

Unity-inspired object-oriented concurrent system development

by

Marlene Maria Ross

Thesis submitted in fulfillment of the requirements for the degree
Doctor of Philosophy in Computer Science
in the Faculty of Natural and Agricultural Sciences
University of Pretoria
Pretoria

January 2001

615237527



UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

2 APR 005.3
115786049 ROSS

Unity-inspired object-oriented concurrent system development

by

Marlene Maria Ross

Thesis submitted in fulfillment of the requirements for the degree
Doctor of Philosophy in Computer Science
in the Faculty of Natural and Agricultural Sciences
Department of Computer Science
University of Pretoria
Pretoria

January 2001

Supervisor: Professor D.G. Kourie

Co-supervisor: Professor R.J. van den Heever

ABSTRACT

The design of correct software remains difficult, especially when dealing with concurrency. The primary goal of the research presented here is to devise a **pragmatic** software development method which

- aids the software designer in producing **reliable** software,
- is **scalable**,
- is **understandable**,
- follows a **unified approach** towards software development (is applicable to different implementation architectures),
- promotes **reuse**,
- has **seamless** transitions between the software development phases,
- guarantees **general availability** and **minimises developmental resources**.

The two main characteristics of the proposed new development method are captured in its name, viz. **Single Location Object-Oriented Programming (SLOOP)**. It is an **object-oriented** method, but its **computational model** is that of a set of statements that execute infinitely often and in any order. A program with such a computational model is called a Single Location Program (SLP). A UNITY program can also be classified as a Single Location Program. In the UNITY theory of programming it was demonstrated how this computational model could **simplify correctness reasoning**, particularly for concurrent systems.

It is this simplification, together with the structuring and reuse features of object-orientation, that is leveraged in the SLOOP method to produce a mechanism whereby **ordinary software practitioners** can take advantage of the benefits of a more rigorous approach towards software development without requiring an in-depth understanding of the underlying mathematics.

The following features of the SLOOP method contribute towards achieving the above goals:

- its **computational model** (it simplifies correctness reasoning, thereby promoting understandability and scalability, and also facilitates designs that are independent of the target implementation architectures),



- ❑ its **object-oriented** nature (apart from promoting reuse of frameworks, design patterns and classes, the SLOOP method provides the necessary mechanisms to facilitate reuse of correctness properties, correctness arguments as well as mappings to implementation architectures),
- ❑ its **emphasis on correctness reasoning** throughout the software development life cycle (its "constructive approach" aids reliability and seamlessness),
- ❑ the **unique** way in which the correctness properties can be specified, reused and reasoned about (this contributes towards understandability and scalability),
- ❑ the **checklist** of useful correctness properties that is provided (this promotes reliability),
- ❑ the **incorporation of existing notations** into the SLOOP syntax (this guarantees general availability, minimises developmental resources and aids understandability).

The main **contribution** of this thesis is that it presents a **unique** way of incorporating the SLP computational model into an object-oriented method with the specific aim of **simplifying informal correctness reasoning** and **promoting reuse**. The notation used for the specification of correctness properties facilitates reuse of correctness properties, ensures the integrity of these specifications and allows one to specify correctness properties at a higher level of abstraction.

The SLOOP method offers a **unique way of modelling concurrency in object-oriented systems** (via its parallel methods), which takes full advantage of the **encapsulation** and **inheritance** features of object-orientation. The issues surrounding **mappings** to implementation architectures are addressed, showing how even mappings can be **reused**. Finally, the **general applicability** of the SLOOP method is demonstrated.

SAMEVATTING

Die ontwerp van korrekte programmatuur bly moeilik, veral wanneer gelyktydigheid ter sprake is. Die primêre doel van hierdie navorsing is om 'n **pragmatiese** programmatuur ontwikkelingsmetode te ontwikkel wat

- die ontwerper sal help om **betroubare** programmatuur te ontwikkel,
- **skaaleerbaar** is,
- **verstaanbaar** is,
- 'n **eenvormige benadering** volg ten opsigte van programmatuurontwikkeling,
- **hergebruik** aanmoedig,
- 'n **gladde oorgang** tussen ontwikkelingsfasies teweegbring,
- **algemene beskikbaarheid** waarborg en die **gebruik van ontwikkelingsbronne** minimeer.

