**PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

PRECISIONSM RESOLUTION TARGETS

**AN OBJECT-ORIENTED REFLECTIVE SIMULATION ENVIRONMENT FOR
DISTRIBUTED ALGORITHMS**

by

ADRIANA LOPES DIAZ

**A Thesis submitted to the Faculty of Graduate Studies
and Research in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy**

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

April 12, 1996

© 1996, Adriana Lopes Diaz

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-19380-2

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture	0729
Art History	0377
Cinema	0900
Dance	0378
Fine Arts	0357
Information Science	0723
Journalism	0391
Library Science	0399
Mass Communications	0708
Music	0413
Speech Communication	0459
Theater	0465

EDUCATION

General	0515
Administration	0514
Adult and Continuing	0516
Agricultural	0517
Art	0273
Bilingual and Multicultural	0282
Business	0688
Community College	0275
Curriculum and Instruction	0727
Early Childhood	0518
Elementary	0524
Finance	0277
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Home Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Philosophy of	0998
Physical	0523

Psychology	0525
Reading	0535
Religious	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language	
General	0679
Ancient	0289
Linguistics	0290
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Medieval	0297
Modern	0298
African	0316
American	0591
Asian	0305
Canadian (English)	0352
Canadian (French)	0355
English	0593
Germanic	0311
Latin American	0312
Middle Eastern	0315
Romance	0313
Slavic and East European	0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0469

SOCIAL SCIENCES

American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
General	0310
Accounting	0272
Banking	0770
Management	0454
Marketing	0338
Canadian Studies	0385
Economics	
General	0501
Agricultural	0503
Commerce-Business	0505
Finance	0508
History	0509
Labor	0510
Theory	0511
Folklore	0358
Geography	0366
Gerontology	0351
History	
General	0578

Ancient	0579
Medieval	0581
Modern	0582
Black	0328
African	0331
Asia, Australia and Oceania	0332
Canadian	0334
European	0335
Latin American	0336
Middle Eastern	0333
United States	0337
History of Science	0585
Law	0398
Political Science	
General	0615
International Law and Relations	0616
Public Administration	0617
Recreation	0814
Social Work	0452
Sociology	
General	0626
Criminology and Penology	0627
Demography	0938
Ethnic and Racial Studies	0631
Individual and Family Studies	0628
Industrial and Labor Relations	0629
Public and Social Welfare	0630
Social Structure and Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0999
Women's Studies	0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture	
General	0473
Agronomy	0285
Animal Culture and Nutrition	0475
Animal Pathology	0476
Food Science and Technology	0359
Forestry and Wildlife	0478
Plant Culture	0479
Plant Pathology	0480
Plant Physiology	0817
Range Management	0777
Wood Technology	0746
Biology	
General	0306
Anatomy	0287
Biostatistics	0308
Botany	0309
Cell	0379
Ecology	0329
Entomology	0353
Genetics	0369
Limnology	0793
Microbiology	0410
Molecular	0307
Neuroscience	0317
Oceanography	0416
Physiology	0433
Radiation	0821
Veterinary Science	0778
Zoology	0472
Biophysics	
General	0786
Medical	0760

EARTH SCIENCES

Biogeochemistry	0425
Geochemistry	0996

Geodesy	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0345
Paleoecology	0426
Paleontology	0418
Paleozoology	0985
Polynology	0427
Physical Geography	0368
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences	0768
Health Sciences	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0382
Public Health	0573
Radiology	0574
Recreation	0575

Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

Pure Sciences	
Chemistry	
General	0485
Agricultural	0749
Analytical	0486
Biochemistry	0487
Inorganic	0488
Nuclear	0738
Organic	0490
Pharmaceutical	0491
Physical	0494
Polymer	0495
Mathematics	0754
Physics	0405
General	0605
Acoustics	0986
Astronomy and Astrophysics	0606
Atmospheric Science	0608
Atomic	0748
Electronics and Electricity	0607
Elementary Particles and High Energy	0798
Fluid and Plasma	0759
Molecular	0606
Nuclear	0610
Optics	0752
Radiation	0756
Solid State	0611
Statistics	0463
Applied Sciences	
Applied Mechanics	0346
Computer Science	0984

Engineering	
General	0537
Aerospace	0538
Agricultural	0539
Automotive	0540
Biomedical	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Heat and Thermodynamics	0348
Hydraulic	0545
Industrial	0546
Marine	0547
Materials Science	0794
Mechanical	0548
Metallurgy	0743
Mining	0551
Nuclear	0552
Packaging	0549
Petroleum	0765
Sanitary and Municipal	0554
System Science	0790
Geotechnology	0428
Operations Research	0796
Plastics Technology	0795
Textile Technology	0994

PSYCHOLOGY

General	0621
Behavioral	0384
Clinical	0622
Developmental	0620
Experimental	0623
Industrial	0624
Personality	0625
Physiological	0989
Psychobiology	0349
Psychometrics	0632
Social	0451

The undersigned hereby recommend
to the Faculty of Graduate Studies and Research
acceptance of the thesis

**An Object-Oriented Reflective Simulation Environment for
Distributed Algorithms**

submitted by Adriana Lopes Diaz, B.Eng. M.C.S.
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Director, School of Computer Science

Thesis Supervisor

External Examiner

E. Annand

Carleton University

July 30, 1996

ABSTRACT

In recent years, distributed computing has received considerable attention in both academia and industry. In particular, the importance of simulating distributed algorithms (for validation and demonstration purposes) is now commonly acknowledged. But the inherent complexity and concurrency of distributed algorithms makes them difficult to simulate. In order to address this problem, the use of computational reflection has been proposed. Given the widespread adoption of the object-oriented paradigm for software development, it follows that the simulation of distributed algorithms should be tackled using a concurrent reflective and object-oriented programming language.

In this dissertation, we investigate this hypothesis. More specifically, we study the relevance of the concurrent reflective object-oriented programming language ABCL/R2 for the simulation of a representative set of distributed algorithms. Our work is three-fold: We first motivate and introduce our set of representative target algorithms. We then justify our choice of ABCL/R2 as the language of implementation. To do so, we survey relevant programming languages and libraries available for object-oriented simulation. But, most importantly, we discuss at length the numerous simulations we have implemented in ABCL/R2. Finally, we conclude by proposing a methodology for the design of simulations of distributed algorithms in a concurrent reflective object-oriented programming language. Our thesis is that this body of work does indeed confirm the relevance and importance of using a concurrent reflective object-oriented programming language for the simulation of distributed algorithms.

To the memory of my grandmother Olga
and
to my husband Donald

ACKNOWLEDGEMENTS

- To Capes for financial support, which enabled me to achieve my goals;
- To my supervisor, professor Jean-Pierre Corriveau for his help, advice, and trust;
- To my husband Donald Corcoran for his unconditional moral support;
- To my family and friends from Brazil and Canada for the words of motivation;
- To Valerie Chaplain for her words of encouragement;
- To professor Eshrat Arjomandi for agreeing to serve as external examiner and for her comments;
- To professors Evangelos Kranakis, Ali Mili and Dorina Petriu for agreeing to serve as examiners and for their comments;
- To all faculty and staff members of Carleton University, in particular Marlene Elliott, Marlene Wilson and Barbara Coleman of the School of Computer Science and Cicely Rayne and Callista Kelly of Interlibrary loans.

TABLE OF CONTENTS

1.0.	Introduction Motivation and Summary of the Dissertation.....	1
Part I - Background		
2.0.	Issues in Distributed Computing.....	9
2.1.	Definitions.....	9
2.2.	Election.....	12
2.3.	Deadlock Detection.....	12
2.4.	Distributed Mutual Exclusion.....	13
2.5.	Consensus in the Presence of Faults and Fault Tolerance.....	13
2.5.1.	Consensus in the Presence of Faults.....	15
2.5.2.	The State Machine Approach.....	16
2.5.3.	The Engineering of Fault-Tolerant Distributed Systems.....	18
3.0.	Reflection.....	20
3.1.	Introduction.....	20
3.2.	Definitions.....	22
3.3.	Reflection in Programming Languages.....	26
3.4.	Applications of Object-oriented Reflection.....	27
4.0.	The Object-oriented Paradigm and Simulation.....	36
4.1.	Introduction.....	36
4.2.	Simulation and Modeling.....	38
4.3.	Object-oriented Simulation.....	40
4.4.	The Relevance of the Object-oriented Paradigm for Simulating Distributed Algorithms.....	42
4.5.	Simulation of Distributed Algorithms.....	43
4.6.	Simulation Languages.....	48

Part II - Simulation of Distributed Algorithms and Object oriented Reflection

5.0.	Simulation Experiments using ABCI/R2.....	63
5.1.	Introduction.....	63
5.2.	Election in a Unidirectional Ring.....	64
5.3.	Communication Deadlock Detection.....	73
5.4.	Distributed Mutual Exclusion.....	84
5.5.	Election in a Complete Network with Sense of Direction.....	95
5.6.	Byzantine Agreement.....	102
5.7.	Fault Tolerant Mutual Exclusion.....	109
5.8.	An Object-oriented Reflective Approach Versus a Class based Approach for Simulating Distributed Algorithms.....	126
6.0.	Reflective Object-oriented Design of Simulations of Distributed Algorithms and a Concurrent Reflective Object-oriented Architecture for Simulating Distributed Algorithms.....	132
6.1.	ROOD - Reflective Object-oriented Design Applied to the Simulations of Distributed Algorithms.....	132
6.2.	The extended ROOD Methodology, a Generalization of the ROOD Methodology.....	164
6.3.	Fault-Tolerant-Reflective-Hybrid-Group (FTRHG) Architecture - A Concurrent Reflective Object-oriented Architecture to Study Fault-tolerance in Distributed Computing.....	170
7.0.	Conclusion.....	175
7.1.	Future Work.....	177
	References.....	178

LIST OF TABLES

Table	Description	Page
1	Comparison of Object-oriented Simulation Environments.....	61

LIST OF FIGURES

Figure	Description	Page
1	Description of the responsibilities accomplished in each level for the simulation of the election algorithm of Chang and Roberts.....	70
2	Messages exchanged during the execution of the election algorithm of Chang and Roberts considering four nodes and node1 as the initiator of the election.....	71
3a	Base-level interaction diagram for the example of execution of the algorithm of Chang and Roberts illustrated in figure 2.....	72
3b	Meta-level interaction diagram for the example of execution of the algorithm of Chang and Roberts illustrated in figure 2.....	72
4	Description of the responsibilities accomplished in each level for the simulation of the algorithm for communication deadlock detection of Chandy, Misra and Haas.....	81

5	Dependent sets and messages exchanged during the execution of the algorithm for communication deadlock detection of Chandy, Misra and Haas considering four nodes.....	82
6a	Base-level interaction diagram for the example of execution of the algorithm of Chandy, Misra and Haas illustrated in figure 5.....	83
6b	Meta-level and metameta-level interaction diagrams for the example of execution of the algorithm of Chandy, Misra and Haas illustrated in figure 5.....	84
7	Description of the responsibilities accomplished in each level for the simulation of the algorithm for distributed mutual exclusion of Ricart and Agrawala.....	92
8	Messages exchanged during the execution of the algorithm for distributed mutual exclusion of Ricart and Agrawala considering three nodes.....	93
9a	Base-level interaction diagram for the example of execution of the algorithm of Ricart and Agrawala illustrated in figure 8.....	94
9b	Meta-level interaction diagram for the example of execution of the algorithm of Ricart and Agrawala illustrated in figure 8.....	95
10	Description of the responsibilities accomplished in each level for the simulation of the election algorithm of Loui, Matsushita and West.....	99

11	Messages exchanged during the execution of the election algorithm of Loui, Matsushita and West considering eight nodes.....	100
12a	Base-level interaction diagram for the example of execution of the algorithm of Loui, Matsushita and West illustrated in figure 11.....	101
12b	Meta-level interaction diagram for the example of execution of the algorithm of Loui, Matsushita and West illustrated in figure 11.....	102
13	Description of the responsibilities accomplished in each level for the simulation of the algorithm for Byzantine agreement of Lamport, Shostak and Pease.....	107
14	Messages exchanged during the execution of the algorithm for Byzantine agreement of Lamport, Shostak and Pease considering four nodes with two faulty nodes: the initiator and node 3 (they corrupt a received message).....	108
15	Description of the responsibilities accomplished in each level for the simulation of the algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu.....	119
16	Messages exchanged during the execution of the algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu considering six nodes and no faults.....	120

17a	Base-level interaction diagram for the example of execution of the algorithm of Chang, Singhal and Liu illustrated in figure 16.....	121
17b	Metameta-level interaction diagram for the example of execution of the algorithm of Chang, Singhal and Liu illustrated in figure 16.....	122
18	Messages exchanged during the execution of the algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu considering six nodes and link failures on links 2-3 and 1-2 at the time that the token must be transfered along these links.....	123
19a	Base-level interaction diagram for the example of execution of the algorithm of Chang, Singhal and Liu illustrated in figures 18.....	124
19b	Metameta-level and meta-level interaction diagram for the example of execution of the algorithm of Chang, Singhal and Liu illustrated in figures 18.	125
20	Counting the number of messages received by an object.....	129
21	Simulation of non-FIFO reception of messages.....	129
22a	Reasoning about messages and performing changes in one level.....	130
22b	Reasoning about messages and performing changes in two levels.....	130

23a	Simulation of a corrupted message.....	131
23b	Simulation of omission on reception of a message.....	131
24	Choosing the levels in which to implement each algorithm of the example of section 6.1.2.1.....	159
25	Reflective object-oriented design of the example of section 6.1.2.1 using the ROOD model.....	160
26	Reflective object-oriented design of the example of section 6.1.2.1 using the extended ROOD model.....	169
27	Unified architecture model of ABCL/R2 and state machine replicas.....	174
A1	Messages exchanged during the execution of the algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu considering ten nodes and link failures on links 6-1, 7-4 and 9-8 at the time that the token must be transfered along these links.....	202
A2	Base-level interaction diagram for the example of execution of the algorithm of Chang, Singhal and Liu illustrated in figure A1.....	203

LIST OF APPENDICES

Appendix A	Additional Simulation Results.....	198
Appendix B	Code of Some Simulation Experiments.....	217

1.0. Introduction, Motivation and Summary of the Dissertation

This dissertation is an attempt to find applications for object-oriented reflection (or reflection¹ in object-oriented programming) by proposing a new approach to the simulation of distributed algorithms with educational purposes. The initial motivation of our work comes from Yonezawa and Watarabe (1988): "We think that developing new applications of reflective capabilities is one of the most interesting research topics" and from Maes (1987): "We need to experiment with reflection in order to distinguish its positive and negative uses". It is our thesis that an object-oriented reflective approach to the simulation of distributed algorithms constitutes an original contribution towards advancing the use of reflection.

In recent years, there has been a strong emphasis on the use of object-oriented paradigm in distributed computing. The power of the object-oriented approach to provide abstractions in helping to understand distributed systems is known and has been exploited. Examples of an "object-oriented distributed approach" can be found in several areas:

a) Programming Languages: Distributed Smalltalk (Bennett 1990), Distributed C++ (Carr et al. 1993), Distributed Eiffel (Gunaseelan and LeBlanc Jr. 1992), Argus (Liskov and Scheifler 1983), Trellis/Dowl (Achauer 1993), Kitara (Guffick and Blair 1989), Vulcan (Kahn et al. 1987), Polka (Davison 1989), Mandala (Ohki et al. 1987), L2IO2 (Pimentel and Troya 1991), Extended Parlog (Diaz et al. 1992);

b) Operating Systems: Mach (Jones and Rashid 1986), Amoeba (Mullender 1985), Ra (Auban et al. 1987), Eden (Almes et al. 1985), Comandos (Horn and Krakowiak 1987), Emerald (Black et al. 1986);

¹ "Reflection is the process of reasoning about and/or acting upon oneself" (Maes 1987)

c) Software Engineering: Protob (Baldassari and Bruno 1991), Object-oriented SDL (Moretti 1990); and

d) Simulation: (Denney 1991), (Bischak and Roberts 1991), (Ozden 1990), (Bezivin 1987), (Roberts and Heim 1988) and (Rogers 1991).

A distributed computing system can be informally defined as "a collection of autonomous processors that share no memory, do not have access to a global clock and communicate only by exchanging messages" (Babaoglu 1990). Throughout this dissertation we will be referring to each autonomous processor as a processing node of a network of processing nodes.

Designing and verifying distributed programs can be very difficult, because such programs are complex. This complexity is partially due to the large amount of components to be considered: variables of each node's computation, messages and processes. Also random communication delays and the relative speed of components make a distributed computation inherently non-deterministic (Tel 1991). In addition, since there is no global clock, it is also hard to visualize the global computation as an aggregate of local computations. If faults² are considered, the complexity becomes even higher. As opposed to parallel computing, where the main concern is performance, distributed computing has to deal with uncertainties. Contrary to sequential, centralized computing where given the program and the input only a single computation is possible, distributed computing can present distinct executions for the same input (initial state) and program. Consequently,

² "Fault is the identified or hypothesized cause of the error, and possibly a failure" (Mili 1990). "A failure occurs when the observed behavior of a program is inconsistent with the correctness of this program with respect to a given specification" (Mili 1990). "We speak of a failure (or fault) when a component shows a behavior that deviates from its program or specification" (Tel 1991)

verifying that a distributed algorithm is correct means to prove that all its possible executions are correct (Tel 1991).

The object-oriented distributed approach can be considered an attempt to use the object-oriented paradigm (Budd 1991) in order to deal with the complexity of distributed computing: "The Object-Oriented approach attempts to manage the complexity inherent in real-world problems by abstracting out knowledge, and encapsulating it within objects. Finding or creating these objects is a problem of structuring knowledge and activities." (Wirfs-Brock et al. 1990). Objects have state and behavior. Also, they communicate through message passing, an approach that matches the model of distributed computing.

In our opinion, reflection (the concept of reflection is explained in Chapter 3) seems to be the next step to cope with the complexity of distributed computing: ". . ., the evolution of programming languages is characterised by a search for more modularity. We believe that explicit support for reflective computation is such a step towards more modularity." (Maes 1987). "The benefits of computational reflection are the abilities to reason and alter the dynamic behavior of computation from within the language framework. This is more beneficial in concurrent/distributed computing, where the complexity of the system is much greater compared to sequential computing;. . ." (Matsuoka et al. 1991b).

Some issues can be identified as specific to distributed computing as opposed to sequential computing. Distributed algorithms have been developed in order to cope with these issues:

- a) Election: (Peterson 1982), (Chang and Roberts 1979), (Franklin 1982), (Hirschberg and Sinclair 1980), (Loui et al 1986);
- b) Distributed termination: (Topor 1984), (Dijkstra et al 1983), (Dijkstra and Scholten 1980), (Rana 1983), (Misra and Chandy 1982);

c) Distributed deadlock detection (resources and communication deadlock): (Chandy et al 1983), (Natarajan 1984), (Kroger et al 1991), (Song et al 1989);

d) Distributed mutual exclusion: (Suzuki and Kasami 1985), (Ricart and Agrawala 1981), (Carvalho and Roucairol 1983);

e) Consensus in the presence of faults: (Lamport et al 1982), (Fischer et al 1986), (Dolev and Strong 1983), (Fischer et al 1985), (Pease et al 1980), (Dolev et al 1987), (Turek and Shasha 1992).

Chapter 2 provides some definitions that are necessary in order to understand some of those algorithms, whose pseudo-code and brief description are presented in Chapter 5.

In recent years, some researchers have shown the importance of simulating distributed algorithms. For example, Love Jr. and Aburdene (1990) believe that a simulation is able to confirm the analysis for algorithms already defined, or provides an idea of the behavior and value of an algorithm without analysis. In addition, Boenheim (1992) has done several simulation experiments to analyze in depth the performance of different rules in order to choose victims to break deadlocks. Also, Flatebo and Datta (1992) have provided experimental analysis of various self-stabilizing algorithms in order to help determine the efficiency of these algorithms and to compare algorithms which solve the same problem. This is necessary since the complexity of self-stabilizing algorithms is difficult to determine. In the opinion of Hirota, Umemoto, Aibara, Yamashita and Ae (1993) simulation is important, since the application of formal methods for showing properties (e.g. correctness, termination, deadlock freedom) of distributed algorithms is not an easy task. We also believe that simulating distributed algorithms for educational purposes is important in order to provide a way for students to better understand these algorithms.

Beyond trying to find an application for object-oriented reflection, we view finding a simulation environment for distributed algorithms as a second goal of our work. We wanted to have an environment where simulations of distributed algorithms could be developed using the advantages that reflection and object-oriented programming can provide:

From object-oriented programming:

- 1) High-level abstractions for modeling complex issues,
- 2) A model that matches the one of distributed computing (nodes as objects that communicate solely by exchanging messages),
- 3) Inheritance.

From reflection:

- 4) The ability to reason about and model dynamic changes in the computation,
- 5) Structured development of open systems.

We decided to use the object-oriented concurrent reflective language ABCL/R2 (Matsuoka et al. 1991b) (Masuhara et al. 1992) to implement some distributed algorithms in order to illustrate the advantages of reflection to the simulation of distributed algorithms. ABCL/R2 appears to be the most appropriate language for our research and implementation purposes because it provides synchronous and asynchronous message passing, reflective capabilities and objects are the units of concurrency. Since the simulation of distributed algorithms requires a simulated network of processing nodes that run in parallel, it makes sense to map these nodes to objects that run concurrently, as in ABCL/R2. We did not consider using object-oriented concurrent languages like Smalltalk-80 (Hopkins and Wolczko 1989) and ConcurrentSmalltalk (Yokote and Tokoro 1987), because in these languages objects are not the units of concurrency, and these languages do not provide

reflective capabilities (see Chapter 4 for further discussion). In our opinion, the reflective architecture of ABCL/R2 provides built-in semantics that would represent a great deal of implementation work in other languages. A similar argument is presented by Rumbaugh. Inheritance is considered one of the main advantages of object-oriented programming. Rumbaugh shows in (Rumbaugh et al 1991) Chapter 16, Section 16.5 the amount of work and the difficulties in implementing inheritance in non-object-oriented languages such as: C, Ada and Fortran. It can be done but it somewhat defeats the purpose of using inheritance.

Maes (1987) showed the relevance of reflection in general. We also have addressed the relevance of object-oriented reflection to the simulation of distributed algorithms in (Diaz and Corriveau 1995a). Maes mentioned that "programming languages do not provide adequate support for its modular implementation. For example, if the programmer wants to follow temporarily the computation, e.g. during debugging, he often changes his program by adding extra statements. When finishing debugging, these statements have to be removed again from the source code, often resulting in new errors." Another example that Maes provides is in simulation, where the code which describes the problem is mingled with the code that specifies how statistics on the simulation should be stored, how the graphical output should be constructed etc. The reflective solution for that would be to have the simulator comprise object-level modules (simulation domain code) and reflective modules (statistics code, graphical output code etc.).

Based on the previous discussion, the main goal of this dissertation is to justify the use of a concurrent reflective object-oriented language for simulating distributed algorithms in order to understand better the complex issues of distributed computing. In order to accomplish this goal, we surveyed concurrent object-oriented languages and libraries

available for simulation. We also performed some simulation experiments. This research can be considered a complement to the analysis of distributed algorithms. We restrict ourselves to discussing typical distributed algorithms for election, communication deadlock detection, mutual exclusion and consensus in the presence of faults. Another contribution of this dissertation is a methodology to obtain the object-oriented reflective design of distributed algorithms while avoiding the disadvantages of reflection mentioned by Maes (1987) and discussed in Section 3.1. In addition, particular attention will be given to the issues related to fault tolerance (see Section 6.3) by obtaining a simulation environment that unifies an object-oriented concurrent reflective architecture and the state machine model (Schneider 1990), which has been used for studying fault tolerance in distributed systems.

The rest of this dissertation is organized as follows: Chapter 2 provides an overview of the basic necessary concepts and terminology in order for the reader to understand the descriptions and discussion related to distributed algorithms presented in Chapters 5 and 6. Chapter 3 begins by presenting basic terminology and definitions relevant to reflection. In addition, it contains a summary of some applications of object-oriented reflection developed to date in order to illustrate that research on reflective object-oriented simulation of distributed algorithms seems to be absent. Chapter 4 is about object-oriented simulation and simulation of distributed algorithms. In that chapter we explain the reasons why we did not choose some of the available object-oriented languages to simulate distributed algorithms. We also present an overview of some non-object-oriented computing environments used for simulating distributed algorithms. We conclude this chapter by motivating our choice of the language ABCL/R2 to simulate distributed algorithms. In Chapter 5 we present our experiments in simulating six distributed algorithms using ABCL/R2. The advantages of the reflective object-oriented approach are

discussed for each simulation. Interaction diagrams (Jacobson et al. 1992) are used as the visualization technique for presenting the results of the simulations. Our simulation experiments cover some important issues of distributed computing: election of a leader, distributed mutual exclusion, communication deadlock detection, consensus in the presence of faults and fault-tolerant algorithms. We conclude Chapter 5 by providing a comparison between the non-reflective object-oriented class-based approach and the reflective object-oriented approach in coping with some situations that can arise in the simulation of distributed algorithms. Chapter 6 introduces our methodology for reflective object-oriented design of simulations of distributed algorithms and includes a proposal for a generalization of this methodology. Our design methodology is based on the responsibility-driven approach used in object-oriented design by Wirfs-Brock et al. (1990). We also propose a reflective object-oriented architecture to study fault tolerance in distributed computing based on the architecture of ABCL/R2 and the state machine model (Schneider 1990). Chapter 7 presents our conclusions and suggestions for future work. Appendix A contains the results of simulation experiments; the algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu (1990) considering ten nodes and link failures and thirty seven cases of Byzantine agreement (Lamport et al. 1982). Appendix B contains the simulation code of experiments with the algorithm for communication deadlock detection of Chandy, Misra and Haas (1983).

2.0. Issues in Distributed Computing

Section 2.1 provides some definitions and terminology related to distributed algorithms that are necessary to understand the material of Chapters 5 and 6. Sections 2.2 to 2.5 describe briefly the issues of distributed computing that are relevant to this dissertation, namely: election, communication deadlock detection, mutual exclusion and consensus in the presence of faults.

2.1. Definitions

In this dissertation we use the following definition of a distributed system provided by Tel (1994): "an interconnected collection of autonomous computers, processes, or processors. The computers, processes or processors are referred to as the nodes of the distributed system." Interconnected means that all nodes must be able to exchange information. Autonomous means that all nodes must have their own private control.

The majority of the definitions provided in the following paragraphs are from (Tel 1991).

Distributed algorithms have been developed based on two models: synchronous and asynchronous. In the synchronous model an algorithm is executed in discrete steps, called rounds (cycles, phases). In this case, a process (processor) starts the computation of round $(i+1)$ as long as it has received all messages that were sent to it in round i . This model resembles a system with a global clock, providing pulses at regular intervals. The asynchronous model does not assume any global clock and message delivery time is assumed to be unbounded.

A synchronous distributed system is one where the operation of the entire system takes place in system-wide cycles. The steps of all processing units are synchronized to take place simultaneously. An asynchronous system does not present any synchronization (unless some form of control is used through message exchanges).

In asynchronous distributed systems, synchronous and asynchronous messages can be exchanged. Synchronous message passing occurs when the sender is ready to send a message and the receiver is ready to receive a message. It is thus a blocking send and blocking receive operation: the execution of the sender and the receiver are deferred until the communication takes place. After that, sender and receiver proceed with their computations. On the other hand, in asynchronous message passing, the sender is not blocked when it wishes to send a message as it sends the message and continues executing (non-blocking send operation). In this case, there is a non-zero finite message delivery time and a receive operation is blocking: the execution of the receiver is deferred when it needs to receive the message. So, synchronous message passing ensures that sending and receiving a certain message take place at the same time while asynchronous message passing ensures that receiving takes place after the sending of a message.

A channel is a connection between the sender and the receiver used for the transport of messages. It can be unidirectional (transports messages from node A to node B but not vice versa) or bi-directional (transports messages from node A to node B and vice versa). A FIFO (First In, First Out) channel is one that allows messages to be received by node A in the same order that they have been sent by node B (in one direction).

In the analysis of distributed systems, it is very common to represent the topology of the network of processing nodes with a graph. The processing nodes are nodes of the graph and the edges are the communication channels.

A N -node ring has nodes v_0 through v_{N-1} and edges $v_i v_{i+1}$ for each i (addition modulo N). A ring is unidirectional if its edges are unidirectional, and bi-directional if its edges are bi-directional. A tree (of N nodes) is a connected bi-directional graph free of cycles with $N-1$ edges. A complete network is a graph in which every pair of nodes is directly connected by a bi-directional edge (or two unidirectional edges). Since a complete network has a large number of edges and each node has a large degree, it is never used as a physical connection structure of processor networks. Nevertheless, it is used as a logical structure for communication primitives on an arbitrary network. The actual physical structure of the network is hidden and (logically) replaced by the complete structure.

In distributed systems, it is not possible for a processing node to determine the global order of events in the whole system. Each processing node has a local clock and can determine a partial ordering of events. If physical clocks are considered, they have to be synchronized (run at the exact same rate) in order to provide an accurate reference for the ordering of events. A local logical clock is a simpler solution to have a reference for the ordering of events (Lamport 1978). Local logical clocks are updated when processors communicate with each other. A very common way of establishing a total order between all the events in the system is to use the time stamping mechanism. A time stamp is the value of the local logical clock of the sender node. Thus, when a message is transmitted from processing node P_i to processing node P_j , P_i stamps the message with the value of its local clock c_i . When P_j receives the message it compares the value of its local clock c_j with the time stamp t_i in the message. If c_j is greater than t_i , nothing is done. If c_j is smaller than t_i , then c_j is updated with the value of t_i . These rules always guarantee that transmission precedes reception of a message. If c_j is equal to t_i , then a comparison of identifiers (a number that identifies each processing node of the network) is necessary in order to

establish an order. Thus, event $E1$ that occurred on site i timestamped $ci(E1)$ precedes $E2$ that occurred on site j and timestamped $cj(E2)$ if and only if: $ci(E1) < cj(E2)$ or [$ci(E1) = cj(E2)$ and $i < j$].

2.2. Election

Election of a leader (Peterson 1982), (Chang and Roberts 1979), (Franklin 1982), (Hirschberg and Sinclair 1980), (Loui et al 1986) in distributed computing occurs when it is necessary to elect a co-ordinator to reorganize a network in the case of failures. The election algorithm makes only one node have the status of "elected". According to Tel (1991), considerable attention has been given to the problem of electing a leader in a network of processors.

