';' or '.' then it will be set to 'more'.

'FFlag' indicates whether the tuple should cease execution, and is set to 'finished' when the **last** keyword occurs in the input, or **i_am** is used.

10.2.4. Tuple Program.

The fourth argument of an 'i' tuple is the program P, which has the form :

< IP, OP, CSP, GP >

IP is a (possibly empty) set of goals which are to be executed initially, OP is the program definition for the class, CSP stores the code section predicates for the object, and GP contains the global predicates.

10.3. A Labelled Transition System for Kernel Polka.

The notation for the system will be a variant of the one explained above, and is defined by the triple :

$$\langle \Gamma_x, T_x, \rightarrow_x \rangle$$

x belongs to the following set of syntactic category labels :

$$\{ \text{exs, ex, goal, message, parlog, code, becomes, send, self, super} \}$$

Γ_x is the set of configurations for the x label.

T_x is the set of terminal configurations for the x label.

In many of the transitions, the subscripts will be omitted since the context makes it clear which is being used.

Γ_x and T_x for all the labels are :

Exs.

$$\begin{aligned} \Gamma_{\text{exs}} &= \{ \langle \text{Nm, GS, S, F, P} \rangle^* \} \vee \{ \text{tt} \} \vee \{ \text{ff} \} \\ T_{\text{exs}} &= \{ \text{tt} \} \vee \{ \text{ff} \} \end{aligned}$$

Ex.

$$\begin{aligned} \Gamma_{\text{ex}} &= \{ \langle \text{Nm, GS, S, F, P} \rangle \} \vee \{ \text{tt} \} \vee \{ \text{ff} \} \\ T_{\text{ex}} &= \{ \text{tt} \} \vee \{ \text{ff} \} \end{aligned}$$

Goal.

$$\begin{aligned} \Gamma_{\text{goal}} &= \{ \langle \text{G, S, FFlag, P} \rangle \} \vee \{ \langle \emptyset, \text{S, FFlag} \rangle \} \vee \\ &\quad \{ \langle \text{GS, S, FFlag} \rangle \} \vee \{ \text{ff} \} \\ T_{\text{goal}} &= \{ \langle \emptyset, \text{S, FFlag} \rangle \} \vee \{ \langle \text{GS, S, FFlag} \rangle \} \vee \{ \text{ff} \} \end{aligned}$$

Message.

$$\begin{aligned} \Gamma_{\text{message}} &= \{ \langle \text{S, OP} \rangle \} \vee \{ \langle \emptyset, \text{S, F} \rangle \} \vee \{ \langle \text{GS, S, F} \rangle \} \vee \{ \text{ff} \} \\ T_{\text{message}} &= \{ \langle \emptyset, \text{S, F} \rangle \} \vee \{ \langle \text{GS, S, F} \rangle \} \vee \{ \text{ff} \} \end{aligned}$$

Parlog.

$$\begin{aligned} \Gamma_{\text{parlog}} &= \{ \langle \text{G, GP} \rangle \} \vee \{ \emptyset \} \vee \{ \text{ff} \} \vee \{ \text{GS} \} \\ T_{\text{parlog}} &= \{ \emptyset \} \vee \{ \text{ff} \} \vee \{ \text{GS} \} \end{aligned}$$

Code.

$$\begin{aligned} \Gamma_{\text{code}} &= \{ \langle \text{G, CP} \rangle \} \vee \{ \emptyset \} \vee \{ \text{ff} \} \vee \{ \text{GS} \} \\ T_{\text{code}} &= \{ \emptyset \} \vee \{ \text{ff} \} \vee \{ \text{GS} \} \end{aligned}$$

Becomes.

$$\Gamma_{\text{becomes}} = \{ \langle \text{BECOMES, S} \rangle \} \vee \{ \text{ff} \} \vee \{ \text{S} \}$$

$$T_{\text{becomes}} = \{ \text{ff} \} \vee \{ S \}$$

Send.

$$\Gamma_{\text{send}} = \{ \langle \text{SEND}, S \rangle \} \vee \{ \text{ff} \} \vee \{ S \}$$

$$T_{\text{send}} = \{ \text{ff} \} \vee \{ S \}$$

Self.

$$\Gamma_{\text{self}} = \{ \langle \text{SELF}, S \rangle \} \vee \{ S \}$$

$$T_{\text{self}} = \{ S \}$$

Super.

$$\Gamma_{\text{super}} = \{ \langle \text{SUPER}, S \rangle \} \vee \{ \text{ff} \} \vee \{ S \}$$

$$T_{\text{super}} = \{ \text{ff} \} \vee \{ S \}$$

The transitions will utilise a number of shorthand forms, the most important of which is the representation of several rules in one definition. For instance, the transition :

$$\frac{A \rightarrow B1 \mid B2}{C \rightarrow D1 \mid D2}$$

is short for :

$$\frac{A \rightarrow B1}{C \rightarrow D1} \quad \text{and} \quad \frac{A \rightarrow B2}{C \rightarrow D2}$$

10.4. The Creation and Linking of the Configuration.

In the following section, and the ones to follow, use is made of functions and procedures for accessing and manipulating data structures in the configuration. Some of their definitions are given informally because the details are unimportant at this level of the presentation.

10.4.1. The Creation of the 'i' Tuple Set.

For a class C, the set of 'i' tuples is specified by :

$$\{ \langle C, GS_C, S_C, F_C, P_C \rangle \} \vee \text{Set}_C$$

where $\langle C, GS_C, S_C, F_C, P_C \rangle$ is generated by a call to 'initialise' :

```

mode initialise( ?, ^ ).
initialise( C, < C, GSC, SC, FC, PC > )

```

'initialise' initialises the arguments for a tuple. For instance, the 'GS' argument is set to ϵ to indicate that no goals are currently pending, the 'Becomes' argument of the S state is set to [] since no **becomes** operations need to be carried out. Also, the 'F' argument gets the value :

```
< nomore, nofinished >
```

Set_C is generated using the predicate call :

```
add_inherits( inherits( SC ), SetC )
```

'inherits' is a function which returns the list of inherited classes held in the state argument, while the purpose of 'add_inherits' is to search through the inherited classes of C and create an 'i' tuple for each one. Its definition is given informally in program 10.1.

```

mode add_inherits( ?, ^ ).
add_inherits( [],  $\epsilon$  ).
add_inherits( [ I | Rest ], { < I, Gi, Si, Fi, Pi > }  $\vee$  Set1  $\vee$  Set2 ) <-
    initialise( I, < Gi, Si, Fi, Pi > ),           % initialise an I tuple
    add_inherits( inherits( Si ), Set1 ),          % add 'i' tuples for I's ancestors
    add_inherits( Rest, Set2 ).

```

Program 10.1 : A 'add_inherits' predicate.

10.4.2. Linking the 'i' Tuples internally.

This task has two parts : the creation of the links which simulate inheritance, and the generation of those for self communication. Rather than specify a precise connection mechanism, the following predicate will be used:

```

mode connect( ?, ^ ).
connect( < output stream >, < input stream > )

```

It guarantees that a message transmitted along the output stream will eventually arrive on the input stream, but says nothing about the manner of the connection.

For inheritance, the links are formed by calling 'iconnect', like so :

```
iconnect( { < C, GSC, SC, FC, PC > }  $\vee$  SetC )
```

'iconnect' is defined by program 10.2.

```

mode iconnect( ? ).
iconnect(  $\epsilon$  ).
iconnect( { < I, Gi, Si, Fi, Pi > }  $\vee$  Set ) <-      % run through each tuple in the set
    iconnect1( inherits( Si ) ),
    iconnect( Set ).

mode iconnect1( ? ).
iconnect1( [] ).
iconnect1( [ IName | Rest ] ) <-                      % run through the ancestors of each tuple
    connect( get_inherits( IName ), dis( IName ) ),
    iconnect1( Rest ).

```

Program 10.2 : Predicates for 'iconnect'.

'iconnect' processes each tuple by calling 'iconnect1' which examines the list of inherited classes for the tuple. It obtains the output stream used for delegating messages and connects it to the default input stream of the tuple representing the inherited class. The output stream is accessed by calling the 'get_inherits' function, while the default input stream is retrieved by using the 'dis' function.

The links for self communication are made in a similar way by calling 'sconnect' :

```
sconnect( { < C, GSC, SC, FC, PC > }  $\vee$  SetC )
```

'sconnect' is defined in program 10.3.

```

mode sconnect( ? ).
sconnect(  $\epsilon$  ).
sconnect( { < I, Gi, Si, Fi, Pi > }  $\vee$  Set ) <-      % run through each tuple in the set
    connect( get_self( Si ), dis( bottom ) ),
    sconnect( Set ).

```

Program 10.3 : A 'sconnect' predicate.

This predicate links the self communication output stream of every tuple to the default input stream of the tuple representing the bottom instance in the inheritance graph. The stream for self communication is accessed via the 'get_self' function, while the name of the bottom instance is stored in 'bottom'.

10.4.3. Linking the 'i' Tuples externally.

The user invokes an instance of class C by calling 'C' with arguments for the visible variables of the class and all of its inherited classes (except their input streams). A list of these visible variables is specified as :

```
visible_OVars( SC ) + Vis
```

where 'visible_OVars' is a function which returns the visible variables from the state, '+' is a concatenation operator, and 'Vis' is the list built by a call to 'visible' :

```
visible( inherits( SC ), Vis )
```

'visible' searches through the inherited classes of C in a depth first order and returns all of the visible variables apart from those of type istream.

The user could use these variables directly, but then the semantics would have to reflect the way in which streams are treated as partially instantiated lists. To hide this the user shall instead use a slightly different list of arguments whose stream variables have an interface between them and the stream variables used in the 'i' tuples. This is done by calling 'io_interface' like so :

```
io_interface( [ visible_OVars( SC ) + Vis ], UserVars )
```

After a call to 'io_interface', the user can start executing the object by calling 'C' with the arguments returned in 'UserVars'.

'io_interface' is defined by program 10.4.

```
mode io_interface( ?, ^ ).
io_interface( [], [] ).

io_interface( [ Var | Rest ], [ UVar | URest ] ) <- % an input stream interface
  is_istream( Var ) :
    input( UVar, Var ), % user variable is UVar
    io_interface( Rest, URest ).

io_interface( [ Var | Rest ], [ UVar | URest ] ) <- % an output stream interface
  is_ostream( Var ) :
    output( Var, UVar ), % user variable is UVar
    io_interface( Rest, URest );

io_interface( [ Var | Rest ], [ Var | URest ] ) <- % state and ostate vars have no interface
  io_interface( Rest, URest ).
```

Program 10.4 : An 'io_interface' predicate.

At the heart of 'io_interface' are calls to 'input' and 'output'. 'input' translates partial instantiations of an input list into calls to 'send_mesg', while 'output' uses 'get_mesg' to obtain messages to partially instantiate an output list.

```

mode input( ?, ^ ) .
input( [], I ) <-                                % deal with input termination
    close_stream( I ) .
input( [ Msg | U ], I ) <-                        % transmit a 'Msg' message
    send_mesg( I, I1, Msg ) &
    input( U, I1 ) .

```

Program 10.5 : An 'input' predicate.

```

mode output( ?, ^ ) .
output( O, U ) <-                                % termination of the output stream
    is_closed( O ) :
    U = [] .
output( O, U ) <-                                % handle a '$wait' message
    get_mesg( O, O1, '$wait'( L, Msg ) ) :
    wait_until( L, Msg, U, U1 ) ,
    output( O1, U1 ) ;
output( O, U ) <-                                % handle an ordinary message
    get_mesg( O, O1, Msg ) :
    U = [ Msg | U1 ] ,
    output( O1, U1 ) .

```

Program 10.6 : An 'output' predicate.

Note how 'input' and 'output' use predicates to manipulate the streams going into and leaving the tuples. This allows the exact nature of the streams to be hidden at this level. 'output' is also interesting because of the way that it handles '\$wait' messages by holding them in a 'wait_until' process until the variables in the 'L' list are all bound.

The modes for 'close_stream', 'send_mesg', 'is_closed', and 'get_mesg' are :

```

mode close_stream( ^ ) .
close_stream( < output stream > ) .

mode send_mesg( ^, ?, ? ) .
send_mesg( < output stream >, < remainder of stream >, < message > ) .

mode is_closed( ? ) .
is_closed( < input stream > ) .

```

and

```

mode get_mesg( ?, ^, ^ ) .
get_mesg( < input stream >, < remainder of stream >, < message > ) .

```

The meaning of 'close_stream' is that it closes the stream it is given as an argument.

The meaning of 'send_mesg' is that the message is put onto the output stream, and the remaining part of the stream enters via the second argument.

The meaning of 'is_closed' is that it succeeds only if its stream argument is closed.

The meaning of 'get_mesg' is that the message is removed from the input stream, returning the remainder of the stream.

'get_mesg' is also used by the input message transitions (section 10.5.3) to pick messages off the 'i' tuple input streams. In fact, these messages were put there by calls to 'send_mesg' in 'input'. Similarly, 'send_mesg' is used by the transitions for the send operation (section 10.5.4.2) to put messages onto the 'i' tuple output streams. It is these which the calls to 'get_mesg' access in 'output'.

'is_closed' and 'close_stream' are also utilised in a number of places, most notably when dealing with the last keyword.

The other job done here (which has not been represented in the code) is to merge the error output streams used by various tuples into a single error stream at the user level. This is achieved by scanning through the list of output streams in 'UserVars' and merging all of the 'Error' streams into a single new output stream.