Die twee hoofeienskappe van die voorgestelde nuwe ontwikkelingsmetode word weerspieël in die naam - **Enkel Posisie Objek-georiënteerde Programmering (EPOP)**. Dit is 'n **objek-georiënteerde** metode, maar die **verwerkingsmodel** is die van 'n groep stellings wat oneindig gereeld en in enige volgorde uitvoer. 'n Program met so 'n verwerkingsmodel word 'n Enkel Posisie Program (EPP) genoem. 'n UNITY program kan ook geklassifiseer word as 'n EPP. In die UNITY teorie van programmering is gedemonstreer hoe hierdie verwerkingsmodel die **beredenering van korrektheid** kan **vereenvoudig**, spesifiek vir gelyktydige stelsels.

Dit is hierdie vereenvoudiging, tesame met die strukturering en hergebruikseienskappe van objek-oriëntasie, wat aangewend word in die EPOP metode om 'n meganisme te produseer waar **gewone programontwerpers** die voordele van 'n meer formele benadering teenoor programmatuurontwikkeling kan benut, sonder om noodwendig die beginsels van die onderliggende wiskunde te hoef te bemeester.

Die volgende eienskappe van die EPOP metode dra by om bogenoemde doelstellings te bevredig:

- die **verwerkingsmodel** (dit vereenvoudig korrektheidsberedenering, dus verhoog dit ook verstaanbaarheid en skaleerbaarheid, en maak ook ontwerpe moontlik wat onafhanklik is van die implementasie argitektuur),
- die **objek-georiënteerde** aard (bo en behalwe die feit dat dit hergebruik van raamwerke, patrone en klasse aanmoedig, voorsien die EPOP metode die meganismes wat die hergebruik van korrektheidseienskappe, korrektheidsargumente, asook afbeeldings na die implementasie argitektuur moontlik maak),
- die deurgaande **klem op korrektheidsberedenering** tydens die programmatuur=ontwikkelingsfasies (die 'konstruktiewe benadering' moedig betroubaarheid en 'n gladde oorgang tussen ontwikkelingsfasies aan),
- die **unieke** manier waarop die korrektheidseienskappe gespesifiseer, hergebruik en beredeneer kan word (dit dra by tot verstaanbaarheid en skaleerbaarheid),
- die **lys van nuttige korrektheidseienskaptipes** wat verskaf word (dit moedig betroubaarheid aan),
- die **insluiting van bestaande notasies** in die EPOP sintaks (dit waarborg algemene beskikbaarheid, minimeer die gebruik van ontwikkelingsbronne en verhoog verstaanbaarheid).

Die **hoofbydrae** van hierdie verhandeling is die feit dat dit 'n **unieke** manier bied om die EPP verwerkingsmodel in 'n objek-georiënteerde metode te inkorporeer, met die spesifieke doel om **informele korrektheidsberedenering te vereenvoudig en hergebruik aan te moedig**. Die notasie wat gebruik word vir die spesifikasie van korrektheidseienskappe maak die hergebruik

van korrektheidseienskappe moontlik, waarborg die integriteit van hierdie spesifikasies en laat die spesifikasie van korrektheidseienskappe op 'n hoër vlak van abstraksie toe.

Die EPOP metode bied 'n **unieke manier om gelyktydigheid in objek-georiënteerde stelsels te modelleer** (by wyse van parallelle metodes), wat ten volle voordeel trek uit die **enkapsulasie en oorervingsseienskappe** van objek-oriëntering. Die kwessies rondom die **afbeeldings** op implementasie argitekture word geadresseer, en wys uit hoe selfs hierdie afbeeldings **hergebruik** kan word. Laastens word die **algemene toepaslikheid** van die EPOP metode gedemonstreer.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to the following people:

Professor D. Kourie, my supervisor, for his stimulating discussions, excellent advice and attention to detail. His encouragement and support have been invaluable.