2.3. Deadlock Detection

It is more difficult to detect deadlocks in distributed systems (Chandy et al 1983), (Natarajan 1984), (Kroger et al 1991), (Song et al 1989) than in centralized systems, because in distributed systems there is no central agent that has complete information about every process. There are two types of deadlock: resource deadlock and communication deadlock. Resource deadlock is the situation where two or more processes are blocked forever because each need a resource that is held by the other. Resource deadlock is characterized by a directed cycle of dependent processes. Communication deadlock is

characterized by a process waiting forever for one or more messages to resume its computation. This situation corresponds to a knot¹ in the graph of waiting processes.

2.4. Distributed Mutual Exclusion

Mutual Exclusion (Suzuki and Kasami 1985), (Ricart and Agrawala 1981), (Carvalho and Roucairol 1983) pertains to controlling access to a certain resource or critical section of code (i.e. only one process at a time must have access to a certain resource or critical section). In a distributed environment without shared memory, mutual exclusion must be achieved with processors sending information through message passing. In addition, equitability and freedom from mutual blocking must be guaranteed (Raynal 1988). This means that any process that wishes to enter the critical section must be able to do so within a finite time.

2.5. Consensus in the Presence of Faults and Fault Tolerance

Consensus is an abstraction of a problem found in designing reliable systems. It is related to achieving fault tolerance by considering a model of replicated data (see Section 2.5.2) that is used by several processors to compute a desired output using a majority voting scheme. The idea is to have faulty processors not affect the output and to have non-faulty processors able to detect faulty processors.

¹ "Two terms, tie and knot, are widely used in dealing with deadlocks. A tie in G is a nonempty set S of idle nodes, with no links from S to $N-S$ " (N is the node set). "A knot is a tie of which any proper subset is not a tie" (Huang 1990)

In distributed systems there are two complementary strategies for coping with failures: the fault avoidance approach in which a system is composed from components that are less likely to fail and the fault tolerance approach in which a system is supposed to continue functioning correctly despite failures. The analysis of fault tolerance (Babaoglu 1990) usually only considers faults due to faulty hardware.

2.5.1. Consensus in the Presence of Faults

Consensus in distributed systems is the situation where a set of agents must agree on a decision. Typically, only two decisions are allowed, but a protocol for two decisions can be extended for any number of decisions. These decisions can be, for instance, "abort" or "commit" in a distributed database system. Formally, as defined by (Babaoglu 1990), a system with n processors where each processor i has an initial local value v_i achieves consensus when a consensus protocol is terminated and the two following conditions are satisfied after each processor computes an n -element vector:

- Agreement condition: all correct processors compute a common vector

$$W = (w_1, w_2, w_3, \dots, w_n);$$

- Validity condition: the j th component of the vector is equal to the initial local value of processor j if j is correct (i.e. $w_j = v_j$ for each j corresponding to a correct processor).

Each processor applies the same deterministic function to the vector W in order to obtain a final value (the final decision) to be used.

Consensus is applied in commit protocols for distributed databases and in ordered atomic broadcast protocols and is closely related to fault tolerance.

A fault-tolerant system is defined as a system that continues to function correctly despite failures. In such a system, resiliency is defined as the maximum number of processors that can fail while the system is still fault tolerant. Analogously, a consensus protocol is t -resilient if it works correctly when at most t processors fail.

Consensus can be easy or difficult to achieve depending on the computing system (synchronous or asynchronous) and on the type of failures considered (Turek and Shasha 1992). The problem of achieving consensus in the presence of Byzantine failures is known as Byzantine agreement (see Section 5.6). It is also known as unanimity or interactive consistency. An initiator of the Byzantine agreement protocol is the processor that sends a message with a value (that represents a decision) to all the other $n-1$ processors. The necessary conditions for Byzantine agreement can be briefly described as follows:

- All non-faulty processors take the same decision;
- If the initiator is non-faulty, then all the non faulty processors take the decision taken by the initiator.

In computer systems, authentication algorithms are the ones that guarantee that signatures of other processors included in received messages are not corrupted by the receiver before relaying these messages. Byzantine failures are not a problem in systems where authenticated messages are used. When there is no authentication, Pease, Shostak and Lamport (1980) show that deterministic Byzantine agreement is possible in a synchronous system if and only if there are at least $3k + 1$ processors (k is the number of processors that can fail) in a complete network.

Dolev and Strong (1983) showed that using authenticated messages, it is possible to reach Byzantine agreement in synchronous systems with at most t faults, assuming that n

$> (t + 1)$. They also proved that $(t + 1)$ phases of message exchanges are necessary independent of whether authenticated messages are used or not.

For asynchronous systems (messages have unbounded but finite transit times), Fischer, Lynch and Paterson (1985) proved that there is no deterministic solution for the Byzantine agreement problem even for a single failure.

Probabilistic algorithms or randomized algorithms are the ones that contain statements of the form: " $x := \text{outcome of tossing a fair coin}$ ". These algorithms use n -sided coins because the tossing is to allow multiway decisions (generation of a random number between 1 and n , in computational terms). The action of tossing a coin is often implicit in a randomized algorithm, such as randomly selecting an item x from a set S or randomly choosing a process with which to communicate. Ben-Or's (1983) randomized algorithm provides the simplest solution for the Byzantine agreement problem in asynchronous systems. The main idea of this algorithm is to use the fact that if independent random choices are made by each processor as a consensus value, eventually a sufficient number of them will pick the same value to allow agreement among correct processors. An agreement is guaranteed if the number of faulty processors, t , is less than one-fifth of the total number of processors.

2.5.2. The State Machine Approach

The state machine (Schneider 1990) is known as a model to implement fault-tolerant systems through the replication of processors. It is used to mask the effects of failures using voting on the outputs of replicas of the computation. In this model, these replicas run on different processors that are physically isolated. A state machine can be considered as

state variables and commands which modify its state. From an object-oriented viewpoint this machine is an object: it has state and behavior. A state machine is not a finite-state automata, because it can present an infinite number of states.

The state machine approach has been used to describe the behavior of distributed computations, since it is a general method to achieve fault tolerance through replication. The replicated system has the properties of a distributed system namely, autonomous processors with no shared memory that communicate by message exchange. Using this approach, a distributed computation can be seen as a collection of objects that on receiving messages perform some computation, might change state or not, and might send messages or not. These operations depend on the object's present state and/or on the received message.

Models that are developed to study fault tolerance can consider the following types of failure: *crash*, when a processor stops running and never resumes; *omission*, when a processor does not send some messages that it is supposed to, but the messages that it sends are always correct; *fail-stop* failures, when a processor before stopping, changes its state so that other processors can detect its failure, finally *Byzantine*, when a processor behaves in an arbitrary way, failing to send some messages, sending messages incorrectly, or even acting arbitrarily and maliciously with other faulty processors to bring the system to a state of confusion (Babaoglu 1990).

Fault-tolerant state machines are implemented by replicating the state machines and running each copy on each of the processors of a distributed system. Copies that run in non-faulty processors, starting in the same initial state and executing the same requests in the same order produce the same output. For faulty processors, assuming that each failure can affect at most one processor (one state machine copy) an agreement has to exist on all

the state machine copies for an output. How this agreement is achieved depends on the type of failure considered.

2.5.3. The Engineering of Fault Tolerant Distributed Systems

According to Babaoglu (1990) there is enough knowledge to incorporate fault tolerance into a large class of applications without having to consider different algorithms for each case. His work aimed at accomplishing the design of fault tolerant distributed computing systems as a traditional engineering process. Also according to Babaoglu, this process requires "solving the appropriate sub-problems, understanding the interactions between the proposed solutions and the properties of the resulting system, and developing the measures necessary for exploring as wide a range of designs as possible in searching trade-offs". Furthermore, Babaoglu believes that there are the necessary theoretical foundations for the engineering of fault-tolerant computing systems, but "what we lack is an understanding of the interactions between the design parameters and the system properties. This is essential if we hope to effectively explore the design space in search of trade-offs". He also stated that the key aspect of the design for a fault-tolerant system (not considering other aspects like total cost and performance) is to guarantee a lower bound on the probability that the system will keep performing correctly after it has been operating for a certain amount of time (i.e. reliability²). He presented data that shows that reliability is a metric that is more suitable than resiliency for the design of a fault-tolerant computing system. His conclusion is: "Although we were able to evaluate several design choices

² "The reliability of a system as a function of time, $R(t)$, is the conditional probability that the system has survived the interval $[0,t]$, given that the system was operational at time $t = 0$ " (Siewiorek and Swarz 1992)

individually using the system reliability as the criterion, a realistic endeavor has to cope with alternatives resulting from varying several parameters simultaneously". These parameters are typically the replication level (number of replicas of the computation), the number of rounds (number of phases of message exchange) necessary to achieve agreement on the output of the replicas and the time interval between failure detections.

The work of Babaoglu suggests that an environment to simulate distributed systems subject to occurrences of different types of failures would be helpful to study fault tolerance.

3.0. Reflection

3.1. Introduction

The concept of reflection has its roots in Artificial Intelligence. Maes (1987) pioneered the work in reflection in object-oriented programming (or object-oriented reflection) by introducing reflection to the object-oriented language KRS (Steels 1986), creating the language 3-KRS. She pointed out that most computational systems exhibit reflective computation (computation about their own computation) in addition to computation about their problem domain. This reflective computation can serve the following purposes: storing performance statistics, storing information for debugging, stepping and tracing, computing what should be done next (for non-deterministic programs), self-optimization, self-modification (as in learning systems). Since a reflective system is able to make modifications to itself, control over computation is shifted from the programmer to the computational system. In Chapter II, Section 6 of her thesis, Maes pointed out the dangers of reflection: less straightforward programming, loss of information, and difficulty in understanding what the computational system actually does.

Programming in a reflective system is less straightforward than ordinary programming because it involves programming several levels of computation. Since the different levels of computation may act upon one another, the computation of a specific level is not only determined by the code for that level, but also by the code of the reflective levels above that level.

The "shift of control" from the programmer to the system happens when the system handles erroneous situations and exceptions on its own without informing the user or the

programmer about them. This implies loss of information. For this reason Maes suggested not to use reflection for exception-handling during the development phase of a system.

Finally, the difficulty in understanding what the computational system actually does is due to the fact that this system may modify itself.

In Ibrahim's (1992) opinion, reflection is very well suited to object-oriented languages for three reasons. The separation that exists between the representation of state and behavior in objects facilitates the implementation of a meta-object that can reflect on the behavior of its associated base-level object (domain object). In addition, encapsulation provides a way for a meta-object to intercept messages before they reach the target object and to reflect on the behavior of this target object. Also, the concept of meta classes that most object-oriented languages have is a mechanism for describing and representing the structure and the computation of the system classes which control the structure and behavior of the application objects.

The first part of this chapter (Sections 3.2 and 3.3) presents some basic terminology on reflection in order to help the reader understand Chapters 5 and 6. The second part (Section 3.4 and its subsections) provides a summary of some applications of object-oriented reflection developed to date. This summary illustrates the fact that research on reflective object-oriented simulation of distributed algorithms seems to be absent.

3.2. Definitions

The following definitions were taken from (Ibrahim 1992) except the ones that contain a reference to other authors.

3.2.1. Definition of Reflection

Smith (1985) defined a reflection hypothesis as follows:

"In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures."

Smith refined this definition years later in a workshop about reflection (Ibrahim 1991):

"Reflection is an entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on, and deals with its primary subject matter."

A short definition was also provided by Maes (1987): "Reflection is the process of reasoning about and/or acting upon oneself".

3.2.2. Meta Level

In order to understand how reflection works it is necessary to assume that computation is performed in at least two levels: the base level and the meta level. The base level is where application computation (or domain computation) is performed. Reflective computation takes place when the system switches to a reflective mode when it interrupts its base-level computation and inspects and reasons about the base-level computation from its meta level.

3.2.3. Meta System

A meta system is a computational system¹ that reasons about and acts upon another computational system. The domain of a meta system is called its object system, which is represented as data of the meta system.

3.2.4. Causal Link

It is possible through reflective computation not only to inspect the base-level computation (application computation), but also to change it. Therefore, the results of changes have to be incorporated into the base-level computation as the system switches back to base-level computation. This relation between base level and meta level has been

¹ "A computational system is something that reasons about and acts upon some part of the world, called the domain of the system. A computational system represents its domain under the form of data. Its program prescribes how these data should be manipulated" (Maes 1987)

called causal link. Causal link is a necessary condition for reflection, since it allows reflective computation to influence and alter the application computation.

Maes (1987) uses the term "Causal Connection link" instead of "Causal link". A causally connected system presents such a link that is set once by the implementer. This guarantees that changes that result from the interaction between the computational system and its domain are incorporated automatically, without any human interaction. Maes provides the following formal definition of a causally connected system:

"(i) when data of the system change, the entities in the domain represented by these data are affected," and

"(ii) when entities in the domain change, the data representing these (or aspects of these) in the system are affected."

Thus, causal connection brings up the important specification question of what aspects of the entities in the domain of a system should be represented as data of the computational system.

3.2.5. Reification

Reification is a term used by Friedman and Wand (1984) to refer to the process of representing and passing the contents of the interpreter components to the application program, so that this program can manipulate them as data. If a system is able to reify, this means that its executing application programs can access and inspect the structure and computing environment of these programs. But there is no guarantee that changes on the inspected items will be incorporated to the executing application program. Consequently, reification is a necessary but not sufficient condition for reflection.

3.2.6. Introspection

Any description or reference to meta level computation is considered introspection. The degree of introspection has been defined (Des Rivieres and Smith 1984) as the number of meta levels of a program beyond which higher levels are of no practical value.

3.2.7. Reversion (Deification)

Reversion is the process by which the system returns to the computation that was being accomplished before reflective computation started. The causal connection link necessary for reflection is achieved this way.

3.2.8. The Reflective Tower

The reflective tower is the group of meta levels that exist in a system. The tower exists since each meta level, say meta level "A" can have a meta level above it, say meta level "AA", which can accomplish introspection about the computation performed in meta level "A".

3.2.9. Reflective system

As defined by Maes (1987), "A reflective system is a causally connected meta system that has as object-system itself." Maes says that what she calls reflection is called introspection by Smith (1986) (see page 35 of Maes' thesis).

3.2.10. Self-representation

The self-representation of a reflective system is the data that is a causally connected representation of itself.

3.2.11. Reflective Computation

A reflective computation is one performed by a system when reasoning about or acting upon itself.

3.3. Reflection in Programming Languages

According to Ibrahim (1992) there are two conditions that have to be fulfilled for a language to be considered reflective:

The first one is that the language must present internal structures that represent the constructs of the language itself and the programs written in this language in the same way that they represent the external domain.

The second one is that the interpreter of such a language must allow programs written in the language to access and manipulate the structures that represent the language, its programs and the state of computation.

Languages like FORTRAN and COBOL are not reflective, they do not fulfill the first condition. Lisp is reflective, because it uses list structures for representing both data and programs. Interpretive languages are reflective, in general (but not always), because there is a direct correspondence between the program code and the executable instructions.

In order for a program to be considered reflective it must be written in a reflective language and it must access the interpreter to modify itself at run time.

3.4. Applications of Object-oriented Reflection

The following subsections provide an overview of some of the research accomplishments with respect to applications for reflection in object-oriented programming. We observe that simulations of distributed algorithms using a reflective object-oriented approach is absent.

3.4.1. ReActalk

ReActalk (Giroux et al. 1993) is a reflective actor kernel for adaptation written on top of Smalltalk-80. Giroux, Senteni and Lapalme think that the object-oriented paradigm is important to heterogeneous systems because of encapsulation, preservation of locality and message passing. But they point out the necessity of integrating all those concepts with an adaptive-open-systems perspective, since adaptiveness and openness can be considered key characteristics to the construction of complex systems. The idea is to design parts as adaptive open systems, put them together to form a system, and let them co-operate and adapt to each other. The purpose of ReActalk is to provide a framework for the study of adaptation in the context of actors and open systems. In ReActalk reflection is used in the context of adaptive processes that evolve in a dynamic environment, as opposed to being used to provide hooks on the programming language and its environment for the user to access and customize them.

3.4.2. Octopus

Octopus (Farkas and Dearle 1993) is a reflective language mechanism that allows the types of values to be abstracted so that the values can be manipulated in a type-independent manner. Octopus stands for Object Closure Transplantable to Other Persistent User Spaces.

The purpose of Octopus is to solve the problem of operating over an infinite number of types, which happens in a certain class of database programs (e.g. object browsers and query tools).

Octopus implements the type-independent manipulation by allowing values from the programming language value space to be hoisted up to a meta level and manipulated in ways which the programming language would not permit. After manipulation is done, those values are dropped back into the value space if they still conform to the language's type system.

3.4.3. A Reflective Invocation Scheme to Realise Advanced Object Management

As a consequence of incorporating distribution in an object-oriented language, it is necessary to deal with object management issues, such as object migration, object replication and granularity control. Also, in a concurrent language, the concurrency control specifications defined for an object by the application programmer must be realised correctly. The reflective invocation scheme proposed by Bijmens, Joosen and Verbaeten (1994) transparently implements solutions for the issues of object migration, object

replication and granularity control in a C++ environment on various distributed memory platforms. The scheme which is based on reference objects, can be extended by a user in order to make it support additional features. This work is part of the XENOOPS (eXecution Environment for Object-Oriented Parallel Software) project (Berbers et al. 1991), an object-oriented language and system for development of applications on a distributed memory platform with automatic support for load balancing and fault tolerance through reusable class libraries which present different strategies.

3.4.4. Silica

Silica (Rao 1991) is the CLOS-based portable window system layer of the Common Lisp Interface manager. In Rao's opinion, Silica serves to show that reflection can be applied in a broad sense, not only to the design of programming languages. He introduces implementational reflection (as opposed to computational reflection) and open implementation (as opposed to a reflective architecture). Implementational reflection was defined by Rao as follows: "Reflection involves inspecting and/or manipulating the implementational structures of other systems used by a program". Rao, also stated that a system with an open implementation provides at least two linked interfaces to its clients: base-level interface, which is the usual interface to the functionality of the system provided by most systems and a meta-level interface, which provides access to some implementation aspects of the base-level interface. This access gives the user the power to add some code (at some specified points) that implements base-level behavior that is different from the default one (semantically and in terms of performance).

Silica presents a base-level interface and a meta-level interface. The base-level interface comprises windows that support input and output operations. The meta-level interface comprises components which implement base-level behavior and the object-oriented protocols which specify how these components interact.

3.4.5. The Reflective BirliX Object Model

The BirliX system (Sonntag et al. 1994) is a type management system with mechanisms for the definition and instantiation of types, and the communication of type instances. The smallest unit of identification and communication in BirliX is the type instance. Instances in BirliX may be considered as objects surrounded by infrastructural properties analogous to an object in a programming language system that encapsulates data and code. Interacting instances distributed among several nodes in a computer network constitute an application program in BirliX. BirliX type instances communicate by invoking methods. The motivation for incorporating reflection into the BirliX object model is to provide the objects with the ability to construct or reconstruct their infrastructure dynamically to support universal adaptability for all objects independent of their type. The reflective BirliX model separates functional (considered during the implementation of the applications) and infrastructural properties (considered during the configuration or reconfiguration of applications).

3.4.6. Lower Layer Concepts for Open-endedness and Dynamic Reconfiguration

Bekker and Putter (1993) presented in their paper some lower layer concepts that help modeling and designing systems ensuring qualities of open-endedness and dynamic reconfiguration by using reflection. Those qualities are important since the requirements for distributed systems tend to change. Consequently, these authors provide some insights about reflective architectures for implementing scalable distributed systems.

In object-oriented analysis and specification of a distributed system, an object can represent an abstraction of a physical device, a logical component or a collection of information. This model can be enhanced by providing meta objects which represent the structure and behavior of the system. Bekker and Putter consider a network management (NM) system to be a causally connected system: "the structure and behavior of a NM system depends on the management policy to which the system conforms, and changes in the policy should lead to changes in the system structure and behavior. In a similar fashion changes in the system structure and behavior must conform to the policy of the system". In their reflective NM system model, meta objects are added to managed objects at runtime. The openendedness of a distributed system is achieved by enhancing those meta objects without affecting their corresponding objects.

3.4.7. A Linguistic Framework for Dynamic Composition of Dependability Protocols

Agha, Frolund, Panwar and Sturman (1992) presented a language framework for describing dependable systems with emphasis on modularity and composition, i.e. a software methodology for supporting dependable services in open systems. One of the purposes of this framework is to achieve separation of design concerns; e.g. dependability features from functionality features. Composition is taken into account in order to support the construction and reuse of general dependability schemes. Another purpose of this framework is to load dependability protocols into a running application (dynamic installation) by using reflection. This can be useful in the situations when a certain dependability protocol is necessary only for some time (during some critical phase of execution). This should overcome the limitation of most methodologies for programming dependable applications where the dependability features of an application are fixed statically (at compilation time). Using MAUD (Meta-level Architecture for Ultra Dependability) created by Agha, Frolund, Panwar and Sturman, an application and its dependability protocols are linked together at compile time.

3.4.8. Muse

Muse (Yokote 1989) is an object model which provides a reflective architecture for an object-oriented distributed operating system (also called Muse). Therefore, the distributed operating system Muse is based on an object-oriented paradigm with a reflective architecture. Yokote, Teraoka and Tokoro claim that an operating system should provide

high-level abstractions to users, be seen as a virtual machine and considered as a programming environment. One of the motivations for implementing Muse is the fact that operating systems do not fulfill the user's requirements for expansion.

In Muse a meta object accepts requests from objects to perform primitive functions such as allocation of local storage and accessing external devices (the meta-object level provides a virtual machine environment to the base-level objects). Each meta meta object corresponds to a CPU and they can be of two types: machine dependent and machine independent. Muse can be ported by modifying only the machine dependent part for target machines. Also a class is a static and immutable entity in Muse. Inter-object communication is uniform (location independent communication) and there are two levels of scheduling: object scheduling (objects are scheduled by their corresponding meta objects) and meta-object scheduling (meta objects in the same host are scheduled by the meta meta object). Storage of objects is managed by their corresponding meta objects and the storage of meta objects is managed by their meta meta objects.

3.4.9. Apertos

Apertos (Yokote 1992) is an example of a distributed object-oriented operating system that uses concepts of reflective computing. The authors of Apertos believe that an operating system should be constructed using object-oriented technology in order to encapsulate in objects everything in the system that is shared and protected. On the other hand, they claim that the system should also provide mechanisms to inspect the internals of an object to facilitate debugging and object management (such as an invocation manager and a scheduler).

In Apertos each object has a metaspace constituted by meta objects. Every system service is implemented by meta objects, for instance, an object's virtual storage is a collection of segment meta objects and managed by a virtual segment manager meta object. These meta objects are also considered as objects, i.e., they have metaspaces as well. The framework in which Apertos is based on is characterized by object/meta object separation, metahierarchy and object migration (migrating an object is equivalent to transferring meta objects to the target metaspace).

3.4.10. RMondel

RMondel (Erradi et al. 1992) is a reflective object-oriented specification language for the description of distributed systems allowing the development of dynamically modifiable specifications. RMondel consists of: a user interface, a translator, a set of constraints, the kernel types and an RMondel interpreter. The user interface allows the user to compose and introduce changes to his specification. The translator takes an RMondel specification and produces a set of RMondel objects according to RMondel semantics. The RMondel interpreter uses a set of predefined static constraints in order to maintain the system in a consistent state. The kernel objects (also used by the interpreter) form a database of RMondel predefined objects. The interpreter is based on a formal semantics definition. Type definitions are accessed and modified at runtime in order to permit the implementation of dynamically modifiable specifications.

3.4.11. Meta-level Architecture Support for Distributed Objects

McAffer (1995) presented a meta-level architecture for transparently adding a wide variety of distributed behaviors to objects without substantially affecting their base-level behavior or code (i.e. "separate what an object does (its base-level) from how it does it (its meta-level)"). The motivation for his work is the importance of providing an environment which allows designers to explore and experiment with various computational and structural models for application objects. The main feature of McAffer's work is to provide the separation of the base-level application code from the meta-level object behavior (e.g. distribution) code. He developed Tj, a platform for describing distributed object behavior. Tj is built on top of CodA, a general meta-level architecture which provides access to the implementation of an object's basic computational aspects, e.g. state accessing, message sending, queuing, method execution. Tj's Distributed Object model comprises mechanisms to deal with object spaces, machines, topologies, remote references, object marshaling, replication and migration. Those mechanisms are open, they can be extended to include user's changes and additions. "A Tj distributed system is a collection of DistributedObjects living in object spaces running on machines and interconnected by transport mechanisms in some system topology".

4.0. The Object-oriented Paradigm and Simulation

4.1. Introduction

Budd (1991) wrote "in object-oriented programming, we have the view that computation is simulation".

The purpose of this chapter is to provide an overview of object-oriented simulation and simulation of distributed algorithms. This is necessary in order to introduce our research in reflective object-oriented simulation of distributed algorithms in Chapters 5 and 6 as an original contribution to the field of object-oriented reflection. Simulation of distributed algorithms can be seen as a special case of message-based discrete-event simulation (Bagrodia et al. 1987).

The material presented in this chapter shows that the contribution of the object-oriented paradigm to discrete event simulation is evident. On the other hand, the contribution of object-oriented reflection to the same field is not so clear, if not absent.

This chapter starts with some basic definitions of object-oriented modeling. Then, it provides some general concepts in simulation and modeling. Furthermore, some issues in object-oriented simulation as well as the contribution of the object-oriented paradigm to discrete event simulation in general are presented. Also the relevance of the object-oriented paradigm to the simulation of distributed algorithms is discussed. Some attempts to provide a programming environment for simulating distributed algorithms are then described. As a continuation on the topic of languages, a summary of the main features of a few object-oriented languages and libraries used in simulation is presented. This chapter concludes by explaining the reasons why we chose the object-oriented concurrent reflective language

ABCL/R2 (Matsuoka et al. 1991b) (Masuhara et al. 1992) to simulate distributed algorithms.

The definitions provided in Sections 4.1.2 and 4.1.3 are from (Jacobson et al. 1991).

4.1.2. Objects, Information Hiding and Encapsulation

An *object* can be defined as a set of operations and a state that stores the results of these operations. An operation of an object is executed (or a behavior is performed) when the object receives a message to do so. An object provides *information hiding* because other objects can only see the operations that are declared public for this object. How these operations are implemented and the effect on the state of the object are not available to the customer object. Consequently, the behavior and state of an object are *encapsulated* in the object.

An object is an instance of a *class*. Thus, a class is a template for the creation of objects, providing a description of the internal structure of an object, i.e. its state and the implementation of its operations.

4.1.3. Inheritance

The usefulness of classes comes from the fact that they can be arranged in a hierarchy according to structural similarities (state and behavior structure of their instances). The common characteristics of several classes are placed into one class and

those classes (child classes) *inherit* from this common class (parent class). This process of extracting and sharing common characteristics is called *generalization*. The process of adding characteristics to a child class in addition to the ones inherited from a parent class is called *specialization*. The use of generalization avoids redundancy. Inheritance can also be useful for updating models more easily: the parent class is updated and the child classes inherit the updated characteristics.

4.2. Simulation and Modeling

The purpose of simulation is to help us understand the world. Ziegler (1976) considers modeling as the study of the relationships between real systems and models (representations of the real systems). He also states that simulation deals with the relationships between models and computers. His theory of simulation and modeling is based on five basic elements: the *real system*, the *experimental frame*, the *base model*, the *lumped model* and the *computer*.

The *real system* is the source of observable data or the system phenomena of interest.

The *experimental frame* is the set of situations to be used in observing or in experimenting with the real system.

The *base model* is a model that fully explains the behavior of the real system. The base model is an intricate set of components and interactions. Being so complex, this model cannot be completely described using a realistic modeling and simulation technique. The base model reflects the worldview of the observer.

The *lumped model* is an abstract, high-level (simplified) model of the base model. The lumped model is valid according to a defined experimental frame. It is developed using decomposition as a first step to deal with the complexity of the real system. The elements that result from the decomposition process are seen as modules, whose internal structures can be refined independently. Each module represents some nameable or recognizable entity from the real system. After decomposition, relationships and interactions between modules are defined. Hierarchies of modules are also defined. These hierarchies, which help dealing with the complexity of the real system, result from the clustering of modules.

Differences may be found between the traditional decomposition and the object-oriented decomposition for building the lumped model (Rogers 1991). Traditional decomposition typically reflects a top-down structure design; each module represents a set of procedures. It focuses on the flow of control within the system to achieve the goals of the client system. Conversely, object-oriented decomposition views the system as a set of independent entities (objects) that collaborate to achieve a certain goal (the overall behavior of the system). An object hierarchy represents a classification of objects, according to their behavior as opposed to the boss-subordinates hierarchy of traditional decomposition. Thus, object-oriented hierarchies reflect commonality of modules as opposed to interaction of modules as in traditional decomposition. The aggregation phase of object-oriented modeling aims at identifying communication relationships instead of subordinate relationships.

The *computer* is the computational device used to experiment with the lumped model. The simulation software, which represents another model is considered part of the computer.

4.3. Object-oriented Simulation

In order to model a simulation, it seems more natural to use the notion of objects and to specify the behavior of these objects, since the real world is composed of objects that interact with each other. This can shorten the gap between the description of the model to be implemented in a computer and the description of the actual problem being modeled.

Extensibility and reusability are known advantages of *object-oriented programming*, but these two aspects alone do not represent a fundamental change in systems modeling. On the other hand, using the *object-oriented paradigm*, emphasizing that objects have a well defined protocol, may be a more powerful tool than the classical top-down approach to deal with complex systems. The object-oriented approach focuses more on defining objects and their interfaces than on their implementation and usage. The use of an object depends on the application at hand and can change during the development and implementation of the model. The main advantage of object-oriented modeling is that the use of an object is likely to change more often than the features of the object itself. On the other hand, finding objects and classes can be controversial and difficult. It is the hardest part of the object-oriented design (Booch 1991). In order for the object-oriented paradigm to be fully used in simulation and modeling, a methodology for identifying objects in the basic and lumped models is necessary.