10.4.4. Executing the 'i' Tuple Set.

The effects on each 'i' tuple will be investigated in section 10.5, but two transitions can be given for the computation of the tuple set as a whole :

1. A successful computation.

$$\frac{\forall i < Nm_i, GS_i, S_i, F_i, P_i > \rightarrow_{ex} tt}{\{ < Nm_1, GS_1, S_1, F_1, P_1 >, \dots < Nm_n, GS_n, S_n, F_n, P_n > \} \rightarrow_{exs} tt}$$

2. A failing computation.

$$\frac{\exists i < Nm_i, GS_i, S_i, F_i, P_i > \rightarrow_{ex} ff}{\{ < Nm_1, GS_1, S_1, F_1, P_1 >, \dots < Nm_n, GS_n, S_n, F_n, P_n > \} \rightarrow_{exs} ff}$$

The first transition states that the 'i' tuple set executes to success (tt) when all of its component tuples do. One property not shown in the transition is the way that the computation of each 'i' tuple side effects the visible variables of the user, and vice versa.

The second transition states that if one of the tuples fails (ff) then the entire set fails.

In this simple model, no account is taken of the problem of deadlock.

10.5. Executing a single 'i' Tuple.

The transitions which manipulate conjunctions of goals in an 'i' tuple will be given in section 10.5.1. Section 10.5.2 will look at the transitions which manipulate a single goal, and will show how calls to **code** section predicates and global predicates are distinguished from kernel Polka operations. The processing of input messages will be considered in section 10.5.3 and this will be followed by the longest section (10.5.4) which looks at kernel Polka operations. That section will be subdivided according to the five main types of operator :

- becomes
- send
- self
- super
- i_am

Section 10.5.5 will outline the transitions needed to deal with **code** section predicates.

10.5.1 Goals Transitions.

An executing 'i' tuple should first carry out any initial actions and this case is specified by transition (1) below. Case (2) defines the execution of a tuple when both goals and input messages can be processed. Case (3) states what happens when there are no goals present. Case (4) concerns an 'i' tuple which can not accept another message until the processing of the current message has been completed. Case (5) specifies how a tuple terminates.

1. Carry out any initial actions.

$ip(P) = \{ G1, \dots Gm \text{ SEP } \}$
 and let $\{ G1', \dots Gm' \}$ be a fresh copy of the conjunction part of the clause
 and checkend(SEP, MoreFlag), which is defined below

$\langle Nm, \epsilon, S, \langle \text{nomore}, \text{nofinished} \rangle, P \rangle \rightarrow_{ex}$
 $\langle Nm, \{ G1', \dots Gm' \}, S, \langle \text{MoreFlag}, \text{nofinished} \rangle, P1 \rangle$

where

'ip' is a function which reads the initial actions from P
 and $P1 := P - ip(P)$, which removes the initial actions from P

'checkend' is :

```
mode checkend( ?, ^ ) .
checkend( '&.', nomore ) .
checkend( '&;', nomore ) .
checkend( '.', more ) .
checkend( ';', more ) .
```

Program 10.7 : A 'checkend' predicate.

2. Either reduce a goal or input a message.

a) Reduce a goal :

$$\begin{array}{l}
 \langle Gi, S, \text{nofinished}, P \rangle \rightarrow_{\text{goal}} \langle \theta 1, S', \text{FFlag} \rangle \mid \text{ff} \mid \langle \{ Gi1, \dots, Gim ; \theta 1 \}, S', \text{FFlag} \rangle \\
 \hline
 \langle Nm, \{ G1, \dots, Gi, \dots, Gn ; \theta 0 \}, S, \langle \text{more}, \text{nofinished} \rangle, P \rangle \rightarrow_{\text{ex}} \\
 \quad \langle Nm, \{ (G1, \dots, Gn) \theta 1 ; \theta 0 \cdot \theta 1 \}, S', \langle \text{more}, \text{FFlag} \rangle, P \rangle \\
 \quad \mid \text{ff} \\
 \quad \langle Nm, \{ (G1, \dots, Gi1, \dots, Gim, \dots, Gn) \theta 1 ; \theta 0 \cdot \theta 1 \}, S', \langle \text{more}, \text{FFlag} \rangle, P \rangle
 \end{array}$$

b) Input a message :

$$\begin{array}{l}
 \langle S, \text{op}(P) \rangle \rightarrow_{\text{message}} \langle \theta 1, S, F \rangle \mid \text{ff} \mid \langle \{ G1i, \dots, Gmi ; \theta 1 \}, S', F \rangle \\
 \hline
 \langle Nm, \{ G1, \dots, Gn ; \theta 0 \}, S, \langle \text{more}, \text{nofinished} \rangle, P \rangle \rightarrow_{\text{ex}} \\
 \quad \langle Nm, \{ (G1, \dots, Gn) \theta 1 ; \theta 0 \cdot \theta 1 \}, S', F, P \rangle \\
 \quad \mid \text{ff} \\
 \quad \langle Nm, \{ (G1, \dots, Gn, G1i, \dots, Gmi) \theta 1 ; \theta 0 \cdot \theta 1 \}, S', F, P \rangle
 \end{array}$$

where

'op' is a function which reads the object program from P

3. No goals to reduce, so input a message.

$$\begin{array}{l}
 \langle S, \text{op}(P) \rangle \rightarrow_{\text{message}} \langle \theta 1, S, F \rangle \mid \text{ff} \mid \langle \{ G1, \dots, Gm ; \theta 1 \}, S', F \rangle \\
 \hline
 \langle Nm, \theta 0, S, \langle \text{more}, \text{nofinished} \rangle, P \rangle \rightarrow_{\text{ex}} \\
 \quad \langle Nm, \{ \theta 0 \cdot \theta 1 \}, S', F, P \rangle \\
 \quad \langle Nm, \{ (G1, \dots, Gm) \theta 1 ; \theta 0 \cdot \theta 1 \}, S', F, P \rangle
 \end{array}$$

4. Process only the current message by reducing all the goals it creates, and then processing the stored **becomes** changes.

a) Reduce a goal.

$$\begin{array}{c}
 \frac{\langle Gi, S, FFlag, P \rangle \rightarrow_{goal} \langle \theta 1, S1, FFlag1 \rangle \mid ff \mid \langle \{ Gi1, \dots, Gim; \theta 1 \}, S1, FFlag1 \rangle}{\langle Nm, \{ G1, \dots, Gi, \dots, Gn; \theta 0 \}, S, \langle nomore, FFlag \rangle, P \rangle \rightarrow_{ex} \langle Nm, \{ (G1, \dots, Gn) \theta 1; \theta 0 \cdot \theta 1 \}, S1, \langle nomore, FFlag1 \rangle, P \rangle} \\
 \mid ff \\
 \langle Nm, \{ (G1, \dots, Gi1, \dots, Gim, \dots, Gn) \theta 1; \theta 0 \cdot \theta 1 \}, S1, \langle nomore, FFlag1 \rangle, P \rangle
 \end{array}$$

b) Process the stored **becomes** changes.

$\langle Nm, \theta, S, \langle nomore, FFlag \rangle, P \rangle \rightarrow_{ex} \langle Nm, \theta, S2, \langle more, FFlag \rangle, P \rangle \mid ff$
 where
 do_becomes(becomes(S), ovars(S), OVars'), which is defined below
 and 'becomes' reads the Becomes list from S
 and 'ovars' reads OVars from S
 and S1 := replace_ovar(OVars, S), which replaces OVars in S with OVars'
 and S2 := nil_becomes(S1), which sets Becomes to []

A simplified definition of 'do_becomes' is given by program 10.8.

```

mode do_becomes( ?, ?, ^ ).
do_becomes( [], OVars, OVars ).
do_becomes( [ bvar( OName, Term ) | Bvs ], OVars, OVars2 ) <- % process each 'bvar'
    dbecomes( OName, Term, OVars, OVars1 ),
    do_becomes( Bvs, OVars1, OVars2 ).

mode dbecomes( ?, ?, ?, ^ ).
dbecomes( OName, Term, [ VAR | OVars ], [ Term | OVars ] ) <- % find right variable in OVars
    name( VAR ) == OName : true ; % and replace with Term
dbecomes( OName, Term, [ V | OVars ], [ V | OVars1 ] ) <-
    dbecomes( OName, Term, OVars, OVars1 ).
  
```

Program 10.8 : Predicates for 'do_becomes'.

5. A tuple will terminate after reducing its remaining goals, processing its stored **becomes** changes, and closing the self stream.

a) Reduce a goal.

$$\begin{array}{c}
 \langle Gi, S, \text{finished}, P \rangle \rightarrow_{\text{goal}} \langle \theta 1, S', \text{finished} \rangle \mid \text{ff} \mid \langle \{ Gi1, \dots, Gim ; \theta 1 \}, S', \text{finished} \rangle \\
 \hline
 \langle Nm, \{ G1, \dots, Gi, \dots, Gn ; \theta 0 \}, S, \langle \text{more}, \text{finished} \rangle, P \rangle \rightarrow_{\text{ex}} \langle Nm, \{ (G1, \dots, Gn) \theta 1 ; \theta 0 \cdot \theta 1 \}, S', \langle \text{more}, \text{finished} \rangle, P \rangle \\
 \mid \text{ff} \\
 \langle Nm, \{ (G1, \dots, Gi1, Gi2, \dots, Gim, \dots, Gn) \theta 1 ; \theta 0 \cdot \theta 1 \}, S', \langle \text{more}, \text{finished} \rangle, P \rangle
 \end{array}$$

b) Process the stored **becomes** changes.

$$\begin{array}{c}
 \langle \theta, S, \langle \text{more}, \text{finished} \rangle, P \rangle \rightarrow_{\text{ex}} \langle \theta, S1, \langle \text{more}, \text{finished} \rangle, P \rangle \mid \text{ff} \\
 \text{where} \\
 \text{do_becomes}(\text{becomes}(S), \text{ovars}(S), \text{OVars}') \\
 \text{and } S1 := \text{nil_becomes}(\text{replace_ovar}(\text{OVars}', S))
 \end{array}$$

c) Close the self stream.

$$\begin{array}{c}
 \langle \theta, S, \langle \text{more}, \text{finished} \rangle, P \rangle \rightarrow_{\text{ex}} \text{tt} \\
 \text{where} \\
 \text{close_stream}(\text{get_self}(S))
 \end{array}$$

10.5.2 Goal Transitions.

The first and second transitions distinguish calls to **code** section predicates from global ones. Transitions for goals which use **code** section predicates are given in section 10.5.5, but no transitions are presented for global predicate goals because of their similarity to the **code** section transitions. If a goal is not a predicate then it will be dealt with by one of the transitions defined for kernel Polka operations in section 10.5.4.

1. Processing code section goals :

$$\begin{array}{c}
 \langle G, CP \rangle \rightarrow_{\text{code}} \theta \mid \text{ff} \mid \{ G1, \dots, Gm ; \theta \} \\
 \hline
 \langle G, S, \text{FFlag}, P \rangle \rightarrow_{\text{goal}} \langle \theta, S, \text{FFlag} \rangle \mid \text{ff} \mid \langle \{ G1, \dots, Gm ; \theta \}, S, \text{FFlag} \rangle \\
 \text{where} \\
 \text{name}(G) / \text{arity}(G) \in \text{csp}(P), \text{ the code section predicates in } P \\
 \text{and } CP = \text{csp}(\text{name}(G), \text{arity}(G), P), \text{ the code section predicate with } G\text{'s name and arity}
 \end{array}$$

2. Processing globally defined goals :

$$\langle G, GP \rangle \rightarrow_{\text{parlog}} \theta \mid \text{ff} \mid \langle G1, \dots Gm ; \theta \rangle$$

$$\langle G, S, FFlag, P \rangle \rightarrow_{\text{goal}} \langle \theta, S, FFlag \rangle \mid \text{ff} \mid \langle \{ G1, \dots Gm ; \theta \}, S, FFlag \rangle$$

where

name(G) / arity(G) $\in$ gp(P), the globally defined predicates in P

and GP = gp(name(G), arity(G), P), the global predicate with G's name and arity

10.5.3 Processing Input Messages.

The processing of input messages is done by matching a message (or messages) against a clause using transitions (1) and (2) defined below. The transitions closely follow the class structure which can be described by the following BNF :

OP ::= OP ; OP % OP is made up of sequential-or clause groups
 | OPC

OPC ::= OC { OC } % OPC is made up of parallel-or clause groups

OC ::= INPUTS '=>' GUARD ':' G { ';' G } SEP % OC is a Polka clause

Transition (3) deals with the successful matching of a message to a clause, while (4) deals with the failure case. At the heart of these transitions is an invocation of the 'match' predicate which first calls 'convert' in order to rewrite the INPUTS message part of a clause as a list of labelled terms. Then a call to 'match1' associates these labelled terms with the relevant input streams, and 'messageL' tries to match the labelled terms with received messages, by using the 'smatch' predicate.