Professor R. J. van den Heever, the co-supervisor, for his helpful suggestions, many stimulating discussions and careful reading of this thesis. His vision and insight regarding the various aspects of Computer Science have been a continued source of inspiration and motivation.

The management team at Azisa, who allowed me to work flexible hours in order to provide me with the opportunity to devote time to my studies.

My husband and parents, for their continued encouragement and all the opportunities that they have given me.

DECLARATION

I hereby declare that, unless otherwise indicated, all the work in this thesis was done by myself and that this thesis has not been submitted to this or any other institution of learning in support of an application for any other degree or qualification.

M. M. Ross
January 2001

DEDICATION

To Victor, for his love, support and understanding

and to Pieter Thomas and Emul, for being two such lovable boys.

TABLE OF CONTENTS

1. INTRODUCTION	1
1.1 The problem	1
1.2 The goals	3
1.3 Towards a new method	3
1.3.1 <i>The object-oriented paradigm</i>	4
1.3.2 <i>UNITY</i>	4
1.3.3 <i>Formal methods</i>	5
1.3.4 <i>Computational reflection</i>	7
1.3.5 <i>The resulting method: SLOOP</i>	7
1.4 Related work	11
1.5 The contribution	13
1.6 Structure of the thesis	14
2. CORRECTNESS, SPECIFICATIONS AND UNITY	17
2.1 Introduction	17
2.2 The software correctness conundrum	18
2.2.1 <i>Validation, a posteriori program verification and the "constructive approach"</i>	18
2.2.2 <i>Categories of correctness properties</i>	19
2.3 Modelling concurrency	20
2.3.1 <i>Interleaving</i>	20
2.3.2 <i>Atomicity and interference</i>	21
2.3.3 <i>Fairness</i>	21
2.4 Formulating correctness properties	23
2.4.1 <i>Temporal logic</i>	23
2.4.2 <i>Temporal operators</i>	24
2.4.3 <i>Which correctness properties?</i>	24
2.4.4 <i>Safety properties</i>	26
2.4.4.1 <i>Partial correctness</i>	26
2.4.4.2 <i>Clean behaviour</i>	26
2.4.4.3 <i>Global and local invariants</i>	26
2.4.4.4 <i>Mutual exclusion</i>	26
2.4.4.5 <i>Deadlock freedom</i>	27
2.4.4.6 <i>Generalised deadlock freedom</i>	27
2.4.5 <i>Liveness properties</i>	28
2.4.5.1 <i>Total correctness</i>	28
2.4.5.2 <i>Intermittent assertions</i>	28
2.4.5.3 <i>Accessibility</i>	28
2.4.5.4 <i>Liveness</i>	28
2.4.5.5 <i>Responsiveness</i>	29
2.4.6 <i>Precedence properties</i>	29
2.4.6.1 <i>Safe liveness</i>	29
2.4.6.2 <i>Absence of unsolicited response</i>	29
2.4.6.3 <i>Fair responsiveness</i>	29
2.5 UNITY	30
2.5.1 <i>Non-determinism</i>	30
2.5.2 <i>Assignments and absence of control flow</i>	30
2.5.3 <i>State transitions</i>	33
2.5.4 <i>Program structuring</i>	33
2.5.5 <i>A logic for the specification, design and verification of UNITY programs</i>	33