The process of modeling a simulation application can be strongly influenced by the features of the simulation language to be used. The idea of having "the best simulation language" depends very much on the needs of the users of the language. Theoretically, any programming language can be used for simulation. Bischak and Roberts (1991) claim that object-oriented simulation does not force the programmer to use a particular simulation

world-view¹, they consider object-orientation as a programming language orientation, not a simulation orientation. Also, Rogers (1991) mentions that object-oriented simulation is currently addressed in the simulation and programming literature as any computer simulation whose code has been written in an object-oriented language. In Rogers' opinion, from a simulation modeling perspective, this is the translation of the lumped model into computer software.

Bezivin (1987) points out the future trends in simulation: large simulation programs will run on distributed systems, using asynchronous techniques with interactive graphical views of the simulation. He believes that Smalltalk-80 is a feasible environment for evaluating different simulation techniques. In (Bezivin 1988b) SIMTALK, an evaluation environment for concurrent simulation, is presented. More information about SIMTALK can be found in Section 4.6.1.1.3.

Key features in discrete-event computer simulation include collection of statistics and the use of a single global clock. Most of the statistics collection is done in a centralized way and shown as tables and histograms. In object-oriented simulation, the use of a single (global) clock can be implemented by an object time which can advance the simulation time and activate processes. Another approach consists in each object possessing its own local simulation clock synchronized periodically with other local clocks according to a global clock.

¹ World views are the different approaches for describing the dynamic structure of a simulation model. The three classical world views are: the event scheduling, the activity scanning and the process interaction (Cota and Sargent, 1992)

4.4. The Relevance of the Object-oriented Paradigm for Simulating Distributed Algorithms

The major features that describe distributed computing are: network topology, faults, fault-tolerance, random communication delays, no shared memory and no global clock.

The fact that objects encapsulate state and behavior communicating through message passing matches the distributed computing model of autonomous processing nodes (without shared memory) that communicate only by exchanging messages. Objects also provide a natural basis for concurrency (Roberts and Heim 1988). Each object could be assigned to a processor and work independently until some co-ordination with other objects becomes necessary.

The ability to develop complex software using high-level abstractions, the encapsulation of state and behavior in objects and inheritance are in our opinion the most important features of the object-oriented paradigm that are relevant to the simulation of distributed algorithms.

High-level abstractions help to build complex models. We can consider almost everything as objects when modeling a distributed system from an object-oriented point of view: processing nodes, communication links, the network of processing nodes etc. Moreover, objects that represent processing nodes can be seen as state machines (Schneider 1990), an abstraction very much used in modeling distributed computing for studying fault-tolerance, as mentioned in Section 2.5.2.

In distributed computing terms, encapsulation means that processing nodes have their own processing power and local memory, not directly accessible by any other node.

Information hiding means that a node does not know how operations are implemented in other nodes: all a node knows is the messages that other nodes can understand.

Finally, inheritance, as it pertains to the possibility of code reuse, is relevant to simulation software as it is to any type of software.

In light of the previous discussion, we believe the object-oriented paradigm is well suited to simulation or to simulation applications. On the other hand, there does not appear to be any research effort to analyse the impact of object-oriented reflection on the field of simulation. Consequently, one of the purposes of this dissertation is to show that object-oriented reflection is particularly useful for simulations of distributed algorithms. The experiments on simulating some distributed algorithms using the reflective object-oriented approach (described in Chapter 5) and the related discussion will illustrate the advantages that reflective capabilities in an object-oriented language can offer for the simulation of distributed algorithms.

4.5. Simulation of Distributed Algorithms

4.5.1. DC/Parasol

DC (Distributed C) is an example of an attempt to have a simulation tool for distributed algorithms. DC was developed at Carleton University by Santoro. It is a collection of constructs that help the users to implement (using the C language) and understand distributed algorithms. The DC version that runs on top of the Parasol simulator (Neilson 1991) is called DC/P. The DC/P system consists of a set of C macros to provide the DC language constructs. These constructs are used to define network topologies, node

states and messages, a run-time debugger to let users display and modify the state of a distributed algorithm during execution, and the Parasol simulation kernel which provides multi-tasking and interprocess communication support. The main purpose of DC/P is to be a teaching tool.

The distributed model of DC is the following: Processing nodes have non-shared memory and clock, and can communicate with neighboring nodes by sending and receiving messages. Each node behaves as an event-driven state machine. Communication links can be uni-directional or bi-directional. Communication links connect nodes according to a particular network topology. Simulated algorithms can be synchronous (not implemented in the language) or asynchronous. By default, algorithms that run in DC/P are asynchronous. For an asynchronous algorithm, no TICK messages are sent by the system, and the algorithm terminates when the number of messages to be processed reaches zero. On the other hand, for a synchronous algorithm, all processes have a local clock, actions can be performed at certain specified times and the system increments all local clocks simultaneously. This means that each process receives a TICK message after the clock of this process has been updated, every message sent in time interval t is received in time interval $t+1$, and the algorithm terminates when some process executes a "terminate-execution" statement. Topologies that are supported by DC/Parasol include uni and bi-directional ring, binary tree and complete graph. There is no mechanism that allows the user to define any network topology. Additional details about the DC/P environment can be found in a manual available at Carleton University written by Robert D. Black ("The DC/P Simulator: A System for Running DC Programs under Parasol").

4.5.2. Examples of Current Research in Simulation of Distributed Algorithms

The following paragraphs enumerate examples of attempts to have programming environments to simulate distributed algorithms.

An example of a distributed discrete event-driven simulator is CNCSIM implemented in CONIC (Kremien et al 1990). It aims at being a tool for quick assessment of decentralised scheduling and load-distribution algorithms for distributed systems. According to the team that worked on this project, formal analysis of the correctness and complexity of distributed algorithms is usually difficult and a trial and error approach is often adopted. They believe that simulation can overcome this problem by providing means for controlling parameters and events, enabling algorithm evaluation. Furthermore, simulation is done in virtual time, not slowed down by network delays.

Another example is the DSS simulation tool (Spirakis et al 1992) (event driven simulator+simulation environment) that aims at providing efficient simulation of distributed algorithms given a user's specification and the topology of the interconnections of the distributed system. The user's specification is modeled as a set of communicating automata with countable number of states. DSS is part of the BRA ALCOM project. A version of DSS has already been used in the industrial area for the simulation of a telecommunication switch whose specifications are written using the specification language SDL (Belina et al. 1991). DICOMP (Spartiotis et al. 1991) was created to be a compiler for SDL, having files containing SDL specification as input and outputting files containing data and C code.

Veda (Jard et al. 1988) is a prototyping tool developed to help designers of protocol modeling and validation. In Veda, the distributed algorithms that represent the protocols are

described in Estelle, a proposed ISO standard for protocol specifications. In (Jard et al. 1988) the use of simulation as a much more powerful tool than prototype implementation for distributed systems is emphasized: implementing a simulation requires less time and costs less, simulation allows more control over all parameters and events of the system being studied, simulation can be done in virtual time and makes it possible to observe the behavior of the distributed system by adding observation devices. But prototyping can be useful (as a complement to simulation) to study real error situations. Verification of protocols is done in Veda with the help of "observers", programs that can see every interaction in the system and internal states of a module halting the simulation, while observing.

ELECTOR (Love Jr. and Aburdene 1990), a program written in OCCAMtm (Burns 1988), a parallel programming language, was developed to simulate distributed election algorithms. The authors of ELECTOR believe that the results that were obtained indicate that simulation can be a better way for evaluating the performance of distributed algorithms. In their opinion simulation can confirm the analysis for algorithms already defined and can provide a general idea of the behavior and value of an algorithm without analysis. Also, in many cases, an analysis is able to determine only the upper bound on the average case behavior of the algorithm. Thus, a simulation can be useful to determine how close the upper bound is to the actual average case behavior. The disadvantage of the approach of Love Jr. and Aburdene is that ELECTOR is useful only for simulating election algorithms. But they point out that with the experience of making ELECTOR, other simulators for other distributed algorithms could be developed. Another limitation of ELECTOR is the fact that OCCAM provides only synchronous communication.

Aburdene and Whaley (1988) present a simulation of an election algorithm in a ring using the connection machine (CM) (Hillis 1985), a single-instruction multiple-data (SIMD) machine. The CM is an array of processors arranged in a hypercube network typically composed of 16K to 64K processors. CM programs are written in dialects of C (similar to C++ with classes replaced by "domains" of parallel data and functions (Rose 1987) (Rose and Steele 1987), Lisp (from Common Lisp) and Fortran 8x (ANSI standard that incorporates parallelism in the array features of the language).

Finkel, Brown and Kiff (1988) simulated the buddy system, a fault-tolerant scheme for a distributed computing system. The simulation language used was SIMAN (Pegden 1986). In SIMAN, two files are built: the model and the experimental frame. The model frame is the description of the simulation model and the experimental frame contains the parameter values. Consequently, one model frame can have several experimental frames. SIMAN allows the use of separately compiled FORTRAN subroutines to extend the capabilities of the language.

The above survey shows that reflective object-oriented simulation of distributed algorithms seems to be a branch of simulation not exploited so far. This emphasizes the originality of our contribution.

4.6. Simulation Languages

Even though, theoretically, any programming language can be used for simulation, object-oriented languages enable software designers to develop simulations more easily. Roberts and Heim (1988) claim that future simulation languages will tend to use more the object-oriented approach. They also believe that it would be more convenient if the simulation model was described with the same terminology that is used to describe the actual problem. One could then think that simulation languages could be specialized to contain specific objects relevant to an application (e.g. objects like transactions, resources, queues, activities, statistical distributions for stock applications or objects like transporters, cranes, carries and layout for manufacturing applications). Roberts and Heim (1988) pointed out that these lists of objects could be endless. So, object-oriented and object-based (without inheritance) simulation languages should provide the user with the ability to build applications based on a consistent platform of concepts offered by the language (mainly the creation and description of objects), i.e. crafting their own objects. The advantage of object-oriented simulation is that it allows the programmer to define new types of objects easily, without the need to match the entities of the simulation model with the limited number of constructs available in a particular simulation language. But in this case, the programmer has to act as a language designer because there are no pre-defined objects. As a consequence, more time is required in order to design and implement the simulation because it involves all the issues of object-oriented design and programming.

Bezivin (1987) mentioned that the present interest in object-oriented programming can be traced back to the Simula 67 language (Nygaard 1978). Also, simulation languages

are currently incorporating ideas from the research in object-oriented programming. So, the idea of using object-oriented languages for discrete-event simulation is not new.

4.6.1. Analysis of the Suitability of some Object-oriented Languages and Environments for the Simulation of Distributed Algorithms

Using object-oriented languages to simulate distributed algorithms Hopkins and Wolczko (1989) point out that the object-oriented model has been considered suitable for the exploitation of parallelism because objects communicate with each other. There are three different approaches to parallelism in this case: message sending like a procedure call, objects as concurrently executable program modules and objects as small units concurrently executable.

In the first approach, control passes from the sender object to the receiver object when a message is sent, and returns when the message has been processed. Processes may be objects, but not all objects are processes. Processes can synchronize using semaphore or monitor objects.

In the second approach, internal activity of an object starts when the object is created and continues after a message is sent. Not all the computation is expressed using message sending (i.e. other means of expressing the functionality within objects are required). Messages are accepted at explicitly specified times.

In the third approach, objects (also called actors (Agha 1986)) can execute concurrently after the sender object sends a message to the receiver object. Messages are enqueued at the receiver, so the computation is expressed as communicating actors and many objects may be active if several messages are sent.

The importance of the object-oriented paradigm to the simulation of distributed algorithms has already been discussed. The following subsections provide an overview of some object-oriented languages and environments and an analysis of their suitability to the simulation of distributed algorithms. A suitable language (environment) must be concurrent in order to simulate parallelism and objects must be the units of concurrency in order to represent processing nodes of a network executing in parallel, i.e. a distributed system. This corresponds to the third approach explained in the previous paragraph. Also, we submit the opinion that the language should provide reflective capabilities. The advantages of using a reflective object-oriented approach in comparison with a class-based one are presented at the end of Chapter 5.

4.6.1.1. Smalltalk

4.6.1.1.1. Smalltalk-80 (ST-80)

ST-80 (Goldberg and Robson 1983) provides some language constructs for concurrency. Nevertheless, the amount of concurrency achieved is very limited.

In ST-80 a process is created by sending a *fork* message to a block context, but there is no *join* to synchronize forked processes. Processes communicate with each other using common variables, which can be temporary variables, instance variables, class variables, pool variables or global variables. Mutual exclusion on the access of these variables is achieved through the use of *semaphores* (instances of the class *Semaphore*). The semantics of message passing in ST-80 are akin to those of a remote call, i.e. the caller process can invoke a process from any position in its program, but the callee process is

only activated by one of its procedures being invoked by the caller. The scheduling of processes occurs in FIFO order with priority, i.e. there is a process switching every time a higher priority process becomes ready for execution. If all processes have the same priority, there is no process switching unless the current process (the process being executed) terminates, becomes blocked in a semaphore or yields the processor to another process. According to Hopkins and Wolczko (1989) this behavior is unfortunate when constructing concurrent programs. They believe that in order to solve this problem, it is necessary to perform changes in the virtual machine² and in the virtual image³. Piumarta implemented an improved process scheduler within Smalltalk itself, but it presented poor performance. The behavior of ST-80 with respect to concurrency characterizes a "non-truly" concurrent language where concurrent execution of the code of different processes is not achieved implicitly by the process scheduler.

In summary, ST-80 cannot be considered a candidate language for simulating distributed algorithms because it is not truly concurrent and objects are not the units of concurrency.

² "The virtual machine consists of a mixture of hardware and software which actually performs operations on the objects in the image" (Hopkins and Wolczko 1989)

³ "The virtual image contains all the objects which make up the Smalltalk system, and is completely portable between implementations" (Hopkins and Wolczko 1989)

4.6.1.1.2. ConcurrentSmalltalk (CST)

ConcurrentSmalltalk (CST) (Yokote and Tokoro 1987) is ST-80 compatible, to take advantage of the fact that ST-80 is widely used. All the language features of ST-80 are also present in CST. Also, CST presents constructs to provide more concurrency than these ones provided by ST-80.

In CST there are two types of communication among objects: synchronous method calls and asynchronous method calls. Synchronous method calls are like the remote calls of ST-80. In an asynchronous method call, an object sends a message and the program proceeds without any need to wait for a reply for the message. But if replies to several asynchronous messages are necessary, Cbox objects are used like mail boxes that store these replies. A Cbox object is created immediately after an asynchronous message is sent and waits for a corresponding reply message. In addition to asynchronous message passing and Cbox objects, CST provides atomic objects. Messages sent to an atomic object are executed serially in FIFO order.

CST is not a good candidate language for simulating distributed algorithms because, despite the features added to the ST-80 language, it is not truly concurrent and objects are not the units of concurrency.

4.6.1.1.3. SIMTALK

SIMTALK (Bezivin 1988a) is based on Smalltalk-80 and is part of a general project to define an experimental evaluation environment for object-oriented simulation. The motivation for this project was the need to study the problems of concurrent simulation.

SIMTALK is composed of a common kernel that implements the basic mechanisms and subsystems that support different styles of simulation programming and interpretation (event-oriented or process-oriented with either an actor network or client-server decomposition). In the actor network approach all simulation entities communicate by exchanging messages. In the client-server approach, the passive entities are the servers and the active entities are the clients. Clients send messages to servers, but the reverse never happens. The actor network approach seems to be an ideal model for simulating distributed algorithms. But unfortunately in SIMTALK, actors are implemented as messages, not as objects. The classical car wash example in (Bezivin 1988) illustrates this.

SIMTALK augments the basic simulation framework of Smalltalk-80 presented in (Goldberg 1983) with several process synchronization schemes: Hoare monitors (Hoare 1974), Kessels monitors (Kessels 1977), Robert and Verjus control modules (Robert and Verjus 1976), Campbell and Habermann path expressions (Campbell and Habermann 1974). However, all these synchronization schemes are useless to the simulation of distributed algorithms where there is no shared memory and processing nodes communicate only by message passing.

SIMTALK is very useful to discrete-event simulation, including distributed simulation. But SIMTALK is not suitable for the simulation of distributed algorithms because the protocol of objects used in the simulations (instances of class `SimulationObject` as in Smalltalk-80) contains only initialization of instance variables, a sequence of activities (tasks, e.g. delay, acquire a resource, release a resource) to be performed during the simulation and messages to describe these activities. All those features do not give support to the model of distributed computing of objects executing in parallel and communicating through message passing.

4.6.1.1.4. SmalltalkAgents (STA)

SmalltalkAgents is based on Smalltalk-80, including a number of extensions that Quasar Knowledge Systems (QKS) has made. The class library is similar to that of Smalltalk-80 but is not a one-to-one mapping. QKS has removed or combined a number of classes and also added many new classes.

The class library of STA has been designed to support interrupt driven threads. The class Thread provides support for the creation of processes (called threads). The notion of threads is similar to processes used in ST-80. The difference is that in ST-80, processes run to completion, unless they explicitly yield the processor to another process or they get blocked in a semaphore's queue. On the other hand, in STA, a round robin policy is used in scheduling threads. Threads can communicate by sending semantic events to each other. Each thread has an event queue, managed in FIFO order. Della Valle (1995) extended the thread feature of STA by creating the class ConcurrentObject. Objects of class ConcurrentObject are units of concurrency and they execute according to the round robin policy of the STA scheduler. Della Valle also created instance methods in the class ConcurrentObject to support asynchronous message passing between concurrent objects.

Considering the extensions made by Della Valle (1995), STA has become a good candidate for the simulation of distributed algorithms. The only present disadvantage of STA is the lack of reflective capabilities, like the ones of ABCL/R2. Appendix B contains the simulation code in STA and in ABCL/R2 of the election algorithm of Chang and Roberts (1979) and the algorithm for communication deadlock detection of Chandy, Misra and Haas (1983). The advantages of using reflective capabilities in the simulation of distributed algorithms are presented in Chapter 5.

4.6.1.2. SIM++

SIM++ is a C++ package of object types and routines especially designed for writing object-oriented parallel simulations that execute on multiprocessors (Lomow and Baezner 1989). "In SIM++, entities are loosely-coupled, independently executing objects that communicate and synchronize their actions by scheduling and receiving events. Each entity has its own separate address space and, generally, cannot access the member variables and functions of other entities or communicate with other entities using shared memory" (Baezner et al. 1990). Each entity is an instance of an entity class. It has an entity identifier and its own separate thread of control. An event is a data structure with attributes: event time, event type and an optional event body. The event time corresponds to the simulation time at which the event will occur. The event type identifies the type of event from a set of different types of events. The event body is used for communicating application-specific information between entities.

The following features of SIM++ support parallel simulation:

- a) Distributed parallel input and output facilities through multiple processors, if this is supported by the operating system and machine architecture,
- b) Built in user level and system level tracing,
- c) Distributed monitoring and debugging tools,
- d) Performance analysis tools that estimate the amount of speedup that can be obtained from a particular simulation program,
- e) Run-time mapping of entities to processors by using a configuration file. Thus, the number of processors used in a simulation can be easily changed with no modifications to the program text.

SIM++ also supports random number generation, data collection and linked list manipulation.

The features related to the parallel execution of objects make SIM++ a good candidate for simulating distributed algorithms. On the other hand, synchronous and asynchronous message passing seem to be absent in SIM++ because objects communicate and synchronize their actions by scheduling and receiving events. SIM++ appears to be suited to distributed simulation.

4.6.1.3. DISC++ (DIScrete Event Simulation in C++)

According to Blair and Selvaraj (1989), C++ is an excellent language base from which to build simulation models independently of the world-view used. There is a natural correspondence between objects (abstract objects or not) of the system to be simulated and C++ classes, such as, "customer", "machines", "bank clerks", "files", "sets", "statistics".

DISC++ is a library of routines written in C and C++ that can be used to implement (in C++) simulation models based on event scheduling and process interaction world-views or a combination of both. It is possible to develop models just using existing library classes or deriving classes from existing library classes.

The motivation for DISC++ is the fact that a truly successful simulation language should support more than one world-view and accommodate a full range of user sophistication and needs. Consequently, the main advantage of DISC++ is that the user can choose the world-view which best fits the system to be modeled, in addition to the level of programming sophistication. DISC++ is totally compatible with C and C++.

The characteristics of DISC++ make it a very useful language for discrete event simulation. But the lack of language constructs to provide concurrent execution of objects communicating through synchronous and asynchronous message passing disqualify it for the simulation of distributed algorithms.

4.6.1.4. DOSE

DOSE (Mak 1991) is a general purpose object-oriented simulation environment in which a simulated system can be specified and constructed modularly and hierarchically using custom and library components. The main goal of this environment is to enable the sharing and reusing of simulation code among several users. In order to accomplish its goal, DOSE comprises a group of library classes in C++ to be linked with a user simulation program. DOSE can be used to structure software as a set of components, which can be viewed as state machines, interconnected through their input and output ports. A component does not have to know, at compile time, which and how many different components it interacts with at run time. In addition, component definitions can be inherited. As opposed to other simulation languages, such as Simula (Dahl and Nygaard 1966), GPSS (Schriber 1974) and Simscript (Kiviat et al. 1973), the programmer does not need to learn another programming language to use DOSE, since it is C++ based.

The state machine approach of DOSE suits very well the model of a distributed system. But since DOSE does not provide parallel or concurrent execution of those state machines it does not qualify as a simulation environment for distributed algorithms.

4.6.1.5. SELF

SELF (Ungar and Smith 1991) is based on simple ideas: prototypes, slots, and behavior. Prototypes provide a simpler and more flexible framework than most object-oriented languages. SELF does not have classes or variables like Smalltalk. A SELF object contains named slots which may store either state or behavior instead of a class pointer. An object is created in SELF by copying (cloning) another object (the prototype).

SELF is not a candidate for the simulation of distributed algorithms because it does not provide any concurrency features that could be used to simulate objects (processing nodes) executing in parallel.

4.6.1.6. The language ABCL/R2 and its architecture

The object-oriented concurrent reflective language ABCL/R2 (Matsuoka et al. 1991b) (Masuhara et al. 1992) is based on the hybrid group reflective architecture presenting features of the individual reflective architecture (Watanabe and Yonezawa 1988) and of the group-wide reflective architecture (Watanabe 1991). In the hybrid group reflective architecture, identities of base level objects (simply objects) are preserved at the meta level in order to control individual objects, i.e. each base level object has its individual tower of meta objects. Also, the meta-level object groups represent global computation, i.e. each base level object belongs to a group which has its tower of meta groups. A meta group comprises: meta objects of the base level objects of the group, the group kernel objects and other meta-level objects, such as a scheduler. The group kernel objects (located at meta level) are the structural and computational causally-connected self representation (CCSR) of

a group of objects at base level. The group tower determines the group behaviour and the individual tower determines the structural and computational aspects of a base level object. The purpose of meta groups in ABCL/R2 is to manage computational resources, such as processors, through co-ordination among the meta objects of the group members and the group kernel objects.

The group kernel objects are composed of: the group manager, the meta object generator, and the evaluator.

The group manager represents and manages the group. It is able to create new groups. When a group is created, its identity is that of the group manager.

The meta object generator of a group is responsible for the generation of meta objects for each member of this group, i.e. it is a template for all meta objects of this group. The creation of an object and its meta object occurs simultaneously. There is a default meta object generator of the system, which can be modified to satisfy a user's application. The meta object generator plays an important role in the simulation of distributed algorithms, because generated meta objects provide a way for monitoring and accomplishing dynamic changes in objects (which simulate the processing nodes of a distributed system) in a clean way, without mixing the monitoring code with the application code, as will be explained in Chapter 5.

The evaluator represents the shared computational resource of the group. It interacts with meta objects and other group kernel objects for group management. There is a default evaluator for the system and it is possible to have multiple evaluators per group for parallelism.

In ABCL/R2, objects are the units of concurrency and synchronous and asynchronous message passing are supported. In addition, the user can easily define any

type of network topology because a network can be seen as an object that contains processing nodes (also objects) connected in a specific way. Furthermore, the reflective capabilities of ABCL/R2 are useful in the simulation of distributed algorithms (as illustrated in Chapters 5 and 6).

We submit the thesis that, in the current state of the art, ABCL/R2 constitutes an adequate choice of language for the reflective object-oriented simulation of distributed algorithms with educational purposes.

Objects in ABCL/R2 are the units of concurrency, scheduled in a fair way. Also, these objects can communicate through synchronous and asynchronous message passing, allowing the implementation of synchronous and asynchronous algorithms. Any network topology can be implemented as an object. Dynamic reconfiguration of a network in this case represents a state change of this network object. Moreover, objects in ABCL/R2 are not black boxes, the reflective capabilities of the language provide the programmer with the ability of manipulating the structural and the computational aspects (that are reified at the meta levels) of objects. Consequently, the programmer can use these reflective capabilities to easily tailor the simulation environment according to his needs for different purposes such as: tracing, debugging, simulation of faults, simulation of message communication delays etc. Also, the group feature of ABCL/R2 is particularly useful for studying fault tolerance by grouping in distinct groups objects that can fail in different ways with different probabilities.

Table 1 below summarizes the comparison of object-oriented languages and libraries presented from subsection 4.6.1.1 to subsection 4.6.1.6.

	FEATURES				
	for Concurrent Programming			for Reflection	
	Objs units	Fair	Synch	Asynch	Reflec
	Conc	Scheduling	Msgs	Msgs	Capab
LANGUAGES					
ST-80	NO	NO	YES	NO	NO
CST	NO	NO	YES	YES	NO
SIMTALK	NO	NO	YES	NO	NO
STA*	YES	YES	YES	YES	NO
SIM++	YES	?	NO	NO	NO
DISC++	NO	NO	NO	NO	NO
DOSE	NO	NO	NO	NO	NO
SELF	NO	NO	NO	NO	NO
ABCL/R2	YES	YES	YES	YES	YES

Table 1. Comparison of object-oriented simulation environments

Legend:

Objs units Conc - supports objects as units of concurrency.

Fair Scheduling - supports a round robin policy for scheduling objects.

Synch Msgs - supports synchronous message passing.

Asynch Msgs - supports asynchronous message passing.

Reflec Capab - supports reflective capabilities.

STA* - STA as modified by Della Valle (1995).

5.0. Simulation Experiments using ABCL/R2

5.1. Introduction

The following sections describe the simulation experiments that were done in ABCL/R2. The distributed algorithms for which simulations were constructed using ABCL/R2 are:

- a) Election of a leader among processors connected in a unidirectional ring (the algorithm of Chang and Roberts, 1979);
- b) Detection of communication deadlock (the algorithm of Chandy, Misra and Haas, 1983)
- c) Distributed mutual exclusion (the algorithm of Ricart and Agrawala, 1981);
- d) Election of a leader in a complete network of processors (the algorithm of Loui, Matsushita and West, 1986)
- e) Consensus in the presence of Byzantine faults (the algorithm of Lamport, Shostak and Pease, 1982); and
- f) Fault tolerant distributed mutual exclusion (the algorithm of Chang, Singhal and Liu, 1990).

In each algorithm (with the exception of the Byzantine agreement), all processing nodes are defined as base-level objects and execute the scripts described for (object) node1 because they inherit these scripts from node1. Inheritance of scripts, as in ABCL/R, is also possible in ABCL/R2 by using reflective capabilities. In order for a node to be able to inherit scripts from node1, it is created as a member of the group of node1. The meta object generator of this group has the capability of dynamic inheritance of scripts. In all the

implementations of these algorithms, the code for monitoring messages that each node receives is placed at a meta level (or at the metameta level), separate from the code of other levels, illustrating one of the advantages of a reflective architecture: separation of concerns in different levels. Other advantages of using reflective capabilities are discussed for each example. The implementation code of some simulations can be found in Appendix B. The simulation results for each algorithm (except for the Byzantine agreement) are presented using the interaction diagrams (Jacobson et al. 1991) as a visualization technique. The results of each simulation (a list with sender id, message, receiver id for each exchanged message) can be stored in a log file and an interaction diagram can be generated automatically from this log file using a visualization tool that we have built in VisualWorks.

In the description of all the algorithms, the variable n (or N) refers to the number of processing nodes in the network, unless stated otherwise. Also, we describe the algorithms using the words processing node, node and processor interchangeably to refer to an independent processing unit that is part of a network of processing units that communicate solely by exchanging messages.

5.2. Election in a Unidirectional Ring - The Algorithm of Chang and Roberts (1979)

The algorithm (for the case where the processor with the highest id number should be the leader) can be briefly described as follows: each processing node (or node or processor) of a network is assumed to know its own identity number and initially passes a message with this number to its neighbor processor in one predetermined direction, let us say right. When receiving a message, a processor compares the number in the message

with its own identity number. If its own number is lower, the processor passes this message again to its right. If its own number is higher than the one received, the processor discards the message and sends another message to its right, containing its number. If the numbers are the same the processor is the leader (elected).

5.2.1. The Algorithm

Description in Pascal-like pseudo-code from (Raynal 1988) with comments on the right:

Each node has the following declarations:

Constant my-number: integer; node's id number

Variables candidate: Boolean; initialized F

winner: integer; initialized 0

When decision (initiate_election) do

candidate <- T;

sendRight (election, my_number);

end do;

(* When a node starts an election

, the node marks itself as a

candidate and sends its id to its

right-hand neighbor *)

When received (election, j) do

case j > my_number then

begin

sendRight (election, j);

(* When a node receives a message

of type election, if number

in the message is greater than its own id

it sends the number to its right-

candidate <- T;	hand neighbor and marks
end case;	itself as a candidate *)
case j < my_number then	(* If number in the message is less
begin	than its own id, it sends its id to
sendRight (election, my_number);	
candidate <- T;	its right-hand neighbor
end case;	and marks itself as a candidate *)
case j = my_number then	(* If number in the message = its id
sendRight(elected, I);	it sends a message of notification to
end case;	its right-hand neighbor informing
end do;	that it is the winner *)
When received (elected, j) do	(*When a node receives a message
	of type elected (notification of the
winner <- j;	winner), it stores the id of the
candidate <- F;	winner and resets its condition to
	participate in other elections *)

Any process that receives the above message (winner or not) resets the candidate condition, so that it can participate in other elections.

```

if j <> my_number then                (* A process that is not the winner
    sendRight(elected, j);           passes this notification message to
end if;                               its right-hand neighbor *)
end do;

```

5.2.2. The Implementation in ABCL/R2

A small example of this algorithm was implemented in DC/Parasol and in ABCL/R2 using four nodes in a unidirectional ring where messages are always transmitted in clockwise direction. Also, an example with six nodes was implemented in SmalltalkAgents. In the DC/Parasol implementation, *define-topology* and *define-size* are used to define the topology and the number of nodes of the network respectively. In the ABCL/R2 implementation, the object *ring* has the topological information of the ring network. In order to simulate a unidirectional ring with more nodes, the corresponding topological information must be included in the object ring. This can be done in a straightforward way. In DC/Parasol, the user cannot control the order in which nodes are placed in the ring network. On the other hand, in ABCL/R2 the user can choose the order of connection of nodes easily. This is possible because the ring network is an object that contains nodes (also objects) and information about the order in which the nodes are connected.