1. Choose a clause from a sequential-or group.

$$\langle S, OP1 \rangle \rightarrow_{\text{message}} \langle \theta, S', F \rangle \mid \text{ff} \mid \langle \{ Gi, \dots Gi ; \theta \}, S', F \rangle$$

$$\langle S, OP1 ; OP2 \rangle \rightarrow_{\text{message}} \langle \theta, S', F \rangle \mid \langle S, OP2 \rangle \mid \langle \{ Gi, \dots Gi ; \theta \}, S', F \rangle$$

2. Choose a clause from a parallel-or group.

$$\frac{\begin{array}{l} \langle S, OPC \rangle \rightarrow_{\text{message}} \langle \theta, S', F \rangle \mid \text{ff} \mid \langle \{ Gi, \dots Gi ; \theta \}, S', F \rangle \\ \hline \langle S, OP \rangle \rightarrow_{\text{message}} \langle \theta, S', F \rangle \mid \text{ff} \mid \langle \{ Gi, \dots Gi ; \theta \}, S', F \rangle \end{array}}$$

3. Successfully match with a message.

and $OC_i = \langle \text{INPUTS} \Rightarrow \text{GUARD} : G_1, \dots G_m \text{ SEP} \rangle$
 and let $\langle \text{INPUTS}' \Rightarrow \text{GUARD}' : G_1', \dots G_m' \rangle$ be a fresh copy of the clause
 and $\text{checkend}(\text{SEP}, \text{MoreFlag})$
 and $\text{match}(\text{INPUTS}', \text{istream}(\text{ovars}(S)), \theta, \text{InV}', \text{FFlag})$, which is defined below
 and $\theta(\text{GUARD}') = \text{true}$

$\langle S, \{ OC_1, \dots OC_i, \dots OC_n \} \rangle \rightarrow_{\text{message}}$
 $\langle \theta, S, \langle \text{MoreFlag}, \text{FFlag} \rangle \rangle \mid$
 $\langle \{ G_1', \dots G_m' ; \theta \}, S', \langle \text{MoreFlag}, \text{FFlag} \rangle \rangle$

where 'istream' extracts the input stream variables from OVars
 and $S' := \text{replace_istream}(\text{InV}', S)$, such that InV' replaces the old istream information in S

The θ case is returned when the body of the OC_i clause is the single goal 'true'

4. Fail to match with a message.

and $\forall OC_i \in \{ OC_1, \dots OC_n \}$
 and let $\langle \text{INPUTS}' \Rightarrow \text{GUARD}' : G_1', \dots G_m' \rangle$ be a fresh copy of the clause
 and $\text{match}(\text{INPUTS}', \text{istream}(\text{ovars}(S)), \text{fail}, _, _)$
 or $\theta(\text{GUARD}') = \text{ff}$

$\langle S, \{ OC_1, \dots OC_i, \dots OC_n \} \rangle \rightarrow_{\text{message}} \text{ff}$

'match' is :

```
mode match(?, ?, ^, ^, ^).
match(INPUTS, InV, theta, InV1, FFlag) <-
  convert(INPUTS, Msgs, FFlag),           % convert INPUTS to Msgs
  match1(Msgs, InV, theta, InV1) : true;    % match Msgs with input messages
match(INPUTS, InV, fail, _, _).
```

Program 10.9 : A 'match' predicate.

'convert' converts the format of the messages in the clauses to one which is easier to manipulate, and returns them in 'Msgs'. Also, the binding of 'FFlag' is set to 'finished' if the **last** keyword occurs in any of the messages, and 'nofinished' in all other cases.

The labelled terms generated by 'convert' have the following general appearance :

label :: [message1, message2, . .]

'match1' is informally defined in program 10.10.

```
mode match1( ?, ?, ^, ^ ).
match1( [], InV, ε, InV ).
match1( [ Label :: last | List ], InV ∨ < Label, Stream >, θ1,
        InV1 ∨ < Label, Stream > ) <-      % deal with the last keyword
    is_closed( Stream ),
    match1( List, InV, θ1, InV1 );
match1( [ Label :: Msgs | List ], InV ∨ < Label, Stream >, θ • θ1,
        InV1 ∨ < Label, Stream1 > ) <-      % match the messages from the
    messageL( Msgs, Stream, Stream1, θ ),      % clauses with those actually input
    match1( List, InV, θ1, InV1 ).
```

Program 10.10 : A 'match1' predicate.

The ∨ is used to access an element in the 'InV' input stream list.

'messageL' is given in program 10.11.

```
mode messageL( ?, ?, ^, ^ ).
messageL( [], Stream, Stream, ε ).
messageL( [ M2 | Msgs ], Stream, Stream2, θ • θ1 ) <-
    get_mesg( Stream, Stream1, M1 ) &      % 'get_mesg' gets a message M1
    smatch( M2, M1 ) = θ ,                  % match it with a message in the
    messageL( Msgs, Stream1, Stream2, θ1 ). % clause head (M2)
```

Program 10.11 : A 'messageL' predicate.

'smatch' can be defined by means of a table based on one used in the Strand language [Foster and Taylor 1989] :

input message M1 clause message M2	variable	constant	structure
variable	M2 <= M1	M2 <= M1	M2 <= M1
constant	suspend	if M1 /= M2 fail	fail
structure	suspend	fail	match arguments

Table 10.1 : Definition for 'smatch'.

10.5.4. Processing Kernel Polka Operations.

The relevant operations can be listed using the following BNF :

POLKA_OP ::= BECOMES | SEND | SELF | SUPER | I_AM

Each of these five operations will be considered in this section.

10.5.4.1. The Becomes Operation.

The BNF for BECOMES in kernel Polka is :

BECOMES ::= VAR becomes TERM

The transition that deals with this is :

$$\frac{\langle \text{BECOMES}, S \rangle \rightarrow_{\text{becomes}} S1 \mid \text{ff}}{\langle \text{BECOMES}, S, \text{FFlag}, P \rangle \rightarrow_{\text{goal}} \langle \epsilon, S1, \text{FFlag} \rangle \mid \text{ff}}$$

The transitions for $\rightarrow_{\text{becomes}}$ are :

- 1) $\langle \langle \text{VAR becomes TERM} \rangle, S \rangle \rightarrow_{\text{becomes}} S1$
 where
 $\text{name}(\text{VAR}) \in \text{names in ovars}(S)$
 and $\text{Becomes} := \text{becomes}(S)$
 and $S1 := \text{replace_becomes}([\text{bvar}(\text{name}(\text{VAR})), \text{TERM}] \mid \text{Becomes}, S)$,
 which replaces the old value of Becomes with the new list
- 2) $\langle \langle \text{VAR becomes TERM} \rangle, S \rangle \rightarrow_{\text{becomes}} \text{ff}$
 where
 $\text{name}(\text{VAR}) \notin \text{names in ovars}(S)$

The idea behind the first transition is to store the **becomes** operation in the 'Becomes' argument of the state, and then process it later using transitions 4 or 5 of section 10.5.1. The second transition deals with an attempt to change an unknown variable.

10.5.4.2. The Send Operation.

The BNF for SEND in kernel Polka is :

SEND ::= VAR '::' TERM | VAR '::' last

This leads to the transition :

$$\frac{\langle \text{SEND}, S \rangle \rightarrow_{\text{send}} S1 \mid \text{ff}}{\langle \text{SEND}, S, \text{FFlag}, P \rangle \rightarrow_{\text{goal}} \langle \epsilon, S1, \text{FFlag} \rangle \mid \text{ff}}$$

The $\rightarrow_{\text{send}}$ transitions are :

- 1) $\langle \langle \text{VAR} ':: \text{TERM} \rangle, S \rangle \rightarrow_{\text{send}} S1$
 where
 $\text{VAR} \in \text{names in ostream(ovars(S))}$
 and $\text{Vr} := \text{get_var(name(VAR), S)}$, which extract the named variable from 'OVars' in S
 and $\text{send_mesg(Vr, Vr', TERM)}$
 and $S1 := \text{replace_var(name(VAR), Vr', S)}$, which replaces the named variable with the Vr' value

- 2) $\langle \langle \text{VAR} ':: \text{last} \rangle, S \rangle \rightarrow_{\text{send}} S$
 where
 $\text{VAR} \in \text{names in ostream(ovars(S))}$
 and $\text{close_stream(get_var(name(VAR), S))}$, which closes VAR's stream

- 3) $\langle \langle \text{VAR} ':: \text{TERM} \rangle, S \rangle \rightarrow_{\text{send}} \text{ff}$
 where
 $\text{VAR} \notin \text{names in ostream(ovars(S))}$

The first transition extracts the output stream from 'OVars' and puts the message onto it using 'send_mesg'. The remaining part of the stream is then stored for future use. The second transition closes the output stream, and the third transition deals with the case when the variable used in the send is not an output stream, and so failure occurs.

10.5.4.3. The Self Operation.

The BNF for SELF in kernel Polka is :

$$\text{SELF} ::= \text{self ':' TERM} \quad | \quad \text{self ':' last}$$

This leads to the transition :

$$\frac{\langle \text{SELF}, S \rangle \rightarrow_{\text{self}} S1}{\langle \text{SELF}, S, \text{FFlag}, P \rangle \rightarrow_{\text{goal}} \langle \epsilon, S1, \text{FFlag} \rangle}$$

The $\rightarrow_{\text{self}}$ transitions are :

- 1) $\langle \langle \text{self ':' TERM} \rangle, S \rangle \rightarrow_{\text{self}} S1$
 where
 Self := get_self(S), which extracts the self stream from S
 and
 send_mesg(Self, Self1, TERM)
 and
 S1 := replace_self(Self1, S), which stores the new self stream in S
- 2) $\langle \langle \text{self ':' last} \rangle, S \rangle \rightarrow_{\text{self}} S$
 where
 close_stream(get_self(S)), which closes the self stream

The first transition outputs the message on the self stream and then stores the remaining part of the stream for future use. The second transition closes the self stream. Since no variable is involved on the left of the ':', no failure transition needs to be defined.

10.5.4.4 The Super Operation.

The BNF for SUPER in kernel Polka is :

$$\text{SUPER} ::= \text{ATOM ':' TERM} \quad | \quad \text{ATOM ':' last}$$

The transition for this is :

$$\frac{\langle \text{SUPER}, S \rangle \rightarrow_{\text{super}} S1 \quad | \quad \text{ff}}{\langle \text{SUPER}, S, \text{FFlag}, P \rangle \rightarrow_{\text{goal}} \langle \epsilon, S1, \text{FFlag} \rangle \quad | \quad \text{ff}}$$

The $\rightarrow_{\text{super}}$ transitions are structured in the same way as the ones for the send operation :

- 1) $\langle \langle \text{ATOM} '::' \text{TERM} \rangle, S \rangle \rightarrow_{\text{super}} S1$
 where
 $\text{ATOM} \in \text{names in inherits}(S)$
 and $\text{Inherit} := \text{get_inherits}(\text{ATOM}, S)$,
 which gets the output stream associated with ATOM from S
 and $\text{send_msg}(\text{Inherit}, \text{Inherit1}, \text{TERM})$
 and $S1 := \text{replace_inherits}(\text{ATOM}, \text{Inherit1}, S)$,
 which stores the new stream with the ATOM name in S

- 2) $\langle \langle \text{ATOM} '::' \text{last} \rangle, S \rangle \rightarrow_{\text{super}} S$
 where
 $\text{ATOM} \in \text{names in inherits}(S)$
 and $\text{close_stream}(\text{get_inherits}(\text{ATOM}, S))$,
 which closes the output stream associated with ATOM

- 3) $\langle \langle \text{ATOM} '::' \text{TERM} \rangle, S \rangle \rightarrow_{\text{super}} \text{ff}$
 where
 $\text{ATOM} \notin \text{names in inherits}(S)$

The first transition sends a message along the output stream associated with the class name ATOM, and then stores the new stream under that name for future use. The second transition closes the stream, while the third transition deals with the case when ATOM is not one of the inherited class names.

10.5.4.5 The I_am Operation.

The BNF for I_AM in kernel Polka is :

$$\text{I_AM} ::= \text{i_am} \langle \text{Parlog literal} \rangle$$

This translates into :

$$\langle \langle \text{i_am} \langle \text{Parlog literal} \rangle \rangle, S, \text{FFlag}, P \rangle \rightarrow_{\text{goal}} \langle \langle \text{Parlog literal} \rangle, S, \text{finished}, P \rangle$$

The idea behind this is that the Parlog literal is handled by the transitions for ordinary goals given in section 10.5.2, but the 'FFlag' is set to 'finished' to indicate that the object should now terminate.

10.5.5 The Code Section.

The code section transitions which are labelled as $\rightarrow_{\text{code}}$ at the top level are based on the following BNF :

```

CP   ::=      CP ';' CP           % CP consists of sequential-or clause groups
          | CPC

CPC  ::=      CC { CC }           % CPC consists of parallel-or clause groups

CC   ::=      HEAD '<-' GUARD ':' BODY      % CC is a clause

```

They produce the following 4 transitions which parallel those for $\rightarrow_{\text{message}}$ in section 10.5.3.

1. Choose a predicate from a sequential-or group :

$$\frac{\begin{array}{l} \langle G, CP1 \rangle \rightarrow_{\text{code}} \theta \mid \text{ff} \mid \{ G1, \dots Gn ; \theta \} \\ \langle G, CP1 ; CP2 \rangle \rightarrow_{\text{code}} \theta \mid \langle G, CP2 \rangle \mid \{ G1, \dots Gn ; \theta \} \end{array}}$$

2. Choose a predicate from a parallel-or group :

$$\frac{\begin{array}{l} \langle G, CPC \rangle \rightarrow_{\text{code}} \theta \mid \text{ff} \mid \{ G1, \dots Gn ; \theta \} \\ \langle G, CP \rangle \rightarrow_{\text{code}} \theta \mid \text{ff} \mid \{ G1, \dots Gn ; \theta \} \end{array}}$$

3. Successfully match with a clause :

and $CCi = \langle \text{HEAD} \leftarrow \text{GUARD} : G1, \dots Gm \rangle$
and let $\langle \text{HEAD}' \leftarrow \text{GUARD}' : G1', \dots Gm' \rangle$ be a fresh copy of the clause
and $\text{smatch}(G, \text{HEAD}') = \theta$
and $\theta(\text{GUARD}') = \text{true}$

$$\frac{}{\langle G, \{ CC1, \dots CCi, \dots CCn \} \rangle \rightarrow_{\text{code}} \theta \mid \{ G1', \dots Gm' ; \theta \}}$$

θ is returned if the body of the clause is the goal 'true'.