2.6 Summary	35
3. THE OBJECT-ORIENTED PARADIGM	39
3.1 Introduction	39
3.2 Object-orientation and concurrency	40
3.2.1 <i>Multiple threads of control</i>	40
3.2.2 <i>Synchronisation</i>	41
3.2.2.1 <i>The semantics of operation invocations</i>	41
3.2.2.2 <i>Race conditions</i>	45
3.2.2.3 <i>Synchronisation constraints</i>	46
3.3 Design patterns and frameworks	47
3.3.1 <i>Categories of patterns</i>	47
3.3.2 <i>Advantages of using design patterns</i>	48
3.3.3 <i>Design patterns and formalisation</i>	49
3.3.4 <i>Frameworks</i>	49
3.4 Specifying object interaction	50
3.5 Object-oriented formal methods	52
3.6 Summary	52
4. THE SLOOP METHOD	55
4.1 Introduction	55
4.2 Overview of the SLOOP method	56
4.2.1 <i>The SLOOP computational model</i>	56
4.2.2 <i>The main components of the method</i>	57
4.2.3 <i>The crux of the method</i>	60
4.3 The structure of a SLOOP program	67
4.3.1 <i>The SLOOP-program</i>	67
4.3.2 <i>SLOOP classes</i>	74
4.3.3 <i>The macros-section</i>	74
4.3.3.1 <i>Purpose of the macros-section</i>	74
4.3.3.2 <i>The syntax</i>	75
4.3.3.3 <i>Example usage</i>	77
4.3.4 <i>The properties-section</i>	79
4.3.4.1 <i>The syntax</i>	79
4.3.4.2 <i>Reuse of correctness properties</i>	82
4.3.4.3 <i>Class properties and method properties</i>	82
4.3.4.4 <i>Definitions of the logical relations</i>	83
4.3.4.5 <i>Method properties and the value returned by a method</i>	84
4.3.5 <i>The methods-section</i>	85
4.3.5.1 <i>The methods-section syntax</i>	85
4.3.5.2 <i>Method invocation</i>	86
4.3.5.3 <i>Parallel method activation and the number of active parallel statements</i>	86
4.3.5.4 <i>Nesting of SLOOP method invocations</i>	87
4.3.5.5 <i>SLOOP objects and events</i>	89
4.3.6 <i>The SLOOP statement-list</i>	90
4.3.6.1 <i>The syntax</i>	90
4.3.6.2 <i>Statements, components and parts</i>	91
4.3.6.3 <i>Evaluation order</i>	95
4.3.6.4 <i>The simplification of reasoning about correctness</i>	97
4.3.6.5 <i>Prevention of deadlock</i>	98
4.3.7 <i>The SLOOP method and inheritance, encapsulation and polymorphism</i>	107
4.4 Seamless analysis, design and implementation	107
4.4.1 <i>The requirements analysis phase</i>	108
4.4.2 <i>The design phase</i>	108
4.4.3 <i>The implementation phase</i>	112



4.4.3.1 Mapping the activation-section	113
4.4.3.2 Mapping a package	114
4.4.3.3 Mapping a class and its methods	114
4.5 Summary	120
5. REQUIREMENTS ANALYSIS FROM THE SLOOP PERSPECTIVE	123
5.1 Introduction	123
5.2 Useful correctness properties	125
5.2.1 <i>Safety properties</i>	127
5.2.1.1 Partial correctness	127
5.2.1.2 Clean behaviour	127
5.2.1.3 Global and local invariants	128
5.2.1.4 Mutual exclusion	128
5.2.1.5 Deadlock freedom	129
5.2.1.6 Generalised deadlock freedom	129
5.2.1.7 Unless property	129
5.2.2 <i>Liveness properties</i>	130
5.2.2.1 Total correctness	130
5.2.2.2 Intermittent assertions	130
5.2.2.3 Accessibility	131
5.2.2.4 Liveness (absence of individual starvation)	131
5.2.2.5 Responsiveness	131
5.2.3 <i>Precedence properties</i>	132
5.2.3.1 Safe liveness	132
5.2.3.2 Absence of unsolicited response	132
5.2.3.3 Fair responsiveness	132
5.2.4 <i>SLOOP checklist of correctness properties</i>	132
5.3 Constructing the class diagram	133
5.3.1 <i>Initial informal problem statement</i>	133
5.3.2 <i>The class diagram</i>	136
5.4 Specifying system behaviour informally	142
5.4.1 <i>Safety properties</i>	142
5.4.1.1 AS1-yy (partial correctness)	142
5.4.1.2 AS2-yy (clean behaviour)	142
5.4.1.3 AS3-yy (global invariants)	146
5.4.1.4 AS4-yy (unless properties)	148
5.4.2 <i>Liveness properties</i>	149
5.4.2.1 AL1-yy (total correctness)	149
5.4.2.2 AL2-yy (intermittent assertions)	149
5.4.2.3 AL3-yy (responsiveness)	150
5.4.3 <i>Precedence properties</i>	150
5.4.3.1 AP1-yy (safe liveness)	150
5.4.3.2 AP2-yy (absence of unsolicited response)	152
5.4.3.3 AP3-yy (fair responsiveness)	152
5.4.4 <i>Consolidation</i>	153
5.4.5 <i>Informal specification of correctness properties of individual classes</i>	159
5.4.5.1 The CommsProviderSimulator class	160
5.4.5.2 The ServiceProviderSimulator class	161
5.4.6 <i>Summary of the requirements analysis phase steps</i>	162
5.5 Summary	163
6. DESIGN FROM THE SLOOP PERSPECTIVE	165
6.1 Introduction	165
6.2 Identifying reusable artifacts	166
6.2.1 <i>Mapping problem domain objects onto solution domain objects</i>	167