The algorithm was simulated in ABCL/R2 considering any of the nodes as the initiator of the election (i.e. the algorithm is supposed to work, no matter which node is the initiator). The initiator in our implementation is the object that receives a message *wake* from the user. Figure 2 shows the four-processor ring with all messages that circulate in order to elect the node with the highest id number, when node 1 is the initiator. In order to start the simulation the user can send a message *wake* to any node, for instance, in this case, to node1: [node1 <= [:wake]]. Figures 3a and 3b show the interaction diagrams for this execution.

This example, albeit small, shows some advantages of reflection: dynamic monitoring of messages received by processing nodes in a clean way (without mixing code of monitoring with code that implements the algorithm) and dynamic inheritance of scripts, as explained in the following paragraphs.

In the DC/Parasol and SmalltalkAgents implementations, monitoring information had to be mingled with the code of the algorithm in order to produce trace messages about the execution of the algorithm: the number of messages, the sender node, the receiver node and the type of each message received by each node. When simulating the same example using ABCL/R2, the same type of monitoring was achieved with a few lines that were included as meta level code, without interfering with the code that implements the algorithm (base-level code). Figure 1 illustrates the clear separation of concerns and code that the structured approach of a reflective object-oriented design and implementation provides by showing the responsibilities that are fulfilled in each level (in this case, two levels: base level and meta level). This example makes clear that object-oriented reflection can be used for monitoring messages without interfering with the application code. In this example, each meta object collaborates with its respective base-level object in an implicit way, as

Figure 1 illustrates, i.e. the meta object does not receive a message (request) from the base-level object in order to fulfill its responsibilities. Whenever a message is received by a meta object (meta node) on behalf of its corresponding base-level object (node), this meta object fulfills its responsibility of monitoring received messages (execution tracing).

Inheritance or dynamic acquisition of scripts happens when an object receives a message for which it does not have any script. In this case, this object inherits the script of another object which is defined as inheritance source. In this example, object node1 has all the necessary scripts defined in it and the other nodes (other objects) just inherit these scripts, because object node 1 is specified as the inheritance source for all the other nodes. This makes repetition of scripts' code unnecessary. There is no inheritance of state variables though, because this is not supported by the current version of ABC1/R2.

NB:

- direction of the flow of messages from one object to another
causal connection

the base level shown in this picture represents all base-level objects and the meta level represents all meta objects

Figure1. Description of the responsibilities fulfilled in each level for the simulation of the election algorithm of Chang and Roberts

Figure2. Messages exchanged during the execution of the election algorithm of Chang and Roberts considering four nodes and node1 as the initiator of the election

5.2.3. Simulation Results

Figure 3a. Base-level interaction diagram for the example of execution of the algorithm of Chang and Roberts illustrated in Figure 2

NB: node ? stands for any node of the network

Figure 3b. Meta-level interaction diagram for the example of execution of the algorithm of Chang and Roberts illustrated in figure 2

A more general simulation program where the number of nodes in the ring is chosen arbitrarily by the user has been also implemented. The user provides information about the number of nodes and desired ring topology. The simulation program creates the nodes dynamically and stores them in a list. In addition, the corresponding topological information (the order in which the nodes are connected in the ring) is included in object ring. For example, `[ring <= [:start 7 [4 3 1 7 2 5 6]]]` defines a ring with 7 nodes connected in the order 4,3,1,7,2,5,6. If, for example, node 5 must be the initiator, then a message *wake* must be sent to the object ring with 5 as an argument, i.e. `[ring <= [:wake 5]]`.

5.3. Communication Deadlock Detection - The Algorithm of Chandy, Misra and Haas (1983)

Chandy, Misra and Haas created an abstract description of a network of communicating processors as a communication model to explain a distributed communication deadlock detection algorithm. In this description, each processor has a dependent set (DS). The DS of a processor contains the identification number of other

processors from which this processor (in idle state) is waiting for a message in order to resume execution. An idle processor starts executing as soon as it receives a message from any processor that belongs to its dependent set; otherwise it does not change its state nor its DS. A processor is terminated if it is idle and its associated dependent set is empty. A nonempty set S of processors is deadlocked if all processors in S are permanently idle. A processor is permanently idle if it never receives a message from any of the processors in its dependent set. Nevertheless, there is a situation when it is impossible to detect permanent idleness: processor A is waiting for a message from processor B and processor B is executing an infinite loop (after which it would send a message to processor A). Thus, Chandy, Misra and Haas adopted the following operational definition of communication deadlock:

A non-empty set of processors S is deadlocked if and only if:

- All processes in S are idle.
- The dependent set of every processor in S is a subset of S .
- There are no messages in transit between processors in S .

A processor is deadlocked if it belongs to some deadlocked set.

5.3.1. The Algorithm

A processor can determine whether it is deadlocked or not by initiating a query computation when it enters the idle state. Processors may exchange messages for the query computation even in idle state because the idle state refers only to the underlying computation (application computation). The processor which initiates a query computation is called the initiator of that query computation. Several processors may initiate query

computations and the same processor may initiate query computations several times. In a query computation, messages are of the form *query*(i, m, j, k) and *reply*(i, m, j, k) meaning that these messages belong to the mth query computation initiated by processor P_i and are being sent from P_j to P_k . So, P_i is the initiator, m is the sequence number, j is the sender and k is the receiver. Properties of the query computations are: if processor P_i is deadlocked when it initiates its mth query computation, it will receive messages *reply*(i,m,j,i) corresponding to every message *query*(i,m,i,j) that it sent and if the initiator P_i has received a message *reply*(i,m,j,i) corresponding to every message *query*(i,m,i,j) that it sent, then it is deadlocked.

Description in Pascal-like pseudo-code from (Chandy et al. 1983) with comments on the right and above the code:

Each node has the following declarations:

Constant N: integer, contains the number of nodes of the network;

Variables latest: Array[N] of integer initialized with zero;

engager: Array[N] of integer initialized with zero;

num: Array[N] of integer initialized with zero;

wait: Array[N] of boolean initialized with false;

When an idle processor P_i initiates a query computation:

begin

(* First it sets the number of the detection (to be distinguished from other detections of the same node) and signals that it is idle. Then it sends a query message to all nodes of its dependent set *)

latest[i] := latest[i] + 1; wait[i] = true;

for all processors P_j in P_i 's dependent set S send query (i, latest[i], i, j);

num[i] := number of elements in S

end

For an executing processor P_k :

On becoming executing, it sets wait[i] = false, for all i (signaling that it is not idle) and discards all received queries and replies while in executing state.

For an idle processor P_k , upon receiving query (i, m, j, k):

if $m > \text{latest}[i]$

(* If it is a new detection in progress, propagate it to dependent set *)

then begin

latest[i] := m; engager[i] := j; wait[i] := true;

for all processors P_r in P_k 's dependent set S send query (i, m, k, r)

```

    num[i] := number of processors in S
end
(* If it is the same detection reply to sender *)
else
    if wait[i] and m = latest[i]
        then
            send reply (i, m, k, j) to Pj

```

An idle processor P_k, upon receiving reply (i, m, r, k):

if m = latest[i] and wait[i]	(* If processor is still idle (since detection started) and
then begin	
num[i] := num[i] - 1;	
if num[i] = 0	received all replies from all nodes
then	of its dependent set
if i = k	and is the one that initiated the
then	detection (initiator), it is deadlocked *)
declare P _k deadlocked	
else	(* If it is not the initiator, it sends
send reply (i, m, k, j) to P _j	reply to node from which it
where j = engager[i]	received the first message query of
end	the detection under consideration *)

5.3.2. The Implementation in ABCL/R2

This algorithm was implemented in ABCL/R2 and in SmalltalkAgents based on the example described in (Chandy et al 1983). This example considers four nodes and initially the dependent sets (marked with arrows linking nodes) are as shown in Figure 5. In this particular example, node 4 is executing and all the others are idle, waiting for messages in order to resume execution. Each node has its own local clock and events (reception of a message, sending of a message and change of state condition from idle to executing and vice versa). The value of this local clock is incremented when an event occurs. In ABCL/R2 the simulation starts with the user sending a message *wake* to (object) node 1, i.e. [node1 <= [:wake]]. The other three nodes are woke up by node 1. The message *wake* is sent to all nodes in order to initialize some of their state variables. After that, node 1 initiates a deadlock detection (simulated by object node 1 sending a message *detect* to itself).

While the detection (query computation) of node 1 is being accomplished, the dynamic behavior of the application program is also simulated through changes in the dependent set of the nodes. The algorithm should detect the existence of a communication deadlock in the presence of those changes. Meta-node 4 simulates the following events of the application: at local time 2, node 4 sends the message to node 3 for which it was waiting in order to resume its execution (simulated by meta-node 4 sending the message *execute* to meta-node 3). Node 4 also stops executing (becomes idle) because it needs a message from node 2 (at this point node 2 and node 4 are deadlocked). Following this, node 4 includes node 2 in its dependent set and becomes idle (through a change of state). This is simulated by meta-node 4 changing the value of the state variables *ds* and *state* of

object node 4 through a reflective procedure. In addition, node 4 starts a deadlock detection (simulated by object node 4 receiving a message *detect* from meta-node 4). Node 3 starts executing when it receives the message for which it was waiting from node 4 (simulated by meta-node 3 receiving the message *execute* from meta-node 4). As a result, node 3 removes node 4 from its dependent set (simulated by meta-node 3 changing the value of the state variable *ds* of object node 3 through a reflective procedure).

Eventually, node 4 comes to the conclusion that it is deadlocked. Node 1 will not declare itself deadlocked because it does not receive a reply from node 3 (node 3 started executing while node 1's detection was in progress). Figures 6a and 6b show the interaction diagrams resulting from this particular execution of this algorithm.

Figure 4 is, like Figure 1, a diagram that shows the responsibilities that are fulfilled in a structured way at different levels (in this case, base level, meta level and metameta level). The monitoring of the execution of the algorithm is accomplished at the metameta level without interfering with the computation of the other levels (see code in Appendix B marked between asterisks). Metameta objects (metameta nodes) and objects (nodes) collaborate in an implicit way in order to fulfill the responsibility of execution tracing, similarly explained in Section 5.2.2. Also, deadlock detection is accomplished at the base level while the underlying computation is simulated at the meta level (see code in Appendix B marked between asterisks). In this case, a meta object (meta node) and a base-level object (node) collaborate explicitly, i.e. a meta object sends a message to its corresponding object in order to initiate a deadlock detection. What can be considered dangerous, in this case, is the fact that the value of the base level variable *time* is updated through meta level programming using reflective capabilities of the language. Updates of the dependent set of nodes 3 and 4 (which are base-level data) are also performed by meta-level code using

reflective capabilities. This situation characterizes violation of encapsulation, since the value of instance variables of base-level objects, that represent the dependent set of each object, is changed by their corresponding meta objects. This illustrates that reflective programming also presents disadvantages. We suggest some design techniques that overcome these disadvantages in Section 6.1.2.2.3. This example reinforces the idea that object-oriented reflection can be useful to build simulation programs of distributed algorithms in a structured way in multiple levels with the additional possibility of using reflective capabilities.

Appendix B also contains the implementation of this algorithm in SmalltalkAgents. The code for monitoring of messages is mixed with the code that implements the algorithm. Also, the code that simulates the execution of the application program that requires communication deadlock detection is implemented as an instance method of the same class (Pdead) that implements the algorithm.

Figure 4. Description of the responsibilities fulfilled in each level for the simulation of the algorithm for communication deadlock detection of Chandy, Misra and Haas

Figure 5. Dependent sets and messages exchanged during the execution of the algorithm for communication deadlock detection of Chandy, Misra and Haas considering four nodes

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Figure 6b. Meta-level and metameta-level interaction diagrams for the example of execution of the algorithm of Chandy, Misra and Haas illustrated in Figure 5

5.4. Distributed Mutual Exclusion - The Algorithm of Ricart and Agrawala (1981)

This algorithm assumes a network of computers whose nodes communicate only by message passing. They do not share any memory. The communication network and nodes are assumed to be error-free, presenting variable transit times in which messages are delivered, not necessarily in FIFO order. Consequently, this algorithm works with distributed control and is not sensitive to the relative speeds of nodes and communication links.

5.4.1. The Algorithm

A node can only enter the critical section after it has sent a message *request* to all other nodes of the network and has received a message *reply* from all of them. When a node receives a message *request*, it replies immediately when it does not wish to enter the critical section. If, when receiving a message *request*, the node also wishes to enter the critical section a decision has to be made about which node should have access to the critical section first. This decision is made based on values of timestamps, the current highest value of timestamp of the receiving node and the timestamp present in the message *request* that shows the local time of the sender node at the time that the message was sent. The node that has the lowest timestamp value should be the one to enter the critical section first. If both nodes present the same timestamp, then the one with the smaller id number, for instance, should enter the critical section first. If the node that receives a message *request* is the one that should have access to the critical section first, it defers its reply to the requesting node(s). After leaving the critical section, this node sends a message *reply* to all nodes that requested access to the critical section after it did (the replies that were deferred).

Description in Pascal-like pseudo-code from (Ricart and Agrawala 1981) with comments on the right and above the code:

Each node has the following declarations:

Constant N: integer; it contains the number of nodes of the network
 my_id: integer; node's id number

Variables my_time_stamp: integer;
 highest_time_stamp: integer initialized with zero;
 reply_count: integer;
 requesting_critical_section: Boolean initialized with false;
 reply_deferred: Array[N] of Boolean initialized with false;

When a processor P invokes mutual exclusion for its node:

```
requesting_critical_section := true;
my_time_stamp := highest_time_stamp + 1;
reply_count := N - 1;
For j = 1 to N do
begin
    if j <> my_id                                (* It requests the
    then                                           permission of all the
        Send request (my_time_stamp my_id) to Pj;    other nodes sending
    end                                             them a message request *)
```

When a processor P receives request (time_stamp_sender id_sender):

```
if highest_time_stamp < time_stamp_sender        (* It updates its own local time
then                                              based on the timestamp of the
    highest_time_stamp := time_stamp_sender;    sender of the message request*)
```

(* If this node also wants to enter the critical section it will have priority if its timestamp is lower than the one of the sending node. In the case of equal values of timestamp, the node with the smallest id number has priority in accessing the critical section *)

```
defer_it := requesting_critical_section and ((time_stamp_sender > my_time_stamp) or
(time_stamp_sender = my_time_stamp and id_sender > my_id))
j := id_sender;
```

<pre>if defer-it then reply_deferred[j] := True; else send_reply to j; endif</pre>	<p>(* If this node has priority, the id of the requesting node should be stored, the requesting node will receive a reply later (after this node leaves the critical section)</p> <p>If this node does not want to enter the critical section, it replies to the requesting node *)</p>
--	---

When a processor P receives reply :

<pre>reply_count := reply_count - 1; if reply_count = 0 then begin "Enter the critical section" After leaving the critical section do</pre>	<p>(* Decrements the number of replies received. If it has received all replies from other nodes, it can enter the critical section *)</p>
--	--

```

begin                                     (* After executing the critical section
    requesting_critical_section := False;    signals that it does not want it *)
    for j =1 to N do
        begin
            if reply_deferred[j]            (* Verifies if there are nodes that also
            then begin                      want to enter the critical section
                reply_deferred[j] := False;    and sends a reply to these nodes *)
                send reply to j;
            end if;
        end for;
    end;
end if

```

5.4.2. The Implementation in ABCL/R2

This algorithm was implemented in ABCL/R2 based on the example described in (Ricart and Agrawala 1981). This example considers three nodes as shown in Figure 8. The simulation starts with the user initially sending a message *wake* to object node1, i.e. [node1<= [:wake]], and after that, sending a message *tryenterCS* to object node3, i.e. [node3 <= [:tryenterCS]]. When object node1 receives a message *wake* it sends a message *wake* to all other objects that represent nodes of the network. The message *wake* must be received by all the nodes of the network before any other message, because it makes these objects initialize a list that contains the name of all nodes of the network so that messages can be broadcasted by any node. When a node receives the message *tryenterCS*, it tries to

enter the critical section by sending a message *request* to all the other nodes. It only enters the critical section after receiving replies from all those nodes as explained in Section 5.4.1.

Figure 8 shows messages that are sent and received among the three nodes during the simulation. The order in which the nodes are granted access to the critical section is: node 2, node 3 and node 1. In (a), node 3 is the first one to request permission to enter the critical section. It sends a message *request* to node 1 and to node 2. Node 1 replies immediately because when it receives the request from node 3, it does not wish to enter the critical section. On the other hand, node 2 also decides to try to enter the critical section before it receives the message *request* from node 3, as shown in (b). This is simulated by meta node3 sending a message *tryenterCS* to object node2 before it enqueues and processes the message *tryenterCS* from the user on behalf of object node3. Node 2 then sends a message *request* to node 1 and to node 3. When node 2 receives the message *request* from node 3 it does not reply immediately (it defers its reply) despite the fact that the timestamp of the request of node 3 is the same as the one of its own request. This happens because node 2's id is smaller than node 3's id. So, node 2 has priority in entering the critical section. Figure 8 (c) and (d) shows that, node 1 replies to the request of node 2 and immediately after that it also decides to try to enter the critical section. This is simulated by meta node1 sending a message *tryenterCS* to object node1 after the message *request* from object node2 is processed. Consequently, node 1 sends messages *request* to node 2 and node 3. Node 2 and node 3 defer their replies to node 1 because they have priority to enter the critical section (they have asked to do so before node 1 has, as shown by the timestamp "2" in the messages *request* from node 1). What is interesting about this part of the simulation is the fact that the messages *reply* and *request* sent from node 1 to node 2 are not received in FIFO order by node 2. In Figure 8 (c), it is shown that the message *request*

arrives first at node 2 and Figure 8 (d) shows the arrival of the message *reply* at node 2. This is simulated by meta node2 exchanging the position of the two messages from object node1 in the queue of messages of object node2. In (e), node 2 receives a message *reply* from node 3 and enters the critical section (node 3 replies to node 2 because node 2's id is smaller than node 3's id, as explained in (b)). In (f), after node 2 leaves the critical section, it sends replies to node 1 and node 3. Node 3 then enters the critical section. In (g), after node 3 leaves the critical section, it sends a message *reply* to node 1. Node 1 then enters the critical section. Figures 9a and 9b show the interaction diagrams resulting from this particular execution of the algorithm for distributed mutual exclusion of Ricart and Agrawala. Figure 7 illustrates the responsibilities that are fulfilled at each level. The meta-level responsibilities of execution tracing and non-FIFO delivery of messages from node 1 to node 2 are fulfilled in a implicit way. The responsibility of implementing the order in which the nodes request to enter the critical section is fulfilled explicitly (a meta object sends a message *tryenterCS* to an object).

This example serves to highlight the relevance of reflection to the simulation of distributed algorithms. In this case, the control of the global order in which the three nodes try to enter the critical section and the non-FIFO delivery of messages from node1 to node2 are completely implemented through meta level programming. The simulation that node 2 tries to enter the critical section at the same time as node 3 (i.e. with the same local timestamp) is handled at the meta level, because reasoning about the time stamp of a received message and the local time stamp of the receiving node can be done at meta level without interfering with the base level code that implements the algorithm for mutual exclusion. Also, the non-FIFO reception of the messages *request* and *reply* by node 2 from node 1 is implemented easily by inverting the order in which those messages are enqueued

in the queue of messages of node 2 and processed. In this way, it is possible to test the algorithm by experimenting a situation which present one or more assumptions that should not make the algorithm fail, i.e. in this case, the non-FIFO reception of messages. This algorithm cannot be tested for the case of receipt of messages in a non-FIFO order using an implementation in DC/Parasol or in Smalltalk where access to the queue of messages of an object is not available to the programmer (the queue of messages of an object is not reified).

NB: direction of the flow of messages from one object to another
causal connection

the base level shown in this picture represents all base-level objects and
 the meta level represents all meta objects

Figure 7. Description of the responsibilities fulfilled in each level for the simulation of the algorithm for distributed mutual exclusion of Ricart and Agrawala

Figure 8. Messages exchanged during the execution of the algorithm for distributed mutual exclusion of Ricart and Agrawala considering three nodes

5.4.3. Simulation Results

Figure 9a. Base-level interaction diagram for the example of execution of the algorithm of Ricart and Agrawala illustrated in Figure 8

Figure 9b. Meta-level interaction diagram for the example of execution of the algorithm of Ricart and Agrawala illustrated in Figure 8

5.5. Election in a Complete Network with Sense of Direction - The Algorithm of Loui, Matsushita and West (1986)

This algorithm considers a complete network (with sense of direction) of asynchronous processors joined by bi-directional links even though the example presented by the authors represents a synchronous execution.

Initially, all processors are active and the algorithm terminates when all processors but the elected leader (the processor with the highest or lowest id number) become passive.

5.5.1. The Algorithm

The following description of the algorithm considers the case when the processor with the highest id number should be elected the leader.

Initially, every processor sends its id to the processor at distance one (adjacent) from it. An active processor becomes passive when it receives a message with an id number greater than its own. When a processor receives a message with an id number smaller than its own, this processor sends a message with its id directly to the sender of the message in order to passivate it. A passive processor just relays messages that it receives, i.e. sends the received messages to the processor that passivated it. Each message contains the distance from the processor whose id is in the message to the processor that receives the message. This distance is used to determine which link a processor should use to send a message.

Description in Pascal-like pseudo-code from (Loui et al. 1986) with comments above the code:

Each node has the following declarations:

Constant	my_id: integer; node's id number
	N: integer; it contains the number of nodes of the network
Variables	d, e, Newid: integer initialized with zero;


```

begin
    d:= N-1;
active:      repeat
(* A node sends its id number and receives id numbers from other nodes, until this node's
id number is smaller or equal than the received one *)
        send(N-d; N-d, id);
        receive(d, Newid);
    until    Newid >= id;
(* If the ids are equal, this node is the leader *)
    if Newid = id then
        announce "elected"
    else
        begin
(* If this node's id is smaller than the received id, this node becomes passive and relays
received messages *)
passive:      receive(e, Newid);
               send(n-d; n-d+e, Newid)
        end
    end
end

```

5.5.2. The Implementation in ABCL/R2

This algorithm was implemented in ABCL/R2 based on the example presented in (Loui et al. 1986) in an asynchronous way (the original implementation is synchronous), in order to verify if the number of exchanged messages would be different. For this particular example the number of exchanged messages is the same.

The example considers eight nodes (eight processors) as shown in Figure 11. The simulation begins with the user sending a message *start* to object *complete_net* with a list that represents the order in which all nodes are positioned in the network ([*complete_net* <= [:start [8 3 5 6 2 7 1 4]]]). The object *complete_net* then stores the topological information of the complete network. When receiving the message *start*, the object *complete_net* sends a message *wake* to all nodes of the network. When receiving a message *wake*, each node of the network sends its id to the neighbor on the right. Figure 11 shows messages that are exchanged among the eight nodes during the simulation of the algorithm, in order to elect the node with the highest id.

In this simulation, as in other simulations, object-oriented reflection makes possible the dynamic monitoring of messages received by processing nodes in a clean way. Figure 10 shows the responsibilities that are fulfilled in each level and Figures 12a and 12b show the simulation results. The collaboration between each meta object (meta node) and its corresponding object (node), in order to fulfill the responsibility of execution tracing, is implicit as has been explained in Sections 5.2.2, 5.3.2 and 5.4.2.

NB: direction of the flow of messages from one object to another
causal connection

the base level shown in this picture represents all base-level objects and
 the meta level represents all meta objects

Figure 10. Description of the responsibilities fulfilled in each level for the simulation of the election algorithm of Loui, Matsushita and West

Figure 11. Messages exchanged during the execution of the election algorithm of Loui, Matsushita and West considering eight nodes

5.5.3. Simulation Results

Figure 12a. Base-level interaction diagram for the example of execution of the algorithm of Loui, Matsushita and West illustrated in Figure 11

Figure 12b. Meta-level interaction diagram for the example of execution of the algorithm of Loui, Matsushita and West illustrated in Figure 11

5.6. Byzantine Agreement-The Algorithm of Lamport, Shostak and Pease (1982)

The Byzantine agreement is an abstraction that represents the problem of handling malfunctioning components that provide conflicting information to different parts of a computing system that should be reliable.

The name Byzantine is due to the situation used to describe this abstraction: a group of generals of the Byzantine army are camped with their troops around an enemy city. The generals communicate only through a messenger in order to agree upon a battle plan. Some of the generals might be traitors. The problem is to find an algorithm to ensure that the loyal generals will reach an agreement. Considering oral messages, there is a solution only if more than two-thirds of the generals are loyal, i.e. for three generals, there is no solution

even with one traitor. Oral messages have the following characteristics: every message that is sent is delivered correctly, the receiver of a message knows who sent it and the absence of a message can be detected. Reaching agreement means to fulfill two conditions, when a commanding general sends an order to its $n-1$ lieutenant generals: all loyal lieutenants obey the same order and if the commanding general is loyal, then every loyal lieutenant obeys the order that the general sends.

Using the terminology of distributed computing, the commanding general is considered the initiator node (or simply the initiator) and the other nodes are the lieutenants.

5.6.1. The Algorithm

The following is the description in pseudo-code from (Lamport et al. 1982), considering n generals and m traitors. Agreement, if possible, is reached after $(m+1)$ rounds of exchange of messages. These rounds of messages exchange characterize a synchronous system.

Algorithm OM(m), $m > 0$

- (1) The commander sends his value to every lieutenant.
- (2) For each i , let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM($m-1$) to send the value v_i to each of the $n-2$ other lieutenants.
- (3) For each i , and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM($m-1$)), or else RETREAT if he receives no such value. Lieutenant i uses the value majority ($v_1, \dots, v_{n-1}$).

5.6.2. The Implementation in ABCL/R2

In our implementation of the Byzantine agreement, we consider the case when a faulty node either relays a value (an order, i.e. attack or retreat) different from the one that it has received (message corruption) or does not relay any value (message omission) with equal probability. The user chooses the size of the network, the nodes that should be faulty (including the initiator), the total number of faults to be generated by the faulty nodes and the value that the non-faulty nodes should agree on, if agreement is possible, in the situation described by the user. For example, consider a network with 7 nodes (with id numbers from 0 to 6) where the initiator, node 0, (node 0 is considered by convention the initiator in any simulation) and node 3 are faulty and two faults must be generated. The user must first send a message *start* to object network, defining the total number of nodes of the network, the id number of the faulty nodes, except the initiator (in a list), and the total number of faults to be generated, i.e. [network <= [:start 7 [3] 2]]. Then, he must send a message to the faulty initiator with the value (for instance, attack) upon which the non-faulty nodes should agree, i.e. [faulty_node0 <= [:wake :attack]]. A similar situation applies to a non-faulty initiator by sending a message *wake* to the object non_faulty_node0.

The faulty nodes and the non-faulty nodes belong to groups called faulty and non-faulty respectively. Those nodes (objects) are created initially (before the simulation starts) according to what is specified by the user in the message *start*, sent to object network. There are two implementation approaches in ABCL/R2 for the algorithm for Byzantine agreement, explained in the next two paragraphs.

In the first approach, whenever a faulty node must relay a value that it has received from another node, it sends a message to its meta object, which determines the type of

failure that should be simulated (depending on a number generated randomly)_ omission or corruption (relaying a different value than the one received). The group feature of the reflective architecture of ABCL/R2 is very useful, in this case, since objects can be separated into two groups, non-faulty processors or faulty-processors with two distinct meta object generators. The first one does not generate faults and the other generates faults.

In the second approach, the simulation of omission and corruption of messages is done by the meta object of the receiver of the message. The meta object verifies whether the sender is a faulty node or not (whether the sender belongs to the faulty group or not) using a group membership function available in ABCL/R2. If the sender is a faulty node, omission or corruption of the received message is simulated. If the sender is not a faulty node, no message faults are generated.

In both approaches, faults are generated at the meta level, independently from the algorithm for consensus implemented at the base level (separation of concerns). Also, if the user wants to run simulations considering different probabilities of occurrence of message omission or corruption faults, this can be done by using different meta objects (generated by different meta object generators). We implemented the algorithm for Byzantine agreement using the second approach because it is simpler than the first one, since it does not require any exchange of messages between objects (nodes) and their respective meta objects.

There are many rounds of messages exchanged, as possible (i.e., $(n - 1)$ rounds), and all the values received in each round with the respective senders ids per node are displayed as the simulation results. If the simulation situation described by the user characterizes an impossibility for agreement, the number of exchanged messages does not modify this scenario, i.e. no matter the number of rounds of messages exchanged,

agreement is not reached. Students usually think that increasing the number of rounds of messages exchanged would make agreement possible.

Figure 14 shows messages that are exchanged among the nodes during the simulation of the algorithm, considering four nodes and two faulty nodes (the initiator and node 3). Each of these faulty nodes cause a message corruption failure. In this situation, it is shown that no agreement is reached despite the fact that all nodes exchanged the maximum possible number of messages. Figure 13 illustrates the responsibilities that are fulfilled in each level. In this case, meta-level responsibilities are fulfilled in a implicit way (from an implicit collaboration between a meta object and its corresponding base-level object). Appendix A provides simulation results for 37 cases.