4. Fail to match with a clause :

	$\forall CCi \in \{ CC1, \dots CCn \}$
and	let $\langle \text{HEAD}' \leftarrow \text{GUARD}' : G1', \dots Gm' \rangle$ be a fresh copy of the clause
and	$\text{smatch}(G, \text{HEAD}') \text{ fails}$
or	$\theta(\text{GUARD}') = \text{ff}$
	$\frac{}{\langle G, \{ CC1, \dots CCi, \dots CCn \} \rangle \rightarrow_{\text{code}} \text{ff}}$

It is relatively straight forward to modify and extend the rules of section 10.5.4 so that they can deal with kernel Polka operations in code section predicates.

10.6. Term Classes.

The introduction of term classes leads to a further increase in the size of the tuple set and the complexity of the transitions, and so their addition will only be outlined. The operational semantics of a term object is very like an ordinary object but is some what simpler because there is no **code** section, inheritance or self communication.

The basic 'i' tuple must be extended to include an argument for the executing term objects :

$$\langle Nm, GS, S, F, P, ToS \rangle$$

ToS is a set of term object tuples, which have a similar structure to the old 'i' tuple, namely :

$$\langle Nm_t, GS_t, S_t, F_t, P_t \rangle$$

P_t will be simpler because of the language restrictions, but S_t will have one new argument for the 'Meta' output stream.

A term object tuple will be added to ToS whenever a 'odemo' goal is executed, and this neccesitates a change to the $\rightarrow_{\text{ex}}$ transitions of section 10.5.1.

At the 'i' tuple level, the **meta** operation is like the **self** operation, and consequently the transition for transmitting a **meta** message is like the transition in section 10.5.4.3 for the **self** operation.

Chapter 11. Implementation.

This chapter will discuss the implementation of kernel Polka by looking at how a class is translated into Parlog. The most important aspect of this is that a class is represented by three types of predicates : the 'top', 'o', and 'i' predicates, which separate out various implementation issues.

The 'top' predicate is called by the user and creates an instance of a class by invoking its 'o' predicate, along with the 'top' predicates for the inherited classes.

The 'o' predicate calls the 'i' predicate for the class, and also deals with the stream connections necessary for inheritance and self communication.

The 'i' predicate contains the clauses and the variables for a class, and is the place where messages are processed.

The result of these calls is to create a parallel conjunction of 'i' predicate goals which correspond to the new instance of the class. In addition, there will be a number of support processes for stream linking and holding suspendible messages, which communicate by partially instantiating shared variables. In essence, this is the process view of objects, although the details are somewhat different from previous approaches, such as the one put forward in [Shapiro and Takeuchi 1983].

The implementation bears many similarities with the semantics developed in chapter 11. The creation and linking operations for the 'i' tuple set can be seen in the actions of the 'top' and 'o' predicates, while an 'i' tuple is very like an 'i' predicate.

In the following discussion, the 'top' predicate will be considered first, followed by the 'o' predicate, and the 'i' predicate. Also, 'higher level' Parlog programs and term classes will be looked at briefly.

11.1. The 'top' Predicate.

The main task of the 'top' predicate is to group visible variables together and assign values to them. It also invokes the 'top' predicates for the inherited classes and the 'o' predicate for its own class.

Every class can be invoked in two ways : directly by the user, or by another object, and so the compilation of a class results in two 'top' predicates. For example, a 'filestore' class in kernel Polka is defined in outline as :

```

filestore  Ins istream, Error ostream
invisible  Fs state
clauses
          :
          :
end.
```

Since it inherits nothing, the following two 'top' predicates are produced :

```
mode filestore( ?, ^ ) .
filestore( Ins, Error ) <-
    o_filestore( [ Ins ], [ Fs ], [ Error ] ) .
```

and

```
mode filestore( ?, ?, ^ ) .
filestore( inherited, Ins, Error ) <-
    o_filestore( inherited, [ Ins ], [ Fs ], [ Error ] ) .
```

Program 11.1 : The two top 'filestore' predicates.

In the first predicate, 'Ins' is an input stream, 'Fs' is an invisible state variable, and 'Error' is an output stream. The 'top' predicate takes these variables and groups them together into input streams, states, and output streams when calling the 'o' predicate. Also, note that 'Fs' does not appear in the head of 'filestore' because it is an invisible variable.

In the second predicate, the extra argument is an 'inherited' flag which is used to call a different 'o' predicate.

One of the other deciding issues behind what a 'top' predicate looks like is whether the class inherits anything. When inheritance is used, the same sort of 'top' predicates are generated, but their bodies become more complicated. For instance, a 'foo' class in kernel Polka is defined in outline as :

```
foo
inherits orange, banana
    Ins istream
clauses
    :
    :
end.
```

The 'top' predicates generated for it are given as program 11.2.

```
mode foo( ?, ^ ) .
foo( Ins, Error ) <-
    o_foo( [ Ins ], [], [], [ OIorange, OIbanana ] ) ,
    orange( inherited, OIorange, Error_orange ) ,
    banana( inherited, OIbanana, Error_banana ) ,
    mergeL( [ Error_orange, Error_banana ], Error ) .
```

and

```
mode foo( ?, ?, ^ ) .
foo( inherited, Ins, Error ) <-
  o foo( inherited, [ Ins ], [], [], [ Olorange, Olbanana ] ),
  clock( inherited, Olorange, Error_orange ),
  random( inherited, Olbanana, Error_banana ),
  mergeL( [ Error_orange, Error_banana ], Error ) .
```

Program 11.2 : The two top 'foo' predicates.

In the body of the first predicate, the 'o' predicate for 'foo' is invoked along with the 'inherited' versions of the 'top' predicates for 'orange' and 'banana'. The three are linked via the fourth argument of the 'o' predicate, which is a list containing output streams for delegating messages from 'foo' to the 'orange' and 'banana' objects. It is also necessary to merge the error streams for each of the inherited objects, so that 'foo' outputs only a single 'Error' stream.

The second 'foo' predicate is not much more complicated since only the 'inherited' argument must be included in the head, and in the call to 'o_foo'.

11.2. The 'o' Predicate.

The 'o' predicate sets up the streams for inheritance and self communication, and also calls the 'i' predicate.

There are four types of 'o' predicate, which are distinguished by the different ways in which a class can be placed in an inheritance graph :

1. The class is at the bottom of the inheritance graph and inherits nothing.
2. The class is inside the inheritance graph and inherits nothing.
3. The class is at the bottom of the inheritance graph and inherits something.
4. The class is inside the inheritance graph and inherits something.

The **inherits** line specifies what a class inherits, and when this is read at compile time, it allows two of the four cases to be eliminated. However, the same class may be called directly, or be one of the inherited classes of another class. This means that every class will generate two 'o' predicates, which either represent cases 1 and 2, or cases 3 and 4. Each of these cases will now be considered.

11.2.1. Bottom / no Inheritance.

A class of this type is called directly and does not inherit any other classes. If such a class is called 'p', then the 'o' predicate generated for it will be :

```
mode o_p( ?, ?, ? ) .
o_p( Is, As, Os1 ) <-
  obj_input( Is, Is1, Self ) ,
  i_p( Is1, As, Os, [ _ ], Self ) ,
  wait_until( Os, Os1 ) .
```

Program 11.3 : A 'bottom/no inheritance' 'o_p' predicate

In the head, 'Is' is a list of the input streams, 'As' is a list of states, and 'Os1' is the list of output streams. In the body, 'Is1' is a list of streams leaving 'obj_input' and entering 'i_p' as its input. 'Self' is the output stream for self messages, which leave 'i_p' and enter 'obj_input'. The fourth argument of 'i_p' is a list which contains output streams for delegating messages from 'i_p'. In this case, since 'p' inherits nothing, the '_' output stream is unused.

This 'o' predicate illustrates how the 'i' predicate is invoked, and how its various inputs and outputs are connected to other processes by utilising 'obj_input' and 'wait_until'.

'wait_until' extracts 'wait' messages from the 'Os' output streams leaving the 'i' process and holds them until their delaying variables are bound. It then puts them back onto the output streams which leave 'o_p' in 'Os1'.

'obj_input' does a number of jobs; one of which is the insertion of messages from the 'Self' stream into the input of the 'i' process. However, the messages are first passed through a 'wait_until' process which holds any 'wait' messages. The messages which come out of this process are then added to the first stream in the 'Is' list which results in a new list of streams called 'Is1'. In addition, 'obj_input' monitors this first stream so that if the user closes it, then the corresponding stream in 'Is1' is also closed, even though it has messages being put onto it from the 'Self' stream.

Clear analogies can be seen between the operation of 'obj_input' and the 'sconnect' and 'input' predicates defined in the semantics chapter. 'sconnect' links the self streams of the inherited instances back into the bottom instance, while 'input' translates user input into a suitable internal form. In a similar vein, the 'wait_until' predicate used here is closely related to the 'wait_until' predicate mentioned in the last chapter.

11.2.2. Inside / no Inheritance.

A class of this type is inherited by another class but does not inherit anything itself. If the class is called 'p', the 'o' predicate for it will be :

```
mode o_p( ?, ?, ?, ? ) .
o_p( inherited, [ [ link( Self ) | Ins ] | Is ], As, Os1 ) <-
  i_p( [ Ins | Is ], As, Os, [ _ ], Self ) ,
  wait_until( Os, Os1 ) .
```

Program 11.4 : An 'inside/no inheritance' 'o_p' predicate.

Most of the differences between this predicate and the one produced for case 1 are due to the need to send 'Self' messages back to the object at the bottom of the inheritance tree. The predicate has one extra argument in the head which acts as a flag to indicate that the corresponding class is inherited by another. Also, the first message on the first stream in the input stream list is always :

```
link( Self )
```

This message contains an input stream linked to the object at the bottom of the tree. The self output stream is unified with this stream, in order to guarantee that all self messages go back to the bottom object. This means that an 'obj_input' call is not required since self messages no longer need to be considered at this level.

11.2.3. Bottom / Inheritance.

A class of this type is called directly and also inherits other classes. If the class is called 'p', then the 'o' predicate generated for it will be :

```
mode o_p( ?, ?, ?, ? ) .
o_p( Is, As, Os1, ToInhs ) <-
  obj_input( Is, Is1, Self ) ,
  i_p( Is1, As, Os, Inhs, Self1 ) ,
  send_inherits( Inhs, Self2, ToInhs ) ,
  merge( Self1, Self2, Self ) ,
  wait_until( Os, Os1 ) .
```

Program 11.5 : A 'bottom/inheritance' 'o_p' predicate.

The head of the predicate contains a new type of argument, called 'ToInhs', which contains a list of output streams, each of which will go to an inherited object. One useful feature of the list is that the order of its streams is the same as the order of the names in 'p's inherits line. Also 'i_p' now has an 'Inhs' variable as its fifth argument, which is a list of output streams which should go to the inherited objects. Both 'Inhs' and 'ToInhs' are passed to 'send_inherits' which connects each stream in

'Inhs' to the corresponding stream in 'ToInhs', although this is only done after a 'link' message has been sent along each 'ToInhs' stream. The 'Self' streams used in each 'link' are merged together and the resulting stream is output as 'Self2' back to the 'o' predicate. This is merged with the self stream coming out of the 'i' process, and the result is passed to 'obj_input'. Thus, the 'Self' stream for every inherited object, as well as 'p' itself, are merged and sent back into 'p' via the 'obj_input' process.

'send_inherits' also generates a number of wait_until processes which hold 'wait' messages until they are ready to be sent to their inherited object destinations.

'send_inherits' plays much the same role as the 'iconnect' predicate defined in the semantics chapter. The aim of 'iconnect' is to link the output streams for delegated messages to the input of the relevant class instances.

11.2.4. Inside / Inheritance.

A class of this type is inherited by another class and also inherits classes of its own. This means that its 'o' predicate will be :

```
mode o_p( ?, ?, ?, ?, ? ).
o_p( inherited, [ [ link( Self ) | Ins ] | Is ], As, Os1, ToInhs ) <-
  i_p( [ Ins | Is ], As, Os, Inhs, Self1 ),
  send_inherits( Inhs, Self2, ToInhs ),
  merge( Self1, Self2, Self ),
  wait_until( Os, Os1 ) .
```

Program 11.6 : An 'inside/inheritance' 'o_p' predicate.

As in section 11.2.2, the predicate has an 'inherited' flag argument in its head. Where it differs from section 11.2.2, is that it has a 'ToInhs' argument for the output streams to the inherited objects, and 'i_p' and 'send_inherits' are used in the same way as in section 11.2.3. This allows messages to be sent to the inherited objects, and all the returned self messages are combined into a single 'Self' stream. However, this is unified with the stream in the 'link' message and so is linked to the object at the bottom of the inheritance tree, rather than to 'p'.

11.3. The Initial Section.

An **initial** section is compiled into a predicate which has a named derived from appending 'i_init_' to the class name. For instance, an **initial** section for the 'filestore' example might be :

```
initial    initialise( Values ),
           Fs becomes Values .
```

The 'i_init' predicate for this is given in program 11.7.

```
mode i_init_filestore( ?, ?, ?, ?, ^ ).
i_init_filestore( Is, [ Fs ], Outs, Inhs, Self ) <-
    filestore_initialise( Values ),
    i_filestore( Is, [ Values ], Outs, Inhs, Self ) .
```

Program 11.7 : The resulting 'i_init_filestore' predicate.

After carrying out the actions of the **initial** section, the 'i_init' predicate calls the 'i' predicate. Kernel Polka operations are translated in the same way as in the 'i' predicate, and so a discussion of these will be deferred until the next section. However, the example above shows how **becomes** is replaced by a call to 'i_filestore' with the 'Fs' argument replaced by 'Values'.