6.2.2	<i>Formal versus informal specification of class properties</i>	175
6.2.3	<i>Reusing existing classes through inheritance</i>	175
6.2.4	<i>Discovering suitable existing classes after design level refinements</i>	176
6.3	Refining the specification	177
6.3.1	<i>The role of correctness properties during design level refinements</i>	177
6.3.2	<i>The impact of the computational model on design level decisions</i>	179
6.4	Formalising correctness properties	181
6.5	Deriving SLOOP statements	182
6.6	Constructing the SLOOP program	193
6.6.1	<i>Using the results of the design phase refinements to determine the contents of the sequential methods of the activation classes</i>	194
6.6.2	<i>Determining the contents of the parallel methods of the activation classes</i>	198
6.7	Making the design more reusable	204
6.8	Summary	204
7.	REASONING ABOUT SLOOP PROGRAMS	207
7.1	Introduction	207
7.1.1	<i>Conveying the semantics</i>	207
7.1.2	<i>Reasoning about correctness</i>	207
7.1.3	<i>Reusability</i>	207
7.1.4	<i>Absence of control flow</i>	209
7.1.5	<i>Using correctness properties to derive SLOOP statements</i>	210
7.2	Conveying the semantics	210
7.2.1	<i>The static nature of a class</i>	210
7.2.2	<i>The dynamic nature of a class</i>	212
7.3	The impact of various SLOOP features on correctness reasoning	218
7.3.1	<i>Safety properties</i>	219
7.3.1.1	<i>Using the correctness arguments of a clean behaviour property to illustrate how the use of macros in SLOOP programs can simplify correctness reasoning</i>	219
7.3.1.2	<i>Demonstrating how the computational model and object-oriented features such as data encapsulation and reusability influence the approach followed during correctness reasoning</i>	221
7.3.1.3	<i>Using a global invariant property to discuss the responsibility of the client object regarding preconditions in correctness properties</i>	231
7.3.1.4	<i>Using an unless property to demonstrate how the correctness properties specified for the class itself as well as those inherited from its parent class are applied in correctness arguments</i>	233
7.3.2	<i>Liveness properties</i>	239
7.3.2.1	<i>Showing why the postconditions of a total correctness property can be used in an ensures relation</i>	239
7.3.2.2	<i>Using an intermittent assertion property to demonstrate how the repeated execution of parallel statements guarantees progress, provided the preconditions hold at some point</i>	241
7.3.2.3	<i>Using a responsiveness property to demonstrate the importance of showing that the preconditions will eventually hold when reusing other correctness properties in the correctness arguments of a liveness property</i>	246
7.3.3	<i>Precedence properties</i>	250
7.3.3.1	<i>Using a safe liveness property to highlight the impact of the postconditions: and postconditions:withArguments: constructs on correctness arguments</i>	250
7.3.3.2	<i>Using an absence of unsolicited reponse property to demonstrate how the characteristics of an ensures relation can be used in correctness arguments</i>	257