Figure 13. Description of the responsibilities fulfilled in each level for the simulation of the algorithm for Byzantine agreement of Lamport, Shostak and Pease

Values received by node 1: attack retreat retreat retreat attack

Values received by node 2: retreat attack attack retreat attack

Values received by node 3: attack attack retreat attack retreat

No consensus is achieved

Figure 14. Messages exchanged during the execution of the algorithm for Byzantine agreement of Lamport, Shostak and Pease considering four nodes with two faulty nodes: the initiator and node 3 (they corrupt a received message)

5.7. Fault Tolerant Mutual Exclusion - The Algorithm of Chang, Singhal and Liu (1990)

The fault tolerant algorithm for distributed mutual exclusion of Chang, Singhal and Liu (1990) is token-based. There is only one token and the node that has the token is the one that can enter the critical section. The algorithm considers a network such that each node has more than one path to the node holding the token. This is what makes this algorithm fault tolerant to communication link and site failures because a node has alternative paths to search for the token, in case of communication link or node failures. When a node requests to enter the critical section, messages *request* are sent between the requesting node and the node that has the token. The token is passed to the requesting node along the same path in reversed direction and, at the same time, the direction of all edges is reversed, so that every path always leads to the node holding the token. This process of reversing edges might generate cycles in the graph, which can cause deadlock (messages *request* circulating in the directed cycle forever). So, in order to avoid this situation, the direction of some edges (links of all outgoing paths) are also reversed. Also, a greedy strategy in granting access to the critical section is used: a requesting node can enter the critical section as soon as it receives the token no matter how many other earlier requests from other neighbors are pending. Moreover, in order to reduce message traffic, a piggyback strategy is used: node *S* includes the number of messages in its requesting queue, when a message *token* is sent to one of its neighboring nodes on an outgoing path (*nearS*) that leads to the node which holds the token. This informs the receiving node that node *S* needs the token back. When a node receives the message *token* with $lqueueS > 0$, it

adds *S* to the end of its queue of requesting nodes as if it had received the token and a message *request(S)*.

Chang, Singhal and Liu (1990) also present procedures to deal with faults of communication links and nodes in several situations. We performed two simulations for the case when the link to *nearS* fails before node *S* sends the token to it (see pseudo code and explanation in Section 5.7.1).

5.7.1. The Algorithm

Description in Pascal-like pseudo-code with comments above the code:

Each node has the following declarations:

Constant *my_id*: integer; it contains the node's id number

Variables

requesting: Boolean; initialized with false (except for the node that has the token initially)

executing: Boolean; initialized with false (except for the node that has the token initially and it is executing the critical section)

near: integer; id number of any neighbor node on an outgoing path that leads to the root node (where the token is)

out: array of integers; it contains the id numbers of all the neighbors of the outgoing paths leading to the root

queue: list of integers; it stores the ids of the nodes that are requesting the token
 begin

procedure Request_CS;

begin

(* the request phase *)

requesting = true;

(* When a requesting node does not have the token with it, it enqueues its id at the beginning of its requesting queue and sends a message request to a node in one of its outgoing paths leading to the root. It then waits for the token*)

if (near $\neq$ Me) then

begin

AQ(queue, Me);

if (lqueue = 1) then send a message Request (Me) to near;

wait until (near = Me);

end;

(* the execution phase *)

(* When a node receives the token it becomes the root of the tree, it can then execute the critical section*)

executing = true;

```
enter CS;  
    executing = false;  
(* the release phase *)
```

(* After leaving the critical section, a node sends the token to the first node in its requesting queue and inverts the direction of the edge that leads to this node, i.e. it includes the node on the set of nodes on its outgoing paths leading to the root *)

```
    if (lqueue1  $\neq$  0) then  
        begin  
            near = DQ(queue);  
            send a message Token(lqueue1, Me) to near;  
            out = out  $\cup$  {near};  
        end;  
        requesting = false;  
    end;
```


procedure Request-Handler;

(* When a node that has the token and is not requesting access to the critical section receives a message *request* from another node, it sends the token to the requesting node and includes the requesting node in the set of nodes on its outgoing paths leading to the root *)

begin (* for message Request(X) *)

EQ(queue, X);

if ((near= Me) and (not (requesting))) then

begin

near = DQ (queue);

send a message Token(lqueue, Me) to near;

out = out U {near};

end;

(* If the node does not have the token, it sends a message request to its neighbor in one of the outgoing paths leading to the root if it has not done it*)

else if (near $\neq$ Me) and (lqueue = 1) then

send a message Request(Me) to near;

end;

```

procedure Token-Handler;
begin (* for message Token(Qsize, X)*)
  (* When a node receives the token, if the size of the queue of requests of the sender is
  greater than zero, this means that the sender wants the token back in order to attend the
  requests in its requesting queue. So, the receiver node puts the sender id in its requesting
  queue *)
    if (Qsize > 0) then EQ(X);
    near = DQ(queue);
    out = out - {X};

  (* In order to avoid directed cycles that could lead to deadlock, the direction of the links of
  all outgoing paths is reversed *)
    for each Y in out do
      begin
        out = out - {Y};
        send a message Reverse(Me, near) to Y;
      end;

  (* If the receiver node is not requesting to enter the critical section, it sends the token to its
  neighbor in one of the outgoing paths *)
    if (near  $\neq$  Me) then
      begin
        send a message Token(lqueue!, Me) to Near;
        out = out U {near};
      end;
end;

```

procedure Reverse-Handler;

(* A node that receives a message reverse, includes the sender of the message in the set of outgoing paths leading to the root *)

begin (* for message Reverse(X,Y) *)

 out = out U {X};

 if (X = Y) then near = X;

end;

Communication link failures:

(a) The link to NearS fails before node S sends a message Request(S) to it

Solution: node S sends a message Request(S) along an alternative path, as follows:

if ($|OutS| > 1$) then

begin

 choose a node X ($\neq$ NearS) from OutS and set NearS = X;

 send a message Request(S) to NearS;

end;

(b) The link to NearS fails before node S sends the token to it

Solution: if there is any other requesting neighbor X in QS, node S sends the token to node X as follows:

if ($|QS| > 0$) then

begin

EQ (QS, NearS);

OutS = OutS - {NearS};

NearS = DQ (QS);

OutS = OutS U {NearS};

send a message Token($|QS|$, S) to NearS;

end

else if ($|QS| = 0$) then

send a message Search(NearS) to every connected neighbor;

Otherwise, node S sends a message *Search* (NearS) to every connected neighbor to search for NearS. The message *Search* (NearS) is repeatedly broadcasted to every neighbor of the message receiver, and stops at a node which is either the node NearS or has received this message for more than one time. When node X satisfying NearS = X receives the message *Search* (NearS), it updates NearX as if it had received a message *Token* (Qsize=0, S) as follows:

```

(* When node X receives a message Search (NearS) *)
if (node X has received this message before) then ignore it
else
begin
    if (NearS = X) then NearX = DQ (QX)
    else send a message Search (NearS) to every connected neighbor;
end;

```

(c) The link to node X (X belongs to OutS) fails before node S sends a message *Reverse*(S, NearS) to it. Although this reduces one alternative path leading to node S at node X, there is no deadlock or starvation. Therefore, node S sends a message *Reverse* (S, NearS) again after the link is fixed.

5.7.2. The Implementation in ABCL/R2

This algorithm was implemented in ABCL/R2 considering the case where some links (determined by the user) can fail just before the token must be sent along them. This corresponds to case b) for communication link failures described above. Four test cases were executed (with some faulty links and without faulty links at all), two considering a network with six nodes and two considering a network with ten nodes. The case with six nodes and no faults is based on the example presented in (Chang et al. 1986) illustrated in Figure 16. A simulation of this algorithm starts with the user sending a message *start* to object network containing the following information: the number of nodes of the network, a list with the id number of a neighbor node on an outgoing path for each node (represented

in the description of the algorithm by variable *near*), another list with the id numbers of all nodes on outgoing paths for each node (represented in the description of the algorithm by variable *out*), a list with the id numbers of all nodes that are connected to each node and a list with the id numbers of the nodes on outgoing paths that represent faulty links for the transmission of the token for each node. For the example illustrated in Figure 16, the simulation starts like this: [network <= [:start 6 [2 3 3 1 1 3] [[2 6] [3 6] [] [1] [1 4] [3]] [[2 4 5 6] [1 3 6] [2 6] [1 5] [1 4] [1 2 3] [[] [] [] [] [] []]].

Figure 15 illustrates the responsibilities that are fulfilled in each level. The responsibility of execution tracing is fulfilled implicitly. The responsibility of dealing with communication link failures is fulfilled explicitly, i.e. a base-level object (node) sends a message to its corresponding meta object (meta node) in order to execute the procedure to deal with a link failure. Figures 17a and 17b show the simulation results for the example in (Chang et al. 1986). The second test case considered the same network with faulty links 2-3 and 1-2. Figure 18 illustrates the execution of this test case and Figures 19a and 19b show the corresponding simulation results. Figure 18 illustrates that the procedure for dealing with link failures proposed by Chang, Singhal and Liu works correctly for the case of just one link failure. It solves the problem of link 2-3 failure, but it does not solve the problem of link 1-2 failure, i.e. node 1 does not receive a message *search* that corresponds to a message *token* for it. Another test case for a network of ten nodes and the corresponding results are presented in Appendix A.

NB:

direction of the flow of messages from one object to another

causal connection

the base level shown in this picture represents all base-level objects, the meta level represents all meta objects and the metameta level represents all metameta objects

Figure 15. Description of the responsibilities fulfilled in each level for the simulation of the algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu

Figure 16. Messages exchanged during the execution of the algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu considering six nodes and no faults

5.7.3. Simulation Results

NB: node ? stands for any node of the network

Figure 17b. Metameta-level interaction diagram for the example of execution of the algorithm of Chang, Singhal and Liu illustrated in Figure 16

(a) Node 3 executes the critical section

(b) Node 6 executes the critical section

(c) Node 6 starts a search for node 2. Message search 2 1 is broadcasted as shown. When node 2 receives this message, it sets its queue as if it had received message token 0 6.

Figure 18. Messages exchanged during the execution of the algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu considering six nodes and link failures on links 2-3 and 1-2 at the time that the token must be transferred along these links

Figure 19a. Base-level interaction diagram for the example of execution of the algorithm of Chang, Singhal and Liu illustrated in Figures 16

Figure 19b. Metameta-level and meta-level interaction diagram for the example of execution of the algorithm of Chang, Singhal and Liu illustrated in Figures 18

5.8. An Object-oriented Reflective Approach Versus a Class-based Approach for Simulating Distributed Algorithms

In this section, we summarize our experience in using reflective capabilities in the simulation of distributed algorithms. We compare the object-oriented reflective approach that we used with the class-based approach.

In distributed computing messages play an important role, since they are the only way nodes can communicate. The following examples will show that the use of reflective capabilities of an object-oriented language can better model some situations that occur in the simulation of distributed algorithms.

Example 1: Counting the number of messages received by an object.

Application: The total number of messages exchanged is one of the complexity measures of distributed algorithms.

In the object-oriented reflective model, this is easily implemented by a meta object with a counter variable.

In the class-based model there are two options. In the first one, when an object receives a message, it invokes its method that performs the counting (i.e., the object sends a message to itself). In the second, when an object receives a message, it invokes the same method on a separate "counter object". See Figure 20.

In other words, in the class-based solution an object has to "worry" about counting received messages (i.e., counting is done explicitly by the object). The method call for counting messages is mixed with the code that the object executes when receiving a message. In the reflective object-oriented solution, the same functionality is achieved

implicitly by a meta object, providing a clear separation of concerns and separation of code between the base level and the meta level.

Example 2: Simulating a non-FIFO reception of messages by an object or simulating communication delays.

Application: Some distributed algorithms are suppose to work under the assumption of non-FIFO delivery of messages, i.e. if object A sends message "d" first and then message "e" to object B, object B will receive and process message "e" before message "d".

In the object-oriented reflective model this can be easily simulated by having the meta object exchange the positions of messages in the queue of received messages of the base-level object.

There is no obvious solution for this problem in the class-based model, since there is no reification of the queue of messages of an object.

See Figure 21.

Examples 3a and 3b: Reasoning about messages and performing computational changes as a consequence.

Application: Some distributed algorithms can require different related and interacting functionalities that can be separated in levels, where these functionalities are performed based on reasoning about received messages. An example of that is the simulation of an algorithm for detecting communication deadlock and the simulation of a program (in which communication deadlock should be detected) being done simultaneously (Diaz and Corriveau 1995c).

In the class-based model, different actions upon receiving a message have to be managed by the object that receives the message (and possibly other objects) with the reasoning code mixed with the base-level code. In addition, if those actions involve the creation of instance variables or inheritance of methods, there is no obvious class-based solution.

See Figures 22a and 22b for the case when actions involve assignment statements of instance variables.

Example 4: Changing the message before processing it.

Application: Simulation of reception of corrupted messages, simulation of omission on reception of messages.

In the reflective object-oriented model any message-related failure can be simulated easily, since received messages are reified into entities that a meta object can manipulate. This is done in an implicit way with separation of concerns and code.

In the class-based model, an extra object and two more messages have to be used to simulate a corruption failure in a explicit way with the code for simulation of failures mixed with the base-level code.

See Figures 23a and 23b.

Figure 20. Counting the number of messages received by an object

Figure 21. Simulation of non-FIFO reception of messages

condition = Boolean expression that can involve the sender object, the value of one or more parameters in the message, the value of one or more instance variables of object X, the value of one or more instance variables of meta X, etc.

reflective object-oriented approach

class-based approach

Figure 22a. Reasoning about messages and performing changes in one level

condition1, condition2 = Boolean expression that can involve the sender object, the value of one or more parameters in the message, the value of one or more instance variables of object X, the value of one or more instance variables of meta X, etc.

reflective object-oriented approach

class-based approach

Figure 22b. Reasoning about messages and performing changes in two levels

reflective object-oriented approach

class-based approach

Figure 23a. Simulation of a corrupted message

reflective object-oriented approach

class-based approach

Figure 23b. Simulation of omission on reception of a message

6.0. Reflective Object-oriented Design of Simulations of Distributed Algorithms and a Concurrent Reflective Object-oriented Architecture for Simulating Distributed Algorithms

6.1. ROOD - Reflective Object-oriented Design Applied to the Simulations of Distributed Algorithms

"Object-oriented programming is not simply a few new features added to programming languages. Rather, it is a new way of thinking about the process of decomposing problems and developing programming solutions" Budd (1991). We think that something similar can be said about reflection in object-oriented programming. It might not represent a completely new paradigm, but it requires a new way of thinking about decomposing problems and developing programming solutions. The difficulty of finding a general reflective object-oriented design methodology resides in the fact that common concepts for different reflective object-oriented languages (or object-oriented languages that provide some sort of reflective capabilities) have not been established so far. For instance, Smalltalk (Goldberg and Robson 1983) uses the concept of meta-classes, instead of meta-objects, as in ABCL/R and ABCL/R2 (Matsuoka et al. 1991a). Also, ABCL/R2 is based on a different reflective architecture than the one of ABCL/R, providing additional features.

The motivation for our design methodology called ROOD is the necessity of designing the simulations of distributed algorithms as a combination of reflective modules to be implemented in a concurrent reflective object-oriented language. ROOD is also an attempt to alleviate some of the disadvantages of reflection mentioned by Maes (1987), as explained in Chapter 3 (Section 3.1).

In his tutorial notes of OOPSLA'95 (Open Implementation Analysis & Design of Substrate Software), Kiczales introduces the notion of reflective modules and open implementation. In his opinion, the best implementation strategy for a module cannot always be determined without knowing how the module will be used in a particular situation: "Black boxes"¹ do not always work because implementation issues are locked in the black boxes by implementors and cannot be changed by clients. Opening an implementation, according to Kiczales, is the process of opening up module implementations to give more control to clients. To do so, "reflective modules" may be used: "we propose the notion of a reflective module, which is one that not only provides its primary functionality but also provides functionality for negotiating the way in which it provides the primary functionality". Kiczales also mentions some software engineering issues related to the use of reflection, in particular, reflective modules.

Reuse of software components has always been the object of study by software engineers. Examples are: subroutines (reuse of sequences of statements), modules (reuse of similar subroutines) and classes (reuse of subroutines and data). The notions of "open implementation" and "reflective module" are new approaches to reuse, but the question of how to use these approaches effectively in software development is still not answered. For example:

a) What qualities must be maintained in a reflective module?

b) Open implementations are very powerful tools. They can provide programmers with enough power to "shoot themselves in the foot". How should this be managed? What methodology and tools can help programmers make proper use of open implementations?

¹ "Black-box abstraction is, we claim, just one such abstraction framework - one in which we break a system down into functional modules, with interfaces that abstract away implementation and keep only functionality explicit" (Kiczales 1995)

c) How does a reflective module affect testability? How do you test a reflective module? How do you test the client of a reflective module?

d) How do we document reflective modules?

Software Engineering has advertised modules as a way of decomposing systems. Modules should hide a design decision from the rest of the system. Thus, it follows that modifications of design decisions should affect (modify) the module (or modules) associated with it. Modules are an attempt to provide locality (Jacobson et al. 1992).

6.1.1. A Reflective Object-oriented Model for ROOD

In the search for a reflective object-oriented model in order to describe our reflective object-oriented design methodology applied to simulation of distributed algorithms, we were certainly influenced by some existing reflective object-oriented architectures and languages (Matsuoka et al. 1991a). In fact, we were strongly influenced by the actor-based concurrent reflective object-oriented language ABCL/R2 (Matsuoka et al. 1991b), since it is the language that we have used in our simulation experiments. Consequently, our model is based on the architecture of ABCL/R2, in which a meta object (on a meta level) is the CCSR of a base-level object which belongs to a group of base-level objects. A meta object is also an object that can have a meta object (a metameta object with respect to the base level), through lazy creation on demand. Similarly, a metameta object can have a meta object (a metametameta object with respect to the base level). The same applies to higher levels, recursively characterizing a reflective tower. We consider a meta object, a metameta object etc., as reflective modules because they are the CCSR of an object, providing the user with the ability of controlling computational aspects of an object. Therefore, in the

ROOD model, a meta-level reflective module and a meta object are the same entity. Also, base-level objects belong to a group of objects. In Section 6.2 we propose an extension of this model, (also based on ABCL/R2) in which the concept of a reflective module and a meta object are distinct.

6.1.2. ROOD's Methodology for Reflective Object-oriented Design of Simulations of Distributed Algorithms

ROOD attempts to unify object-oriented design and reflective design of simulations of distributed algorithms. More precisely, ROOD integrates reflective design with the collaboration responsibility of object-oriented design of Wirfs-Brock et al. (1990). See Subsection 6.1.2.2. The next subsection provides a specification of a distributed computation. It involves the simultaneous execution of several distributed algorithms² to be used as an example to illustrate all the phases of our design methodology explained in Section 6. .2.2.

6.1.2.1. An Example for Explaining the ROOD Methodology

In this subsection we provide an example that is used for explaining the ROOD methodology applied to the simulation of several distributed algorithms simultaneously in Subsection 6.1.2.2.2.

² These algorithms are strictly for illustrative purposes, and are not meant to be seen as correct.

Requirements specification:

Each of the following distributed algorithms assume a complete network topology. Each node has a unique identification number (an integer), called `nodeId`. There are 20 nodes. Messages between nodes 1 and 7 and between nodes 3 and 7 as related to the execution of the algorithm for mutual exclusion are delivered in an arbitrary order. Each node has a local clock that is updated according to the rules established by Lamport (1978) as explained in Section 2.1. Each node executes the same computation which is described below (in pseudo code).

Local data of each node:

`nodeId` : integer; it identifies a node.

`requesting` : Boolean; it is false initially, it shows whether a node is requesting access to the critical section or not.

`executing` : Boolean; it is false initially, it shows whether a node is executing the critical section or not.

`queue of requesting nodes` : a list; initially empty. It stores the ids of the nodes that are requesting the token to have access to the critical section.

`DS` : a set; initially empty, it is the set of nodes from which this node is waiting for a message to resume execution.

`detector` : id of the node from which this node receives a deadlock detection message.

`detecting` : Boolean; it is false initially, it shows whether a node has initiated a deadlock detection or not.

`deadlocked` : Boolean; it is false initially. It shows whether a node is deadlocked or not.

clock : integer; initially zero. It is updated each time a message is sent or received.

tentative : integer; initially zero. It shows the number of times that a node has been communication deadlocked.

success : integer; initially zero. It shows the number of times that a node managed to receive all reply messages that it was waiting for.

receivedreplyhello: Boolean; initially false. It shows whether a node has received a (REPLY_HELLO, id) message or not.

Description of the algorithm for mutual exclusion in the access of the critical section (CS) :

When receiving message (TRY_ENTER_CS)

 requesting = true;

 broadcast message (REQUEST_TOKEN, nodeId);

 update clock

When receiving message (REQUEST_TOKEN, Id)

 update clock;

 if (token is present in this node) and not requesting

 then begin

 send message (TOKEN, queue_of_requesting_nodes) to sender

 identified by Id;

 update clock;

 end;

 else append the Id received in the message (REQUEST_TOKEN, Id)

in the queue_of_requesting_nodes

When receiving message (TOKEN, queue)

update clock;

append queue to queue_of_requesting_nodes;

if requesting

then begin

executing := true;

Execute the critical section;

Leave the critical section;

executing := false;

requesting := false;

end

If (queue_of_requesting_nodes is not empty)

then begin

dequeue the first element (id) of queue_of_requesting_nodes;

send a message (TOKEN, queue_of_requesting_nodes) to node

with corresponding id;

update clock;

end

Code to be executed in the critical section:

Sleep for a random amount of time;

Description of the algorithm for detecting communication deadlock :

When receiving message (DETECT_DEADLOCK, id)

update clock;

detector := id;

If (DS is not empty)

then begin

send a message (DETECT_DEADLOCK, nodeId) to all nodes of DS;

update clock;

end

else send a message (REPLY_DEADLOCK, nodeId) to sender;

update clock;

When receiving message (REPLY_DEADLOCK, Id)

update clock;

If (all DS nodes replied)

if detecting

then begin

deadlocked = true;

detecting := false;

end

else send a message (REPLY_DEADLOCK, nodeId) to detector;

else wait for other replies

When receiving message (DETECT_DEADLOCK)

detecting := true;

send a message (DETECT_DEADLOCK, nodeId) to all nodes of DS

Application program :

/* A node finishes executing this program when this node has tried three times to receive at least one message REPLY_HELLO from one of three other nodes (chosen randomly), but became communication deadlocked or when this node succeeds in receiving six messages REPLY_HELLO from the other three nodes. When a node is executing the critical section, it should not reply to a message HELLO*/

When receiving a message (WAKE)

update clock;

While ((tentative < 4) and (success < 7)) do

begin

receivedreplyhello := false;

Choose three nodes randomly;

Sleep for a random amount of time;

Send a message (HELLO, nodeId) to each of the three nodes;

update clock;

Include the ids of these three nodes in DS;

Send a message (TRY_ENTER_CS) to itself;

if (not receivedreplyhello)

then send a message (DETECT_DEADLOCK) to itself;

```

Wait until deadlocked or receivedreplyhello;
If (deadlocked) then tentative := (tentative + 1);
end

```

When receiving a message (HELLO, Id)

```

update clock;
if not executing
send a message (REPLY_HELLO, nodeId) to the sender;
update clock

```

When receiving a message (REPLY_HELLO, Id)

```

update clock;
receivedreplyhello := true;
success := (success + 1);
remove the sender's id from DS

```

When program is finished, generate execution reports.

The algorithms and application program should be tested with respect to fault tolerance, considering the following situations:

a) Some of (REQUEST_TOKEN, nodeId) messages are corrupted, the value of nodeId in the message is changed. This can cause nodes that are not requesting the critical

section to receive the token. In this case, the token is forwarded to a node whose id is the first of the requesting queue.

b) Some of (REQUEST_TOKEN, nodeId) messages are not received (omission failures). In this case, the requesting node might not receive the token, which grants it access to the critical section.

c) Some of (REPLY_DEADLOCK, nodeId) messages are corrupted, the value of nodeId in the message is changed. In this case, since each node knows the contents of DS, the recovery procedure is simply to start a communication deadlock detection again.

d) Some (REPLY_HELLO, id) messages are not received (omission faults).

The algorithm for mutual exclusion should also be tested for the case when some messages between node1 and node7 and between node3 and node7 are not delivered in FIFO order.

The following information about the execution of the algorithms and the application program should be generated (execution reports):

Execution report #1:

- 1) A trace of the execution of the application program.

Execution report #2:

- 1) The total number of messages received by each node for each algorithm at the end of the execution of each algorithm.
- 2) A trace of the execution for each algorithm.

Execution report #3:

- 1) The number of REQUEST_TOKEN messages received by each node.
- 2) The value of the local time of each node every time this node was granted access to the critical section.

6.1.2.2. The ROOD Methodology

6.1.2.2.1. The Methodology Applied to the Simulation of One Distributed Algorithm

Our methodology is based mainly on finding responsibilities of base level objects and meta objects and collaborations between all these objects. We do not take into account the need to find classes, as we assume that inheritance is achieved through reflection in an actor-based computational model, as employed by ABCL/R2.

The methodology applied to the simulation of just one distributed algorithm is very straightforward, as follows. In the simulation of distributed algorithms, nodes of the network can be mapped into objects easily due to the similarities of the object-oriented model and the distributed computing model (see Chapter 3). Assuming a distributed algorithm where all nodes execute the same code, it is possible to write code (scripts, using ABCL/R2 terminology) for one node, let us say node1 (object node1), and make the other nodes reuse this code. Since ABCL/R2 does not provide classes, objects inherit scripts from other objects (specified as inheritance source) directly. Therefore, all the objects that represent processing nodes should inherit scripts from object node1 through reflection. Since ABCL/R2 only provides inheritance of scripts, state variable definitions have to be

specified for all nodes. So, the first design step of our methodology consists of defining the scripts and state variables of a processing node. This should be easy, assuming that a distributed algorithm fully describes the states of a node, the messages that are exchanged and what happens when a node receives each message. Assuming that the base level code takes care of the simulation of the algorithm, the second step is an exploration of responsibilities that could be fulfilled at the different meta levels. We call this exploration the "macro view". The third step deals with the interaction between objects of different levels in order to accomplish the responsibilities determined in step 2. We call this the "micro view".

Summarizing:

- a) Find the states and scripts that implement the distributed algorithm.
- b) Taking into account the characteristics of the algorithm that should be executed at the base level, try to assign responsibilities to meta levels, such as: monitoring of exchanged messages, monitoring of changes of node state, non-FIFO delivery of messages, generation of faults, reconfiguration of the network, implementation of a local clock etc. These are common to distributed algorithms. In general, these responsibilities are related to the operations to be implemented in a modular way, i.e. the code that corresponds to these operations should not be mingled with the code of the algorithm. These operations can also require the use of reflective capabilities, only available at higher levels than the base level.
- c) Build interaction diagrams (Jacobson et al. 1992) including the base level objects and their meta objects for each message that a base-level object can receive in order to verify if the responsibilities of each level are fulfilled. These interaction diagrams should represent the collaboration relationship between base-level objects and meta objects. This collaboration relationship comprises messages that are exchanged between the base level

and the meta levels and dynamic base level changes that are performed at the meta levels through the use of reflective capabilities.

These steps lead to:

- 1) A list of objects and meta objects.
- 2) A description of responsibilities to be accomplished at each level.
- 3) A description of the collaborations between objects and meta objects in order to accomplish base-level and meta-level responsibilities.

After the implementation of base level and meta level objects, an interaction diagram for different executions of the algorithm can be produced. These diagrams can serve as a validation of the design and implementation of the simulation, assuming that the algorithm is correct. In (Diaz and Corriveau 1995c), we illustrate the methodology by applying it to a case study: the algorithm for communication deadlock detection of Chandy, Misra and Haas (1983).

6.1.2.2.2. The Methodology Applied to the Simulation of Several Distributed Algorithms Simultaneously

In this subsection, we present a more detailed description of our methodology applied to the example described in Subsection 6.1.2.1 which requires more than one distributed algorithm to be simulated simultaneously.

Phase 1. Finding Base-level Objects

Since we consider an actor-based model, there is no need to establish a hierarchy of classes. Only objects must be found.

"Objects can be found from the requirements specification or from the description of the design goals" (Wirfs-Brock 1990). "Objects are naturally occurring entities of the application domain, the ones that are relevant to the system are the ones to be considered" (Jacobson et al. 1992).

In the given example, there are twenty "node objects" that are responsible for the execution of the (distributed) application program and the (distributed) algorithms for mutual exclusion and communication deadlock detection. The application program could possibly be executed at any level, but in order to have a uniform approach, we assume that it should always be executed at the base level. The other algorithms that execute based on the state of the application program are implemented at higher levels (where reflective capabilities are available with respect to the level immediately below) than the base level. The scripts and state variables of the application program should then be defined from the specification.

Phase 2. Finding Meta Objects for Each Level above the Base Level

As a general rule, a meta object is necessary whenever a base-level object has to present several unrelated functionalities that require dynamic inspection or changes of the base-level object's computational aspects. This makes a meta object look like "an object that comes from inside a base-level object" in the sense that in some situations there is a need to

break the black-box model of an object. In addition, from an object-oriented point of view, meta objects can also be used as reflective modules (software components) even when reflective capabilities are not used. If the use of reflective capabilities is necessary, then meta objects should be identified after base-level objects have been identified. Identifying base-level objects and meta objects requires different approaches. Finding base-level objects is known as the process of mapping logical objects of the problem space to actual base-level objects in the program. A meta object (or a metameta object and so on) is found whenever it is necessary to inspect or change computational aspects of the corresponding base-level object (or meta object and so on) dynamically. In the example of Section 6.1.2.1, two distributed algorithms have to be executed simultaneously: one to ensure mutual exclusion in the access of the critical section and the other to verify the existence of a communication deadlock. Since these two algorithms deal with two distinct issues (that represent distinct functionalities), separate reflective modules (meta objects) in distinct meta levels should be used for each.

The need to use reflective capabilities for the two algorithms and the application program is explained in the following paragraphs:

The application program requires tracing of its execution which should be done at a higher level (where messages are reified) than the base level in order not to mingle the code for tracing with the application code.

When access to the critical section is granted to a node by executing the algorithm for mutual exclusion, a variable must indicate this fact in the application program and in the code of the algorithm. The base-level object's instance variable *executing* serves this purpose and must be updated during the execution of the algorithm for mutual exclusion. In the application program, when this variable has the value true, a reply is forbidden for a

HELLO message (i.e. if a HELLO message is received when the node is executing the critical section, this message is discarded). The statistics demanded by the execution report #3 requires that the value of the local logical clock (represented by the base-level object's instance variable *clock*) and the value of the base-level object's instance variable *requesting* be inspected when a node receives a message TOKEN and is granted access to the critical section. Consequently, reflective capabilities must be used, if possible, in order to update the variable *executing* from the level where the algorithm for mutual exclusion is executed. Also, for the sake of separation of concerns, the reasoning about messages (HELLO and TOKEN in this case) which require inspecting the base-level variables *executing* and *requesting* should be done at a higher level than the one where the algorithm for mutual exclusion is executed (where reflective capabilities with respect to the base level are not available).