If a class contains an **initial** section, then the 'o' predicates described in section 11.2 will call the 'i_init' predicate rather than the 'i' predicate. However, this does not textually alter their definitions by very much, because the arguments of the two kinds of predicate are the same.

11.4. The 'i' Predicate.

The compilation of a class results in a single 'i' predicate whose name is generated by appending 'i_' to the class name. For instance, the compilation of the kernel Polka 'filestore' class outlined in program 11.8 will lead to the production of an 'i_filestore' predicate. Nearly all 'i' predicates have five arguments, and are tail recursive because of their basis on the process view of objects. Also, there will be a 'i' predicate clause for each clause in the class.

```

filestore  Ins istream, Error ostream
invisible  Fs state

clauses
  Ins :: add( Key, Val, Reply )    =>
        insert( Key, Val, Fs, Reply, NewFs ) ,      % update Fs
        Fs becomes NewFs .

        :      % several more clauses
        :

  Ins :: last      =>      Error :: last ;           % terminate

  Ins :: Message   =>      Error :: Message .        % output unrecognised
                                                         % message

end.

```

Program 11.8 : A 'filestore' class.

The resulting 'i' predicate is shown as program 11.9.

```

mode i_filestore( ?, ?, ?, ^ ).
i_filestore( [[ add( Key, Val, Reply ) | Ins ]], [ Fs ], Outs, Inhs, Self ) <-
  filestore_insert( Key, Val, Fs, Reply, NewFs ) ,      % process an input message
  i_filestore( [ Ins ], [ NewFs ], Outs, Inhs, Self ) . % and change state

:      % one 'i' clause for each clause in the class
:

i_filestore( [[ ]], As, [ Error ], Inhs, [] ) <-          % output a message
  Error = [] ;                                           % and terminate

i_filestore( [[ Message | Ins ]], As, [ Error ], Inhs, Self ) <-
  Error = [ Message | Error1 ] ,                        % more complex output
  i_filestore( [[ Ins ]], As, [ Error1 ], Inhs, Self ) .

```

Program 11.9 : The resulting 'i_filestore' predicate.

The first argument of the predicate is a list of input streams used by the object, the second is a list of state and ostate variables, and the third is a list of output streams. The fourth is a list of output streams going to the inherited objects, and the fifth is an output stream for **self** messages.

The structure of the arguments for an 'i' predicate is very similar to the state argument of the 'i' tuple discussed in the semantics chapter, which is :

< OVars, Inherits, Self, Becomes >

'OVars' contains information about the global variables including their name, type and value. 'Inherits' is a list of the names of the inherited classes, 'Self' contains the self communication output stream, and 'Becomes' is a list for storing **becomes**

operations. The compiler removes the need for a 'Becomes' list, as discussed in section 11.4.3, but 'OVars', 'Inherits', and 'Self' clearly map onto the five arguments of the 'i' predicate.

The actual details of the translation will be considered in the following sections which will refer back to the clauses in the 'i_filestore' predicate in program 11.9.

11.4.1. Variables.

The variable declarations in a class lead to the creation of the first three arguments of the 'i' predicate which are lists of variables subdivided by their types : input stream, state (both input and output), and output stream.

The various variables are assigned initial values when the 'i' predicate is invoked by the 'o' predicate.

11.4.2. Input Messages.

Generally, an input message is compiled into a partial instantiation of its stream variable in the 'i' predicate head. For example, the message :

```
Ins :: hello( X ) => ...
```

is translated into :

```
[ hello( X ) | Ins ]
```

Two examples of this translation method can be seen in the 'filestore' example, in the first and last clauses.

The multiple message :

```
Control :: ( m1, m2 ) => ...
```

becomes :

```
[ m1, m2 | Control ]
```

The main exception to this translation technique is the **null** keyword which leaves all of the input stream variables unchanged in the 'i' predicate head.

The **last** keyword :

```
Ins :: last => ...
```

is converted into :

```
[ ]
```

in the 'Ins' position in the predicate head. An example of this is shown in the penultimate clause of the 'filestore' example.

The last keyword also terminates the object, and this is achieved in the 'i' predicate by simply not recursing.

11.4.3. Separators.

There are four clause separators in Polka : '.', ';', '&.', and '&.'. The first two result in 'i' predicate clauses which are separated by '.' and ';' as expected. The other two extend this idea since they force the Polka clause to have succeeded before another message can be processed. In Parlog terms, this means creating a clause which does not recursively call the 'i' predicate until its body has succeeded. This is achieved by placing a '&' conjunction in the body. For example, the clause in a class 'bar' :

```
Ins :: msg => p1, p2 &;
```

becomes the Parlog clause :

```
i_bar( [[ msg | Ins ], As, Os, Inhs, Self ) <-
  ( p1 , p2 ) &                                % sequential-and
  i_bar( [[ Ins ]], As, Os, Inhs, Self );        % sequential-or
```

This forces 'p1' and 'p2' to succeed before 'i_bar' recurses, and so starts processing another message.

11.4.4. Becomes.

The BNF syntax for the **becomes** operation in kernel Polka is :

```
BECOMES ::= VAR becomes TERM
```

Essentially, each **becomes** operation is translated into an argument replacement in the recursive call of the 'i' predicate. For instance, if the following :

```
X becomes 2
```

occurs, then it will mean that the 'X' argument position in the corresponding 'i' clause will have the value 2 when it recurses. It is for this reason that updates due to **becomes** do not take effect until the end of a clause. Another example of this can be seen in the first clause of the 'i_filestore' example (program 11.9) at the start of

section 11.4. The 'NewFs' value replaces the old 'Fs' value in the recursive call to 'i_filestore'.

11.4.5. Sends.

The syntax for this type of operation in kernel Polka is :

```
SENDS ::= VAR ':' TERM | VAR ':' last
```

The basic technique for dealing with sends is very simple : a message is converted into a partial instantiation of the output stream stored in the 'i' process head. After all the messages have been sent, the tail of the stream is stored in the same argument position in the recursive call to the 'i' predicate. However, complications arise out of the need to deal with message sends within predicates, and conjunctions of sends. The first problem is discussed in the **code** section part of this chapter, but in essence it means that an output stream must sometimes be passed to a predicate so that the predicate can transmit messages. Conjunctions of sends cause difficulty because of the way that two (or more) send operations are related in their Parlog translation. For instance, the following two sends :

```
Out :: a & Out :: b
```

can be translated into the goals :

```
Out = [ a | Out1 ] & Out1 = [ b | Out2 ]
```

and 'Out2' will be stored as the new stream variable. In fact, kernel Polka rules out the sequential conjunction of messages, but this example is included to illustrate the more efficient compilation possible when '&' is used instead of ','.

If the two sends are separated by a parallel conjunction :

```
Out :: a , Out :: b
```

Then the translation is :

```
merge( Out1, Out2, Out ) ,  
Out1 = [ a | Out3 ] , Out2 = [ b ]
```

and 'Out3' will become the new stream variable. This translation tries to mimic the non-deterministic nature of the parallel conjunction of the two sends.

A special case arises out of the use of **last**. For instance :

```
Out :: last
```

is translated into :

```
Out = [ ]
```

A suspendible messages in Polka might be :

```
Out on Flag :: hello
```

In kernel Polka, this is represented as :

```
Out :: '$wait'( [ Flag ], hello )
```

As expected, this is translated into the following goal :

```
Out = [ '$wait'( [ Flag ], hello ) | Out1 ]
```

A message of this type is held in the object until its delaying variable is bound. This is achieved by passing the output stream through a 'wait_until' process which captures and holds '\$wait' messages. The 'wait_until' process is created at the 'o' predicate level, and is invisible to the 'i' process.

11.4.6. Inheritance.

The syntax for the super operation in kernel Polka is :

```
SUPER ::= ATOM ':' TERM | ATOM ':' last
```

Before any super operations are parsed, the **inherits** line will have been read and made into a list. This list is used to create the 'Inhs' argument of the 'i' predicate, which contains an output stream variable for each inherited object. For example, the line :

```
inherits circle, frame
```

would lead to an 'Inhs' list of the form :

```
[ Olcircle, Olframe ]
```

This allows super operations to be translated into partial instantiations of these streams, in the same way as send operations. For instance :

```
circle :: perimeter( Radius, Ans )
```

would become the goal :

```
Olcircle = [ perimeter( Radius, Ans ) | Olcircle1 ]
```

The new 'OIcircle1' variable is stored in the 'Inhs' list used in the recursive call of the 'i' process.

Also, `circle :: last`
 becomes `OIcircle = [ ]`

11.4.7. Self Communication.

The syntax for the self operation in kernel Polka is :

```
SELF ::= self ':' TERM | self ':' last
```

The 'Self' variable of the 'i' predicate is used as the output stream for messages, which allows an operation like :

```
self :: hello
```

to be rewritten as :

```
Self = [ hello | Self1 ]
```

'Self1' is then stored in place of 'Self' in the recursive call to the 'i' predicate.

When last is used, the 'Self' variable is set to [].

11.4.8. I_am.

The compilation of a literal with an **i_am** prefix is done in the same way as for an ordinary literal, but the resulting clause will not be recursive.

11.4.9. The Code Section.

The local scoping of **code** section predicates is achieved by renaming the predicates so that they can only be invoked by the 'i' predicate for the class. Also, if a **code** section predicate uses **send**, **self** or **super** operations, then extra stream arguments are added to it. Program 11.10 contains an example which illustrates these transformations :

```

foo Ins istream, Out ostream
clauses
  Ins :: double( Msg ) => dub( Msg ) .           % call 'dub'

code
  mode dub( ? ) .                                % 'dub' calls 'single' twice
  dub( Msg ) <-
    single( Msg ) & single( Msg ) .

  mode single( ? ) .                             % 'single' outputs 1 message
  single( Msg ) <-
    Out :: Msg .

end.

```

Program 11.10 : Augmented predicates in a 'foo' class.

Kernel Polka does not permit sequential conjunction of body goals, and so the 'dub' predicate in program 11.10 would not be generated by the compiler. However, the rule is relaxed here in order to show the different possible compilation strategies for augmented code section predicates.

The 'dub' and 'single' predicates can be rewritten as program 11.11.

```

mode foo_dub( ?, ^, ? ) .                        % 'dub' is renamed
foo_dub( Msg, Out2, Out ) <-                     % pass in the 'Out' stream
  foo_single( Msg, Out2, Out1 ) &
  foo_single( Msg, Out1, Out ) .

and

mode foo_single( ?, ^, ? ) .                     % 'single' is renamed
foo_single( Msg, Out1, Out ) <-                 % use the 'Out' stream
  Out1 = [ Msg | Out ] .

```

Program 11.11 : Rewritten 'dub' and 'single' predicates.

The compiled clause in the class is :

```

i_foo( [[ double( Msg ) | Ins ]], As, [ Out1 ], Inhs, Self ) <-
  foo_dub( Msg, Out1, Out ) ,                    % call the renamed 'dub'
  i_foo( [ Ins ], As, [ Out ], Inhs, Self ) .

```

The main consequence of this approach is that the **code** section must be parsed after the variable declarations of the class, but before the **initial** section and the clauses of the class. The reason for this is that the **code** section predicates must be analysed to find the output streams they directly or indirectly use. Then each predicate must be changed to include two extra arguments for each stream : one input, and one output argument. When this is done, the predicate calls in the clauses of the class can be rewritten to have the right name and number of arguments.

If 'dub' conjoins the two invocations of 'single' with a ',' :

```
mode dub( ? ) .
dub( Msg ) <-
    single( Msg ) ,           % parallel-and conjunction
    single( Msg ) .
```

Program 11.12 : And- parallel 'dub' predicate.

Then it would be rewritten as program 11.13.

```
mode foo_dub( ?, ^, ? ) .
foo_dub( Msg, Out3, Out ) <-
    merge( Out1, Out2, Out3 ) ,      % use merge on the two outputs
    foo_single( Msg, Out2, [] ) ,    % first output
    foo_single( Msg, Out1, Out ) .   % second output
```

Program 11.13 : Rewritten and- parallel 'dub' predicate.

This translation more accurately reflects the non-deterministic order of the transmission of messages by the two 'single' calls.

A similar idea allows messages to be sent to **self** and to named classes, by rewriting the messages as instantiations of the 'Self' variable, and the streams stored in 'Inhs'.

11.5. Higher Level Parlog.

The implementation of the higher level Parlog part of Polka is achieved by setting a flag in the compiler when the **parlog** keyword is read. The flag disallows certain constructs including any form of message related to inheritance or self communication.

The resulting code for an 'i' predicate is much the same, but the head and recursive call are simplified since arguments for inheritance, and self communication do not need to be added.

An example of the simplification can be seen when 'merge' is compiled. The class is given in program 11.14.

```

parlog merge X, Y istream, Z ostream
clauses
  X :: Mesg    =>    Z :: Mesg .

  Y :: Mesg    =>    Z :: Mesg .

  X :: last    =>    Y = Z .

  Y :: last    =>    X = Z .
end.

```

Program 11.14 : A 'merge' class.

The 'i' predicate produced by its compilation is shown as program 11.15.

```

mode i_merge(?,?,?).
i_merge( [[ Mesg | X ], Y ], As, [ Z ] ) <-
  Z = [ Mesg | Z1 ],
  i_merge( [ X, Y ], As, [ Z1 ] ).

i_merge( [ X, [ Mesg | Y ] ], As, [ Z ] ) <-
  Z = [ Mesg | Z1 ],
  i_merge( [ X, Y ], As, [ Z1 ] ).

i_merge( [ [], Y ], As, [ Z ] ) <-
  Y = Z .

i_merge( [ X, [] ], As, [ Z ] ) <-
  X = Z .

```

Program 11.15 : The resulting 'i_merge' predicate.