7.3.3.3 Using a fair responsiveness property to illustrate the importance of showing via correctness arguments that the selection of the constituent classes of a system will indeed result in the behaviour as described in the specification of the system	259
7.4 Deriving SLOOP statements from correctness properties	264
7.5 Summary	269
8. THE IMPLEMENTATION PHASE	273
8.1 Introduction	273
8.2 Mappings to various architectures	274
8.2.1 <i>Sequential architectures</i>	274
8.2.2 <i>Synchronous shared-memory architectures</i>	274
8.2.3 <i>Asynchronous shared-memory architectures</i>	275
8.2.4 <i>Distributed systems</i>	276
8.2.5 <i>Comparison with UNITY mappings</i>	276
8.3 Deriving executable programs on various architectures	277
8.3.1 <i>Sequential architectures</i>	277
8.3.2 <i>Synchronous shared-memory architectures</i>	278
8.3.3 <i>Asynchronous shared-memory architectures</i>	279
8.3.4 <i>Distributed systems</i>	282
8.3.4.1 <i>Guaranteeing absence of deadlock in a mapping to a distributed architecture</i>	284
8.3.4.2 <i>Identifying the objects that need to be reserved</i>	305
8.3.4.3 <i>The middleware infrastructure</i>	307
8.4 Mapping macros	308
8.5 Mapping SLOOP statements	309
8.5.1 <i>Mapping quantified-statement-lists</i>	309
8.5.1 <i>Mapping statement-components and component-parts</i>	310
8.6 The use of reflection in mappings of SLOOP programs	312
8.6.1 <i>A reflective computation infrastructure</i>	312
8.6.2 <i>Using reflective computation for control purposes</i>	317
8.6.3 <i>Using reflective computation for assertion checking</i>	320
8.7 Modifying the level of parallelism in a SLOOP design	320
8.8 Summary	321
9. INCORPORATING DESIGN PATTERNS INTO A SLOOP DESIGN	323
9.1 Introduction	323
9.2 Architectural patterns	323
9.2.1 <i>Pipes and filters</i>	323
9.2.2 <i>Reflection</i>	325
9.3 Creational design patterns	326
9.3.1 <i>The Factory Method</i>	326
9.3.2 <i>Singleton</i>	330
9.4 Structural design patterns	332
9.4.1 <i>Adapter</i>	332
9.4.2 <i>Flyweight</i>	333
9.4.3 <i>Proxy</i>	334
9.5 Behavioural design patterns	334
9.5.1 <i>Iterator</i>	334
9.5.2 <i>State</i>	334
9.5.3 <i>Template</i>	344
9.5.4 <i>Strategy</i>	347
9.6 Summary	349
10. CONCLUSIONS	353



10.1 Evaluation of the SLOOP method	353
10.1.1 <i>Increasing the reliability of systems developed via this method</i>	353
10.1.2 <i>Scalability of the method</i>	355
10.1.3 <i>Understandability of the method</i>	356
10.1.4 <i>Unified approach</i>	357
10.1.5 <i>Reusability</i>	357
10.1.6 <i>Seamlessness</i>	357
10.1.7 <i>General availability and minimisation of developmental resources</i>	358
10.2 Concluding remarks and future research directions	358
Appendix A. SLOOP SYNTAX QUICK REFERENCE	361
A.1 Notational conventions	361
A.2 The SLOOP program structure	361
A.3 Complete and partial class descriptions	362
A.4 The macros-section	362
A.5 The properties-section	363
A.6 The methods-section	365
A.7 SLOOP statements	365
A.8 Comments in a SLOOP program	366
Appendix B. A SLOOP PROGRAM FOR A CALL CENTRE	367
B.1 Scope of the first level of refinement of the design	367
B.2 The CC_Activation class	370
B.3 The CC_SimulationActivation class	376
B.4 The Configuration class	378
B.5 The EventSimulator class	383
B.6 The CommsProviderSimulator class	388
B.7 The Connection class	391
B.8 The ServiceCategoryAllocator class	394
B.9 The ServiceRequest class	397
B.10 The ServiceCategory class	400
B.11 The TimerServices class	404
B.12 The TimeoutElement class	412
B.13 The ServiceProviderSimulator class	415
Bibliography	421