The algorithm for communication deadlock detection requires inspecting a set (that contains the ids of the nodes from which a node is waiting for a message in order to resume execution) called the dependent set (DS) of this node (represented by a base-level object instance variable DS). This set must be updated in the application program whenever a node begins waiting for a message from another node and whenever a node receives a message from one of the nodes of the set. Thus, this set is defined as data of the distributed application, but should be inspected during the execution of the communication deadlock detection algorithm (since, in this algorithm, a node that wants to detect whether it is deadlocked or not sends a message DETECT_DEADLOCK to all nodes in its DS). Also, this algorithm requires updating the base-level instance variable *deadlocked*. Therefore, reflective capabilities should be used, if possible to inspect the value of DS and to update the value of the variable *deadlocked*.

A logical clock (represented by a base-level instance variable *clock*) is used in both algorithms and in the application program. It must be updated in the application program whenever a message is sent or received and it must also be updated by the two algorithms whenever a message is sent or received. So, the variable *clock* must be updated from higher levels than the base level (where the algorithms are implemented) using reflective capabilities, if possible. Depending on the level where the algorithms are implemented a causal-connection link to the base level might not exist.

In both algorithms faulty messages must be simulated, which must be done in a level higher (where messages are reified) than the one where each algorithm is implemented. The same observation applies to the statistics collection on the number of messages and the tracing of the algorithms. On the other hand, the simulation of non-FIFO delivery of messages required by the algorithm for mutual exclusion must be done at a level *immediately* above the level where this algorithm is implemented because it is where the queue of messages for this algorithm is reified.

The decision of executing one algorithm or the other at a meta level (using meta objects) or at a metameta level (using metameta objects) can influence the number of messages exchanged between meta levels since in ABCL/R2 the structural and computational aspects of a base-level object are reified only by its meta object (in the level right above the base level). For instance, if we assume that the algorithm for communication deadlock detection is implemented at the meta level and the algorithm for mutual exclusion is implemented at the metameta level, the following situation will arise: each time the base-level instance variable DS needs to be inspected from the meta level, this inspection can be done using available reflective capabilities by the meta object. If this algorithm was to be implemented at the metameta level, DS would have to be inspected

from the metameta level, the metameta object would have to send a message to the meta object or to the base-level object (asking for the value of DS) and receive a message from the meta object or from the base-level object (with the value of DS). This would defeat the purpose of using reflective capabilities from different levels to reduce the number of messages exchanged between levels in order to simplify the execution of the algorithms. Similar arguments are applicable to the base-level variable *deadlocked* that must be updated by the algorithm for communication deadlock detection.

A similar situation occurs with respect to the implementation of the algorithm for mutual exclusion, since the base-level object instance variable *executing* must be updated by this algorithm from a level above the base level. In addition, in both situations the logical clock represented by the base-level instance variable *clock* would require updating messages. Figure 24 illustrates those two situations. Based on the number of messages that must be exchanged in order to update and inspect base-level variables in both situations, we decided to place the algorithm for communication deadlock detection at the meta level and the algorithm for mutual exclusion at the metameta level. At this point it is clear that the use of reflective capabilities in order to change the state of the computation is a violation of encapsulation, which is a trade off in order to obtain an open simulation environment.

Phase 3. Finding the Responsibilities of Each Reflective Module

Responsibilities, according to Wirfs-Brock (1990), comprise the knowledge that an object maintains and the actions that an object can perform.

Responsibilities represent the services (publicly available) that the object (working as a server) can provide in the form of contracts. A contract between two objects is a list of

services that one object can request to the other. A base-level object can play the role of a server for another base-level object that requests services from it.

This client-server relationship also exists in the case when a base-level object requests a service directly to its meta object by sending a message to its meta object. In this case, we consider the relationship *explicit*. In addition, there is a client-server relationship between a base-level object and its corresponding meta object whenever a meta object provides its services at the time that a message is received and processed on behalf of its corresponding base-level object. In this case, we consider the relationship *implicit*.

From the specification of the application program (of the example), it is possible to identify functionalities (that require reflective capabilities, called "reflective responsibilities") to be implemented at the meta level or at a higher level, e.g. tracing of the execution of the application program.

From the specification of the algorithm for mutual exclusion it is possible to identify reflective responsibilities to be implemented at the metameta level: update the value of *executing* and update the value of *clock* (defined at the base level). The following reflective responsibilities to be implemented at higher levels than the metameta level are: simulate omission and corruption faults of messages REQUEST_TOKEN, simulate non-FIFO delivery of messages between nodes 1 and 7 and between nodes 3 and 7, trace the algorithm, monitor the number of messages (related to the execution of this algorithm) received, monitor the number of messages REQUEST_TOKEN received and monitor the value of the local time when a node is granted access to the critical section.

From the specification of the communication deadlock detection algorithm, it is possible to identify reflective responsibilities to be implemented at the meta level: inspect the value of DS, update the value of *deadlocked* and update the value of *clock* (defined at

the base level). The following reflective responsibilities to be implemented at higher levels than the meta level are: simulate corruption of some messages `REPLY_DEADLOCK`, trace the algorithm, monitor the number of messages (related to the execution of this algorithm) received.

In assigning reflective responsibilities, the guidelines of Wirfs-Brock (1990) also apply: distribute system intelligence evenly, state responsibilities in a general way, as much as possible, keep behavior with related information, if any, keep information about one thing in one place and share responsibilities among related objects.

We explore these heuristics below.

Distribute System Intelligence Evenly

In object-oriented design, the amount of responsibilities that an object has is a measure of the amount of intelligence that it embodies. For reflective object-oriented design, intelligence should be distributed among base-level objects, meta objects, metameta objects etc. in a way that makes it easy to modify the system (for scalability and maintainability).

The responsibilities of a meta object (i.e. reflective responsibilities) associated to a base-level object are in fact responsibilities of the base-level object related to a necessity of reasoning about and changing the state of the base-level computation. The same situation applies recursively to any level and the level above it. In terms of responsibilities, distribution and replacement, a reflective system presents advantages because meta objects can serve as software components. A meta object can be "plugged to" and "unplugged

from" a base-level object, associating or disassociating responsibilities and services (that are provided in a implicit or explicit way) respectively.

The disadvantage of the ROOD model (based on the architecture of ABCL/R2) is that a meta-level reflective module and a meta object are the same entity (similarly for other levels, i.e. a metameta level, a metameta-level reflective module and a metameta object are the same entity etc.). This makes a fine-grained distribution of reflective responsibilities at any level impossible. In Section 6.3 we propose a more flexible design model (Extended ROOD model) that overcomes this disadvantage.

State Responsibilities in a General Way

If reflective responsibilities are stated in a general way, it is easier to find common reflective responsibilities shared by meta objects, metameta objects etc. In the given example, updating the local clock, monitoring the number of messages received, tracing the execution and generating message faults are common reflective responsibilities of the algorithm for communication deadlock detection and of the algorithm for mutual exclusion. Also, tracing the execution is also a reflective responsibility of the application program.

A separate meta level (meta level X) above the levels that implement each algorithm and the application program should be used just with the purpose of fulfilling the responsibility of generating message faults. An additional level should be used for monitoring messages, reasoning about messages and tracing the execution (of the algorithms and the application program), since messages that are sent to levels below level X are intercepted at level X. Each of these messages is a parameter of each message received (intercepted) at level X, i.e. messages to levels below level X are reified at level

X. So, those responsibilities are fulfilled in an implicit way with the exception of the reasoning of messages. The reasoning of messages requires the inspection of a base-level instance variable as explained in phase 2. This inspection must be done in an explicit way, since reflective capabilities with respect to the base level are not available above the levels that implement each algorithm. Updating the local clock (represented by the base-level variable *clock*) during the execution of the algorithm for communication deadlock detection can also be done in an implicit way. This is possible because this algorithm is implemented at the meta level and there is a causal connection link between the meta level and the base level. On the other hand, for the algorithm for mutual exclusion the responsibility of updating the local clock must be done in an explicit way because this algorithm is implemented at the metameta level, where there is no causal connection link with the base level. This distribution of responsibilities in meta levels consists in making use of the "reflective tower" feature of the ROOD model. This is why in this case the design is "vertical" (see Figure 25). Extended ROOD is more flexible, allowing "horizontal" and "vertical" design (explained in Section 6.3), i.e. for a base-level object, more than one meta object can be placed in a meta level (similar situation applies to any level).

Keep Behavior with Related Information (Locality)

Keeping behavior and its related information means to encapsulate data and operations that manipulate this data in one place. This provides a better understanding of the system and a better way of changing the system without interfering with other parts of the system. Meta objects, like objects, can be considered as entities that encapsulate data and operations with the additional advantage of providing reflective capabilities.

Unfortunately the ROOD model tends to make locality difficult to achieve because it provides a limited way of placing responsibilities in meta levels, since a meta level, a meta-level reflective module and a meta object are the same entity.

Keep Information about one Thing in one Place

In object-oriented design, this guideline aims at preventing the sharing and duplication of information, which could cause inconsistencies and worsen understandability. A meta object can be a way of preventing the duplication of information, assuming that any base-level object information can be accessed by the corresponding meta object through the use of reflective capabilities. The causal-connection link between the two levels definitely prevents inconsistencies since changes made at the meta level are immediately updated at the base level or vice-versa. Conversely, the use of these reflective capabilities does not make the system easier to understand.

Share Responsibilities

In object-oriented design, sometimes it is necessary to divide a responsibility into several smaller ones in order to better assign them to different objects. It can make sharing of responsibilities an easier task. This is also applicable to reflective object-oriented design. For instance, in the given example, if a reflective module is only responsible for monitoring the number of messages received, it can be used in three situations: to monitor all messages received in the execution of the communication detection algorithm, to monitor all messages received in the execution of the mutual exclusion algorithm and to monitor all messages

REQUEST_TOKEN received. However, as mentioned before, it is very difficult to obtain a fine-grained distribution of reflective responsibilities in any level using ROOD.

Figure 25 shows the responsibilities associated with each reflective module (based on the discussion in phases 1 and 2) for the example using the ROOD design model and methodology.

Phase 4. Finding the Collaborations between Objects and Meta objects

Responsibilities and collaborations are related. According to Wirfs-Brock (1991), "Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A collaboration is the embodiment of the contract between a client and a server." In object-oriented design, collaborations are one-way interactions and they exist each time an object cannot fulfill a responsibility alone and consequently needs the help of other objects. In this case the object exchanges one or more messages with one or more supplier objects. Consequently, a responsibility may be fulfilled by one or more collaborations. In object-oriented design, collaborations serve to show the flow of control and information during the execution of the program. Collaborations in reflective object-oriented design also have the purpose of showing the points in the computation where reflective capabilities are used and where encapsulation is being violated (as a consequence of the use of reflective capabilities with the purpose of changing the state of the computation). The tasks of identifying client objects, server objects, contracts and collaborations are interrelated (e.g. if there is a client and a server, there must be a

corresponding collaboration represented by a contract to fulfill a certain responsibility. If any of these entities exist without any of the others, there is a flaw in the design).

A base-level object or a meta object, or a metameta object, etc. can fulfill a responsibility by executing actions as long as it has the appropriate knowledge to do so. If it does not have such knowledge, it is necessary to state clearly the requirements of the object and which other objects (base-level, meta-level, metameta-level, etc.) can provide relevant services. These requirements can be provided implicitly or explicitly depending on the type of client-server relationship (explained in phase 3). In the given example, all collaborations are implicit except in the following cases:

a) When a base-level object sends a message to its meta object to execute a communication deadlock detection,

b) When a base-level object sends a message to its metameta object to execute mutual exclusion,

c) During the execution of the algorithm for mutual exclusion (when a metameta object sends a message to its base-level object to update the value of the variable *clock* and when a metameta object sends a message to its base-level object to update the value of the variable *executing*),

d) When message reasoning is performed in two situations. First, when a message HELLO is received, a message is sent to the base-level object in order to retrieve the value of the variable *executing*. Second, when a message TOKEN is received, a message is sent to the base-level object in order to retrieve the value of the variable *requesting*. Also, when *requesting* is true, another message is sent to the base-level object in order to retrieve the value of the variable *clock*.

This list illustrates how dangerous the use of reflective capabilities can be by making almost all collaborations implicit.

Phase 5. Hierarchies

In object-oriented design, according to Wirfs-Brock (1990), the purpose of this phase is to analyze what has been done in the other phases in order to accomplish a final design. The inheritance hierarchies are structured in order to take the most advantage of the object-oriented technology. Graphical and conceptual tools are necessary to provide a global understanding of the design, necessary to the analysis, such as: hierarchy graphs, Venn diagrams and contracts. Graphical tools can also help visualizing the composition of reflective modules in different levels. Figure 26 is an example of that.

Figure 24. Choosing the levels in which to implement each algorithm of the example of section 6.1.2.1

Figure 25. Reflective object-oriented design of the example of section 6.1.2.1 using the ROOD model

Phase 6. Visualizing the Collaborations between Objects of Different Levels

We use interaction diagrams (Jacobson et al. 1992) as a visualization technique for the collaborations between objects in order to fulfill responsibilities. These diagrams include base-level objects, meta objects, metameta objects etc. An interaction diagram is built for each message that any object can receive, in order to verify if the responsibilities of each level are fulfilled.

6.1.2.2.3. A Proposal to Alleviate the Disadvantages of Reflection as Pointed by Maes (1987)

In Chapter 3 (Section 3.1) we describe the disadvantages of reflection as pointed by Maes. In this section, we suggest a few design techniques that could alleviate those disadvantages with respect to reflection in object-oriented programming.

In order to make reflective programming (i.e. programming using reflective capabilities) more straightforward, it is necessary to show the interactions between the base level and the meta level. The same applies to the meta level with respect to the metameta level, etc. A meta object is an object that performs an action only in two situations: whenever it receives a message that is directed to its corresponding base-level object or whenever it receives a message that is directed to itself. So these situations must be monitored in order to determine if the computation of the meta level is acting upon the computation of the base level. This monitoring can be done statically through the meta-

object protocol specification which must contain information about the possible computational changes of the corresponding base-level object.

Using the example of Section 6.1.2.1:

The protocol for the meta object that executes the algorithm for communication deadlock detection:

message DETECT_DEADLOCK :

The following services are provided implicitly to each node that receives this message:

- value of base-level instance variable DS is inspected, but not modified
 - at the level where the algorithm is executed (meta level);
- sender, message and receiver are printed (trace)
 - at a higher level than the one where the algorithm is executed;
- number of messages received is incremented and printed
 - at a higher level than the one where the algorithm is executed;

message **REPLY_DEADLOCK** :

The following services are provided implicitly to each node that receives this message:

- value of base-level instance variable **DS** is inspected, but not modified;
- value of base-level instance variable *deadlocked* is updated (if node is deadlocked);
- sender, message and receiver are printed (trace)
 - at a higher level than the one where the algorithm is executed;
- number of messages received is incremented and printed
 - at a higher level than the one where the algorithm is executed;
- corruption of message is simulated randomly
 - at a higher level than the one where the algorithm is executed;

In order to make the computation more understandable, interaction diagrams can help as a visualization technique, as long as they mark the points in the computation where changes can happen (due to the use of reflective capabilities) and show the changes. For clarity, several interaction diagrams should be drawn, one for each level.

We think that those two guidelines complement each other very well. The protocol specification of a meta object provides static information about the computational changes through the use of reflective capabilities, and the interaction diagrams present the dynamics of the computation, showing the points in the computation where these changes occur.

6.2. The Extended ROOD Methodology, a Generalization of the ROOD Methodology

The generalization of the ROOD methodology (called E-ROOD) is based on an extension of the model used for ROOD. In this extended model each level may contain one or more reflective modules and one or more non reflective modules. A non reflective module does not contain any meta object, meta meta object etc. A reflective module of the meta level must contain at least one meta object. A reflective module of the metameta level must contain at least one metameta object etc. Also, a meta-level reflective module may contain meta-level objects. A metameta-level reflective module may contain metameta-level objects etc.. In this model causal connection links can be set arbitrarily. For instance, a metametameta object can have a causal connection link to a base-level object and another metametameta object can have a causal connection link to a meta object. Causal-connection links can also exist between reflective modules or between a reflective module and a non reflective module. In this case, any component of the higher level (reflective) module is a CCSR of any of the components of the lower level (reflective or non reflective) module. A base-level object is also allowed to have more than one meta-object associated to it. The same situation is applicable to higher levels. We call meta objects, meta-level objects, metameta objects, metameta-level objects etc. components. So, reflective and non reflective modules consist of one or more components.

Meta-level objects do not reify any computational aspects of base-level objects. They just serve as auxiliary objects for meta objects, which are causally-connected representations of base-level objects. A similar situation applies to other levels. A message to a base-level object is received by the highest level and made accessible to all levels

components, in order for them to perform reasoning about the message if they require so. Nevertheless, the message queue is only reified in components that are the CCSR of another component.

As opposed to the ROOD methodology, in the extended ROOD methodology the user must find meta-level objects (metameta-level objects etc.) in addition to meta objects (metameta objects etc.) and base-level objects. Meta-level objects (metameta-level objects etc.) are necessary in order to map functionalities (responsibilities) to be performed at the meta level (and higher levels) that do not require the use of reflective capabilities. Meta-level objects help meta objects fulfil their responsibilities. A similar situation applies to higher levels, i.e. metameta-level objects help metameta-objects etc. A meta-object can, for instance, reason upon the computational state of a base-level object and as a consequence of this reasoning distribute some activities to be performed among meta-level objects.

Using the extended ROOD methodology, for the example described in Subsection 6.1.2.1, the decision of executing one algorithm or the other using reflective modules at the meta level or reflective modules at the metameta level does not depend on any consideration about the number of exchanged messages between the base level and higher levels (when reflective capabilities are not available). The reason is that the extended ROOD model assumes that the computational aspects of a base-level object can be reified by a meta object, a metameta object etc., due to the ability of choosing where to place the causal connection links. We will assume that the algorithm for communication deadlock detection and the algorithm for mutual exclusion are placed in the same reflective module at the meta level in order to provide locality (all algorithms can be found in the same place).

Figure 26 shows the design of the example using the extended ROOD model and methodology. This design shows a fine-grained decomposition of functionalities, i.e. one functionality is assigned to a component. This allows the reuse of components more easily.

Finding the Responsibilities of Components

Distributing System Intelligence Evenly

The fine-grained decomposition that the extended ROOD model provides certainly allows a better distribution of intelligence among components. As opposed to the ROOD model, where a reflective module and a meta object are the same, the smallest unit of the extended ROOD model is the component, which can be placed in any module in any level.

State Responsibilities in a General Way

Reflective responsibilities can be stated in a general way in both models _ ROOD model and extended ROOD model. The difference is that the extended ROOD model is more flexible, allowing "vertical" and "horizontal" design, providing locality with fine-grained distribution of reflective responsibilities.

Keep Behavior with Related Information (Locality)

Components, reflective and non reflective modules and distinct levels represent several different ways to provide locality in extended ROOD. The duplication of

information is not necessary if reflective capabilities are used to access data that is defined in a component by another component which is the CCSR of the former. In the given example, locality is provided in levels: the base level deals with the application program, the meta level deals with the algorithms that the application program uses to detect communication deadlock and for providing mutual exclusion, the metameta level deals with issues related to message delays (simulation of non-FIFO delivery of messages), the metametameta level deals with simulation of faults in messages (omission and corruption) and the metametametameta level deals with the statistics related to messages and message reasoning in two separate modules. In each module, locality is also provided for each component. For instance, in the metametameta module, simulation of omission in the reception of messages is done by one component and simulation of corrupted messages is done by another component of the same module.

Share Responsibilities

Extended ROOD easily allows the sharing of reflective responsibilities among modules and components because of the fine-grained decomposition. For example, Figure 26 shows that the necessity of simulating corrupted messages is common to both algorithms. This reflective responsibility is fulfilled by one metametameta component (N.B. the causal-connection link from the metametameta component to the entire meta-level module). But, simulation of omission in the reception of messages is necessary only for the algorithm for mutual exclusion. This responsibility is also fulfilled by a metametameta component (N.B. the causal-connection link from one metametameta component to the meta-level component that implements the algorithm for mutual exclusion). Similarly,

statistics on received messages is used by both algorithms, but the application program only requires tracing of execution. The causal connection links that come from the metametametameta level show this. As for the algorithms, the causal-connection link goes from the entire meta-level module to the base level, since both algorithms have to fulfill reflective responsibilities with respect to the application program.

Finding the Collaborations between Objects, Meta objects and Meta-level objects

Objects, meta objects and meta-level objects of all levels collaborate with each other in order to fulfill responsibilities. A component can fulfill a responsibility by executing actions, as long as it has the appropriate knowledge to do so. If it does not have such knowledge, it is necessary to state clearly the necessities of this component and which other components (modules) can provide such necessities. Since the extended ROOD model permits placing causal-connection links from any component or module to any component or module, all the collaborations and services provided by the collaborations can be made implicit.

Figure 26. Reflective object-oriented design of the example of section 6.1.2.1 using the extended ROOD model

6.3. Fault-Tolerant-Reflective-Hybrid-Group (FTRHG) Architecture - A Concurrent Reflective Object-oriented Architecture to Study Fault-tolerance in Distributed Computing

Motivated by the work of Babaoglu (see Section 2.5.3) and the experience of implementing simulations of distributed algorithms in ABCL/R2, we now propose a reflective architecture based on the one of ABCL/R2 to study fault-tolerance in distributed computing. Our proposed architecture is a unification of the architecture of ABCL/R2 and the state machine model (Schneider 1990) explained in Section 2.5.2, since the state machine model has been used to study fault tolerance in distributed computing.

6.3.1. Common Features of ABCL/R2 and the State Machine Model

The fundamental characteristics of the State Machine model match the way objects can behave in ABCL/R2. A State Machine processes requests one at a time following causality rules:

(a) If a single client issues request r_1 before request r_2 to a State Machine sm , r_1 will be processed before r_2 by sm . In ABCL/R2, if object X sends a message b after sending a message a to object Y , object Y receives message a before message b .

(b) If client c_1 makes a request r_1 to a State Machine sm that causes client c_2 to make request r_2 to sm , sm processes r_1 before r_2 . In ABCL/R2, if object X sends a message a to object Y that causes object W to send a message b to object Y , object Y receives message a before message b .

(c) According to the state machine model, a computation is replicated on processors that are physically isolated. This can be modeled in ABCL/R2 by considering each replica of each computation as a base-level object belonging to a group of base-level objects.

(d) The state machine approach allows separation of fault-tolerance from other aspects of the system design. The reflective approach of ABCL/R2 also allows a separation of different design concerns, which can be encapsulated in several meta levels.

In addition, the outputs of a State Machine are completely determined by the sequence of requests it processes. They do not depend on time and any other activity in a system. Objects in ABCL/R2 behave according to the processing that is done caused by the reception of messages, independently of time.

6.3.2. Unifying the State Machine Model and a Reflective Concurrent Object-oriented Architecture: A Reflective State Machine

Our proposed unification (Diaz and Corriveau 1995b) is based on modeling the characteristics of the state machine in the object-oriented concurrent reflective architecture of ABCL/R2 in order to simulate a distributed computation subject to failures. Figure 28 illustrates the Fault-Tolerant-Reflective-Hybrid-Group architecture — our unified architectural model, which presents the following features:

(a) A faulty replica of a computation is an object that is placed in a group of faulty objects (faulty group). Each faulty group is characterized with a type of failure and the probability that this failure can occur.

(b) Non-faulty replicas of a computation are objects placed in the non-faulty group.

(c) The type of failure and the probability that occurs for a group of objects (replicas) is implemented in the meta object generator (defined by the programmer). Therefore, in this model the probability that an object presents any type of failure is defined by its meta object. This might sound like a contradiction for Byzantine failures because they happen in a completely arbitrary way. Nevertheless, it is possible to consider that a processor presents arbitrary behavior with a certain probability. Since each object belongs to a group that has a unique meta object generator, objects of different groups are necessary to simulate the occurrence of different types of failures with different probabilities.

(d) Each replica of the computation is seen by its corresponding meta object as an instance of a state machine with a set of state variables and a set of commands. The computation execution of each replica can be monitored by its corresponding meta object. This monitoring is possible because each time a message is sent to an object, the message is first received by the corresponding meta object. A meta object can then inspect its object's state and/or behavior before and after the receipt of a message and store this information as meta-level data. These meta-level data can be used to generate simulation reports. The meta object has also the opportunity to reason about the message, and perform changes to its object's state and/or behavior by using the available reflective capabilities (i.e. reflective procedures). The occurrence of failures can be part of this process of reasoning. Consequently, simulation of occurrence of failures is done at the meta level, independently of the computation performed at the base level (achieving separation of concerns).

(e) According to the state machine model, fault tolerance can be implemented in a group of state machine copies by removing state machine copies running on faulty processors and adding copies running on repaired processors (reconfiguration). This

should be modeled in ABCL/R2 through dynamic inclusion in and removal of objects (replicas) from a group by the group manager. Presently, ABCL/R2 provides dynamic creation of objects in a group but no dynamic removal of objects from a group.

(f) Reports can be generated through meta-level programming in a structured way, without interfering with the code of other meta levels or base level. These reports can provide information such as: the number of messages exchanged, the number of rounds of exchanges among the objects and a log file of all the messages received by an object when executing an algorithm for agreement on the output value.

With our unified approach (called a reflective state machine), we are not expecting to solve problems of fault tolerance that were already proven to be unsolvable. We intend to obtain an object-oriented reflective simulation environment for the study of fault tolerance in which no ad hoc coding is required because of the availability of reflective capabilities.

Figure 27. Unified architecture model of ABCL/R2 and state machine replicas

7.0- Conclusion

This dissertation has proposed the simulation of distributed algorithms for teaching purposes as an application for object-oriented reflection. We believe that finding applications for object-oriented reflection is an important research topic.

The use of reflection in object-oriented programming might not represent a new paradigm per se, but it requires a new way of designing. It offers more expressive power than simulation languages/environments and preserves the advantages of object-oriented programming. However, reflective object-oriented programming requires a design methodology that is different than the ones developed to date.

Meta levels as modules provide separation of concerns. This is very useful when programming complex simulations in which the code of each level corresponds to one or more issues to be simulated that should not be mixed with issues of other levels. Separation of concerns is not a new idea in software engineering. In particular, object-oriented programming emphasizes modularity. What is new is the use of reflective capabilities to support an open implementation approach.

Therefore, it is our thesis that reflective capabilities can be very useful for tailoring a simulation according to the user's needs. But some problems can arise due to the use of these capabilities (e.g. violation of encapsulation in a non-controlled way). Reflective object-oriented programming can be only successful if those problems are overcome.

This work presents several original contributions:

- We illustrate the use of the concurrent object-oriented reflective language ABCL/R2 as a simulation environment for distributed algorithms.
- We justify choosing ABCL/R2 as a result of a survey of relevant object-oriented languages and libraries used for simulation.
- We also propose a reflective object-oriented design methodology to build simulations of distributed algorithms as well as a generalization of this methodology. Suggestions to overcome the problems generated by the use of reflective capabilities are addressed in this design methodology.
- In our opinion, this work establishes the first steps to broader research for using object-oriented reflection for message-based discrete-event simulation.

ABCL/R2 was not designed with the intention of being a simulation language for any specific purpose. However, we found out that it turned out to be the best language that we could find to implement the simulations of distributed algorithms without programming in an ad hoc way. In addition, we were able to easily and straightforwardly map the entities of a distributed computing environment into objects of the language.

7.1. Future Work

Object-oriented languages provide flexibility to programmers of discrete-event simulations because entities of the simulation can be mapped to different objects, as opposed to being mapped to a limited number of constructs available in a simulation language. As suggested in this dissertation, the next step seems to be the use of reflective object-oriented languages in discrete-event simulation in order to provide adaptability to different users' needs.

Several other issues remain unexplored:

- A common conceptual model and terminology for reflective object-oriented languages.
- A standard notation for reflective object-oriented design.
- Automatic visualization techniques for reflective object-oriented design and programming.
- The application of object-oriented reflection to the design of embedded systems using the multiactivity paradigm (Krieger 1995).
- Reflective object-oriented testing techniques.

REFERENCES

- Achauer, B. "The Dowl distributed object-oriented language", *Communications of the ACM*, vol. 36, no. 9, September 1993, 48-55.
- Aburdene, M.F. and Whaley, R. 1988. "Simulation of a Distributed Algorithm Using the Connection Machine", *Modeling and Simulation ISA 19*, 1039-1047.
- Agha, G. 1986. "Actors-A Model of Concurrent Computation in Distributed Systems", The MIT Press, 144 pp.
- Agha, G., Frolund, S., Panwar, R. and Sturman, D. 1992. "A Linguistic Framework for Dynamic Composition of Dependability Protocols". *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications*, International Federation of Information Processing Societies, preprints, 345-363.
- Almes, G., Black, A., Lazowska, E. and Noe, J. 1985. "The Eden System: A Technical review", *IEEE Software Engineering*, vol SE-11, no. 1, January 1985, 43-58.
- Auban, J.B., Hutto, P. and Khalidi, M.Y. 1987. "The Architecture of the Ra Kernel", Technical Report GIT-ICS-87/35, Georgia Institute of Technology.

Babaoglu, O. 1990. "The Engineering of Fault-Tolerant Distributed Computing Systems". Fault-Tolerant Distributed Computing, Lecture Notes in Computer Science, vol. 448, Springer-Verlag, 262-273.

Baezner, D., Lomow, G. and Unger, B. W. 1990 "Sim++TM: The Transition to Distributed Simulation". Distributed Simulations: Proceedings of the SCS Multiconference on Distributed Simulation, 1990, 211-218.

Bagrodia, R.L., Chandy, K.M. and Misra, J. 1987. "A Message-Based Approach to Discrete-Event Simulation", IEEE Transactions on Software Engineering, vol. SE-13, no. 6, June 1987, 654-665.

Baldassari, M. and Bruno, G. 1991. "Protob: an object oriented methodology for developing discrete event dynamic systems". Computer Languages (Pergamon Press), vol.16, no.1, 39-63.

Bekker, C. and Putter, P. 1993. "Reflective Architectures: Requirements for Future Distributed Environments". Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems, 112-118.

Belina, F., Hogrefe D. and Sarma, A. 1991. "SDL with applications from protocol specification", Hemel Hempstead, England, Prentice-Hall, 275 pp.