Input streams, output streams, and states are still grouped together, but partial evaluation techniques exist which can remove the superfluous argument groupings.

Since this predicate is simpler than the typical 'i' predicate, no 'o' predicate is needed, and the 'top' predicate can invoke it directly, like so :

```

mode merge(?, ?, ^).
merge( X, Y, Z ) <-
  i_merge( [ X, Y ], [], [ Z ] ).

```

Program 11.16 : The top 'merge' predicate.

11.6. Term Classes.

Term classes are implemented by being compiled into Parlog terms, which can then be manipulated at run time by various built-in meta level predicates.

Since the syntax of a term class is so close to that of an ordinary class, its compiler is very similar in structure to that for a class. However, the class compiler produces an internal form which is then converted into Parlog predicates, whereas a term class is executed by using an meta interpreter predicate, called 'odemo'. This means that code generation need not be carried out for a term class, since the internal form produced by the compiler is utilised.

Part 4. Conclusions.

Chapter 12. Conclusion.

12.1. Design Aims.

The three main design aims behind Polka are :

- the combination of OOP and concurrent LP
- support for 'higher level' Parlog
- support for meta level programming

When the OOP and concurrent LP paradigms are combined, the result is more than just a sum of their parts because of the way that inadequacies in one paradigm are compensated for by strengths in the other. For example, features from concurrent LP allow the development of powerful forms of object composition, concurrency, and stream manipulation, which are commonly weak points in traditional OOP languages. Conversely, OOP replaces the rather ad hoc module mechanisms proposed for LP by the object, which is more suitable for 'programming in the large'.

Many Parlog programs describe systems of communications processes, and abstractions in Polka allow these to be coded in a much more intuitive manner. In this situation, Polka is being used as a source of syntactic conventions for representing the more complex Parlog notation, and so is acting as a 'higher level' Parlog. The practical benefit of this are code size reduction, less errors, and an increase in understandability, because Polka directly supports the paradigm which underpins these programs.

The meta level programming component of the language demonstrates how a familiar idea from LP can be profitably applied to OOP. In programming terms, this enables term classes to be manipulated inside objects and predicates, and to be treated as messages. Many other uses are outlined in chapter 6.

12.2. A Measure of Success.

Section 1.1 listed eleven attributes which could be used as a measure of the success of a programming language : each of these will be briefly examined with respect to Polka.

1. A well defined syntactic and semantic description.

The syntax for Polka is defined by the BNF productions in the appendix, while an operational semantics based on Plotkin's state transition approach is put

forward in chapter 10. However, a denotational semantics for Polka has still to be developed.

2. Orthogonality.

This is the property of a language whereby its features are separately understandable and free from interaction when combined. This attribute enables Polka to be used either as a 'higher level' Parlog, or as an OOP language.

3. Machine Independence.

Polka is independent of any machine, and as demonstrated by the graphics example in chapter 7, machine dependent applications can be made more portable by utilising the class mechanism.

4. Generality.

This property means that the features of a language should be based on a few basic concepts. This is the case with Polka, although the concepts underlying the language come from concurrent LP and OOP. For example, concurrent LP supplies Polka's model of concurrency and its pattern matching mechanism, while OOP is the motivation behind the class structure, message passing and inheritance.

5. Uniformity.

Uniformity relates to the idea that similar things should have similar meanings. This is adhered to in Polka, as can be seen with message passing which uses the '::' operator for output streams, inherited objects and self communication. The major failure of this property shows itself in the inconsistencies between ordinary classes and term classes.

6. Extensibility.

Extensibility is one of the key attributes of OOP, and in Polka it is straight forward to add new data abstractions and operations in the form of new classes. This can be taken further by extending the functionality of term classes in ways similar to those outlined in section 6.6.

7. Ability to use language subsets.

This is an important property of Polka, and helps to ease the learning process. Thus, naive users can be unaware of the term class mechanism, and some of the more complex parts of the inheritance and self communication mechanism.

However, useful programs, such as the set of graphics classes in chapter 7, can still be written with this subset.

8. Consistency with commonly used notation.

The LP notation in Polka is based closely on Parlog, although it has been extended in several places. An example of this are the two additional clause separators '&.' and '&:'. The notation for OOP is not related to any particular language, although **self**, and **super** are based on operations in Smalltalk.

9. Reliability.

Polka is a research language, and so the more experimental parts do not have totally robust implementations, although all of the features described in this thesis are reliable.

10. Fast translation.

The Polka compiler is written in Parlog, and could be speeded up by increasing the number of parallel operators it uses. A Polka program goes through two main compilation phases - firstly from Polka to Parlog, and then to machine code. An obvious improvement of this is to combine the actions of these two phases, and this may be done during the next stage of development.

11. Efficient object code.

The measure of efficiency used at the moment is how close the Parlog code generated by the Polka compiler is to code written directly in Parlog. This is difficult to measure objectively, but a class which utilises the **parlog** keyword compiles into a predicate which is almost identical to one that a Parlog programmer would write for the same task.

12.3. Future Research.

In future work, it may be useful to borrow an idea from the A'UM language [Yoshida and Chikayama 1987] and annotate the message variables which are intended as streams or states. This would permit these variables to be used safely in Polka operations, because compile time type checking could be carried out.

Plans are beginning on a run time debugger for Polka, and a browsing tool for class hierarchies would also be a useful addition to the environment.

The new Parlog system will support Polka classes a little more directly. For instance, it will include a low level 'write' predicate which places a term onto a stream with a constant time delay, in a similar manner to constructs proposed in [Shapiro and Safra 1986]. This primitive could be used as the compiled result of a Polka message send, rather than the present partially instantiated list. Not only would the resulting Parlog code be simpler, but also more efficient. Also under investigation is the use of arrays to store object variables, which may speed up their access and manipulation. Utilising destructive assignment to change values may also be of use, especially for reducing the amount of garbage collection required during object execution.

Some of the ideas from Pandora [Bahgat and Gregory 1988] about combining Parlog and Prolog might be utilised by Polka for introducing don't know non-determinism into objects. In particular, it may be helpful for searching an inheritance tree for a suitable clause, and for ignoring a message if no clause is found.

Pandora uses a deadlock detection mechanism which enables deadlocked goals to be manipulated at run time. In OOP terms, this might mean that suspended objects could be rewritten as a single object, by executing a clause like :

```
taxi( In1 ), customer( In2 ) <- merge( In1, In2, In ), busy_taxi( In )
```

Operationally, this replaces the suspended 'taxi' and 'customer' objects by a 'busy_taxi' object with input generated from the old objects. This type of clause is reminiscent of ones found in some of the Prolog-based OOP languages discussed in chapter 3 [Monteiro 1984; Conery 1987], and the techniques they employ may be applicable.

The use of Open Path Expressions (OPEs) for dealing with the separation of synchronisation and functionality in classes is of considerable interest [Campbell and Habermann 1974]. Recent OPE specification languages have a number of things in common with concurrent OOP [Tam 1988], which suggests that they might easily be simulated in Polka. In addition, Polka's LP background may throw light on ways to improve the specification languages.

One of the most important future design goals for Polka is support for distributed programming. OOP is a very suitable abstraction for distributed programming, since it naturally contains computationally independent entities which communicate by message passing. However, extensions are required for dealing with such things as the explicit mapping of objects to processors, load balancing, and recovery from processor failure. Fortunately, the utilisation of past work on the distribution of concurrent LP, along with term classes, make these elements relatively simple to incorporate into the language [Davison 1989b].

Appendix. BNF Description of Polka.

The BNF used here is very close to the traditional notation [Horowitz 1984], but its operations will be defined below for the sake of clarity.

Capitalised words are non-terminals, while bold lowercase words, or symbols in quotes, are terminals.

$X ::= Y Z$ means X parses as Y and Z .

$[X]$ states that X is optional.

$\{ X \}$ permits X to occur 0 or more times.

$X | Y$ means either X or Y can occur. When a choice can be made between complex descriptions, each will be placed on a separate line.

(X) is equivalent to X , but brackets help to make the scope of the operators inside X more explicit.

$\%$ indicates the start of a comment.

The BNF is subdivided into three parts : the BNF for a class, the term class BNF, and the code section BNF.

Some short cuts have been taken in defining certain rules, as can be seen for **TERM**, which is :

$$\text{TERM} ::= \text{Parlog term}$$

This states that **TERM** is equivalent to a Parlog term, such as $[]$, which seems preferable to specifying a set of rules which parallel those for Parlog. A BNF for Parlog can be found in [Conlon 1989].

The **code** section of a class allows the definition of Parlog predicates but, as can be seen from its BNF description, the language is slightly different from Parlog. For instance, output streams can be manipulated in a predicate. Restrictions include the absence of neutral operators, and brackets around conjunctions in clauses, or around the clauses of a predicate.

The most important simplification of the BNF is the absence of the $\#$ notation. This considerably reduces the size of the syntax since Polka literals and terms become equivalent to Parlog literals and terms. As explained in chapter 9, the $\#$ notation can always be replaced by the **becomes** operator.

The only other simplification is related to the **parlog** flag, which disables the OOP features of the language. However, the corresponding changes to the BNF are not reflected here, because it was felt that this would over complicate it. The flag

causes restrictions in two areas : inheritance and self communications, which means that the **inherits** line is disallowed, and output messages to **self**, **super**, **all**, and inherited classes are not permitted.

1. Class BNF.

CLASS	::=	[parlog] ATOM [inherits ATOM { ',' ATOM }] CLASS_BODY [TERM_CLASSES] [CODE_SECTION] end '
CLASS_BODY	::=	VDEFINITION [initial ACTIONS ['&'] ' '] clauses CDEFINITION ['&'] ' '
VDEFINITION	::=	[[visible] VDEF] [invisible VDEF]
VDEF	::=	VARDEF { ' ' VARDEF }
VARDEF	::=	VAR TYPE ['<=' TERM] VAR { ' ' VAR } TYPE
TYPE	::=	istream ostream state ostate
CDEFINITION	::=	LINE { SEP LINE }
SEP	::=	['&'] (' ' ' ')
LINE	::=	INPUTS '=>' ACTIONS
INPUTS	::=	null RECEIPT { ' ' RECEIPT }
RECEIPT	::=	[VAR '::'] MESSAGES

MESSAGES	::=	MESSAGE '(' MESSAGE { ',' MESSAGE } ')'
MESSAGE	::=	last TERM
ACTIONS	::=	[ACTS ':'] ACTS
ACTS	::=	PRED { CONJ PRED }
CONJ	::=	'&' ','
PRED	::=	BECOMES SEND SELF SUPER I_AM Parlog literal
BECOMES	::=	VAR { ',' VAR } becomes B_TERM { ',' B_TERM }
B_TERM	::=	TERM Parlog arithmetic expression
SEND	::=	VAR [on VARS] '::' MESSAGES
VARS	::=	VAR { ',' VAR }
SELF	::=	self [on VARS] '::' MESSAGES
SUPER	::=	ATOM [on VARS] '::' MESSAGES super [on VARS] '::' MESSAGES all [on VARS] '::' MESSAGES [RETURN]
RETURN	::=	returning VAR returning '(' VAR { ',' VAR } ')'
I_AM	::=	i_am Parlog literal
ATOM	::=	Parlog identifier

VAR ::= Parlog variable

TERM ::= Parlog term

2. Term Class BNF.

TERM_CLASSES ::= **termClass** **TERM_CLS** { **TERM_CLS** }

TERM_CLS ::= **ATOM**
VDEFINITION
[**initial** **TO_ACTIONS** [**'&'**] **'.'**]
clauses **TO_CDEFINITION** [**'&'**] **'.'**
end

TO_CDEFINITION ::= **TO_LINE** { **SEP** **TO_LINE** }

TO_LINE ::= **INPUTS** **'=>'** **TO_ACTIONS**

TO_ACTIONS ::= [**TO_ACTS** **'.'**] **TO_ACTS**

TO_ACTS ::= **TO_PRED** { **CONJ** **TO_PRED** }

TO_PRED ::= **TO_SEND** | **BECOMES** | **I_AM** |
Parlog literal

TO_SEND ::= **meta** [**on** **VARS**] :: **MESSAGES** |
VAR [**on** **VARS**] :: **MESSAGES**

3. Code Section BNF.

CODE_SECTION	::=	code CS_PREDICATE { CS_PREDICATE }
CS_PREDICATE	::=	MODE CS_CLAUSE { CS_SEP CS_CLAUSE } '.'
MODE	::=	mode ATOM ['(' MODE_ARG { ',' MODE_ARG } ')'] '.'
MODE_ARG	::=	'?' '^'
CS_SEP	::=	'.' ';' ':'
CS_CLAUSE	::=	Parlog literal ['<-' [CS_ACTS ':'] CS_ACTS]
CS_ACTS	::=	CS_PRED { CONJ CS_PRED }
CS_PRED	::=	SEND SELF SUPER Parlog literal

References.