LIST OF FIGURES

Figure 3-1(a)	Synchronous operation invocations in a sequential program	44
Figure 3-1(b)	A scenario for deadlock during synchronous operation invocations	44
Figure 4-1	Overview of the SLOOP method	58
Figure 4-2	Scenarios representing atomic executions	62
Figure 4-3	Call centre example	70
Figure 4-4	Package structure of the call centre	73
Figure 4-5	Example combinations of methods from different categories	88
Figure 4-6	SLOOP statement structure	92
Figure 4-7	The role of <i>if</i> clauses in the parallel statement hierarchy	104
Figure 4-8(a)	Cyclic invocation of the sequential methods of an object	105
Figure 4-8(b)	Example of an invalid execution sequence	106
Figure 4-9	Requirements analysis phase steps	109
Figure 4-10(a)	Design and implementation phases (Part 1 of 2)	110
Figure 4-10(b)	Design and implementation phases (Part 2 of 2)	111
Figure 5-1	Architecture of a call centre and its environment at a high level of abstraction	135
Figure 5-2	Class diagram of the call centre and its environment (without simulation classes)	140
Figure 5-3	Class diagram of the call centre and its environment (with simulation classes)	141
Figure 6-1	Mapping problem domain classes onto solution domain classes	174
Figure 8-1	Scenario for deadlock when there are no shared objects, but the objects share processors in a single process per processor distributed architecture....	286
Figure 8-2(a)	Resource allocation graph showing deadlock	289
Figure 8-2(b)	Deadlock is prevented is the resource allocation requests are made in an (arbitrary) order	289
Figure 8-3	Deadlock as a result of the granting of multiple reservation requests.....	290
Figure 8-4(a)	Handling of local reservation requests (shared local objects).....	292
Figure 8-4(b)	Handling of local reservation requests (no shared local objects).....	293
Figure 8-5	Scenario illustrating how resources may be requested by and granted to parallel statements at processor A.....	298
Figure 8-6	Scenario illustrating that a local parallel statement is always allowed to execute if it does not send messages to remote objects and also not to local objects that have been allocated to or requested by a remote parallel statement.....	300
Figure 8-7	Scenario illustrating that a local parallel statement for which all resources have been granted may be executed while another local parallel statement is still waiting for remote resources to be granted. The two statements do not share local objects.....	302
Figure 8-8(a)	Deadlock in the case of reservation request collision when both local and remote reservation requests are granted at each processor.....	303
Figure 8-8(b)	Deadlock when a local reservation request is granted while a local object is currently allocated to a remote parallel statement.....	303
Figure 8-9(a)	Deadlock prevention in the case of reservation request collision.....	304
Figure 8-9(b)	Deadlock prevention because a reservation request for a local object is queued while a local object has been allocated to a statement at a remote processor.....	304
Figure 8-10	Class diagram of the ALBEDO meta-object infrastructure based on Smalltalk.....	313
Figure 9-1	Pipes and filters in the call centre system.....	325
Figure 9-2	Incorporating the State design pattern into the Connection class.....	335

Figure B-1	Call centre object diagram.....	368
Figure B-2	Contents of the various call centre packages.....	369
Figure B-3(a)	Structures used by aTimerServices (currentTick = 3).....	405
Figure B-3(b)	Structures used by aTimerServices (currentTick = 0).....	406