Bennett, J.K. 1990. "Experience With Distributed Smalltalk", *Software-Practice and Experience*, vol.20(2), February 1990, 157-180.

Ben-Or, M. 1983. "Another advantage of free choice. completely asynchronous agreement protocols. Proceedings of the 2nd Annual ACM Symposium on the Principles of Distributed Computing, ACM, New York, 27-30.

Berbers, Y., Joosen, W., Moons, H. and Verbaeten, P. 1991 "The XENOCOPS Project". *Proceedings of the 1991 International Workshop on Object-Orientation in Operating Systems*, October 1991, 144-146.

Bezivin, J. 1987. "Some Experiments in Object-Oriented Simulation" *OOPSLA'87*, 394-405.

Bezivin, J. 1988a. "An evaluation environment for concurrent object-oriented simulation", *Distributed Simulation*, pages 140-154.

Bezivin, J. 1988b. "Design and Implementation Issues in Object-Oriented Simulation", *Simuletter*, vol. 19, no. 2, June 1988, 47-53.

Bijnens, S., Joosen, W. and Verbaeten, P. 1994. "A Reflective Invocation Scheme to Realise Advanced Object Management". *Object-Based Distributed Programming, ECOOP'93 workshop proceedings*, 139-151.

3

OF/DE

3

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Bischak, D. P. and Roberts, S. D. 1991. "Object-Oriented Simulation". Proceedings of the 1991 Winter Simulation Conference, 194-203.

Black, A., Hutchinson, N., Jul, E. and Levy, H. 1986. "Object Structure in the Emerald System". University of Washington, Seattle, TR 86-04-03, April. Also in ACM SIGPLAN Notices, vol. 21, no. 11, November 1986.

Blair, E.L. and Selvaraj, S. 1989. "DISC++: A C++ Based Library for object oriented simulation". Proceedings of the 1989 Winter Simulation Conference, 301-305.

Booch, G. 1991. "Object-Oriented Design with Applications". Redwood City, CA: Benjamin/Cummings.

Boehm, C.K. 1992. "Performance study of deadlock resolution in distributed systems". University of Illinois at Urbana-Champaign, IL, USA, report no. UIUCDCS-R-1779, 146 pp.

Budd, T. 1991. "An Introduction to object-oriented programming", Addison/Wesley, 399 pp.

Burns, 1988. "Programming in OCCAM2", Wokingham, England, Addison-Wesley.

Campbell, R.H. and Habermann, A.N. 1974. "The specification of process synchronization by path expressions". Lecture notes in Computer Science, vol.16, Springer Verlag.

Carr, H., Kessler, R. and Swanson, M. 1993. "Distributed C++", Sigplan Notices, vol.28, no.1, January 1993, 81.

Carvalho, O.S.F. and Roucairol, G. 1983. "On Mutual Exclusion in Computer Networks", Technical Correspondence of Communications of the ACM, vol. 26, no. 2, February 1983, 146-148.

Chandy, K.M., Misra, J. and Haas, L. M. 1983. "Distributed Deadlock Detection", ACM Transactions on Computer Systems, vol. 1, no. 2, May 1983, 144-156.

Chang, E. and Roberts R. 1979. "An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of processes", Communications of the ACM, vol. 22, no. 5, May 1979, 281-283.

Cota, B. A. and Sargent, R. G. 1992. "A modification of the Process Interaction world View", ACM Transactions on Modeling and Computer Simulation, vol. 2, no. 2, April 1992, 109-129.

Dahl, O.J. and Nygaard, K. 1966. "SIMULA - An ALGOL Based Simulation Language", Communications of the ACM, vol. 9, 671-678.

Davison, A. 1989. "Polka: A Parlog Object-Oriented Language, Ph.D. thesis, University of London.

Della Valle, C. 1995. "Implementing a model of time-constrained memory for reader-based text comprehension", Directed Studies, Carleton University, Ottawa, Canada.

Denney, V. P. 1991. "Object-Oriented Discrete-Event Simulation using Smalltalk with mixed-paradigm models". Proceedings of the 1991 Summer Computer Simulation Conference, 69-74.

Des Rivieres, J. and Smith, B.C. 1984. "The Implementation of Procedurally Reflective Languages". Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, August 1984, 348-355.

Diaz, A.L. and Corriveau, J.P. 1995a. "Using the Object Oriented Concurrent Reflective Language ABCL/R2 to Simulate Distributed Algorithms". SCS Winter Object-Oriented Simulation Conference, Las Vegas, Nevada, USA, January 1995, 113-119.

Diaz, A.L. and Corriveau, J.P. 1995b. "Looking for a Reflective Object-Oriented Architecture to study Fault Tolerance in Distributed Systems". SCS Summer Simulation Conference, Ottawa, Ontario, Canada, July 1995, 155-159.

Diaz, A.L. and Corriveau, J.P. 1995c. "Object-Oriented Reflective Design of Simulations of Distributed Algorithms - A Case Study". TOOLS USA, Santa Barbara, California, USA, August 1995, 229-239.

Diaz, M., Pimentel, E. and Troya, J.M. 1992. "A Declarative Environment for Distributed Programming". Information processing 92, Vol. I, Elsevier Science publishers B.V. (North-Holland), 163-169.

Dijkstra, E. W., Feijen, W. H. J. and Gasteren, A. J. M. 1983. "Derivation of a termination Detection Algorithm for Distributed Computations", Information Processing Letters, vol. 16, no. 5, June 1983, 217-219.

Dijkstra, E. W. and Scholten, C. S. 1980. "Termination Detection for Diffusing Computations", Information Processing Letters, vol. 11, no. 1, August 1980, 1-4.

Dolev, D., Dwork, C. and Stockmeyer, L. 1987. "On the Minimal Synchronism Needed for Distributed Consensus", Journal of the ACM, vol.34, no.1, January 1987, 77-97.

Dolev, D. and Strong, H. R. 1983. "Authenticated Algorithms for Byzantine Agreement", SIAM Journal of Computing, vol. 12, no. 4, November 1983, 656-666.

Erradi, M., Bochmann, G.V. and Hamid, I.A. 1992. "Dynamic Modifications of Object-Oriented Specifications", Compeuro 1992 Proceedings (Computer Systems and Software Engineering), 654-659.

Farkas, A. and Dearle, A. 1993. "Octopus: A Reflective Language Mechanism for Object Manipulation". Proceedings of the fourth International Workshop on Database Programming Languages, Object Models and Languages (DBPL-4), 50-64.

Finkel, D., Brown, D.J. and Kiff, T.J. 1988. "Simulation of a Fault Tolerant Distributed Computing System", Modeling and Simulation ISA 19, 997-1001.

Fischer, M. J., Lynch, N. A. and Paterson, M. S. 1985. "Impossibility of Distributed Consensus with One Faulty Process", Journal of the ACM, vol. 32, no. 2, April 1985, 347-382.

Flatebo, M. and Datta, A.K. 1992. "Simulation of self-stabilizing algorithms in distributed systems". Proceedings 25th Annual Simulation Symposium, 32-41.

Franklin, W. R. 1982. "On an improved Algorithm for Decentralized Extrema Finding in Circular Configurations of Processors", Communications of the ACM, vol. 25, no. 5, May 1982, 336-337.

Friedman, D.P. and Wand, M. 1984. "Reification: Reflection without Metaphysics". Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, 348-355.

Fron, A. 1992. "Multiprocessor Simulation Using Object-Oriented Programming". *Parallel Processing: CONPAR 92-VAPPV, Second Joint International Conference on Vector and Parallel Processing*, pages 819-820.

Giroux, S., Senteni, A. and Lapalme, G. 1993. "Adaptation in Open Systems Reflection as a Backbone". *Proceedings of International Conference on Intelligent and Cooperative Information Systems*, 114-123.

Goldberg, A. and Robson, D. 1983. "Smalltalk-80, the language and its implementation", Addison-Wesley, 714 pp.

Guffick, I.M. and Blair, G.S. 1989. "Kitara: A Parallel Object-Oriented Language for Distributed Applications", *Parallel Computing* 89, Elsevier Science Publishers B.V. (North-Holland).

Gunaseelan, L. and LeBlanc Jr., R.J. 1992. "Distributed Eiffel: A Language for Programming Multi-Granular Distributed Objects on the Clouds Operating System". *Proceedings of the 1992 International Conference on Computer Languages*, IEEE Computing Society Press, 1992, 331-340.

Hillis, W.D. 1985. "The connection machine". MIT Press, Cambridge, Mass.

Hirota, N., Umemoto, H., Aibara, R., Yamashita, M., Ae, T. 1993. "A distributed simulator for distributed algorithms", Transactions of the Information Processing Society of Japan, vol. 34, no. 11, 2421-30.

Hirschberg, D. S. and Sinclair J. B. 1980. "Decentralized Extrema-Finding in Circular Configurations of Processors", Communications of the ACM, vol. 23, no. 11, November 1980, 627-628.

Hoare, C.A.R. 1974. "Monitors: an operating system structuring concept", Communications of the ACM, vol. 17, no. 10, October 1974, 549-557.

Hopkins, T.P. and Wolczko, M.I. 1989. "Writing Concurrent Object-Oriented Programs using Smalltalk-80", The Computer Journal, vol. 32, no. 4, 341-350.

Horn, C. and Krakowiak, S. 1987. "Object Oriented Architecture for Distributed Office Systems". Proceedings of the 1987 Esprit Conference, North Holland, September 1987.

Huang, S-T 1990. "A Distributed Deadlock Detection Algorithm for CSP-like Communication". ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, January 1990, 102-122.

Ibrahim, M.H. 1991. "Report on the OOPSLA/ECOOP'90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming" in OOPSLA/ECOOP'90 Addendum to the proceedings, special issue of OOPS messenger (1991), 73-80.

Ibrahim, M.H. 1992. "Reflection in Object-Oriented Programming", International Journal on Artificial Intelligence Tools, vol. 1, no. 1, 117-136.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. 1992. "Object-Oriented Software Engineering, A Use Case Driven Approach", ACM Press/ Addison-Wesley, 524 pp.

Jard, C., Monin, J-F, Groz, R. 1988. "Development of Veda, a Prototyping Tool for Distributed Algorithms", IEEE Transactions on Software Engineering, vol. 14, no. 3, march 1988, 339-351.

Jones, M. and Rashid, R. 1986. "Mach and matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems", proceedings OOPSLA '86, 67-77.

Kahn, K., Tribble, E.D., Miller, M.S. and Bobrow, D.G. 1987. "Vulcan: Logical Concurrent Objects". Research Directions in Object-Oriented Programming, the MIT Press, 75-112.

Kessels, J.L.W. 1977. "An alternative to event queues for synchronization in monitors", Communications of the ACM, vol. 20, no. 7, July 1977, 500-503.

Kiviat, P.J., Villanueva, R. and Markowitz, H.M. 1973. "SIMSCRIPT II.5 Programming Language", CACI, Los Angeles.

Kremien, O., Kramer, J. and Magee, J. 1990. "CNCSIM - A Distributed Simulator for Decentralised Algorithms". Real-Time Systems, Theory and Applications, Elsevier Science Publishers B.V. (North-Holland), 271-287.

Krieger, M. 1994. "The Multiactivity Paradigm: An Approach for the Design of Embedded Systems by Application Specialists". Paper submitted to IEEE Transactions on Systems, Man and Cybernetics.

Kroger, B., Luling, R., Monien, B. and Vornberger, O. 1991. "An improved Algorithm to Detect Communication Deadlocks in Distributed Systems". Distributed algorithms. 4th International Workshop Proceedings, Springer Verlag, 90-101.

Lamport, L. 1978. "Time, Clocks, and the Ordering of Events in a Distributed System". Communications of the ACM, vol. 21, no. 7, July 1978, 558-565.

Lamport, L., Shostak, R. and Pease, M. 1982. "The Byzantine generals problem", ACM Transactions on Programming Languages and Systems, vol.4, no.3, July 1982, 382-401.

Liskov, B. and Scheifler, R. 1983. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM Transactions on Programming Languages and Systems, vol. 5, no. 3, July 1983, 381-404.

Lomow, G. and Baezner, D. 1989 "A tutorial introduction to object-oriented simulation and Sim++TM". Proceedings of the 1989 Winter Simulation Conference, 140-146.

Loui, M. C., Matsushita, T. A. and West, D. B. 1986. "Election in a Complete Network with a Sense of Direction", *Information Processing Letters*, vol. 22, no. 4, April 1986, 185-187.

Love Jr., A. E. and Aburdene, M. F. 1990. "Simulation of distributed algorithms using OCCAM", *Simulation*, vol. 55, no. 2, August 1990, 86-95.

McAffer, J. 1995. PhD Thesis. "A Meta-Level Architecture for prototyping Object Systems", University of Tokyo, Tokyo, Japan, September 1995, 128 pp.

Maes, P. 1987. "Computational Reflection". PhD Thesis. Laboratory for artificial Intelligence, Vrije Universiteit Brussel. Brussels, Belgium, 187 pp..

Mak, V. W. 1991. "DOSE: A Modular and Reusable Object-Oriented Simulation Environment". *Proceedings of the SCS Multiconference on Object-Oriented Simulation*, vol. 23, no. 3, 3-11.

Masuhara, H., Matsuoka, S., Watanabe, T. and Yonezawa, A. 1992. "Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently". *OOPSLA'92*, 127-144.

Matsuoka, S., Watanabe, T., Ichisugi, Y. and Yonezawa, A. 1991a. "Object-Oriented Concurrent Reflective Architectures". *Object-Based Concurrent Computing Workshop ECOOP'91, Lecture Notes in Computer Science*, no.612, Springer-Verlag, 211-226.

Matsuoka, S., Watanabe, T. and Yonezawa, A. 1991b. "Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming". Proceedings of ECOOP'91, Lecture Notes in Computer Science, no. 512, Springer-Verlag, 231-250.

Mili, A. 1990. "Program Fault Tolerance, a structured programming approach". Prentice Hall International (UK), 352 pp.

Misra, J. and Chandy, K. M. 1982. "Termination Detection of Diffusing Computations in Communicating Sequential processes", ACM Transactions on Programming Languages and Systems, vol. 4, no. 1, January 1982, 37-43.

Moretti, R. 1990. "SDL and object-oriented design: a way for producing quality software". CSELT Technical reports, vol. XVIII, no.2, April 1990, 131-134.

Mullender, S. 1985. "Principles of Distributed Operating System Design", Mathematisch Centrum, Vrije Universiteit, Amsterdam, October 1985.

Natarajan, N. 1984. "A Distributed Scheme for Detecting Communication Deadlocks", IEEE Transactions on Software Engineering, vol. SE-12, no. 4, April 1986.

Neilson, J.E. 1991. "PARASOL: a simulator for distributed and/or parallel systems". Carleton University technical report SCS-TR-192.

Nygaard, K. and Dahl, O.J. 1978. "The development of the Simula language", ACM conference on History of Programming Languages, Los Angeles, June 1978.

Ohki, M., Takeuchi, A. and Furukawa, K. 1987. "An Object-Oriented Programming Language Based on the Parallel Logic Programming Language KL1", proceedings of the 4th International Conference on Logic Programming, the MIT Press, Cambridge, Massachusetts, 894-909.

Ozden, M. H. 1990. "Automatic Programming of Simulation Models in an Object-Oriented Environment". Modelling and Simulation, Proceedings of the 1990 European Simulation Multiconference, 64-69.

Pease, M., Shostak, R. and Lamport, L. 1980. "Reaching Agreement in the Presence of Faults", Journal of the ACM, vol.27, no.2, April 1980, 228-234.

Pegden, C.D. 1986. "Introduction to SIMAN, Systems Modeling Corp., State College, Pennsylvania.

Peterson, G. L. 1982. "An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem", ACM Transactions on Programming Languages and Systems, vol. 4, no. 4, October 1982, 758-762.

Pimentel, E. and Troya, J.M. 1991. "A Concurrent Object Oriented Logic Language", Atti del Sesto Convegno sulla Prog.Log. GULP'91.

- Rana, S. P. 1983. "A Distributed Solution of the Distributed Termination Problem", *Information Processing Letters*, vol. 17, no. 1, July 1983, 43-46.
- Rao, R. 1991. "Implementational Reflection in Silica". *Proceedings of ECOOP'91, Lecture Notes in Computer Science*, no. 512, Springer-Verlag, 251-266.
- Raynal, M. 1988. "Distributed Algorithms and Protocols". John Wiley & Sons Ltd., 163 pp.
- Ricart, G. and Agrawala, A.K. 1981. "An Optimize' Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, vol. 24, no. 1, January 1981, 9-17.
- Robert, P. and Verjus, J.P. 1976. "Towards autonomous descriptions of synchronization modules". *Research Report no. 47, ENSIMAG, Grenoble, October*, 24 pp.
- Roberts, S. D. and Heim, J. 1988. "A perspective on object-oriented simulation". *Proceedings of the 1988 Winter Simulation Conference*, 277-281.
- Rogers, R. V. 1991. "Understated implications of object-oriented simulation and modeling". *Proceedings of the 1991 IEEE International Conference on Systems, Man, and Cybernetics*, 285-289.
- Rose, J.R. 1987. "C*: A C++-like language for data parallel computation", *Thinking Machines Technical Report PL-87-8, Thinking Machines Corporation, Cambridge, Mass.*

Rose, J.R. and Steele Jr, G.L. 1987. "C*: An extended C language for data parallel programming". Proceedings Second International Conference on Supercomputing, vol. 2, 2-16.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. 1991 "Object-Oriented Modeling and Design", Prentice hall, 500 pp.

Schneider, F.B. 1990. "The State Machine Approach: A Tutorial". Fault-Tolerant Distributed Computing, Lecture Notes in Computer Science, vol. 448, Springer-Verlag, 18-41.

Schriber, J.J. 1974. "Simulation Using GPSS", JohnWiley & Sons, New York.

Siewiorek, D.P. and Swarz, R.S. 1992. "Reliable Computer Systems Design and Evaluation", Digital Press, second edition.

Smith, B.C. 1985. "Prologue to Reflection and Semantic in a Procedural Language" in Readings in Knowledge Representation, Morgan Kaufmann Publishers, Los Altos, CA.

Song, G., Park, D., Lee, D., Park, K.H. and Kim, M. 1989. "A Distributed Deadlock Detection Algorithm: Distributed Graph Reconstruction Algorithm", Information Processing Letters, vol. 30, March 1989, 245-252.

Sonntag, S., Hartig, H. Kowalski, O., Kuhnhauser, W. and Lux, W. 1994. "Proceedings of the Twenty-Seventh Hawaii International Conference on Systems Sciences, 383-392.

Spartiotis, D., Spirakis, P. and Tampakos, B. 1991. "DICOMP, a Compiler for SDL into Distributed System Simulator Specs". Internal Report (ESPRIT 2080) REX-WP4, April 1991.

Spirakis, P., Tampakas, B., Papatriantafillou, M., Konstantoulis, K., Vlachodimitropoulos, K., Antonopoulos, V., Kazazis, P., Metallidou, T. and Spartiotis, D. 1992. "Distributed System Simulator (DSS)". Proceedings of STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, 615-616.

Steels, L. 1986. "The KRS Concept System", Vrije Universiteit Brussel. Artificial Intelligence Laboratory. Technical Report 86-1, Brussels, Belgium.

Suzuki, I. and Kasami, T. 1985. "A Distributed Mutual Exclusion Algorithm", ACM Transactions on Computer Systems, vol. 3, no. 4, November 1985, 344-349.

Tel, G. 1991. "Topics in Distributed Algorithms", Cambridge International Series on Parallel Computation 1, Cambridge University Press, 207pp.

Tel, G. 1994. "Introduction to Distributed Algorithms", vol. 1. Cambridge University Press, 1994, 534 pp.

Topor, R. W. 1984. "Termination Detection for Distributed Computations", *Information Processing Letters*, vol. 18, no. 1, January 1984, 33-36.

Turek, J. and Shasha, D. 1992. "The many faces of Consensus in Distributed Systems", *Computer*, June 1992, 8-17.

Ungar, D. and Smith, R.B. 1991. "SELF: The Power of Simplicity". *Lisp and Symbolic Computation: An International Journal*, 4,3.

Watanabe, T. 1991. "An Actor-Based Metalevel Architecture for Group-Wide Reflection". *Foundations of Object-Oriented Languages REX School/Workshop Proceedings, Lecture Notes in Computer Science* no. 489, Springer-Verlag, 405-425.

Watanabe, T. and Yonezawa, A. 1988. "Reflection in an Object-Oriented Concurrent Language". *OOPSLA'88*, 306-315.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L. 1990. "Designing Object-Oriented Software". Prentice Hall, 341 pp.

Yokote, Y. and Tokoro, M. 1987. "Concurrent Programming in ConcurrentSmalltalk". *Object-Oriented Concurrent Programming, MIT Press Series in Computer Systems*, 129-158.

Yokote, Y., Teraoka, F. and Tokoro, M. 1989. "A Reflective Architecture for an Object-Oriented Distributed Operating System". Proceedings of ECOOP'89, Lecture Notes in Computer Science, no. 707, 89-106.

Yokote, Y. 1992. "The Apertos Reflective Operating System: The Concept and its Implementation". Proceedings of OOPSLA '92, 414-434.

Yonezawa, A. (editor) 1990. "ABCL: An Object-Oriented Concurrent System", The MIT Press.

Yonezawa, A., Briot, J.-P. and Shibayama, E. 1986. "Object-Oriented Concurrent Programming in ABCL/1". OOPSLA '86, 258-268.

Yonezawa, A. and Watanabe, T. 1988. "An Introduction to Object-Based Reflective Concurrent Computation". OOPSLA '88 workshop on Object-Based Concurrent Programming Proceedings, 50-54.

Ziegler, B.P. 1976. "Theory of Simulation and Modeling". New York, Ny:Wiley.

Appendix A - Additional Simulation Results

This appendix presents the following simulation results:

1) The algorithm for fault tolerant distributed mutual exclusion of Chang, Singhal and Liu considering ten nodes and link failures on links 6-1, 7-4 and 9-8 at the time that the token must be transferred along these links. This particular example illustrates that an infinite loop of messages is generated because the algorithm works only for the case of one link failure.

2) The results of simulation experiments of the Byzantine agreement algorithm of Lamport, Shostak and Pease.

(a) Request messages that are issued while node 7 is executing the critical section

(b) Node 6 executes the critical section

(c) Node 2 executes the critical section

(g) Node 3 executes the critical section

(h) Node 7 sends the token to node 6, because link 7-4 is

(i) At this point an infinite loop of exchanged messages begin

(j) This configuration is identical to the one illustrated in (h)

(k) This configuration is identical to the one illustrated in (i)

Figure A1. Messages exchanged during the execution of the algorithm of Chang, Singhal and Liu considering ten nodes and faulty links 6-1, 7-4 and 9-8 at the time that the token must be sent along them. An infinite loop of messages is generated, illustrated in (h), (i), (j), (k)

Figure A2. Base-level interaction diagram for the execution of the algorithm of Chang, Singhal and Liu as illustrated in figure A1.

BYZANTINE AGREEMENT

THE GOAL IS TO ATTACK

CASE 1: 3 nodes, initiator is faulty
initiator sends a faulty message to node 2
each node receives 2 messages

ABCL/R2>[network <= [:start 3 [] 0]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 1 numRETREAT= 1

node 2 numATTACK= 1 numRETREAT= 1

CASE 2: 3 nodes, node 1 is faulty
each node receives 2 messages

ABCL/R2>[network <= [:start 3 [1] 1]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 2 numRETREAT= 0

node 2 numATTACK= 1 numRETREAT= 1

CASE 3: 3 nodes, node 2 is faulty
each node receives 2 messages

ABCL/R2>[network <= [:start 3 [2] 1]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 1 numRETREAT= 1

node 2 numATTACK= 2 numRETREAT= 0

CASE 4: 4 nodes, initiator is faulty
 initiator sends a faulty message to node 2
 each node receives 5 messages

ABCL/R2>[network <= [:start 4 [| 0]]
 NIL
 ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 3 numRETREAT= 2
 node 2 numATTACK= 4 numRETREAT= 1
 node 3 numATTACK= 3 numRETREAT= 2

CASE 5: 4 nodes, node 1 is faulty
 each node receives 5 messages

ABCL/R2>[network <= [:start 4 [1] 1]]
 NIL
 ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 5 numRETREAT= 0
 node 2 numATTACK= 3 numRETREAT= 2
 node 3 numATTACK= 3 numRETREAT= 2

CASE 6: 4 nodes, node 2 is faulty
 each node receives 5 messages

ABCL/R2>[network <= [:start 4 [2] 1]]
 NIL
 ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 3 numRETREAT= 2
 node 2 numATTACK= 5 numRETREAT= 0
 node 3 numATTACK= 3 numRETREAT= 2

CASE 7: 4 nodes, node 3 is faulty
each node receives 5 messages

ABCL/R2>[network <= [:start 4 [3] 1]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 3 numRETREAT= 2

node 2 numATTACK= 3 numRETREAT= 2

node 3 numATTACK= 5 numRETREAT= 0

CASE 8: 4 nodes, nodes 1 and 3 are faulty
each node receives 5 messages

ABCL/R2>[network <= [:start 4 [1 3] 2]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 2 numRETREAT= 3

node 2 numATTACK= 1 numRETREAT= 4

node 3 numATTACK= 2 numRETREAT= 3

CASE 9: 4 nodes, nodes 1 and 2 are faulty
each node receives 5 messages

ABCL/R2>[network <= [:start 4 [1 2] 2]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 2 numRETREAT= 3

node 2 numATTACK= 2 numRETREAT= 3

node 3 numATTACK= 1 numRETREAT= 4

CASE 10: 4 nodes, nodes 2 and 3 are faulty
each node receives 5 messages

ABCL/R2>[network <= [:start 4 [2 3] 2]]
NIL
ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 1 numRETREAT= 4
node 2 numATTACK= 2 numRETREAT= 3
node 3 numATTACK= 2 numRETREAT= 3

CASE 11: 4 nodes, initiator and node 3 are faulty
initiator sends a faulty message to node 2
each node receives 5 messages

ABCL/R2>[network <= [:start 4 [3] 1]]
NIL
ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 2 numRETREAT= 3
node 2 numATTACK= 3 numRETREAT= 2
node 3 numATTACK= 3 numRETREAT= 2

CASE 12: 5 nodes, initiator is faulty
initiator sends a faulty message to node 2
each node receives 16 messages

ABCL/R2>[network <= [:start 5 [] 0]]
NIL
ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 11 numRETREAT= 5
node 2 numATTACK= 15 numRETREAT= 1
node 3 numATTACK= 11 numRETREAT= 5
node 4 numATTACK= 11 numRETREAT= 5

CASE 13: 5 nodes, node 1 is faulty
each node receives 16 messages

ABCL/R2>[network <= [:start 5 [1] 1]]
NIL
ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 16 numRETREAT= 0
node 2 numATTACK= 11 numRETREAT= 5
node 3 numATTACK= 11 numRETREAT= 5
node 4 numATTACK= 11 numRETREAT= 5

CASE 14: 5 nodes, node 2 is faulty
each node receives 16 messages

ABCL/R2>[network <= [:start 5 [2] 1]]
NIL
ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 11 numRETREAT= 5
node 2 numATTACK= 16 numRETREAT= 0
node 3 numATTACK= 11 numRETREAT= 5
node 4 numATTACK= 11 numRETREAT= 5

CASE 15: 5 nodes, node 3 is faulty
each node receives 16 messages

ABCL/R2>[network <= [:start 5 [3] 1]]
NIL
ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 11 numRETREAT= 5
node 2 numATTACK= 11 numRETREAT= 5
node 3 numATTACK= 16 numRETREAT= 0
node 4 numATTACK= 11 numRETREAT= 5

CASE 16: 5 nodes, node 4 is faulty
each node receives 16 messages

ABCL/R2>[network <= [:start 5 [4] 1]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 11 numRETREAT= 5

node 2 numATTACK= 11 numRETREAT= 5

node 3 numATTACK= 11 numRETREAT= 5

node 4 numATTACK= 16 numRETREAT= 0

CASE 17: 5 nodes, nodes 1 and 3 are faulty
each node receives 16 messages

ABCL/R2>[network <= [:start 5 [1 3] 2]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 10 numRETREAT= 6

node 2 numATTACK= 6 numRETREAT= 10

node 3 numATTACK= 10 numRETREAT= 6

node 4 numATTACK= 4 numRETREAT= 12

CASE 18: 5 nodes, nodes 2 and 4 are faulty
each node receives 16 messages

ABCL/R2>[network <= [:start 5 [2 4] 2]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 6 numRETREAT= 10

node 2 numATTACK= 10 numRETREAT= 6

node 3 numATTACK= 4 numRETREAT= 12

node 4 numATTACK= 10 numRETREAT= 6

CASE 19: 5 nodes, initiator and node 3 are faulty
 initiator sends a faulty message to node 2
 each node receives 16 messages

ABCL/R2>[network <= [:start 5 [3] 1]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 8 numRETREAT= 8
 node 2 numATTACK= 11 numRETREAT= 5
 node 3 numATTACK= 11 numRETREAT= 5
 node 4 numATTACK= 8 numRETREAT= 8

CASE 20a: 6 nodes, initiator is faulty
 initiator sends a faulty message to node 2
 each node receives 65 messages

ABCL/R2>[network <= [:start 6 [] 0]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 49 numRETREAT= 16
 node 2 numATTACK= 64 numRETREAT= 1
 node 3 numATTACK= 49 numRETREAT= 16
 node 4 numATTACK= 49 numRETREAT= 16
 node 5 numATTACK= 49 numRETREAT= 16

CASE 20b: 6 nodes, initiator is faulty
 initiator sends a faulty message to node 3 and to node 4
 each node receives 65 messages

ABCL/R2>[network <= [:start 6 [] 0]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 33 numRETREAT= 32
 node 2 numATTACK= 33 numRETREAT= 32
 node 3 numATTACK= 48 numRETREAT= 17
 node 4 numATTACK= 48 numRETREAT= 17
 node 5 numATTACK= 33 numRETREAT= 32

CASE 21: 6 nodes, node 1 is faulty
each node receives 65 messages

ABCL/R2>[network <= [:start 6 [1] 1]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 65 numRETREAT= 0
node 2 numATTACK= 49 numRETREAT= 16
node 3 numATTACK= 49 numRETREAT= 16
node 4 numATTACK= 49 numRETREAT= 16
node 5 numATTACK= 49 numRETREAT= 16