- Agha, G. 1985. "Semantic Considerations in the Actor Paradigm of Concurrent Computation", In *Seminar on Concurrency*, (eds.) Brookes, S.D., Roscoe, A.W., and Winskel, G., LNCS 197, Springer-Verlag, pp. 151-179.
- Agha, G. 1986. *Actors : A model of Concurrent Computation in distributed systems*, The MIT Press, Cambridge, MA.
- Aît-kaci, H., and Nasr, R. 1986. "LOGIN : A Logic Programming Language with built-in Inheritance", *Journal of Logic Programming*, 3 (1986), pp. 185-215.
- America, P., De Bakker, J., Kok, J.N., and Rutten, J. 1986. "Operational Semantics of a Parallel Object-Oriented Language", In *Conf. Record of the 13th ACM Symp. on Principles of Programming Languages*, pp.194-208.
- Anjewierden, A. 1986. "How about a Prolog Object ?", *Journées Langages Orientés Objet*, Vol. 1, No. 3, pp.167-176.
- Atkinson, R., and Hewitt, C.E. 1979. "Specification and Proof Techniques for Serializers", *IEEE Trans. on Software Engineering*, SE-5, No.1.
- Attardi, G., and Simi, M. 1981. "Semantics of inheritance and attributions in the description system Omega", In *Proc. IJCAI 81*, Vancouver, Canada, August.
- Bahgat, R., and Gregory, S. 1988. "PANDORA : Non-deterministic Parallel Logic Programming", Dept. of Computing, Imperial College, London, November.
- Baker, H. 1987. "Actor Systems for Real-Time Computation", Technical Report 197, MIT Laboratory for Computer Science.
- Bancilhon, F. 1986. "A Logic-Programming / Object-Oriented Cocktail", *SIGMOD RECORD*, Vol. 14, No. 3, pp. 11-21.
- Bartual, R. 1989. "LPA Prolog and Flex Expert", *Program Now*, Vol. 3, No. 2, February, pp.43-47.
- Ben-Ari, M. 1982. *Principles of Concurrent Programming*, Prentice-Hall, Englewood Cliffs, NJ.
- Birtwistle, G.M. 1979. *DEMOS : A system for discrete-event modelling on SIMULA*, The MacMillan Press Ltd.
- Birtwistle, G.M., Dahl, O-J., Myhrhaug, B., and Nygaard, K. 1973. *SIMULA BEGIN*, Petrocelli / Charter, New York.
- Bishop, J.M. 1986. *Data Abstraction in programming languages*, Addison Wesley, Reading, MA.
- Black, D., and Manley, J. 1987. "A Logic-based Architecture for Knowledge Management", In *Proc. of 10th IJCAI*, Vol. 1, Milan, Italy, pp. 87-90.

- de Boer, F.S., Kok, J.N., Palamidessi, C., and Rutten, J.J.M.M. 1989. "Semantic Models for a version of Parlog", In *Logic Programming Proceedings of the 6th Int. Conf.*, (eds.) Levi, G., and Mortelli, M., MIT Press, pp.621-636.
- Bowen, K.A. 1985. "Metalevel Programming and Knowledge Representation", *New Generation Computing* 3 (1985), pp. 359-383.
- Bowen, K.A., and Kowalski, R.A. 1982. "Amalgamating language and metalanguage in logic programming", In *Logic Programming*, (eds.) Clark, K.L., and Tarnlund, S.A., Academic Press, New York, pp. 153-172.
- Brachman, R.J., and Schmolze, J.G. 1985. "An overview of the KL-ONE knowledge Representation system", *Cognitive Science* (9), pp.171 - 216.
- Broda, K., and Gregory, S. 1984. "Parlog for discrete-event simulation", Research Report DOC 84 / 5, Imperial College, London.
- Campbell, R.H., and Habermann, A.N. 1974. "The specification of process synchronization by path expressions", In *LNCS 16*, Springer-Verlag, pp. 89-102.
- Cardelli, L., and Wegner, P. 1985. "On understanding types, data abstraction and polymorphism", *Computing Surveys*, Vol. 17, No. 4, pp.471-522.
- Chen, W., and Warren, D.S. 1988. "Objects as Intensions", In *Logic Programming : Proc. 5th Int. Conf. and Symposium*, (eds.) Kowalski, R.A., and Bowen, K.A., Seattle, Washington, Sponsored by ALP and IEEE, pp. 404-419.
- Chernicoff, S. 1987. *Macintosh revealed, Volume 1 : Unlocking the Toolbox* and *Macintosh revealed, Volume 2 : Programming with the Toolbox*, Hayden.
- Chikayama, T. 1984. "ESP Reference Manual", ICOT Technical Report : TR-044, Tokyo, Japan.
- Chusho, T., and Haga, H. 1986. "A multilingual modular programming system for describing knowledge information processing systems", In *Information Processing 86*, (ed.) Kugler, H.-J., North Holland, pp.903-908.
- Clark, K.L., and Foster, I.T. 1987. "A Declarative Environment for Concurrent Logic Programming", Presented at *TAPSOFT '87*.
- Clark, K.L., and Gregory, S. 1984. "Notes on Systems Programming in Parlog", Presented at the *Int. Conf on Fifth Generation Computer Systems*, Tokyo.
- Clark, K.L., and Gregory, S. 1986. "PARLOG: Parallel Programming in Logic", *ACM Trans. on Programming Languages and Systems*, 8 (1), pp. 1-49.
- Clark, K.L., and Gregory, S. 1987. "PARLOG and Prolog United", Presented at the *4th Int. Logic Programming Conf.*, Melbourne, Australia, May.

- Cleary, J., Goh, K.-S., and Unger, B. 1984. "Discrete-event simulation in Prolog", Research Report No. 84/171/29, Dept. of Computer Science, University of Calgary, Alberta, Canada, November.
- Clinger, W.D. 1981. "Foundations of Actor Semantics", AI-TR-633, MIT AI Lab.
- Clocksin, W.F., and Mellish, C.S. 1981. *Programming in Prolog*, Springer-Verlag, New York.
- Conery, J.S. 1987. "Object Oriented Programming with First Order Logic", CIS-TR-87-09, Univ. of Oregon.
- Conery, J.S. 1988. "Logical Objects", In *Logic Programming : Proc. 5th Int. Conf. and Symposium*, (eds.) Kowalski, R.A., and Bowen, K.A., Seattle, Washington, Sponsored by ALP and IEEE, pp. 420-434.
- Conlon, T. 1989. *Programming in Parlog*, Addison Wesley, Reading, MA.
- Coscia, P., Franceschi, P., Levi, G., Sardu, G., and Torre, L. 1988. "Meta-level Definition and Compilation of Inference Engines in the Epsilon Logic Programming Environment", In *Logic Programming : Proc. 5th Int. Conf. and Symposium*, (eds.) Kowalski, R.A., and Bowen, K.A., Seattle, Washington, Sponsored by ALP and IEEE, pp.359-373.
- Courtois, P.J., Heymans, F., and Parnas, D.L. 1971. "Concurrent Control with 'Readers' and 'Writers'", CACM, Vol. 14, No. 10, October, pp. 667-668.
- Cowan, W.R. 1988. "Polka Programming Environment Tools", Thesis for PGDip in Computing and Artificial Intelligence, South Bank Polytechnic, London.
- Cox, B.J. 1986. *Object-Oriented Programming : An Evolutionary Approach*, Addison-Wesley, Reading, MA.
- Cutcher, M.G. 1986. "Techniques of Parallel Logic Programming", Technical Report, Systems and Architectures, ICL, Reading, England.
- Danforth, S., and Tomlinson, C. 1988. "Type Theories and Object-Oriented Programming", ACM Computing Surveys, Vol. 20, No. 1, pp. 29-72.
- Davison, A. 1987a. "POOL : A Parlog Object Oriented Language", Dept. of Computing, Imperial College, London, April.
- Davison, A. 1987b. "Blackboard systems in PARLOG", Dept. of Computing, Imperial College, London, June.
- Davison, A. 1987c. "Polka : A Parlog Object Oriented Language", British Computer Society, Parallel and Distributed Object Orientated Programming Workshop, Univ of London, October.
- Davison, A. 1987d "Blackboard systems in Polka", Int. Journal of Parallel Processing, Vol. 16, No. 5.
- Davison, A. 1988a. "Simulation techniques in Polka", Dept. of Computing, Imperial College, London, June.

- Davison, A. 1988b. "User Guide for Polka", Dept. of Computing, Imperial College, London, July.
- Davison, A. 1989a. "BMS and Polka", Dept. of Computing, Imperial College, London, January.
- Davison, A. 1989b. "Distribution Issues in Polka", Dept. of Computing, Imperial College, London, August.
- Dieng, R., Fornarino, M., and Pinna, A-M. 1988. "OTHELO : Objects, Typing, Inheritance and Logic Programming", Tech. Report, INRIA, Valbonne, France, March.
- Dijkstra, E.W. 1972. "Hierarchical Ordering of Sequential Processes", In *Operating Systems Techniques*, (eds.) Hoare, C.A.R., and Perrott, R.H., Academic Press, New York, 1972, pp. 72-93.
- Dijkstra, E.W. 1975. "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", CACM, Vol. 18, No. 8, August 1975, pp. 453-457.
- Elshiewy, N.A. 1987. "Logic Programming for Real Time Control of Telecommunications Switching Systems", Tech. Report, Computer Science Lab, Ericsson Telecom, Stockholm, April.
- Elshiewy, N.A. 1988. "Modular and Communicating Objects in SICStus Prolog", Tech. Report, KISTA, Stockholm, Sweden.
- Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Raj Reddy, D. 1980. "The Hearsay II speech understanding system : Integrating knowledge to resolve uncertainty", Computing Survey, Vol. 12, No. 2, pp.213-253.
- Eshghi, K. 1988. "The Naming Relation in Meta Language", Dept. of Computing, Imperial College, London, April.
- Feldman, J.A. 1979. "High Level Programming for Distributed Computing", CACM, Vol. 22, No. 6, June, pp. 353-368.
- Fellous, J-M., Rizk, A., and Tueni, M. 1988. "An Object-Oriented Formalism in the Concurrent Logic Programming Language Parlog", BULL and INRIA Technical Report, Massy, France, August.
- Filman, R.E., Friedman, D.P. 1984. *Coordinated Computing : Tools and Techniques for Distributed Software*, McGraw-Hill.
- Fishman, G.S. 1978. *Principles of discrete-event simulation*, John Wiley.
- Foster, I., and Taylor, S. 1989. *Strand : New Concepts in Parallel Programming*, Prentice Hall, Englewood Cliffs, NJ.
- Fukunaga, K., and Hirose, S. 1986. "An Expierence with a Prolog-based Object-Oriented Language", In *OOPSLA '86 Proceedings*, SIGPLAN Notices, Vol. 21, No. 11, pp.224-231.

- Furukawa, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M., and Ueda, K., 1984. "Mandala : A Logic Based Knowledge Programming System", In *Proc. Int. Conf. 5th Gen. Computer Systems*, ICOT, Tokyo, Japan, pp. 613-622.
- Futo, I., and Szeredi, J. 1984. "System Simulation and Co-Operation Problem-Solving on a Prolog Basis", In *Implementations of Prolog*, (ed.) Campbell, J.A., Ellis Horwood, pp. 163-174.
- Gallaire, H. 1986. "Merging Objects and Logic Programming : Relational Semantics", In *Proc. AAAI 86*, Philadelphia, PA, pp. 754-758.
- Goguen, J.A., and Meseguer, J. 1986. "Extensions and Foundations of Object-oriented Programming", *SIGPLAN Notices*, Vol. 21, No. 10, October, pp. 153-162.
- Goldberg, A., Robson, D., and Ingalls, D. 1983. *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA.
- Gregory, S. 1987. *Parallel logic programming in PARLOG*, Addison Wesley, Reading, MA.
- Gregory, S., Neely, R., Ringwood, G. 1985. "Parlog for specification, verification and simulation", In *7th Int. Symp. on computer hardware, description languages and their applications*, Tokyo, August, (eds.) Koomen, C.J., and Moto-oka, T., pp.139-148.
- Gullichsen, E. 1985. "BiggerTalk : Object-Oriented Prolog", MCC Technical Report, No. STP-125-85, Austin, Texas, November.
- Hayes-Roth, B. 1984. "BB1 : An architecture for blackboard systems that control, explain, and learn about their own behaviour", Tech. Report No. STAN-CS-84-1034, Stanford University, December.
- Hayes-Roth, B. 1985. "A Blackboard architecture for control", *Artificial Intelligence* (26), pp.251-321.
- Hayes-Roth, B., Hayes-Roth, F., Rosenschein, F., and Cammarata, S., 1979. "Modelling Planning as an incremental opportunistic process", In *6th IJCAI*, Los Altos, CA, pp.375-383.
- Hayes-Roth, B., and Hewitt, M. 1985. "Learning control heuristics in BB1", Tech. Report STAN-CS-85-1036, Stanford University, January.
- Hennessey, M., and Plotkin, G.D. 1979. "Full Abstraction for a Simple Parallel Programming Language", *Proc. 8th MFCS, LNCS 74*, Springer, Berlin/New York pp. 108-120.
- Hewitt, C.E. 1977. "Viewing Control Structures as Patterns of Passing Messages", *Journal of Artificial Intelligence*, 8-3, pp.323-362.
- Hewitt, C.E. 1985. "The Challenge of Open Systems", *BYTE*, April, pp. 223-242.
- Hewitt, C.E. 1986. "Concurrency in Intelligent Systems", *AI Expert*, Premier Issue, pp. 45-50.