CASE 22: 6 nodes, nodes 1 and 3 are faulty
each node receives 65 messages

ABCL/R2>[network <= [:start 6 [1 3] 2]]

NIL

ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 48 numRETREAT= 17
node 2 numATTACK= 33 numRETREAT= 32
node 3 numATTACK= 48 numRETREAT= 17
node 4 numATTACK= 29 numRETREAT= 36
node 5 numATTACK= 29 numRETREAT= 36

CASE 23a: 6 nodes, initiator and node2 are faulty
initiator sends a faulty message to node 2
each node receives 65 messages

ABCL/R2>[network <= [:start 6 [2] 1]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 49 numRETREAT= 16
node 2 numATTACK= 64 numRETREAT= 1
node 3 numATTACK= 49 numRETREAT= 16
node 4 numATTACK= 49 numRETREAT= 16
node 5 numATTACK= 49 numRETREAT= 16

CASE 23b: 6 nodes, initiator and node2 are faulty
 initiator sends a faulty message to node 3 and to node 4
 each node receives 65 messages

ABCL/R2>[network <= [:start 6 [2] 1]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 27 numRETREAT= 38
 node 2 numATTACK= 33 numRETREAT= 32
 node 3 numATTACK= 38 numRETREAT= 27
 node 4 numATTACK= 38 numRETREAT= 27
 node 5 numATTACK= 27 numRETREAT= 38

CASE 24a: 7 nodes, initiator is faulty
 initiator sends a faulty message to node 2
 each node receives 326 messages

ABCL/R2>[network <= [:start 7 [] 0]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 261 numRETREAT= 65
 node 2 numATTACK= 325 numRETREAT= 1
 node 3 numATTACK= 261 numRETREAT= 65
 node 4 numATTACK= 261 numRETREAT= 65
 node 5 numATTACK= 261 numRETREAT= 65
 node 6 numATTACK= 261 numRETREAT= 65

CASE 24b: 7 nodes, initiator is faulty
 initiator sends a faulty message to node 3 and to node 4
 each node receives 326 messages

ABCL/R2>[network <= [:start 7 [] 0]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 196 numRETREAT= 130
 node 2 numATTACK= 196 numRETREAT= 130
 node 3 numATTACK= 260 numRETREAT= 66
 node 4 numATTACK= 260 numRETREAT= 66
 node 5 numATTACK= 196 numRETREAT= 130
 node 6 numATTACK= 196 numRETREAT= 130

CASE 25: 7 nodes, node 1 is faulty
each node receives 326 messages

ABCL/R2>[network <= [:start 7 [1] 1]]
NIL
ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 326 numRETREAT= 0
node 2 numATTACK= 261 numRETREAT= 65
node 3 numATTACK= 261 numRETREAT= 65
node 4 numATTACK= 261 numRETREAT= 65
node 5 numATTACK= 261 numRETREAT= 65
node 6 numATTACK= 261 numRETREAT= 65

CASE 26: 7 nodes, nodes 1 and 3 are faulty
each node receives 326 messages

ABCL/R2>[network <= [:start 7 [1 3] 2]]
NIL
ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 260 numRETREAT= 66
node 2 numATTACK= 196 numRETREAT= 130
node 3 numATTACK= 260 numRETREAT= 66
node 4 numATTACK= 186 numRETREAT= 140
node 5 numATTACK= 186 numRETREAT= 140
node 6 numATTACK= 186 numRETREAT= 140

CASE 27: 7 nodes, nodes 1, 3 and 5 are faulty
each node receives 326 messages

ABCL/R2>[network <= [:start 7 [1 3 5] 3]]
NIL
ABCL/R2>[non_faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 195 numRETREAT= 131
node 2 numATTACK= 131 numRETREAT= 195
node 3 numATTACK= 190 numRETREAT= 136
node 4 numATTACK= 116 numRETREAT= 210
node 5 numATTACK= 185 numRETREAT= 141
node 6 numATTACK= 116 numRETREAT= 210

CASE 28a: 7 nodes, initiator and node 4 are faulty
 initiator sends a faulty message to node 2
 each node receives 326 messages

ABCL/R2>[network <= [:start 7 [4] 1]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 212 numRETREAT= 114

node 2 numATTACK= 261 numRETREAT= 65

node 3 numATTACK= 212 numRETREAT= 114

node 4 numATTACK= 261 numRETREAT= 65

node 5 numATTACK= 212 numRETREAT= 114

node 6 numATTACK= 212 numRETREAT= 114

CASE 28b: 7 nodes, initiator and node 4 are faulty
 initiator sends a faulty message to node 3 and to node 4
 each node receives 326 messages

ABCL/R2>[network <= [:start 7 [4] 1]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 212 numRETREAT= 114

node 2 numATTACK= 212 numRETREAT= 114

node 3 numATTACK= 276 numRETREAT= 50

node 4 numATTACK= 260 numRETREAT= 66

node 5 numATTACK= 212 numRETREAT= 114

node 6 numATTACK= 197 numRETREAT= 129

CASE 29a: 7 nodes, initiator and nodes 2 and 6 are faulty
 initiator sends a faulty message to node 2
 each node receives 326 messages

ABCL/R2>[network <= [:start 7 [2 6] 2]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 212 numRETREAT= 114
 node 2 numATTACK= 260 numRETREAT= 66
 node 3 numATTACK= 207 numRETREAT= 119
 node 4 numATTACK= 207 numRETREAT= 119
 node 5 numATTACK= 207 numRETREAT= 119
 node 6 numATTACK= 261 numRETREAT= 65

CASE 29b: 7 nodes, initiator and nodes 2 and 6 are faulty
 initiator sends a faulty message to node 3 and to node 4
 each node receives 326 messages

ABCL/R2>[network <= [:start 7 [2 6] 2]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 98 numRETREAT= 228
 node 2 numATTACK= 162 numRETREAT= 164
 node 3 numATTACK= 137 numRETREAT= 189
 node 4 numATTACK= 137 numRETREAT= 189
 node 5 numATTACK= 88 numRETREAT= 238
 node 6 numATTACK= 130 numRETREAT= 196

CASE 29c: 7 nodes, initiator and nodes 2 and 6 are faulty
initiator sends a faulty message to node 2 and to node 3
each node receives 326 messages

ABCL/R2>[network <= [:start 7 [2 6] 2]]

NIL

ABCL/R2>[faulty_node0 <= [:wake :attack]]

node 1 numATTACK= 163 numRETREAT= 163
node 2 numATTACK= 211 numRETREAT= 115
node 3 numATTACK= 207 numRETREAT= 119
node 4 numATTACK= 158 numRETREAT= 168
node 5 numATTACK= 143 numRETREAT= 183
node 6 numATTACK= 211 numRETREAT= 115

Appendix B - Code of Some Simulation Experiments

- 1) The algorithm for communication deadlock detection of Chandy, Misra and Haas considering four nodes implemented in ABCL/R2.
- 2) The algorithm for communication deadlock detection of Chandy, Misra and Haas considering four nodes implemented in SmalltalkAgents.

;; THE ALGORITHM FOR COMMUNICATION DEADLOCK DETECTION OF
CHANDY, MISRA
;; AND HAAS (1983) CONSIDERING FOUR NODES

```
(defmacro inherit (&rest args)
  `(reify-args-to-meta :inherit ,@args))
```

;;; THE METAMETA OBJECT GENERATOR

;;; The metameta object generator for execution tracing

```
[object the-meta-meta-gen
 (group (group-of (group-of group)))
 (script
  (=> [:new StateSpec LexEnv Scripts Eval GMgr . Extra] @ R
   [object metaobject
    (state [queue := [queue-gen <== :new]]
      [state := nil]
      [scriptSet := scripts]
      [evaluator := Eval]
      [Group-manager := GMgr]
      [mode := ':dormant]
      [countmsg := 0]
      id)
```

```
(script
  ;; No script for inheritance is necessary at this level
```

```
(=> [:message Message Reply Sender]
```

```
(temporary idSender)
```

```
[queue <= [:enq [Message Reply Sender]]]
```

```
;;
*****
```

```
(match Message
 (is [:message M R S]
```

```
(if (and (not (eq S [den [den Me]])) (not (eq (car M) :wake)))
 [
```

;; Monitoring of the number of messages received from other nodes

```
[countmsg := (+ 1 countmsg)]
(format t "~&node ~S received ~S messages so far" id countmsg)
```

;; Monitoring of the messages received and the sender (execution tracing)

```

[idSender := (second M)]
(format t "~&NODE ~S RECEIVED ~S FROM NODE ~S" id M idSender)
)
))

```

```

;;
*****

```

```

(when (eq mode ':dormant)
  [mode := ':active]
  [Me <= :begin]))

(=> [:myid anId]
  [id := anId])

(=> :begin
  (match [queue <== :deq]
    (is [Message Reply Sender]
      (match (find-script Message Reply Sender scriptSet)
        (is [Bindings ScriptBody]
          [evaluator
            <= [:do
              ScriptBody [env-gen <== [:new Bindings state]]
              Me Group-manager evaluator]
            @ [cont _
              [Me <= :end]]]))
      (otherwise
        (warn "~S cannot handle the message ~S"
          [den Me] Message))))))

(=> :end
  (if (not [queue <== :empty?])
    [Me <= :begin]
    [mode := ':dormant]))

;; reflective operations
(=> :state !state)
(=> [:set-state s]
  [state := s])
(=> :queue @ R
  [queue <= :listify @ R])
(=> :dequeue @ R
  [queue <= :deq @ R])

```

```

(=> [:enqueue item]
  [queue <= [:enq item]])
(=> :scripts !scriptSet)
(=> [:find-script Message]
  !(search-script Message nil nil scriptSet))
(=> [:find-script Message Reply Sender]
  !(find-script Message Reply Sender scriptSet))
(=> [:add-script new-script]
  [scriptSet := (cons new-script ScriptSet)])
(=> [:remove-script Message]
  [scriptSet := (remove-script Message nil nil ScriptSet)])
(=> :evaluator !evaluator)
(=> [:set-evaluator e] [evaluator := e])
)]
;; initialize state variables
[Eval <= [:do-evlis (stateSpec-initforms StateSpec)
  LexEnv Metaobject GMgr Eval]
  @ [cont initValues
    [Metaobject
      <= [:set-state
        [env-gen
          <== [:new (pairlis (stateSpec-varNames stateSpec)
            initValues)
            LexEnv]]]]
    [R <= Metaobject]]]])]

```

;; THE META OBJECT GENERATOR

```

;; The meta object generator for the inheritance of scripts and simulation of
;; the application program that requires communication deadlock detection.
;; Generated meta objects have two additional scripts:
;; [:inherit ...] for the inheritance
;; :execute to simulate the resumption of the application program
;; Other definitions are the same as the ones of the original meta object.

```

```

[object the-meta-gen
  (group (group-of group))
  (script
    (=> [:new StateSpec LexEnv Scripts Eval GMgr . Extra] @ R
      [object metaobject
        (meta-gen the-meta-meta-gen)
        (state [queue := [queue-gen <== :new]]
          [state := nil]
          [scriptSet := scripts]
          [evaluator := Eval])

```

```

[Group-manager := GMgr]
[mode := ':dormant])

(script
;; The body of pseudo reflective procedure is as follows.
(=> [:inherit Args Env Id Gid Eval] @ C
[Eval <= [:do-evlis Args Env Id Gid Eval]
@ [cont {message reply sender inherit-source}
;; A inquiry of a script is sent to the metaobject of the
;; inheritance source.
(match [[meta inherit-source]
<== [:find-script message reply sender]]
;; If a script is found, start execution of the script.
(is [bindings scriptbody]
[Eval
<= [:do scriptbody [env-gen <== [:new bindings Env]]
Id Gid Eval]
@ C])
(otherwise
(warn "Inheritance from ~S is failed." inherit-source)
[C <= nil]]))])

(=> [:message Message Reply Sender]

;; time is incremented to mark the reception of messages from other nodes

(if (and (not (eq [den Me] Sender)) (not (eq (car Message) :wake)))

[(let ((val-time [state <== [:value-of 'time]]))
[state <= [:set 'time (1+ val-time)])]

;; At local time = 2, node 4 sends a message to node3 (node3 was waiting
;; for this message to resume execution)
;; Node 4 becomes idle and waits for a message from node2

(if (and (eq [den Me] node4) (= (1+ val-time) 2))

[[[meta node3] <= :execute]
[state <= [:set 'condition 'idle]]
[state <= [:set 'time (+ 3 val-time)]]]

;; time was incremented to mark the sending of a message to another node
;; time was also incremented to mark a change of state condition of node4

(format t "~&NODE 4 BECOMES IDLE AND WAITS FOR A MESSAGE FROM NODE
2 AT TIME ~S"(+ 3 val-time))

```

```

        (let ((val-DS [state <== [:value-of 'DS]]))
          [val-DS := (append [node2] val-DS)]
          [state <= [:set 'DS val-DS]])
        [[den Me] <= [:detect 'META-NODE4]]
      )))
    ])

    [queue <= [:enq [Message Reply Sender]]]
    (when (eq mode ':dormant)
      [mode := ':active]
      [Me <= :begin]))

;; *

(=> :execute from Sender
    (temporary)

;; Time is incremented to mark the reception of message execute and the state is
;; changed to executing, i.e. node3 resumes execution
;; Node4 (the sender of the message) is removed from node3's dependent set

    [state <= [:set 'condition 'executing]]
    (let ((val-time [state <== [:value-of 'time]]))
      (val-DS [state <== [:value-of 'DS]]))
      [state <= [:set 'time (1+ val-time)]]
      [state <= [:set 'DS (Remove [den Sender] val-DS)]]
      (format t "~&NODE3 RESUMES EXECUTION AT TIME ~S" (1+ val-time))
    )
  )

;; *

(=> :begin
  (match [queue <== :deq]
    (is [Message Reply Sender]
      (match (find-script Message Reply Sender scriptSet)
        (is [Bindings ScriptBody]
          [evaluator
            <= [:do
              ScriptBody [env-gen <== [:new Bindings state]]
              Me Group-manager evaluator]
            @ [cont _
              [Me <= :end]]]))
        (otherwise
          (warn "~S cannot handle the message ~S"
            [den Me] Message))))))

```

```

(=> :end
  (if (not [queue <== :empty?])
    [Me <= :begin]
    [mode := :dormant]))

;; reflective operations
(=> :state !state)
(=> [:set-state s]
  [state := s])
(=> :queue @ R
  [queue <= :listify @ R])
(=> :dequeue @ R
  [queue <= :deq @ R])
(=> [:enqueue item]
  [queue <= [:enq item]])
(=> :scripts !scriptSet)
(=> [:find-script Message]
  !(search-script Message nil nil scriptSet))
(=> [:find-script Message Reply Sender]
  !(find-script Message Reply Sender scriptSet))
(=> [:add-script new-script]
  [scriptSet := (cons new-script ScriptSet)])
(=> [:remove-script Message]
  [scriptSet := (remove-script Message nil nil ScriptSet)])
(=> :evaluator !evaluator)
(=> [:set-evaluator e] [evaluator := e])
)]
;; initialize state variables
[Eval <= [:do-evlis (stateSpec-initforms StateSpec)
  LexEnv Metaobject GMgr Eval]
  @ [cont initValues
    [Metaobject
      <= [:set-state
        [env-gen
          <== [:new (pairlis (stateSpec-varNames stateSpec)
            initValues)
            LexEnv]]]]]
    [R <= Metaobject]]]])]

```

;;; A group definition for using the inheritance mechanism.

```

[group a-group
  (meta-gen the-meta-gen)]

```

;;; The application program


```

[object node1
  (state [condition := 'idle]
    [time := 0]
    [DS := nil]
    (SetQ LATEST (Make-Array '(4)))
    (SetQ NUM (Make-Array '(4)))
    (SetQ ENGAGER (Make-Array '(4)))
    (SetQ WAIT (Make-Array '(4)))

    [nodeId := 1]
  )

```

```

(script
  (=> [:wake]

```

```

;; When node1 wakes, it wakes all the other nodes
;; Wake serves to initialize some state variables that have to be initialized

```

```

(SetF (Aref LATEST 0) 0) (SetF (Aref LATEST 1) 0) (SetF (Aref LATEST 2) 0)
(SetF (Aref LATEST 3) 0)
(SetF (Aref NUM 0) 0) (SetF (Aref NUM 1) 0) (SetF (Aref NUM 2) 0)
(SetF (Aref NUM 3) 0)
(SetF (Aref ENGAGER 0) 0) (SetF (Aref ENGAGER 1) 0) (SetF (Aref ENGAGER 2) 0)
(SetF (Aref ENGAGER 3) 0)

```

```

;; message wake also makes this node inform its metameta object the value of its
;; id number to be used in the execution tracing of the algorithm

```

```

[[meta [meta Me]] <= [:myid nodeId]]

```

```

;; WAIT does not need to be initialized because all its elements contain nil
;; which corresponds to false

```

```

(if (= nodeId 1) [[DS := [node2 node3]]
  [node2 <= [:wake]]
  [node3 <= [:wake]]
  [node4 <= [:wake]]

```

```

;; After node1 wakes all the other nodes, it starts detecting if it is
;; deadlocked

```

```

  [Me <= [:detect nodeId]]
  ])
(if (= nodeId 2) [DS := [node4]])
(if (= nodeId 3) [DS := [node1 node4]])
(if (= nodeId 4) [DS := []])
!nil

```

```

)

(=> [:detect idSender]

    (temporary l size [ii := (- nodeId 1)])

;; If the dependent set of a node is empty, the node is not deadlocked

    (if (eq condition 'idle)
        [
            (if (eq DS NIL)
                [(format t "~&NODE ~S IS NOT DEADLOCKED" nodeId)])

;; If a node is idle and is supposed to detect if it is deadlocked, it first
;; sets the number of the detection (to be distinguished from other detections
;; of the same node) and signals that it is idle

                [[l := (Aref LATEST ii)]
                 [l := (+ l 1)]]
                (SetF (Aref LATEST ii) l)
                (SetF (Aref WAIT ii) T)
                [size := (Length DS)])

;; It then starts the detection by sending query messages to all nodes of
;; its dependent set

                (DoList (eachnode DS)
                    [eachnode <= [:query nodeId nodeId l]]
                    [time := (+ time 1)])
                    (setF (Aref NUM ii) size)))
        ])

)

(=> [:query idSender i m] from j

    (temporary l w size ii)

    (if (eq condition 'idle)
        [
            [ii := (- i 1)]
            [l := (Aref LATEST ii)]
            [w := (Aref WAIT ii)]
            (if (> m l) [(SetF (Aref LATEST ii) m)
                        (SetF (Aref ENGAGER ii) j)
                        (SetF (Aref WAIT ii) T)]

;; If the dependent set of the node is empty, then it is not deadlocked

```

```

                (if (eq DS NIL)
                    [(format t "~&NODE ~S IS NOT DEADLOCKED,
THUS NODE ~S IS NOT DEADLOCKED" nodeId i)
                    (SetF (Aref NUM ii) 0)])

```

```

;; Receiving a query message that corresponds to a new detection,
;; a node must send a query message to all nodes of its dependent set
;; (propagate the query)

```

```

                [(DoList (eachnode DS)
                    [eachnode <= [:query nodeId i m]]
                    [time := (+ time 1)])]
                [size := (Length DS)]
                (SetF (Aref NUM ii) size)))]

```

```

;; If the node has received a query message of the same detection, then it
;; replies to the sender of this query message

```

```

                (if (and (= m l) (eq w T))
                    [[j <= [:reply nodeId i m]]
                     [time := (+ time 1)])]
                )
)

```

```

(=> [:reply idSender i m]

```

```

    (temporary l n w j ii)

```

```

    (if (eq condition 'idle)
        [
            [ii := (- i 1)]
            [l := (Aref LATEST ii)]
            [w := (Aref WAIT ii)]

```

```

            (if (and (= m l) (eq w T))
                [[n := (Aref NUM ii)]
                 [n := (- n 1)]
                 (SetF (Aref NUM ii) n)]

```

```

;; If a node has received replies from all the nodes of its dependent set, it
;; is the initiator of this detection and it is still idle (since the detection
;; started), then it is deadlocked

```

```

                (if (= 0 n) [(if (= i nodeId)
                                .
                                [[condition := 'deadlocked]
                                 (format t "~&NODE ~S IS
DEADLOCKED" nodeId)

```

|

;; If this node is not the initiator of this detection, it sends a reply to
 ;; the node from which it received the first query message of this detection

```
[[j := (Aref ENGAGER ii)]
 [j <= [:reply nodeId i m]]
 [time := (+ time 1)]]))
```

))

)

)

(group a-group)]

[object node2

```
(state [condition := 'idle]
 [time := 0]
 [DS := nil]
 (SetQ LATEST (Make-Array '(4)))
 (SetQ NUM (Make-Array '(4)))
 (SetQ ENGAGER (Make-Array '(4)))
 (SetQ WAIT (Make-Array '(4)))
```

[nodeId := 2]

)

(script

```
(=> Any @ Reply from Sender
 (inherit Any Reply Sender node1)))
(group a-group)]
```

[object node3

```
(state [condition := 'idle]
 [time := 0]
 [DS := nil]
 (SetQ LATEST (Make-Array '(4)))
 (SetQ NUM (Make-Array '(4)))
 (SetQ ENGAGER (Make-Array '(4)))
 (SetQ WAIT (Make-Array '(4)))
```

[nodeId := 3]

)

(script

```
(=> Any @ Reply from Sender
 (inherit Any Reply Sender node1)))
```

(group a-group)]

;; Initially node4 is the only one that is executing

[object node4

 (state [condition := 'executing]
 [time := 0]
 [DS := nil]
 (SetQ LATEST (Make-Array '(4)))
 (SetQ NUM (Make-Array '(4)))
 (SetQ ENGAGER (Make-Array '(4)))
 (SetQ WAIT (Make-Array '(4)))

 [nodeId := 4]

)

(script

 (=> Any @ Reply from Sender
 (inherit Any Reply Sender node1)))
 (group a-group)]

"THE IMPLEMENTATION OF THE ALGORITHM FOR COMMUNICATION DEADLOCK DETECTION OF CHANDY, MISRA AND HAAS (1983) IN SMALLTALKAGENTS"

UserLib

```

    name: #Pnode
    superclass: UserLib@#ConcurrentObject
    flags: 0x1000
    category: nil
    classInstanceVars: nil
    namedInstanceVars: nil
    classPoolVars: nil
    pools: nil
    structures: nil
    storageSize: 0

```

```

UserLib@#Pnode compileMethodSource: (
'my_id

```

```

^id')

```

UserLib

```

    name: #Pcdead
    superclass: UserLib@#Pnode
    flags: 0x1000
    category: nil
    classInstanceVars: nil
    namedInstanceVars: #(latest num engager wait ds time state id)
    classPoolVars: #(processors mutex)
    pools: nil
    structures: nil
    storageSize: 0

```

```

(UserLib@#Pcdead) metaclass compileMethodSource: (
(Text from: 'newWithID: anInteger

```

```

| processor |

```

" Creation of a processing node with its identification number"

```

processor:= super new initialize: anInteger.
processors at: anInteger put:processor.

```

```
(UserLib@#Pcdead) metaclass compileMethodSource: (
(Text from: 'reset
```

```
" Resetting the threads before running a simulation"
```

```
(mutex isNil)
```

```
  ifTrue: [mutex:= Semaphore newForMutualExclusion].
processors do: [:procl proc terminate].
processors:= Dictionary new.
```

```
(UserLib@#Pcdead) metaclass compileMethodSource: (
(Text from: 'simulate
```

```
" Starts the simulation after creating 4 processing nodes by sending the message detect to
node 1 (node 1 starts a communication deadlock detection"
```

```
self reset.
self newWithID:1.
self newWithID:2.
self newWithID:3.
self newWithID:4.
```

```
(processors at:1) detect.
```

```
UserLib@#Pcdead compileMethodSource: (
(Text from: 'detect
```

```
  | l idnode |
```

```
idnode:= self my_id.
```

```
" If the dependent set of a node is empty, this node is not deadlocked "
```

```
(state = "idle")
  ifTrue:[(ds isEmpty)
    ifTrue:[mutex critical:[
      Console nextPutAll:("Processor #",
        idnode asString," is not deadlocked"); cr.]]
```

```
" If a node is idle and is supposed to detect if it is deadlocked, it first sets the number of the
detection (to be distinguished from other detections of the same node) and signals that it is
idle"
```

```
  ifFalse:[l:= latest at:idnode.
    l:= l + 1.
```

```

latest at: idnode put: l.
wait at: idnode put: true.
num at: idnode put: (ds size).

```

" It then starts the detection by sending query messages to all nodes of its dependent set "

```

ds do:[anode
  (processors at: anode) asynchronousSend: #query:and:and:
  withParameters: (List with: self with: l with: self).
  time:= time + 1.].]

```

UserLib@#Pcdead compileMethodSource: (
(Text from: 'reply: initiator and: number and: source

```

! l n w j idinitiator idnode l

```

```

idinitiator:= initiator my_id.

```

```

idnode:= self my_id.

```

"Monitoring of received messages"

```

"*****"

```

```

mutex critical:[Console nextPutAll:("Processor #",
  idnode asString,"received message #reply with ",
  (idinitiator) asString,
  number asString,
  (source my_id) asString); cr.].

```

```

"*****"

```

```

time:= time + 1.

```

```

changed:= self checkChanges.

```

```

(changed)

```

```

ifFalse:[

```

```

  (state = "idle")

```

```

    ifTrue:[l:= latest at: idinitiator.

```

```

      w:= wait at: idinitiator.

```

```

      ((number = l) and: (w = true))

```

```

        ifTrue:[n:= num at: idinitiator.

```



```

n:= n - 1.
num at:idinitiator put:n.
(n = 0)

```

" If a node has received replies from all the nodes of its dependent set, it is the initiator of the detection and it is still idle (since the detection started), then it is deadlocked "

```

ifTrue:[(idinitiator = idnode)
  ifTrue:[state:= "deadlocked".
    mutex critical:[
      Console nextPutAll:("Processor #",
        idnode asString," is deadlocked");
    cr.]]

```

" If this node is not the initiator of the detection, it sends a reply to the node from which it received the first query message of the detection under consideration"

```

ifFalse:[j:= engager at:idinitiator.
  j asynchronousSend:#reply:and:and:
  withParameters:(List with:initiator with:number with:self).
  time:= time + 1.].
]]

```

]

```

UserLib@#Pcdead compileMethodSource: (
  'Text from: 'query:initiator and:number and:source

```

```

  | l w idnode changed |

```

```

idnode:= self my_id.

```

```

" Monitoring of received messages"

```

```

"*****"

```

```

mutex critical:[Console nextPutAll:("Processor #",
  idnode asString," received message #query with ",
  (initiator my_id) asString,
  number asString,
  (source my_id) asString); cr.].

```

```

"*****"

```

```

time:= time + 1.

```

```

" Changes occur to node 4 at time 1"

```

```

changed:= self checkChanges.

```

```

(changed)
ifFalse:[

(state = "idle")
  ifTrue:[ l:= latest at:(initiator my_id).
    w:= wait at:(initiator my_id).
    (number > l)
      ifTrue:[ latest at:(initiator my_id) put:number.
        engager at:(initiator my_id) put:source.
        wait at:(initiator my_id) put: true.

" If the dependent set of the node is empty, then it is not deadlocked "
      (ds isEmpty)
        ifTrue:[mutex critical:[
          Console nextPutAll:("Processor #",
            idnode asString," is not deadlocked");
          cr.].
          num at:(initiator my_id) put: 0.]

" Receiving a query message that corresponds to a new detection,
a node must send a query message to all nodes of its dependent set
(propagate the query)"
        ifFalse:[num at:(initiator my_id) put:(ds size).
          ds do[:anode (processors at:anode)
            asynchronousSend:#query:and:and:
              withParameters:(List with: initiator with: number with: self).
              time:= time + 1.]].].

" If the node has received a query message of the same detection, then it replies
to the sender of this query message"

          ((number = l) and: (w = true))
            ifTrue:[source asynchronousSend:#reply:and:and:
              withParameters:(List with:initiator with:number with:self).
              time:= time + 1.]].].
]

```

```

UserLib@#Pcdead compileMethodSource: (
(Text from: 'executeFrom:source

```

```

| idnode |

```

```

idnode:= self my_id.

```

" Time is increased to mark the reception of message execute and the state is changed to executing, i.e. node 3 resumes execution and node 4 is removed from node 3's dependent set"

```
time:= time + 1.
state:= "executing".
ds remove:(source my_id).
```

```
mutex critical:[Console nextPutAll:("Processor #",
    idnode asString,"resumes execution at time",
    time asString); cr.].
```

```
UserLib@#Pcdead compileMethodSource: (
    (Text from: 'my_id
```

" Returns the id number of a node"

```
^id
```

```
UserLib@#Pcdead compileMethodSource: (
    (Text from: 'checkChange3
```

" At local time 1, the following happens: node 4 sends a message to node 3 (node 3 was waiting for this message to resume execution), node 4 becomes idle and waits for a message from node 2 "

```
((self my_id) = 4) and: (time = 1))
    ifTrue:[state:= "idle".
        ds add:2.
        (processors at:3)
            asynchronousSend:#executeFrom: with:self.
        time:= time + 2.
        mutex critical:[
            Console nextPutAll:("Processor #",
                (self my_id) asString,
                " becomes idle and waits for message from node2 at time ",
                time asString); cr.].
        self detect.
```

```

    ^true]
    ifFalse:[^false].

```

```

UserLib@#Pcdead compileMethodSource: (
(Text from: 'initialize: anInteger

```

```

" Initialization of instance variables before any communication deadlock
detection starts "

```

```

latest:= List new: 4.
latest atAllPut:0.
num:= List new: 4.
num atAllPut:0.
engager:= List new: 4.
engager atAllPut:0.
wait:= List new: 4.
wait atAllPut:false.

```

```

time:= 0.
id := anInteger.

```

```

(anInteger = 1) ifTrue: [ds:= (List with: 2 with: 3).
                        state:= "idle".].
(anInteger = 2) ifTrue: [ds:= (List with: 4).
                        state:= "idle".].
(anInteger = 3) ifTrue: [ds:= (List with: 1 with: 4).
                        state:= "idle".].
(anInteger = 4) ifTrue: [ds:= (List new).
                        state:= "executing".].

```

END

1 3-0 6-9 7

FIN