- Hewitt, C.E., and Agha, G. 1988. "Guarded Horn Clause Languages: Are They Deductive and Logical?", In *Proc. of the Int. Conf. on 5th Generation Computer Systems*, Vol 2, pp.650-656.
- Hewitt, C.E., and Baker, H. 1977. "Laws for Communicating Parallel Processes", In *1977 IFIP Congress Proceedings*, August, pp.987-992.
- Hewitt, C.E., Reinhardt, T., Agha, G., and Attardi, G. 1985. "Linguistic Support of Receptionists for Shared Resources", In *Seminar on Concurrency*, (eds.) Brookes, S.D., Roscoe, A.W., and Winskel, G., LNCS 197, Springer-Verlag, pp. 330-359.
- Hirsh, S., Kahn, K. M., and Miller, M.S. 1987. , "Interming : Unifying Keyword and Positional notations ", Tech. Report, Xerox PARC, Palo Alto, CA.
- Hogger, C.J., and Kowalski, R.A. 1987. "Logic Programming", In *Encyclopaedia of Artificial Intelligence*, (ed.) Shapiro, S.C., John Wiley & Sons, New York, NY, pp.544-558.
- Honiden, S., Uchihira, N., and Kasuya, T. 1985. "MENDEL : Prolog Based Concurrent Object Oriented Language", ICOT Technical Memorandum : TM-0144, Tokyo, Japan, November.
- Horowitz, E. 1984. *Fundamentals of Programming Languages*, Computer Science Press.
- Iline, H., and Kanoui, H. 1987. "Extending Logic Programming to Object Programming : The System LAP", In *Proc. 10th IJCAI*, Vol. 1, Milan, Italy, August, pp. 34-39.
- Ishikawa, Y., and Tokoro, M. 1986. "Orient84/K : An Object-Oriented Concurrent Programming Language for Knowledge Representation", In *Object Oriented Concurrent Programming*, (eds.) Yonezawa, A., and Tokoro, M., MIT Press.
- Johnson, M. 1988a. Email message to Comp.lang.prolog, johnson@cslu.UUCP, 2 June.
- Johnson, M. 1988b. Email message to Comp.lang.prolog, johnson@cslu.UUCP, 13 June.
- Kahn, K.M. 1982. "Intermission - Actors in Prolog", In *Logic Programming*, (eds.), Clark, K.L., and Tarlund, S.A., pp. 213-228.
- Kahn, K.M. 1986. "Uniform - A language based upon Unification which unifies (much of) Lisp, Prolog and Act 1, In *Logic Programming : Relations, Functions and Equations*, (eds.) DeGroot, D., and Lindstrom, G. Prentice Hall, Englewood Cliffs, NJ, pp.411-438.
- Kahn, K.M. 1987. "Intergrating high level Abstractions in a concurrent logic programming framework", Xerox PARC, Palo Alto, CA.
- Kahn, K.M., and Miller, M.S. 1988a. "Language Design and Open Systems", In *The Ecology of Computation*, (ed.) Huberman, B., North Holland.

- Kahn, K.M., and Miller, M.S. 1988b. "Objects with State in Prologs with Freeze", Xerox PARC, Palo Alto, CA.
- Kahn, K.M., Tribble, D., Miller, M.S., and Bobrow, D.G. 1987. "Vulcan : Logical Concurrent Objects", In *Concurrent Prolog : Collected Papers*, (ed.) Shapiro, E.Y., MIT Press, Cambridge, MA, Vol. 2, Chapter 30, pp.274-303.
- Karam, G.M. 1988. "Prototyping Concurrent Systems with Multilog", Tech. Report, Dept. of Systems and Computer Eng., Carleton Univ., Canada, January.
- Kauffman, H., and Grumbach, A. 1986. MULTIALOG : MULTiple worlds in LOGic programming", In *Expert Systems Conf. 1986*, Vol. 1, Brighton, England, pp.291-305.
- Kornfeld, W.A. 1983. "Equality for Prolog", In *Proc. 8th IJCAI*, Karlsruhe, Germany, August, pp.514-519.
- Koseki, Y. 1987. "Amalgamating Multiple Programming Paradigms in Prolog", In *Proc. 10th IJCAI*, Vol. 1, Milan, Italy, August, pp. 76-82.
- Kowalski, R.A. 1979. *Logic for problem solving*, North Holland, New York, NY.
- Kowalski, R.A. 1986. "Report on S-LONLI", Tech. Report, Dept. of Computing, Imperial College, London, June.
- Kwok, C.S. 1988. "A Survey of Structuring Mechanisms for Logic Programs", Tech. Report, Dept. of Computing, Imperial College, London, November.
- Lam, M. 1987. Personal Communication.
- Levy, J. 1986. "CFL : A Concurrent Functional Language, Embedded in a concurrent logic programming environment", The Weizmann Institute of Science, CS86-26, Israel, October.
- Manning, C.R. 1986. "Acore : An Actor Core Language Reference Manual", Internal Memo, MIT Message Passing Semantics Group Design Note : 7, September.
- McCabe, F.G. 1987a. "Logic and Objects", Research Report DOC 86/9, Dept. of Computing, Imperial College, May.
- McCabe, F.G. 1987b. "Denotational Graphics", Dept. of Computing, Imperial College, July.
- McCabe, F.G. 1988. "Logic and Objects", PhD Thesis, Dept. of Computing, Imperial College, December.
- Mello, P., and Natali, A. 1988. "Programs as Collections of Communicating Prolog Units", In *ESOP 86*, LNCS 213, (eds.) Robinet, B., and Wilhelm, R., Springer-Verlag, pp.274-288.
- Miller, M., Bobrow, D., Tribble, E.D., and Levy, J. 1987. "Logical Secrets", In *Concurrent Prolog : Collected Papers*,. (ed.) Shapiro, E.Y., MIT Press, Cambridge, MA, Vol. 2, Chapter 24, pp.140-161.

- Misra, J. 1986. "Distributed discrete-event simulation", *Computing Surveys*, Vol.18, No. 1, pp.39-65.
- Miyoshi, H., and Furukawa, K. 1987. "Object-Oriented Parser in the Logic Programming Language ESP", Technical Report, ICOT, Tokyo, Japan.
- Monteiro, L. 1984. "A Proposal for Distributed Programming in Logic", In *Implementations of Prolog*, (ed.) Campbell, J.A., Ellis Horwood, Chicester, pp.329-340.
- Murakami, M. 1988. "A New Declarative Semantics of Parallel Logic Programs with Perpetual Processes", In *Proc. of FGCS '88*, pp.374-381.
- Nakashima, H. 1984. "Knowledge Representation in Prolog/KR", In *1984 Int. Symp. on Logic Programming*, Boston MA, pp. 126-130.
- Naish, L. 1985. "Negation and Control in Prolog", Tech. Report 85/12, DoCS, Univ. of Melbourne, Australia.
- Newton, M., and Watkins, J. 1988. "The Combination of Logic and Objects for Knowledge Representation", *Journal of Object Oriented Programming*, November/December, pp.7-10.
- Nii, H.P. 1986a. "Blackboard systems : The blackboard model of problem solving and the evolution of blackboard architectures", Part One, *The A.I Magazine*, Summer, pp.38-53.
- Nii, H.P. 1986b. "Blackboard systems : Blackboard application systems, blackboard systems from a knowledge engineering perspective", Part Two, *The A.I Magazine*, August, pp.82-106.
- Nii, H.P., Feigenbaum, E.A., Anton., J.J., and Rockmore, J. 1982. "Signal-to-Symbol transformation : HASP/SIAP case study", *The A.I. Magazine* 3(2), pp.23-35.
- Ohki, M., Takeuchi, A., and Furukawa, K. 1988. "An Object-Oriented Programming Language based on the Parallel Logic Programming Language KL1", In *Logic Programming : Proc. of the 4th Int. Conf.*, (ed.) Lassez, J.L., Vol. 2, MIT Press, Cambridge, MA, pp. 894-909.
- Pereira, L.M., and Nasr, R. 1984. "Delta-Prolog : A Distributed Logic Programming Language", In *Int. Conf. on 5th Generation Systems*, Tokyo, Nov, pp.283-291.
- Plotkin, G.D. 1981. "A Structural Approach to Operational Semantics", DAIMI FN-19, Computer Science Dept., Aarhus University, Aarhus, Denmark, September.
- Plotkin, G.D. 1983. "An Operational Semantics for CSP", In *Formal Description of Programming Concepts II*, (ed.) Bjørner, D., North-Holland, Amsterdam, pp. 199-223.
- Porto, A. 1983. "Logical Action Systems", In *Proc. Int. Workshop on Logic Programming*, Praia da Falésia, Portugal, pp. 192-203.

- Quintus Computer Systems Inc. 1988. ProWINDOWS Reference Manual", Evaluation Version, CA, August.
- Rauen, J. 1987. "Lexically scoped FCP", Presented at the *Concurrent LP workshop*, Xerox PARC, Palo Alto, CA, Sept.
- Richard, G., and Rizk, A. 1988. "Semantics of Parlog, A Parallel Logical Language", INRIA report, France.
- Ringwood, G.A. 1988. "Parlog86 and the Dining Logicians", CACM, Vol. 31, No. 1, pp. 10-25.
- Sacerdoti, E.D. 1974. "Planning in a hierarchy of abstraction spaces", Artificial Intelligence 5, pp.115-135.
- Safra, S., and Shapiro, E.Y. 1986. "Meta-interpreters for real", In *Proceedings of IFIP-86*, North Holland, pp. 271-278.
- Saraswat, V.A. 1985. "Partial Correctness Semantics for CP [$\downarrow$, |, &]", LNCS 206, Springer-Verlag, pp. 347-368.
- Saraswat, V.A. 1987. "CP as a concurrent constraint logic programming language", Concurrent LP workshop report, Xerox PARC, September.
- Saraswat, V.A. 1988. "Concurrent Logic Programming Languages", PhD Thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh.
- Sato, H., and Matsumoto, H. 1986. "Intelligent support for office work with a Prolog-based Object-Oriented Programming Language ESP", ICOT Technical Report : TR-172, Tokyo, Japan, April.
- Shapiro, E.Y. 1983. "A subset of Concurrent Prolog and its interpreter", ICOT technical report, TR-003, Tokyo, Japan.
- Shapiro, E.Y. 1987. "Or-Parallel Prolog in FCP", In *Concurrent Prolog : Collected Papers*, (ed.) Shapiro, E.Y., MIT Press, Cambridge, MA, Vol. 2, Chapter 34, pp.415-441.
- Shapiro, E.Y. 1989. "The Family of Concurrent Logic Programming Languages", CS89-08, The Weizmann Institute of Science, Israel, May.
- Shapiro, E.Y. and Safra, S. 1986. "Mutliway merge with constant delay in Concurrent Prolog", New Generation Computing, 4(2), pp. 211- 216.
- Shapiro, E.Y. and Takeuchi, A. 1983. "Object Oriented Programming in Concurrent Prolog", New Generation Computing 1 (1983), pp.25-48.
- Stabler, E.P. 1986. "Object-Oriented Programming in Prolog", AI Expert, Vol. 1, No. 3 pp.46-57.
- Stefik, M., and Bobrow, D.G. 1986. "Object Oriented Programming : Themes and Variations", The A.I Magazine, Vol. 6, No. 4, January, pp. 40-62.
- Sterling, L., and Shapiro, E.Y. 1987. *The Art of Prolog : Advanced Programming Techniques*, MIT Press Series in Logic Programming, Reading, MA.

- Takeuchi, I., Okuno, H., and Ohsato, N. 1986. "A List Processing Language TAO with Multiple Programming Paradigms", *New Generation Computing* 4 (1986), pp. 401-444.
- Tam, C.C. 1988. "On Evaluating and Improving Open Path Languages", Dept. of Computing, Imperial College, Draft, May.
- Terry, A. 1983. "The CRYNALIS Project : Hierarchical control of production systems", Tech. Report HPP-83-19, Stanford University.
- Theriault, D. 1982. "A Primer for the Act-1 Language", A.I. Memo 672, MIT AI Lab, April.
- Theriault, D. 1983. "Issues in the Design and Implementation of Act2", Technical Report 728, MIT AI Lab, June.
- Thomas, D. 1989. "What's in an Object?", *BYTE*, March, pp.231-240.
- Trehan, R., Wilks, P., Buckley, M. 1988. "Object Models in the Committed Choice Non-deterministic Logic Languages", DAI Research Paper No. 368, Univ. of Edinburgh.
- Trenouth, J. 1988. Email message to Comp.lang.prolog, jtr@expya.UUCP, 15 April.
- Tribble, E.D., Miller, M.S., Kahn, K., Bobrow D.G. and Abbott, C. 1987. "Channels : A generalization of streams", in J.-L. Lassez (ed.), *Proc 4th International Conference of Logic Programming*, MIT Press, pp. 839-857.
- Warren, D.H.D. 1982. "Higher-order extensions to Prolog : are they needed?", In *Machine Intelligence 10*, (eds.) Hayes, J., Michie, D., and Pao, Y.-H., Ellis Horwood, Chichester, pp. 441-453.
- Watt, J. 1984. *Learning with Apple Logo*, McGraw-Hill.
- Wegner, P. 1989. "Learning the language", *BYTE*, March, pp.245-253.
- Weinbaum, D., and Shapiro, E.Y. 1986. "Hardware Description and Simulation using Concurrent Prolog", In *Concurrent Prolog : Collected Papers*, (ed.) Shapiro, E.Y., MIT Press, Cambridge, MA, Vol. 2, Chapter 36, pp.470-490.
- Weyhrauch, R.W. 1980. "Prologomena to a theory of Mechanized Formal Reasoning", *Artificial Intelligence* 13, pp.133-170.
- Yang, R. 1988. "Programming in Andorra-I", Technical Report, Dept. of Computer Science, Univ. of Bristol.
- Yardeni, E., and Shapiro, E.Y. 1987. "A Type System for logic Programs", In *Concurrent Prolog : Collected Papers*, (ed.) Shapiro, E.Y., MIT Press, Cambridge, MA, Vol. 2, Chapter 28, pp.211-244.
- Yoshida, K., and Chikayama, T. 1987. "A'UM - Parallel Object-Oriented Language upon KLI", ICOT Technical Report : TR 335, Tokyo, Japan.

Young, S.J. 1983. *An Introduction to Ada*, Ellis Horwood.

Zaniolo, C. 1984. "Object-Oriented Programming in Prolog", Presented at the *1984 Symposium on Logic Programming*, Atlantic City, NJ, February, pp. 265-270.